Adding an Iambic Keyer to your DDS Development Kit

By Bruce Hall, W8BH

This article will describe how to add a very simple iambic keyer to your DDS Development kit. But wait a second, why bother? There are already good keyers available, with loads of built-in functions. I use a TiCK keyer chip in some of my QRP transmitters, and it works very well. I added keyer code to the DDS kit for three reasons. First, I wanted to learn how to do it. Second, I wondered if I could create something in code (for free) that would save me a few bucks on the keyer chip. And third, I wanted to integrate the keyer with the DDS kit, so that message memories displayed on the LCD as they were sent.

The first thing to figure out is where to attach the key! We need at least 3 I/O lines for a keyer: two input lines for the paddles, left and right, and one output line to key the rig. I already used a bunch of I/O lines for my keypad (see http://w8bh.net/avr/AddKeypadFull.pdf), so my options were somewhat limited. I decided to use the two available pins on Port C for my paddles, and the one remaining pin on Port D for my keyer output.

Bit Number	Port B	Port C	Port D
0	Keypad C2	LCD	DDS
1	Keypad C3	LCD	DDS
2	Keypad R1	LCD	Encoder
3	Keypad R2	Led	Encoder
4	Keypad R3	(unused)	Encoder
5	Keypad R4	(unused)	DDS
6	xtal	Reset	(unused)
7	xtal	---	Keypad C1

WA2MZE has pointed out that you could free up 4 of the keypad I/O lines if you ‘multi-purposed’ outputs of the 74HC164 shift register. This is a great idea, and I might need to rewrite the keypad routines to take advantage of the savings. Unfortunately, these four lines aren’t readily available – you’ll need to solder some wires to the LCD module or PCB if you want to use them.

Calculating Code Speed

Even though I’ve been using CW on and off (pun intended) for many years, I didn’t know much about code speed and timing until a few weeks ago. Perhaps most of you know your Iambic A from your Iambic B. I didn’t! After hooking up my key to pins PC4 and PC5 (and common to ground), I wanted to be able to send some dits. Pretty easy! It is just like blinking the LED, at the start of my keypad experiments. The algorithm looks like this: key down, wait a while, key up, wait a while, done. But how long is a dit? How long do we wait? It depends on the code speed of course, but how?

Code speed, in words per minute, is defined by the number of 5 character words that are sent within one minute. Specifically, the word PARIS is used to calculate speed. PARIS contains exactly 50 elements, where each element is the length of a dit, and 3 elements is the length of a dah. There is also one element between dits/dahs, 3 elements between characters, and 7 elements between words. Here is the breakdown:

P = dit-dah-dah-dit	= 2 + 4 + 4 + 1 + 3 char spacing	= 14 elements
A = dit-dah	= 2 + 3 + 3 char spacing	= 8 elements
R = dit-dah-dit	= 2 + 4 + 1 + 3 char spacing	= 10 elements
I = dit-dit	= 2 + 1 + 3 char spacing	= 6 elements
S = dit-dit-dit	= 2 + 2 + 1 + 7 word spacing	= 12 elements

A better explanation of all this is found on the Kent Engineers website at http://www.kent-engineers.com/codespeed.htm. At 5 WPM, five of these PARIS words or 250 elements are sent in 60 seconds. Therefore the element length is 60/250 = 0.24 seconds. A little algebra gives us a general formula of dit-length (in milliseconds) = 1200/code speed (in WPM).

Getting dit-length in milliseconds is quite handy, because the DDS kit includes a very accurate millisecond timer. The 20.48 MHz crystal frequency (thanks Diz), divided by 1024 internally, yields an interrupt clock frequency of 50 microseconds. This clock is further divided by 200 in the DDS software to give us a 1 millisecond delay. Call the wait routine with the number of milliseconds (say 120) and viola, a timer for 10 WPM! We know enough now to write half of the keyer code:

```
.equ          KeyOut = PD6

DIT:
       rcall  KeyDown
       rcall  DitWait                    ;key down for 1 dit
       rcall  KeyUp
       rcall  DitWait                    ;key up for 1 dit
       ret
DAH:
       rcall  KeyDown
       rcall  DahWait                    ;key down for 1 dah
       rcall  KeyUp
       rcall  DitWait                    ;key up for 1 dit
       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                  ;120ms = 10 WPM code
       rcall  wait
       ret

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

The only tricks here are using the CBI 'clear bit I/O' instruction to set our outputs active-low, and the SBI 'set bit I/O' instruction to set our outputs high again.

Paddle Inputs

These routines compile just fine, but they don't do anything yet. We have to look at our paddle inputs, and call the 'dit' and 'dah' routines accordingly. The AVR instruction set gives us several different ways to check the value of an I/O pin. I use SBIS 'skip if I/O bit is set' to branch according to the value of the paddle input. Here is the code:

```
.equ          LPaddle = PC5
.equ          RPaddle = PC4

CHECKKEY:
;      Checks to see if either of the paddles has been pressed.
;      Paddle inputs are active low
       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
       ret

LPADDLEDOWN:
;      Come here is the left (dit) paddle is pressed
       rcall  Dit                        ;just send a dit for now
       ret

RPADDLEDOWN:
; Come here is the left (dit) paddle is pressed
       rcall  Dah                        ;just send a dah for now
       ret
```

CheckKey does all the work, looking to see if either paddle is pressed. If the left paddle is down we do a dit, and if the right paddle is down we do a dah. For this to work we must periodically check the paddle inputs. We can't check it all the time, since our DDS needs to look for other input, too. But we can put a call to CheckKey in our main program loop, ensuring the paddles are checked many, many times a second.

I got this far and was very pleased with the results. I got dits from the left paddle, and dahs from the right paddle. But a minute later when I tried a CQ, I realized what's missing: there isn't any iambic action. Squeezing both keys together just gives you dits, courtesy of the first SBIS 'Is the left paddle down?' instruction. I've used iambic ever since I built my Heathkit HD-1410 keyer in the 70's. If you favor your single-paddle or cootie key you can stop here, but I needed more! First, we need to be able to detect the squeeze, when both paddles are pressed. I modified my PaddleDown routines to check for the opposite paddle, using SBIS, and then branch to a new Iambic routine if the other paddle was also pressed:

```
LPADDLEDOWN:
;       Come here is the left (dit) paddle is pressed
        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   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
;       Now what do I do???
```

Iambic Mode

What is supposed to happen when both paddles are squeezed together? The keyer is supposed to go 'di-dah, di-dah, di-dah', in iambic rhythm, alternating dits and dahs until you let go. But it is a little more complicated and technically true only if you pressed the dit-paddle first. If you pressed the dah paddle first, then the squeeze would give you dah-di, dah-di, dah-dit instead. (For you poetry scholars this *not* iambic, but trochaic rhythm.)

To get an iambic/trochaic rhythm, we need to know what the previous element was, and then send the other element. If the last thing sent was a dah, we now send a dit, and vice versa. To remember the previous state means that we'll need to store that information in a register or memory byte somewhere. I chose to use a single bit of the flag byte I used in my last project. You can store it somewhere else if you want. Every time we send an element, dit or dah, we store what it is that we sent. Then, when both keys are down, we look at what was sent last and send the other element. Here is the revised code:

```
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
```

I am using the second-lowest bit in temp2 to keep track of dits and dahs. When a dit is sent I clear this bit with the CBR instruction. When a dah is sent I set the bit with SBR. Now when we get to the Iambic routine we can decide what to do: just check the bit and do the opposite of what was done before. This is called Iambic Mode A, which is what I learned and am used to. A good discussion of Mode A and Mode B can be found at the Jackson Harbor website or at http://wb9kzy.com/modeab.pdf. To code for Mode B you'll need to add an extra element at the completion of the squeeze.

What's Missing

This keyer works great for me, but it took a little practice before I stopped dropping elements. Why? Because this keyer does not have 'dit-memory', like almost all keyers do. You can tell the difference if you send a Q: dah-dah-di-dah. With dit-memory you can squeeze-send your dit anytime during the second dah - way before the dit is expected - and it will be sent at the correct time. But with this keyer you must have the paddles squeezed when the second dah has completed and the dit is expected. If not, the dit gets dropped. To add dit-memory you'll need to record paddle-presses *within* an element, and this probably means tying the input lines to interrupts. Let me know if you do!

Also, most keyers will let you change the weighting, or duration, of the elements. I haven't bothered yet, but dividing the element time by 2 will let you change the length of each element in increments of half-a-dit. Dividing by four would give you quarter-dit resolution.

It would be nice to change the code speed without recompiling. You can add a code speed setting in EEPROM, and then let the user change it by turning the encoder knob. See my EEPROM article at http://w8bh.net/avr/EEPROM.pdf for code on how you can save data.

Finally, I still intend to add memory functions to this keyer, so that I can automate my CQ's and other messages. Stay tuned.

Full Code

The entire keyer program takes only 50 instructions, more or less. Amazing!

This code is taken directly from my system, and may include or assume code that I've written for one or more of my preceding projects. I've commented out a lot of the initialization section that applies only to previous projects. Remove the first-column semicolons on these code lines if you are using my previous code.

For the VFO to work correctly you should uncomment the calls to CheckEncoder and CheckButton in the new main program loop, and use the code from my VFO memory project at http://w8bh.net/avr/AddMemories.pdf.

Alternatively, if you just want this keyer code and nothing else, copy the new initialization code into the original source code and put a call to CheckKey in the original main program loop.

```
; Before compiling, manually make the following changes to the source code:
; Just below the label "Menu: ;main program", add this line
;     rjmp W8BH  ; new main program loop
; In dseg, add a single byte variable called flags.

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

W8BH:

;       PORT B SETUP
        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

;       PORT C SETUP
        ldi     temp1,$0F                   ;binary 0000.1111
        out     DDRC,temp1                  ;set PC0-PC3 as outputs
        ldi     temp1,$38                   ;binary 0011.1000
        out     PORTC,temp1                 ;set pullups on PC4-5 & LED off

;       PORT D SETUP
        ldi     temp1,$E3                   ;b1110.0011 (add bits 6&7)
        out     DDRD,temp1                  ;set PD0,1,5,6,7 outputs

;       VARIABLES
        clr     temp1
;       sts     mode,temp1                  ;start mode0 = normal operation
        sts     flags,temp1                 ;nothing to flag yet
;       sts     preset,temp1                ;start with first preset
;       sts    speed,temp1                ;start with default code speed
;       clr    release                    ;no button events on startup
;       clr    hold                       ;no holds on startup

;       COUNTERS/TIMERS
;       ldi    temp1, $07                 ;set timer2 prescale divider to 1024
;       sts    TCCR2B,temp1
;       ldi    temp1, $01                 ;enable TIMER2 overflow interrupt
;       sts    TIMSK2,temp1

;       MISC STARTUP CODE
;       rcall  CheckEE                    ;make sure EEPROM is initialized
;       ldi    temp1,1
;       rcall  DisplayLine1               ;startup message


;*********************************************************
;*  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  CheckKey                   ;check for paddle action
;       rcall  Keypad                     ;check for keypad action
        rjmp   Main                       ;loop forever




;*********************************************************
;*  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   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   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

GETDELAY:
       ldi    delay, 120                 ;set speed at 10 WPM
       ret

DITWAIT:
       rcall  GetDelay                   ;get # of milliseconds for dit
       rcall  wait                       ;and wait that long
       ret

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


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