

A few weeks ago the Four States QRP Group asked me if I would be willing to develop a PIC-based CW keyer that they could sell as a fund raiser for OzarkCon. I wanted to do a keyer for a long time - to implement one with exactly the features I wanted, but never got around to it. This was a good excuse for doing it, and a good cause, so I agreed.

Well, I developed the code and made a board for them and we have a completed product now. The Four States QRP Group has announced its availability yesterday (11/12/2009) and sales are jumping! Initial interest has been VERY strong.

I've updated my web page ([www.cbjohn.com/aa0zz](http://www.cbjohn.com/aa0zz)) to show the keyer and to provide pointers to the 4SQRG group in case any of you may be interested in playing with it. It's cheap - only \$17 shipped (US). I won't talk about the features here since you can get them from the web page or the 4SQRG web page.

I thought some of you might be interested in hearing about the development process. I developed this CW keyer entirely on my PIC-EL III board. In addition, I'm programming all of the production PICs on my PIC-EL board. It is working great.

From the very beginning I really wanted to have 3 pushbuttons for three messages. From my contesting experience I think three messages will handle the basic exchanges without needing to reprogramming the messages during a contest. I also wanted to put the PIC into SLEEP mode when the keyer was not operating, eliminating the need for a power switch. (The ported version of Steve Elliott, K1EL's K8 keyer to the PIC-EL does not use SLEEP mode. It can't.) These two issues were closely intertwined. Here's why. In order to use SLEEP mode, you obviously need to have some trigger for waking it up. I wanted it to use the PIC's port monitoring feature, where any activity on these ports would wake up the PIC, and I wanted to make it wake up on a paddle press or a pushbutton press. However, there are only 4 port pins (RB4-RB7) that the PIC looks at for this wake-up functionality. I resolved the issue with a little "trickery" in the way I implemented the three pushbuttons. Two of them are configured as normal but the third is implemented with a couple of diodes which makes pushing the third pushbutton activate BOTH Pushbutton 1 and Pushbutton 2. Now I just had to implement code to watch for both pushbuttons simultaneously as the trigger for sending message number 3. Yes, it works great. (See schematic for details.)

I decided to run the keyer with the PIC's internal 4 MHz oscillator. It is not as accurate as a crystal but, for the purposes of a keyer, it is very sufficient. This reduced the overall cost of the keyer, since a crystal would have been one of the more expensive parts. Yes the code can still be run on the PIC-EL with the crystal connected. You just configure the internal oscillator (with a different CONFIG statement) and the PIC simply ignores the crystal connection.

I decided to use the same basic command structure that Steve used in his K8 keyer and that we ported it to the PIC-EL. I started out with the standard PIC-EL hardware connections (paddles, speaker, keyed output) that we used in the K8 keyer. This was sufficient to get the basic keyer mechanism working; however, only one of the PIC-EL's pushbuttons was really functionally unique, since two of them use the same PIC pins as the paddles. I then implemented "stubs" - short temporary pieces of code to force execution of areas of the code that I couldn't get to "naturally". For example, I had to do this to force execution of message 2 and message 3, since these pushbuttons weren't physically there.

In the end, when I had the hardware board implemented, I ended up putting two #DEFINE statements in the source code for "PICEL" and "STANDALONE". This was done so I could move the paddles and the three pushbuttons to RB4-RB7 so I could use them for the wake-ups. Then I could simply change one line in the source code, re-assemble it, and change from one configuration to the other. Two different sets of PORT definitions are coded and, with #ifdef and #ifndef statements around the affected code segments, I could turn on the right code at the right time. It works like a charm.

I must share one more trick that I used in developing the CW keyer. Obviously, the timing was critical. I had to figure out how to make the speaker operate at the desired sidetone frequency of 600 Hz. This is a bit lower than the 800 Hz tone that Steve implemented in the K8 but 600 Hz is often thought to be "optimum" for hams as they get older and have more trouble hearing high frequency tones. I designed timing loops that toggled the speaker port with the right timing to make the 600 Hz sidetone.

How did I check it? Two ways. First I loaded the code into a PIC simulator. I really like the PIC simulator by OshonSoft ([www.oshonsoft.com](http://www.oshonsoft.com)). I put breakpoints in the code at appropriate locations and counted the instructions for one complete tone cycle. Knowing that the time to execute one instruction was 1 uS I was able to adjust the constants which control the instructions in the timing loops. Then, in the end, I checked it with my oscilloscope and it was right on.

I did some math, calculating exactly what the theoretical timing should be for ideal Morse code. The three-to-one DAH-to-DIT ratio was implemented and the time between elements was also set to be one DIT-time. I built up constants for loop routines that take exactly one DIT-time to execute, and constructed loops in such a way that they would work when the sidetone was on or off. The tone cycles are a basic building block of these timing loops, of course. Then I made a routine for a "DAH element". It was constructed by simply calling the DIT-time loop three times. You can see that it gets a bit complicated and confusing so I kept careful notes (as comments in the code symbolics) to keep my bearings. I set up the basic timing cycles to have perfect timing with the keyer running at 25 words per minute. Then I simply used proportions to make the constants for the other speeds. I checked the timing with the simulator and verified it on the oscilloscope. All written in one sentence, but it took a long time to actually do.

By the way, just for fun, try figuring out the number of DIT-elements there are in the word PARIS. It actually contains exactly 50 of the basic DIT-elements when you include the 7 DIT-elements for the inter-word spacing at the end. The word PARIS is often used as a message timing standard because it contains the dit-to-dah ratio that is typical of an "average word" in English. Obviously sending a "five" five times does not take the same length of time as sending "zero" five times. Sending PARIS 5 times at 5 WPM should take one minute so that gives 240 mS per DIT-element. Sending PARIS 25 times at 25 WPM should also take one minute, or 48 mS per Dit-element.

There you have it. A neat, fully functional keyer, all developed on the PIC-EL. It was a real pleasure to implement it this way.

In case you are wondering, I am not going to be publishing the source code for this keyer for a while. This is by agreement with the Four States QRP Group - to protect their fund-raising investment. I expect to release it eventually. Code protection is also turned on in the production PICs - so it can't be downloaded and copied. This is different than anything I have done in the past but I hope you understand the reasoning.

I would be glad to answer any questions you might have.

Best Regards,  
-Craig, AA0ZZ