PIC Key, PIC Key

This simple CW keyer is a great way to learn about PICs.

Vladimir A. Skrypnik UY5DJ
Pravdinska - 58
Kharkiv - 107
Ukraine - 310107
[uy5dj@yahoo.com]

Do you recall this phrase: "These simple projects should whet your appetite to learn more about the little PIC microcontrollers you see so frequently"? This comment preceded the article "Using PIC Microcontrollers in Amateur Radio Projects," by John Hansen W2FS, in the October 1998 issue of *QST*. This prediction was certainly true for me! Before reading that article, I was not at all familiar with PIC microcontrollers. To me, they were *terra incognita*. But the article encouraged me to begin learning about PICs by experimenting with programming and by producing my first projects. This article is a direct result of John Hansen's prediction.

PIC microcontrollers, a new generation of electronic components, provide us with fascinating possibilities of eliminating early-on rather large numbers of discrete elements by utilizing the power of programming to provide needed functions. The large printed circuit board, with its multiple conductors performing functional connections between parts of schematics, is supplanted by an invisible program stored in memory inside a single, small chip. The small size of a circuit board containing a single PIC microcontroller, along with a very few discrete components to accomplish input/output functions, belies the latent power of the program stored within the PIC.

The main challenge for the PIC designer is to create a program to implement the project idea. This is a daunting first-time task—at least it seems so before you begin your study of microcontrollers. I have found that the best way to study is learning-by-doing. To begin with, all you need for your home lessons is David Benson's book (see Notes at end of article). This easy-to-understand manual will introduce you to PIC microcontrollers from the inside. Stepping from page to page, you will acquire increasing ability by learning to write simple programs and then checking them with the MPLAB media (see Notes).

Algorithm of simple keyer program

Let's review how an ordinary keyer works. Let's assume that the keyer's output is connected to the transmitter keying circuitry. Inputs are connected to the left and right contacts of the keyer's paddle. Normally, the keyer is in the idle condition: The output is open (or high) and the transmitter is not activated. When the operator presses the paddle handle to the "Dot" contact, the output becomes active and drives either a relay or a transistor connecting the keying circuit to ground and the transmitter starts sending a Morse code dot. The keyer supplies the appropriate length of the dot, as well as a pause in sequence. The durations of both the dot and the pause are equal. When the paddle returns to the neutral position, the keyer, once again, assumes the idle condition. If the paddle is pressed and held in the "Dot" position, the keyer performs a precise series of dots and pauses. The same is true when the operator presses the paddle to the "Dash" position. However, the length of a dash is three times longer than the length of a dot.

Forming precise dots, dashes, and pauses, as described above, will be accomplished by the PIC's program, and a coherent microcontroller program must have a coherent plan; such a plan is generally called an algorithm. **Fig. 1** depicts the algorithm for our project keyer. Referring to **Fig. 1,** keep in mind that the microcontroller, PIC16F84, has 5 lines of port A and 8 lines of port B.

Any line of port A or B can be used as either an input or an output. In this project, we will connect the dot and dash paddle contacts to the port A lines, which will function as inputs. Keyer output and piezo buzzer for audio monitoring will be connected to port B as outputs.

Now, let's examine the operation algorithm diagram, **Fig. 1**. All working steps are marked with rectangular boxes. Box "Start" is the point where the program will actually start to run. When supply voltage is applied to the keyer, the first step in the program is to instruct all port A lines to function as inputs. In the next steps, all lines of port B are instructed to function as outputs, and they also are switched to normal low output levels. Up to this point, the program has only prepared the PIC microcontroller. But continuing from this point, the program will begin to run in the normal idle operation of the keyer. This is marked by the label "BEGIN" on the diagram.

Let me digress a bit from the algorithm diagram and explain how certain dot, dash, or pause durations will occur in the keyer operation. PIC microcontrollers act by stepping under internal clock pulses. Every step is called out as a cycle. Each command instruction has some quantity of instruction cycles. I will not describe each one, or how many cycles it will require. I only want to point out that in order to produce a certain length of dot, we need to calculate how many instruction cycles the microcontroller will use for providing the operation, and how many to add for delaying cycles to establish the proper relationship between transmitting speed and Morse code elements. Delay duration depends upon the delay constants we will incorporate into the program. There are three different constants used: one each for the dot, dash, and pause. There are three counters nominated in the file register's internal memory area. To provide the desired delay, the constants will be put into their appropriate counters.

But let's now return to the algorithm. The box closest to the "BEGIN" label is initialization of the counters. Initialization means to clear counters

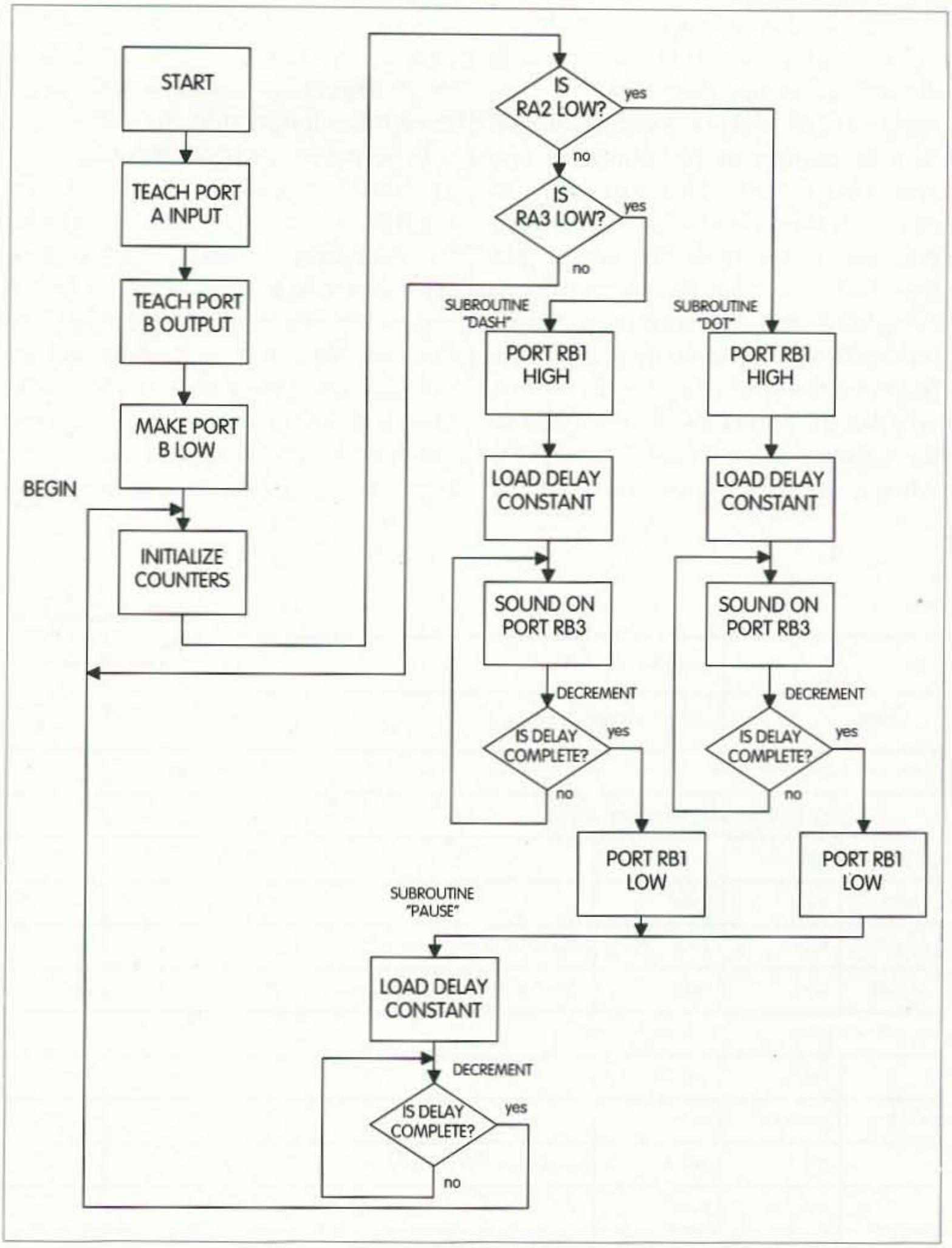

Fig. 1. *Operation algorithm for the PIC-controlled keyer.*

to make them ready for the next operation. The keyer program now sequentially checks to determine if the dot or dash paddle contacts are closed or not. First, it checks the dot input. If port line RA2 is low, the program will call the subroutine "Dot." This is depicted by the right comparison rhomb corner marked with "Yes." If not, RA2 is still high, which means that the dot paddle contact was not closed, and the program will go to check the status of the dash input. If the dash paddle contact is pressed to make port RA3 low (yes), the program will call subroutine "Dash" to form a dash. If not (the dash paddle contact not closed), it will return to the beginning point and continue to run this loop until "Yes" (a dot or dash paddle contact closure) occurs on one of the comparison rhombs.

Let's consider what will happen when the dot is pressed and the keyer begins forming the duration of the dot mark to key the transmitter. First, we have to make the keying output port line RB1 go high. This will cause the transmitter connected to the keyer to start transmitting a dot.

The next box on the algorithm diagram tells us that we have to load the delay constant into the counter. After that, the program will start to generate a sound pulse sequence to operate the monitoring buzzer.

The next rhomb is for decreasing the

counter number by one unit and checking to see if it is equal to zero or not. If the answer is no, this loop will continue until the delay is completed, and then the number in the counter will be decreased to zero. This will cause an exit from this point to the "Yes" direction, and it will make the output port line RB1 low. This means that dot is completed and the transmitter stops transmitting. The same procedures are followed for producing the dash—except that the program will operate under the control of the "Dash" subroutine when it will find a low level on the input port line RA3. The only difference is the delay constant, which is much larger to produce the dash that is three times longer than the dot.

In both cases, when either the "Dot" or "Dash" routine is completed, and the RB1 port line goes low, it will start the subroutine "Pause." This routine must generate a pause between Morse code elements equal to the length of one dot. Note that here we are not including the provision of the audio monitoring signal, which takes some amount of instruction cycles. This pause is controlled by another delay constant—a bit larger one—than the one used for the dot. Subroutine "Pause" works in the same manner as the routines for forming the length of the dot and dash, except that it has its own unique constant loaded into its counter. The delay constant number in the pause counter is decreased by one unit until it is zero. When pause is completed, the program returns to the point labeled "BEGIN" to check for dot or dash inputs by the operator, and the keyer's PIC microcontroller continues to repeat this action until power is turned off.

```
;-------------------------------------------------------------------------------
list            p=16f84
__config        0x3ff3 ; RC clock oscillator
;-------------------------------------------------------------------------------
;               CPU equates (memory map)
porta   equ     0x05
portb   equ     0x06
count1  equ     0x0c    ; for DOT delay constant
count2  equ     0x0d    ; for PAUSE  delay constant
;-------------------------------------------------------------------------------
        org     0x000
start   mov1w   0xff
        tris    porta   ; teach  port A inputs
        mov1w   0x00
        tris    portb   ; teach port B outputs
        clrf    portb   ; all port B lines low
;
begin   clrf    count1  ; initialize counters
        clrf    count2
        clrf    count3
;---------------------   DOT   ---------------------------------------------
        btfsc   porta,2 ; is RA2 low (dot pressed)?
        goto    dash?
        call    dot     ; calling subrouting DOT
        goto    begin
;---------------------   DASH   --------------------------------------------
dash?   btfsc   porta,3 ; is RA3  low (dash pressed)?
        goto    begin
        call    dash    ; calling subrouting DASH
        goto    begin
;---------------------   subroutine DOT   ----------------------------------
dot     bsf     portb,1 ; RB1=1, dot begins
        mov1w   d'12'   ; delay constant
        movwf   count1  ; load const to counter
rptdot  bsf     portb,3 ; sound on
        bcf     portb,3 ; sound off
        decfsz  count1,f ; decrement counter
        goto    rptdot  ; not 0
        bcf     portb,1 ; RB1=0, end dot
        call    pause   ; start  PAUSE subroutine
;---------------------   subroutine DASH   ---------------------------------
dash    bsf     portb,1 ; RB1=1, dash begins
        mov1w   d'37'   ; delay constant
        movwf   count3  ; load const to counter
rptdsh  bsf     portb,3 ; sound on
        bcf     portb,3 ; sound off
        decfsz  count3,f ; decrement counter
        goto    rptdsh  ; not 0
        bcf     portb,1 ; RB1=0, end dash
;
        call    pause   ; start PAUSE subroutine
        return
;---------------------   subroutine PAUSE   --------------------------------
pause   mov1w   d'14'   ; delay constant
        movwf   count2  ; load counter with delay const
rptpau  decfsz  count2, f ; decrement counter
        goto    rptpau  ; not 0
        return          ; counter 0, end pause
;
;---------------------   END of program   ----------------------------------
        end
```

Table 1. *An assembly language program for PIC keyer.*

An assembly language program

The assembly language program for the keyer is presented in **Table 1**. Assembler software will examine this program and ignore all lines beginning from the semicolon. Others perform assembly source code. This part will be assembled by MPASM, the compiler included into the MPLAB integrated development environment from Microchip. The assembler will convert readable text files into hexadecimal code for programming the PIC microcontroller.

The line beginning with the word "list" informs the assembler what type of a PIC microcontroller is used. The next line determines the type of internal clock oscillator built into the device. In this case, it is an RC-type oscillator.

The next five lines are equating statements which assign hexadecimal addresses to file registers in the PIC memory area. The line with ORG (origin) defines the address in memory where the program code starts.

The line with the label "start" in the first column of the program will teach all port A lines to function as inputs by loading hexadecimal FF (or binary 1111 1111) into a special tristate register. Actually, this instruction only needs five "1's," because port A has five input/output lines (named as RA0–RA4) in this type of PIC. Therefore, the three "1's" in the left "F" are functionally superfluous. In like manner, the program will teach all port B lines to function as outputs by loading hexadecimal 00 (binary equivalent is 0000 0000) into this register. The Port B register is also cleared, which means low level statements for each of the eight output lines RB0–RB7.

The label "BEGIN" shows the point where delay counters are cleared. When all three counters are ready, the program begins the "Dot" portion. Here the program checks for the low level at the input port RA2. Electrically, this point is wired to the dot contact of the paddle. If bit 2 of port A is high (paddle is not pressed to dot) in accordance with the instruction "goto," the program goes to the "Dash" portion. However, if bit 2 of port A is low, the next executed instruction will be "call." This means call the Dot subroutine.

In the first subroutine, the line labeled as "dot," bit 1 of the port B is set to "1." This high level will activate the transmitter's keying circuitry to start transmitting a dot. The next two program lines load decimal value "12" (the delay constant) into counter 1. Following this step, the program begins to generate signals for the buzzer. Instruction "bsf" sets to "1" bit 3 of port B. If you remember, previously we made all port bits low. Now RB3 goes high and the buzzer produces one click. But in the next step instruction, "bcf" makes this output low, causing a new click. A fast repetition rate transforms the clicks into a tone.

Instruction "decfsz" decrements counter 1 contents by one unit and compares the result with zero. Until zero has been reached, the instruction "goto" loops to the label "rptdot" to produce new clicks, and continues to decrement the counter until the content of the counter becomes zero—then the following instruction "bcf" will make RB1 low. The dot is now over and the transmitter no longer transmits RF energy.

But subroutine "Dot" isn't over. Instruction "call" will execute another subroutine, "Pause." This begins by loading decimal value "14" (delay constant) to counter 2. The next instruction decrements this counter until the delay is complete and the counter is clear. Note that output port lines RB1 and RB3 are not used in this subroutine. We do not need to either key the transmitter or produce sound. We only need to get a standard length pause equal to the length of one dot. The pause for the audio signal is controlled by counter 2 and a much larger delay constant.

When subroutine "Pause" is over, instruction "return" returns us to subroutine "Dot." But the last instruction here also is "return," and the program goes back to the "Dot" portion. From there the program jumps to the point labeled "BEGIN" to initialize counters again, and starts checking which contact on the paddle is being pressed.

Subroutine "Dash" is the same as

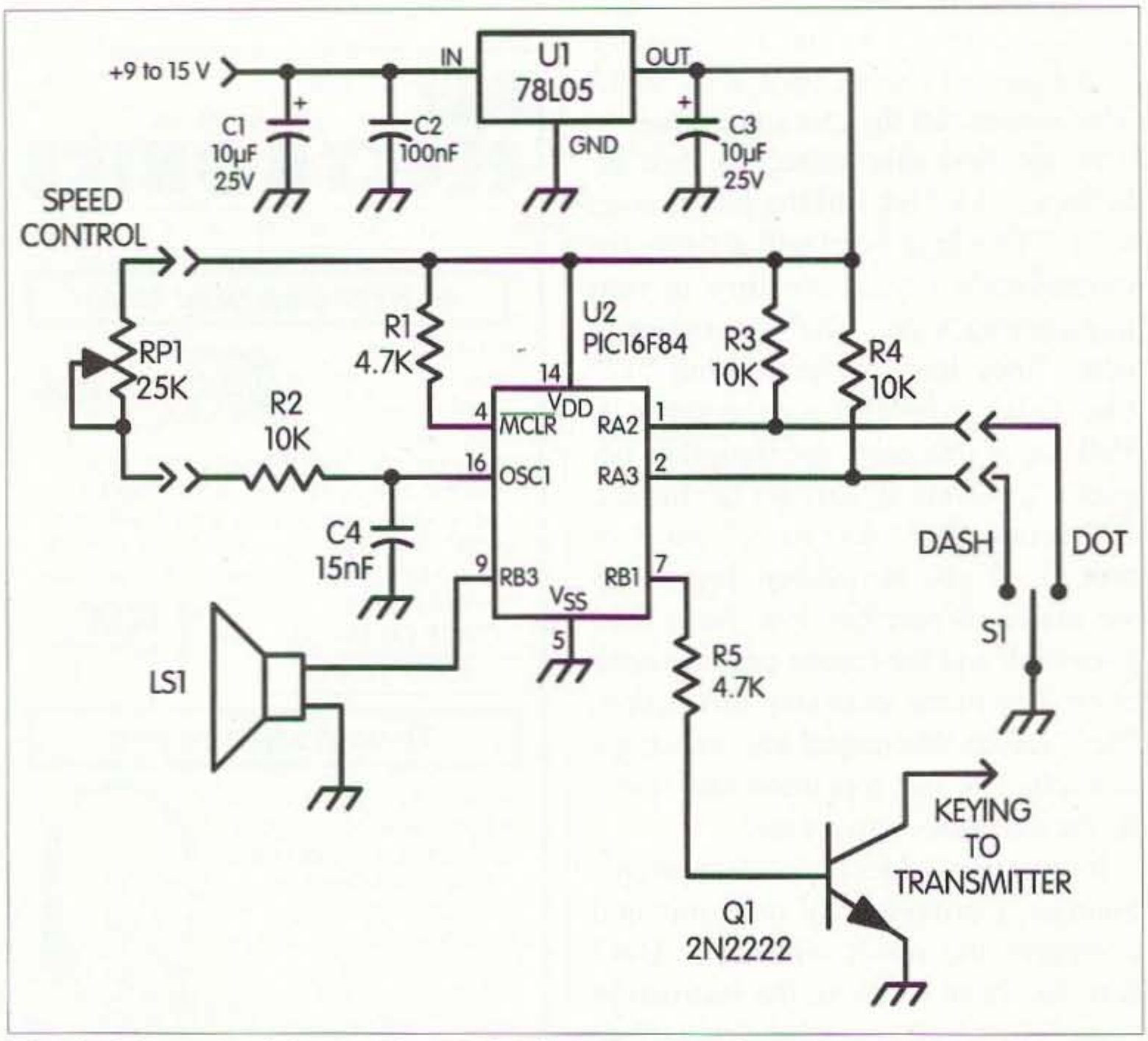

Fig. 2. *Schematic of the simple PIC CW keyer. Unless otherwise specified, resistors are 1/4 W, 5% tolerance, carbon-composition or film units. Appropriate equivalent parts from Digi-Key (DK) can be substituted as shown in* ***Table 2****.*

subroutine "Dot." It is followed by subroutine "Pause" as well. The only difference is in delay constant value. The decimal equivalent is "37," which makes the dash duration almost three times larger than the dot or pause.

Circuit description

Refer to the schematic diagram, **Fig. 2**. The circuit is powered from +5 V voltage regulator U1. Capacitors C1 and C3 provide clear DC, and C2 is for suppression of incoming RF energy from the transmitter.

The keyer itself is microcontroller U2. Resistor R1 keeps the reset input on pin 4 high. Resistors R3 and R4 are pull-up resistors for inputs RA2 and RA3. They provide high idle level at the paddle's dot and dash contacts. Note that I do not specify left or right contacts on the paddle because that is a matter of the operator's taste.

Onboard components R2 and C4 together with outboard potentiometer RP1 are the RC circuitry for the internal clock oscillator. With the component values shown here, the transmitting speed varies from approximately 5 wpm to over 30. To make a more narrow speed range, you may use a higher value of R2.

Signal from pin 7 of U2 is used for keying the transmitter. Q1 functions as a bipolar switch to key the transmitter keying circuitry. When port RB1 goes high it turns Q1 on, thereby connecting the collector network to ground. Resistor R5 is for limiting base current.

The piezo buzzer, connected to pin 9, monitors the transmitted Morse code text. There is another unusual function of the buzzer. You will notice that the buzzer's pitch is related to the clock speed of the microcontroller. When the operator varies the Morse transmission speed by rotating the knob on RP1, it will also vary the sound pitch. At first this may seem like a disadvantage, but the positive effect of this is to make it possible to estimate desired Morse speed just by listening to the pitch of the tone. The lower the tone of one dot, the lower the Morse speed. No need to overload the band with a series of dots to check the transmitting speed. This, of course, is true only for the buzzer's tone, not for the signals heard from your station headphones! Your transceiver uses other methods to get a monitoring tone.

Construction

The keyer was built on a 30 x 35 mm glass-epoxy single-sided PC board (see **Fig. 3**). If you notice my name and callsign, you will understand why metric sizes were cited. Customary English dimensions are approximately 1-1/4 x 1-3/8 inches. I am not familiar with companies outside of the Ukraine that produce custom boards in small quantities. However, I think it is normal practice for radio amateurs to make their own boards.

The assembled board can be installed into almost any transceiver. Limitations will be either not enough room in its case (which seems incredible) or some specific feature of the keying circuitry such as keying with high sink current or high voltage above ground. In this situation, transistor Q1 should be used to drive a small relay with open contacts. Don't forget to include a small silicon diode across the relay coil to manage the inductive spike when the relay coil is de-energized (and, of course, do ensure that

Parts List

Designation	Part
S1	Any type CW keyer paddle
C1, C3	10 µF, 25 V electrolytic or tantalum (DK P5148-ND)
C2, C4	Ceramic (C2: DK P4924-ND; C4: DK P4905-ND)
LS1	Piezo buzzer element (DK P9924-ND)
RP1	25 k potentiometer (DK CT2266-ND)
U1	78L05 small 5 V positive regulator (DK NJM78L05-ND)
U2	PIC16F84 microcontroller (DK PIC16F84-04/P-ND)
Q1	2N222 or any general purpose NPN silicon transistor (DK PN2222ADICT-ND)

Table 2. *Parts list.*

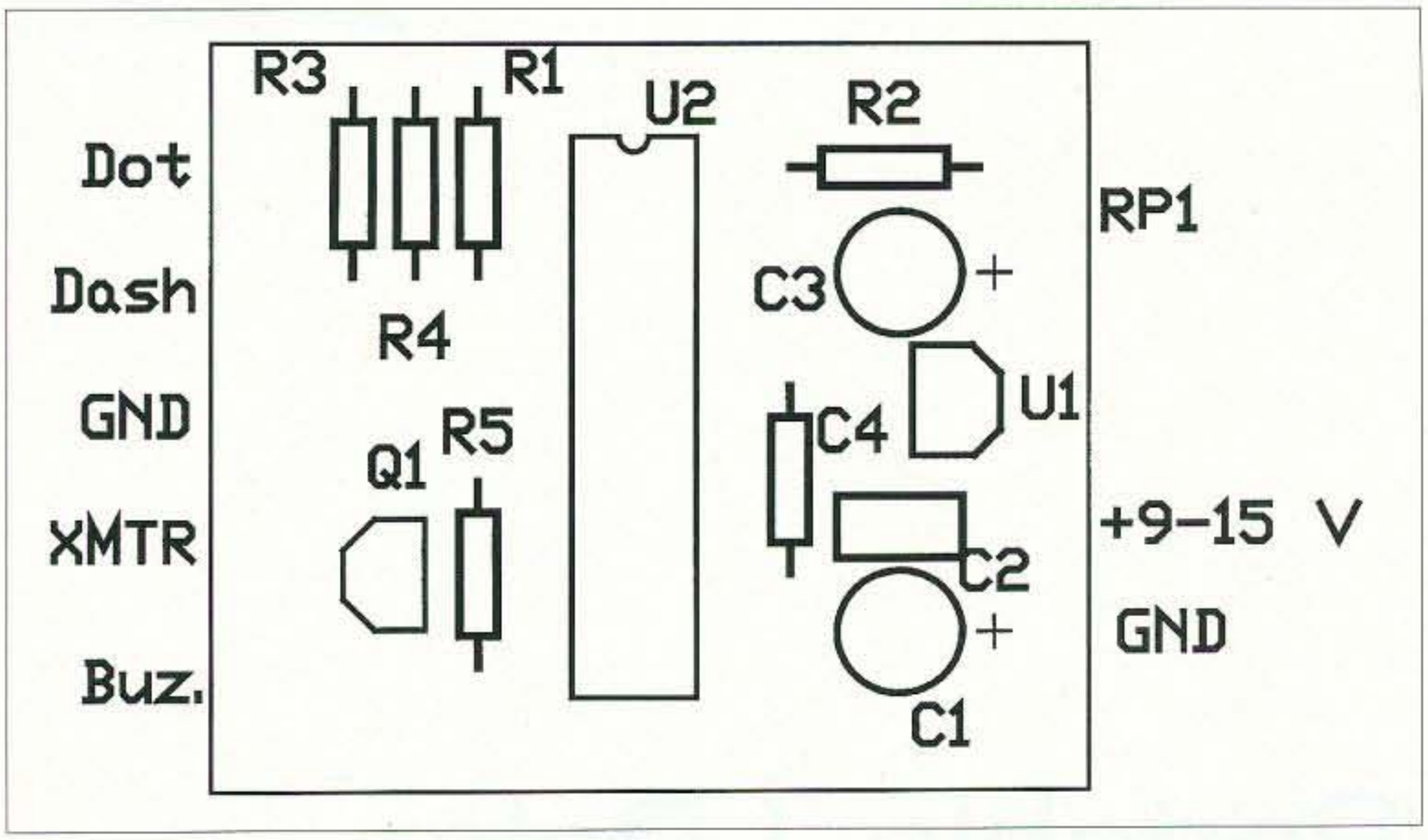

Fig. 3(a). PIC keyer printed circuit board, component side.

the polarity is correct and the diode is not DC conductive when the relay coil is energized).

It is wise to install the keyer board directly onto the keyer paddle assembly. This will ensure the shortest possible input wires, and keep it away from strong RF fields. A metal enclosure to further shield against RF energy in your shack is also a wise idea. The accompanying photo reveals that my keyer is an improvisation (which in the Ukraine is standard procedure due to the cost of living and scarcity of manufactured electronic parts). It is mounted on an old-fashioned telephone polarized relay modified as a paddle. But this is also the amateur radio tradition, and I'm sure you will conjure up your own unique improvisations.

Programming

First of all, you have to work with your assembler program in Microchip's MPLAB software. This is the best environment for design and debugging your programs. This software can be obtained free from the Microchip Web site (see Notes). The assembler compiler MPASM is included in MPLAB and also supplied separately. It may be used to obtain source files for the programmer, but I prefer to use the whole MPLAB package. On the Web site, you will also find a manual for the newest version of MPLAB, with detailed explanations on how to work with this software.

The results of your work in MPLAB will be a file with extension *.hex. It should be used in programming software PIX (see Notes, note 3). Also, you will need the programmer itself. I am using the simple serial port programmer, which was included (along with a detailed description and operating procedure) in W2FS's article.

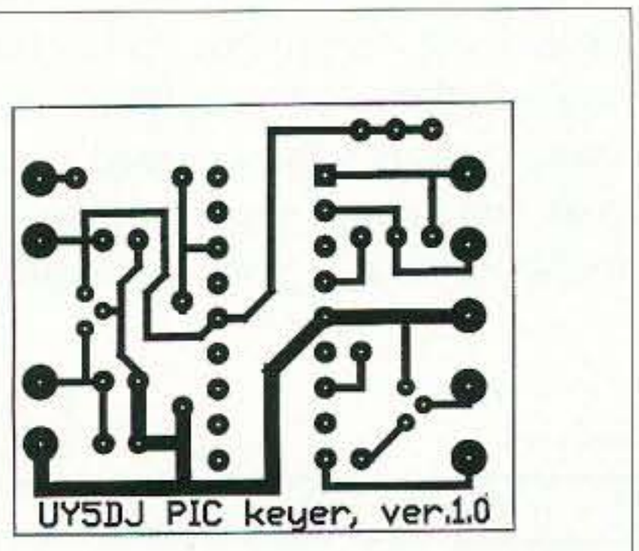

Fig. 3(b). PIC keyer printed circuit board, soldering side.

Summary

This simple keyer is an example of gaining knowledge and skills by self-study, experimentation, and construction—and you end up with a very useful station accessory as well! And, of course, like most amateur radio projects, the project itself is ripe for further improvements and modifications. Keep in mind that the program described in this short article utilizes only a very small part of PIC16F84 capabilities.

In closing, I would like to express my heartiest gratitude to my friend Dave Evison W7DE for his valuable remarks and comments.

Notes

1. David Benson, "Easy PIC'n. A Beginner's Guide to Using PIC 16/17 Microcontrollers." Version 3.0, Square 1 Electronics, 1997.

2. Available at [http://www.microchip.com].

3. Available at [http://home5.swipnet.se/~w53783].

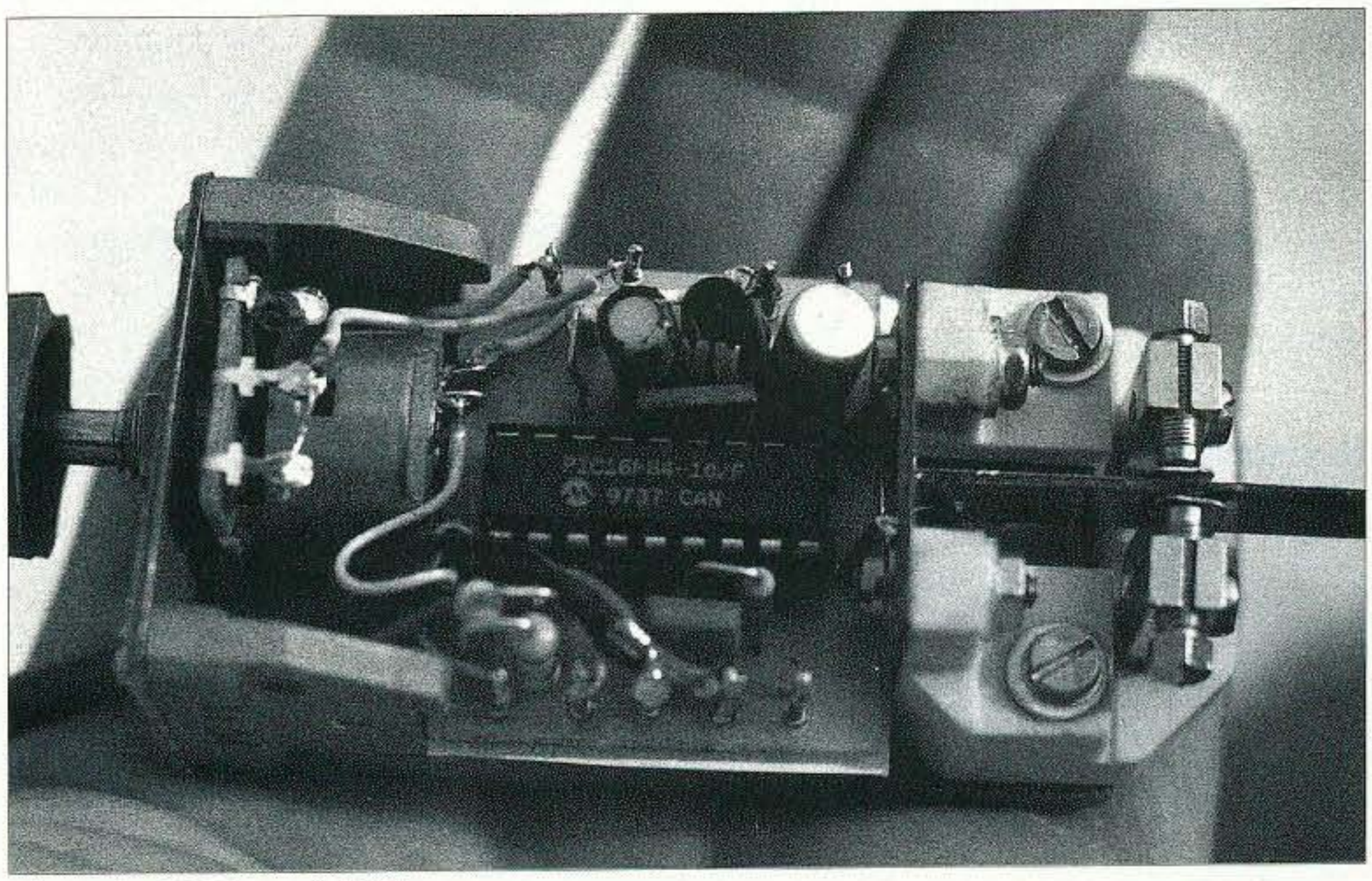

***Photo A.** PIC-based CW keyer.*