

Beacon Keyers Using PIC's

Dave Powis, DL/G4HUP

What are PIC's

PIC's (Programmable Intelligent Controllers?) are the range of embedded microcontroller parts manufactured by Arizona Microchip [1]. Similar ranges of devices are available from other manufacturers, such as Amtel AVR and SGS Thompson. Within these families there exists a wide variety of packages, inbuilt peripheral options and a wide range of computing and storage capacity - from a few hundred bytes to many kB of RAM. This paper is particularly about my experiences in learning to use the small PIC devices to build CW beacon identification units.

Within the device is a processing unit, program storage memory and data storage memory. Input/output (I/O) buses and timer/prescaler functions exist on all devices, and A/D, PWM and LCD driver features are available on some devices.

Programming is performed via a serial link to a PC - the device package ranges include One Time Programmable (OTP), UV EPROM and flash versions (not all devices have a flash version available). The EPROM devices (in non-flash ranges) are ideal for testing code, but are too expensive for general use - the proven code is better loaded into an OTP device, and the EPROM washed ready for the next project. Within each device family there is also a range of package styles, from the traditional DIP to SO and J lead types.

So what do you need & where do you get it...

On the hardware side, you need a PC (Win 3.1/95/98 etc), a programmer, and for non-flash device families, you will need a UV EPROM 'washer'.

Programmers

There are many types available - Microchip, of course, make their own product, but there are many other sources, although they don't all handle all device families, unlike the Microchip offering. Many of these alternative programmers are designed only to program the device that was of specific interest to the designer! Some sources of programmer (both designs for construction, and ready built) are given in [2].

It is worth noting that the 'dodgy satellite decoder' market has given a great boost to the availability of cheap programming devices for the 16C84 range!

Software

I will now attempt to cover the basic areas needed to get going with code development for these chips. There are two initial requirements - a development environment and your application.

Development Environment

Microchip MPLAB Integrated Development Environment (IDE) is a complete suite of development tools covering the entire range - containing editor, compiler, linker, programmer driver, simulation and de-bugging tools running in a Windows environment. This suite comes 'free' with their programmer hardware, but is also downloadable free from their website [3] - but beware, it is quite a large zipfile, and may take sometime, especially at 14k4 or lower speeds! All the examples in this paper will be based on the use of the Microchip IDE and programmer.

Your Application

As with most things, you need to know what you are trying to achieve! Programming cannot be learnt by just reading the manuals. For a first application you should look for some very simple, possibly even mundane or repetitive task - even flashing a sequence of LED's etc. After all, if you are using a flash or EPROM device for testing, it's totally re-usable once you get on to more meaty projects.

I will be using an example of a project developed for the 12C508/9 chips – a simple callsign test generator. It was the use of this device to provide the keyer for the GB3MHS beacon that started my interest – this original keyer was developed and provided by G4FRE [4]. A second example will be discussed during the talk.

12C50x devices

These are particularly attractive and useful devices for small applications, as they are only 8 pin chips, have an internal oscillator and 5 I/O lines - so for applications where the timing not too critical, there are minimal external components - in many cases, none. Just connect the chip to +5v and ground and away it goes. In the applications used here, the accuracy of the internal clock is entirely adequate – the relative timings will be correct, although the character speed may not be precisely as intended, and can change noticeably over wide temperature variations. Fig 1 shows the connections for the applications used in this paper.

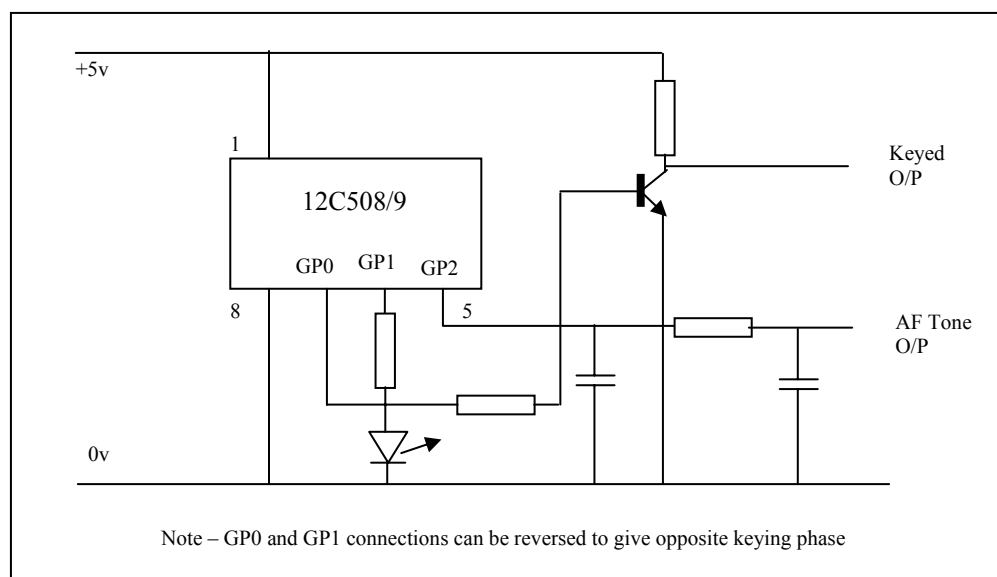


Fig 1 – Schematic diagram for 12C508/9 Keyer

Program and Data Storage

The chip contains two main areas of memory – registers (RAM) and program memory (ROM). The registers are used for storing data variables – some are specific function registers dedicated, for example, as the accumulator (w), port registers (GPIO) or timer (TMR0), whilst others are free for use within your code – they are all manipulated by a common set of instructions.

ROM is used primarily for program code storage, and can also be used for look-up data tables, provided that the data can be programmed at programming time. ROM cannot be used to hold program variables. Some ranges do have devices with EEPROM areas – these could be used for applications where look-tables may need to be modified by the program.

Table 1 shows the RAM/ROM sizes for the 12C508/9

| Device | EPROM | RAM |
|-----------|-----------|-----|
| PIC12C508 | 512 x 12 | 25 |
| PIC12C509 | 1024 x 12 | 41 |

Table 1 – Memory sizes for the 12C508 and 12C509 chips

Code Structure

High level code structure dictates certain positions for certain parts of the program. Firstly, the editor must know which chip your code is for, so that the compiler will carry out the correct conversions. A 'LIST p=12C508' instruction is required to set this up. A Configuration Word is also required, to select the oscillator type for the chip. It sets other global parameters, as well, but these are not significant in this discussion. Following the Configuration Word, the declaration assignments must be carried out - assigning the 'label' that you will use for each RAM location that you need to use.

HINT No 1: Although all of the code examples I have seen do not show the need to declare the dedicated registers before use, I have found it necessary to do so – it doesn't take up any more code space within the chip.

The next section is your main program loop, followed by any sub-routines, and finally any data look-up tables. OK – let's look at an application....

Example – Simple CW Test Identification Generator

The purpose of this design is to generate a repetitive CW message, eg 'test de DL/G4HUP' followed by a period of continuous tone. The code is designed to give two outputs from the chip - one is a positive logic keying waveform, the other negative logic – ie antiphase keying outputs. This facilitates modulation of Tx's needing either sense of keying line.

The full code listing for this project is given in Appendix 1.

CW Storage

The method of storing the CW for this example is to convert the dots and dashes into 0's and 1's based on the timing of a dot – ie a dot is worth one 1 or 0 in timing terms - thus 'T' becomes 111000, where the 111 is the dash, and the 000 is the space before the next character; similarly G becomes 1110101000. The interword space is 5 '0's. The desired CW is converted in its entirety, then the list of 0 and 1's is broken up into blocks of 8 characters, and converted into Hex for easier handling. Thus the first 8 digits of the G above, would become EA in Hex – written as H'EA'. The final aspect to the storage is identifying when the message is complete. This is achieved by inserting a hex word that cannot possibly be a valid CW character – I have used H'00', as there would never be eight successive 0's in any CW message, and it is also easy to test for within the software. Fig 2 shows the CW file for the message 'test de DL/G4HUP'.

| | |
|-------------|------|
| RETLW H'e2' | ; te |
| RETLW H'2a' | ; s |
| RETLW H'38' | ; t |
| RETLW H'3a' | ; d |
| RETLW H'88' | ; e |
| RETLW H'ea' | ; D |
| RETLW H'2e' | |
| RETLW H'a0' | ; L |
| RETLW H'ea' | |
| RETLW H'e8' | ; / |
| RETLW H'3b' | ; G |
| RETLW H'a2' | |
| RETLW H'ab' | ; 4 |
| RETLW H'8a' | ; H |
| RETLW H'a2' | |
| RETLW H'b8' | ; U |
| RETLW H'bb' | ; P |
| RETLW H'a0' | |
| RETLW H'00' | |

Fig 2 - Code file for message 'test de DL/G4HUP'

This technique for storing the CW was developed by Dave Robinson, G4FRE/WW2R, and has been published by him in a number of articles and designs [5, 6]. The CW look-up table can be either written into the chip code directly, or just as modern windows programs allow the insertion of a file into documents, so the MPLAB editor allows externally generated

files to be included in source code for applications. Note – this technique can also be used to insert code modules for routines into programs – a ‘building block’ approach.

Designing the Keyer Application

The first stage is to break the task down into the component operations that must be carried out – Fig 3 shows a flow chart of the keyer.

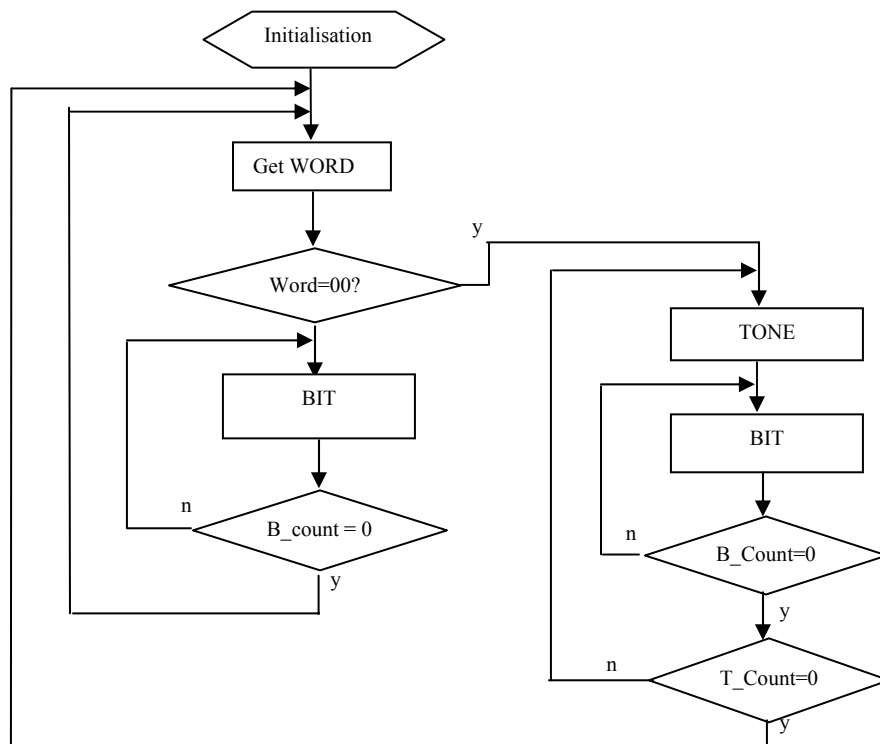


Fig 3 Program Flow Chart for Keyer Application 1

Following the initialisation, shown in Fig 4, the first task is to read the first CW character word from memory, then to process this word, bit by bit, to the output ports until that byte has been completed. The next word is then fetched and the process repeated, testing whether the fetched word is H'00' each time. If it is not, then outputting the word continues. If it is H'00', then the keyer must move on to outputting the tone for the required duration, before starting all over again.

Within this application, there are therefore a number of easily identifiable activities, eg get a CW message byte, step through the bits, load a tone byte... Each of these tasks is also cyclic – CW bytes are read in succession until the end of the message is reached; bits are stepped through until the byte has been read, and ‘tone’ bytes are output until the wanted duration of tone has been created. They all therefore need some means of tracking the progress – the method of detecting the end of the CW message has already been stated, but the other operations use ‘variables’ to count the steps. In each case, the variable is ‘preset’ to a certain value at the start of each piece of code using that variable – then as the code runs and loops around, the variable is decremented by 1 each time, and tested to see when it reaches 0: the loop of code then ends, and the program passes control to the next routine. Fig 4 shows the Declarations and Initialisation for the program. Each register used, including the dedicated ones (STATUS & GPIO), is declared against its physical memory location (in hex). The initialisation section sets up the initial port conditions – keyed output (GP0) to 0, and antiphase output (GP1) to 1.

Next, the CW message handler routine is shown as a code fragment in Fig 5. A counter (W_COUNT) still has to be used, to hold the offset value for the data look-up table. In this case, however, W_COUNT uses an incrementing counter, since the offset of the data in the look-up table will increase by one with each pass.

```

STATUS      EQU H'03'
GPIO        EQU H'06'
B_COUNT     EQU H'07'           ; for setting number of bits to be counted
W_COUNT     EQU H'08'           ; for setting number of text words to be output
T_COUNT     EQU H'09'           ; for setting number of tone words to be output
LOOP1       EQU H'0A'           ; for timing loops in DELAY
LOOP2       EQU H'0B'           ; for timing loops in DELAY
CURRENT     EQU H'0C'           ; temporary location for current data word

ORG 000H

movlw H'fc'           ; port control word
tris GPIO             ; write to set GP0 and GP1 as outputs
bcf GPIO, 0           ; initialise GP0 to 0
bsf GPIO, 1           ; initialise GP1 to 1

```

Fig 4 Initialisation and Declarations for the Simple Keyer Example

```

WORD1:
clrW             ; clear W reg
MOVWF W_COUNT    ; save in reg
WORD:
MOVF W_COUNT, 0  ; read offset into W
CALL TEXT        ; get data word from memory
MOVWF CURRENT    ; save in reg
clrW             ; clear W reg
iorwf CURRENT,0  ; Inclusive OR W with CURRENT, result in W
btfss STATUS, H'02' ; test zero bit in STATUS
GOTO WORD2       ; if false jump
GOTO TONE1       ; true then goto TONE1
WORD2 CALL BIT    ; call bit output routine
INCF W_COUNT, 1  ; decrement W_COUNT
GOTO WORD        ; loop back to word if W_COUNT not 0

```

Fig 5 Code fragment for CW handler routine

Fig 6 shows the code fragment for the BIT handling routine. This uses a pre-loaded value of 8 as the counter, which is decremented by 1 each time a bit is output. Bits are output by shifting the word through the Carry register bit (it's part of the STATUS register). The Status register is bit testable, unlike the accumulator or other variables. If the bit in Carry is a 1 then I/O 0 is set to 1, I/O 1 is set to 0, and vice versa. The counter is then decremented and tested to see whether it has reached 0 - if so the subroutine ends and control is passed back to the calling routine – if not, the operation loops back and outputs the next bit.

```

BIT:
    MOVLW .8           ; initialise bit counter to 8
    MOVWF B_COUNT      ; save in reg
BIT1:
    RLF CURRENT, 1      ; rotate CURRENT left through carry
    BTFSC STATUS, H'00' ; if carry = 1, then
    BSF GPIO, 0         ; set GP0 to 1
    BTFSS STATUS, H'00' ; else set it
    BCF GPIO, 0         ; to 0
    BTFSC STATUS, H'00' ; if carry = 1, then
    BCF GPIO, 1         ; set GP1 to 0
    BTFSS STATUS, H'00' ; else set it
    BSF GPIO, 1         ; to 1
    CALL DELAY          ; call DELAY routine to give correct cw speed
    DECFSZ B_COUNT, F   ; decrement B_COUNT
    GOTO BIT1           ; loop if B_COUNT not 0
    RETURN              ; return

```

Fig 6 Code fragment for BIT handling routine

In this application, it is also through the BIT routine that the timing for the CW is applied – obviously with a 4MHz processor clock the port operation must be slowed down to give humanly readable CW speeds. Each time the bit is set on the port lines a DELAY loop is called before any other actions are taken – thus the port is left set to the value for a ‘dot’ period each time. Again, the DELAY loop is based on decrementing counters, with preset loaded values - see the code fragment in Fig 7. It uses nested loops (one inside the other) to achieve long delays.

```

DELAY:  ; when running set LOOP1 to .125 and LOOP2 to .200
        ; this will give approx 125 ms per CW dot - approx 12 wpm
        MOVLW .125          ; set LOOP1 value
        MOVWF LOOP1

OUTER:
        MOVLW .200          ; set LOOP2 value
        MOVWF LOOP2

INNER:
        NOP
        NOP
        DECFSZ LOOP2, F    ; decrement LOOP2 counter

        GOTO INNER          ; loop if not 0
        DECFSZ LOOP1, F    ; decrement LOOP 1 counter
        GOTO OUTER
        RETURN

```

Fig 7 DELAY code fragment

The TONE routine works in a similar way to the BIT routine - see Fig 8. A variable sets the number of TONE words to be output, and the equivalent of continuous tone (H'FF') is loaded into the CURRENT register as the CW pattern. BIT is then called, and handles the output and timing. After each TONE word is completed, the counter is tested for 0 and either loops back or jumps to the start of the program.

```

TONE1:
        MOVLW .25          ; initialise tone counter to 25
        MOVWF T_COUNT      ; save in reg

TONE:
        MOVLW H'FF'        ; set TONE data word
        MOVWF CURRENT      ; save in CURRENT reg
        CALL BIT            ; call bit output routine
        DECFSZ T_COUNT, F   ; decrement T_COUNT
        GOTO TONE           ; loop until tone finished (ie T_COUNT not 0)
        GOTO WORD1         ; loop back to start

```

Fig 8 Code fragment for TONE routine

Code Compilation

Once the code has been created via the editor, it must be compiled – this is a process of converting the mnemonics and instructions of the ‘source’ code into the ‘object’ code that will be stored in the ROM locations in the chip. All labels are replaced with their memory location addresses, as are the data register references and all instructions are replaced by their hexadecimal equivalents. The ‘Build All’ function carries out compiling and linking of relevant files – eg any ‘include’ files, and writes the hex code file for simulation and programming into the chip.

The Microchip compiler will indicate errors and warnings from its operation on your code – warnings do not necessarily need attention, but may be a source of mis-operation of your application. Errors must be corrected before you can simulate the operation of your application – the .hex file for the application will not be created if errors are found in the compiling process. Fortunately, the IDE is well integrated, and a mouse click on the error report will take you to the offending line in the program - but beware! – the actual fault is often in the preceding line or statements!

HINT 2 – When using data look-up tables in ROM, if you happen to have a data value in the table that is the same as the hex address of a label, the compiler will substitute the label for that data value in the table on compilation! This is easily cured by adding a NOP (no operation) statement as a padding statement to change the hex address of the label.

Program Debugging

Having successfully compiled your code, you now need to simulate its operation and de-bug it. Successful compilation only proves that you have followed the rules for writing statements (the syntax) - it provides no indication at all that your program will actually do what you meant it to..

Simulation is easily carried out – I find the single step mode most useful, where the program is advanced one operation for each mouse click. Windows to show the current position within the code, the register contents and the special function registers (by bit) can all be brought up on screen so you can step through and check all functions – see fig 9.

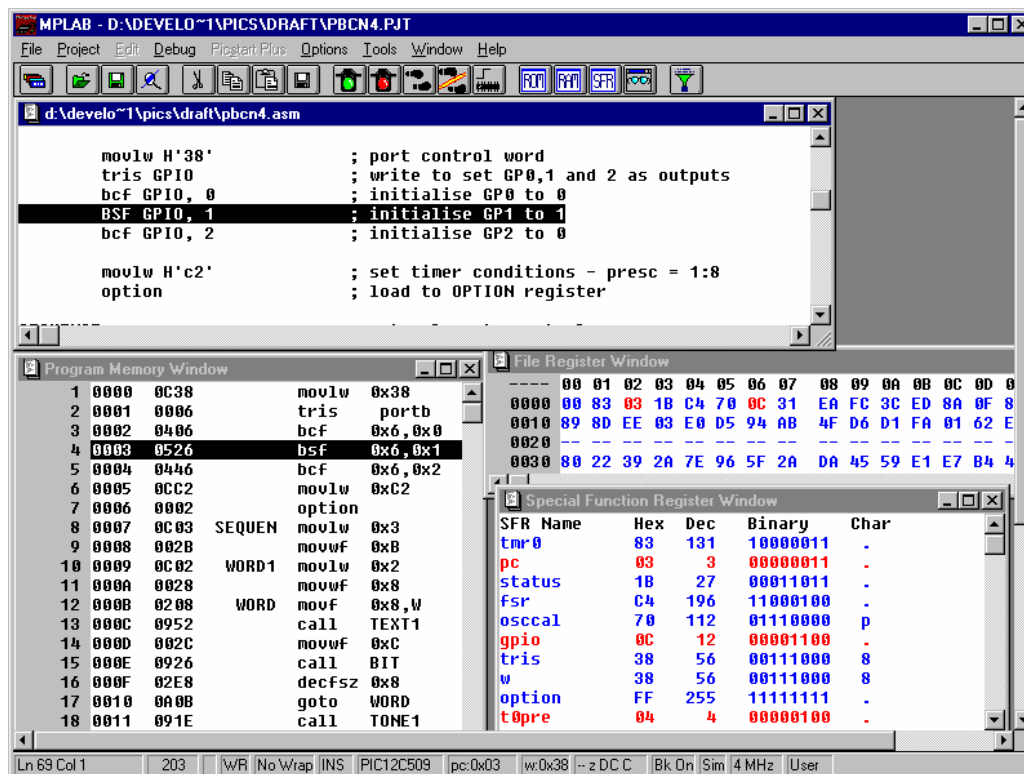


Fig 9. Code de-bugging windows

Where loops with counters are used, especially in timing loops, there is no point in tediously stepping through a large loop using the actual counter preset values – it takes a long time, and gives you cramp! There are only two situations of interest – that the loop does loop when the counter has not reached its final value, and that the loop ends when the counter has reached its final value. So for de-bugging I set all such loop counter presets to a value of 2 - the loop will then execute once in each mode – before simulating code. You must remember to re-edit the counter preset values and re-compile the code before programming a chip!

Another approach which is useful in testing sections of programs without having to step through from the start is to use the Run-To feature – by setting a marker in the editor code window, the code can be made to run through at normal speed until it reaches the marker, and you can then single step from there – all the registers etc will be in the correct states because the code has been executed from the beginning.

For more complex new routines or techniques it is always worth developing a ‘dummy’ program so that you can validate the routine in isolation – this is far easier than trying to sort it out in the middle of a new application program. Write the code

for the routine, and just provide enough support code around it for it to be testable. You can then compile, debug and test this piece of code without the risk of any other program problems confusing the issue – once you know the routine does what you want you can then incorporate it into the application – a bonus of this is that you will also know how the routine works in detail before you get into the serious de-bugging of the application.

Final proving of code is done by blowing it into either an EPROM or flash device, and putting it into the final circuit. This is especially important for applications involving timing, such as those shown here, since the simulation environment, even if allowed to run at ‘normal’ speed gives no indication of the speed at which the output of the chip will operate – there are no visible or audible indicators in real time run mode. I have also found that calculated loop and delay values do not always give the speeds you expect or want – fine tuning has to be done by modifying the timer/counter values, re-compiling, re-programming and listening again!

Further Reading

There is a lot of literature about for these devices, and many source code examples are available free via the Internet sites. There is also a wide range of books, [7] many of which are also accompanied by sample software on disc or CD, and full and fragment listings in the text - many ideas can be found hidden inside other programs – several of the features used in these examples were gained in this way.

Acknowledgements

Firstly, thanks to Dave Robinson, G4FRE/WW2R, for his work in developing the memory of beacon keyers from diode matrices, through EPROMS to PIC devices! Also, of course, I must acknowledge the extremely useful material made available by Peter Anderson - his site has been useful in solving a number of problems, and the very practical books from Nigel Gardener at Bluebird Electronics. The trade marks of Microchip are also acknowledged.

Conclusions

The intention of this paper was to show how relatively easy it is to get started with simple application programming for these useful ranges of devices. The example used is part of a range of applications I developed over a period of approx four months, from a background of no previous experience. I am now quite convinced of the usefulness of these products, where complex logic functions and switching control can be simply implemented – one only has to look at the frequency with which PIC (or similar) based designs are appearing in the amateur press to appreciate their popularity.

The author can be contacted at: dl4mup@qsl.net

References

- [1] Arizona Microchip Website – <http://www.microchip.com>
- [2] Useful PIC URL's – there are many links from these pages:
Peter Andersens PIC page – http://www.phanderson.com/programmer_docs.htm
Bluebird Electronics - <http://www.bluebird-electronics.co.uk>
- [3] MPLAB Integrated Development Environment - <http://www.microchip.com/>
- [4] New 13cm Beacon at Martlesham – GB3MHS: Dave Powis, G4HUP, 43 UKW Tagung, Weinheim, 1998
- [5] A microwave beacon callsign generator and keyer, , Dave Robinson G4FRE, RSGB Microwave Manual, Vol 2, p 9.21
- [6] G4FRE web pages: <http://www.flash.net/~g4fre>
- [7] Bluebird Technical Press, e-mail info@bluebird-electronics.co.uk:
A Beginners Guide to the Microchip PIC
PIC Cookbook Vol 1

PIC Cookbook Vol 2
Greatest Little PIC Book

APPENDIX 1 – Code Listing for CW Identification Generator

```
; pbcn6.asm (12C508/9)
;
; Program to provide 'personal beacon' function. Program will output a continuous message from GP0, comprising
; the text 'test de xxxxx' followed by approx 25 sec of tone, where xxxxx indicates the callsign of the user.
; This code uses an 'include' file to import the CW message.
;
; The output speed is approx 12 wpm. GP1 gives an inverted output, which can be used to drive an LED, whilst GP0 drives the modulator.
;
; The duration of the message can be modified by changing the text stored in the data table.
;
; The program gets a data word from memory and outputs it as CW - it then steps through the remaining words until it encounters a data word of '00'
; - an illegal character in CW. The CW is stored in normal order, unlike earlier pbcn programs. This program is thus compatible with work by
; G4FRE/WW2R.
;
; The duration of the tone can be changed by modifying the value in t_count, and the speed of the cw can be altered by changing the value of 125 in
; the 'loop' routine.
; t_count adjusted to 25 (should give 25s of tone) 3/1/99.
; LOOP values adjusted to give better timing 6/1/99.
;
; version 0.A 21st Feb 1999 Initial code changes on pbcn1.asm
; version 0.B 8th March 1999 CW message replaced by include file
;
; copyright Dave Powis (G4HUP) Feb 21st 1999
```

```
LIST p=12c509
```

```
__config 0a ; configuration bits MCLR=0, CP=1, WDT=0, INTRC=10
```

```
STATUS EQU H'03'
GPIO EQU H'06'
B_COUNT EQU H'07' ; for setting number of bits to be counted
W_COUNT EQU H'08' ; for setting number of text words to be output
T_COUNT EQU H'09' ; for setting number of tone words to be output
LOOP1 EQU H'0A' ; for timing loops in DELAY
LOOP2 EQU H'0B' ; for timing loops in DELAY
CURRENT EQU H'0C' ; temporary location for current data word
```

```
ORG 000H
```

```
movlw H'fc' ; port control word
tris GPIO ; write to set GP0 and GP1 as outputs
bcf GPIO, 0 ; initialise GP0 to 0
BSF GPIO, 1 ; initialise GP1 to 1
```

```
WORD1:
    clrw ; clear W reg
    MOVWF W_COUNT ; save in reg

WORD:
    MOVF W_COUNT, 0 ; read offset into W
    CALL TEXT ; get data word from memory
    MOVWF CURRENT ; save in reg
    clrw ; clear W reg
    iorwf CURRENT, 0 ; Inclusive OR W with CURRENT, result in W
    btfss STATUS, H'02' ; test zero bit in STATUS
    GOTO WORD2 ; if false jump
    GOTO TONE1 ; true then goto TONE1

WORD2:
    CALL BIT ; call bit output routine
    INCF W_COUNT, 1 ; decrement W_COUNT
    GOTO WORD ; loop back to word if W_COUNT not 0

TONE1:
    MOVLW .25 ; initialise tone counter to 25
    MOVWF T_COUNT ; save in reg

TONE:
    MOVLW H'FF' ; set TONE data word
    MOVWF CURRENT ; save in CURRENT reg
    CALL BIT ; call bit output routine
    DECFSZ T_COUNT, F ; decrement T_COUNT
    GOTO TONE ; loop until tone finished (ie T_COUNT not 0)
```

```

        GOTO WORD1                ; loop back to start

BIT:
    MOVLW .8                      ; initialise bit counter to 8
    MOVWF B_COUNT                ; save in reg

BIT1:
    RLF CURRENT, 1               ; rotate CURRENT left through carry
    BTFSC STATUS, H'00'          ; if carry = 1, then
    BSF GPIO, 0                  ; set GP0 to 1
    BTFSS STATUS, H'00'          ; else set it
    BCF GPIO, 0                  ; to 0
    BTFSC STATUS, H'00'          ; if carry = 1, then
    BCF GPIO, 1                  ; set GP1 to 0
    BTFSS STATUS, H'00'          ; else set it
    BSF GPIO, 1                  ; to 1
    CALL DELAY                   ; call DELAY routine to give correct cw speed
    DECFSZ B_COUNT, F             ; decrement B_COUNT
    GOTO BIT1                    ; loop if B_COUNT not 0
    RETURN                       ; return

DELAY: ; when running set LOOP1 to .125 and LOOP2 to .200
        ; this will give approx 125 ms per CW dot - approx 12 wpm
        MOVLW .125               ; set LOOP1 value
        MOVWF LOOP1

OUTER:
        MOVLW .200               ; set LOOP2 value
        MOVWF LOOP2

INNER:
        NOP
        NOP
        DECFSZ LOOP2, F           ; decrement LOOP2 counter

        GOTO INNER               ; loop if not 0
        DECFSZ LOOP1, F           ; decrement LOOP 1 counter
        GOTO OUTER

        RETURN

TEXT:
        ADDWF H'02', 1            ; add offset to PC
        include "cw_hup1.asm"

        END

```