Software Manual

This note describes the installation and use of the PIC software currently available for the General Purpose PIC Controller. Three Ready-to-Go packages can be supplied, either as .ASM/ HEX code for you to do your programming, or (at additional cost) as ready programmed PICs.

Note on Assembly Files and structure

All PIC assembler code is supplied with these packages for you to play with and use as a basis for modifying and developing your own code. Many routines in here are common across most of the software designed for this module. A list of the most used ones are given at the end. I strongly advise you to at least look at the code in the .ASM files, in conjunction with a data sheet for the PIC device open at the Instruction Set Summary. See also ‘A Bit More on (my) Assembly File Structure’ at the end of this document

4-Channel Voltmeter

A 16F819 device with its integral A/D converter contains the code ***4chanVmeter*** This PIC has a maximum of 5 analogue input channels, although for our purposes one of these, A3, is allocated to the voltage reference. A MAX6004 4.096V precision reference should be installed on the PCB in the position marked.

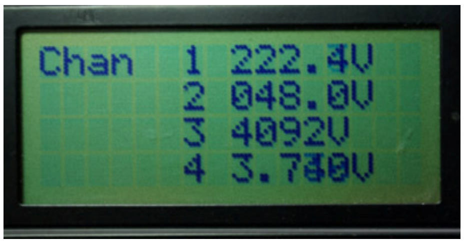
The remaining four analogue inputs go to pins labelled 0,1,2,4 on the 8-way header, which pass to the PICs pins via optional series resistors and filtering / attenuation. For voltages in the range 0 – 4.095, connect a resistor in the range 300Ω to 1kΩ in series in the R1 – R3 and R5 position). R4 should no be installed unless you need to make use of the voltage reference output for any external analogue conditioning such as adding voltage offsets. Capacitors whose values are non-critical can be added as nose filtering in the C1-C3 and C5 positions.

The ***4chanVmeter*** software allows the position of the decimal point in the display to be shifted to cope with voltage dividers of 10, 100 and 1000. For 10X, allowing voltages of 0 – 40.95V to be displayed, a 10:1 potential divider can be made from a 9.1kΩ resistor in parallel with a 820kΩ in the R1-R5 positions (piggy-back the two) and by adding 1kΩ resistor in the C1-C5 positions. If filtering capacitors are still wanted here, they can be piggy backed on top of the resistors.

For X100 operation, the divider is made up from the same 1kΩ resistors to ground, but the series resistor of 99kΩ is best made up from three 33kΩ resistors mounted off board to allow the higher voltage to be shared across all of them, and keep it away from the sensitive PIC module. This is certainly essential for X1000 4kV measurements!

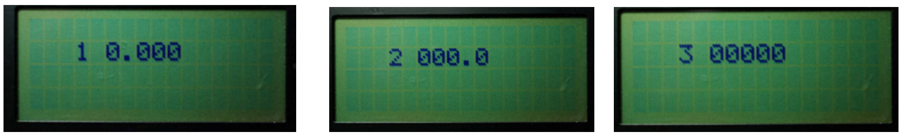
Changing the Display Format

When turned on the display will briefly show an introductory message, then start showing the four voltages in a display similar to that shown. Chan 1 is the voltage on the pin labelled 0 on the 4-way header, Chan 2 that on pin 1, Chan 3 on pin 2 and Chan 4 on pin 4.



```
Chan  1 222.4V
      2 048.0V
      3 4092V
      4 3.780V
```

To change the position of the decimal point in any channel, push down and hold the push button for two seconds. The display should then change to that of Figure 2 .



```
1 0.000
```

```
2 000.0
```

```
3 00000
```

Rotating the rotary controller will then shift the display between 00000/0.000/00.00/000.0 When it is showing the correct format for Channel , press the button briefly, and the display will change to showing the format for channel 2. Rotate the knob to obtain the desired format and push briefly to store the format for this channel. Repeat for channels 3 and 4. Pressing the button will rotate through all four channels repeatedly allowing you to alter the position of the decimal point. When all are set to your satisfaction, press and hold down the button for 2 seconds to save the format in non-volatile memory. These settings will remain until altered again in the same way.

Voltmeter Software Details

Refer to the source code listing in *4chanVmeter.asm* . The PIC code selects each analogue channel input in turn and connects this to the internal 10-bit A/D converter. The A/D output is a number in the range 0 – 1023, corresponding to 0 – 4.092V on the input with a step size of 4mV. In the subroutine ReadADAv 16 successive measurements are made on the selected channel and the results summed to give a total in the range 0 – 16383. This combining of 16 readings performs averaging on the input signal and serves to increase the measurement resolution. According to the maths of statistics, an improvement equal to the square root of the averaging factor can be expected; so 16 times averaging ought to give four times, or two extra bits worth of resolution. Which is very convenient, as the resulting 12 bits effective accuracy from the A/D is equal 4096, which just happens to be the reference voltage!

In the routine ReadScaledAD the summed result is multiplied by a constant number stored in one of four EEProm locations – one corresponding to each channel. The default values for all four are fixed at 64 for each channel, so the final measurement for the first channel is a number in the range zero to 16383 * 64 = 104851 This is divided by 256 by the simple expedient of throwing away the least significant byte, leaving a two byte value holding a number in the range 0 – 4095. This is converted to BCD and sent to the display. The whole process is then repeated for the other three measurement channels

The main loop is continually testing the button input line press and measuring the duration of any press. Action is then taken on long or short presses accordingly. Study the source code listing for full details. The names of all the appropriate subroutines should be self explanatory, and all relevant sections of the PIC code are commented.

To gain experience with PIC programming, try adjusting the four constants stored in EEProm to give scaling factors other than just the powers of 10 corresponding to decimal point placement. A value of 32 stored in the appropriate location, for example, will cause a value to be displayed in the range 0 – 2.047 (or 20.47, or 204.7) for that channel. Other numbers can be selected for arbitrary scaling in the range 1 – 255, allowing multiplication of between 1/64 to 255/64 (.0156 to 3.98) to be applied to the displayed value. To change these EEProm –stored values, a PIC programmer such as the PicKit 2 and a code assembler like MPASM will be needed .

For further enhancements, modify the code to for stored offsets – addition subtraction routines can be found in other listings within this archive.

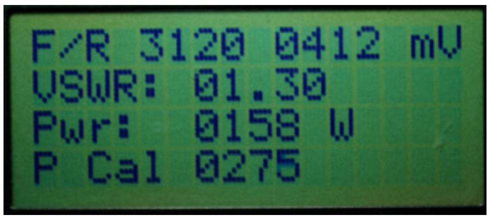
Auto SWR Bridge Display

To be able to use this PIC code, you will first have to make/obtain/modify an RF head to deliver two DC output voltages corresponding to Forward and Return signals. Ensure this is properly balanced, ie gives similar voltages on FWD and REF when the direction of RF flow is reversed, and that the voltage output during normal operation, including under mismatch conditions does nor exceed 4.09V For accuracy and linearity, it is usually better to design for a higher voltage out from the diodes and use a potential divider to bring this down to the input voltage range – which also allows for calibration and balancing.

Three analogue measurement channels are needed for the full SWR and power indication; Chan 1 (Pin-0 on the 8-way header) takes the FWD voltage, Pin-1 the REF and pin-2 the power calibration input. The power calibration uses the 4.096V reference output, so a resistor of a value determined below needs to be installed in the R4 position. Non-critical valued resistors of a few hundred ohms to 1kΩ can be installed at R1 – R3 positions with C1-C3 to add RF decoupling and filtering.

SWR is calculated by measuring the voltage on the FWD and REF inputs and calculating the ratio between them. Provided the analogue channels are matched, no other calibration is needed. Power measurement, on the other hand, requires user calibration, which can only really be made in conjunction with another, calibrated power meter. Therefore, the digital power reading here is obtained by measuring the FWD voltage reading, squaring it (as Power is proportional to V^2) then multiplying by a value obtained by measuring a voltage on the third channel. This is turn is set during the calibration process and can be defined by a preset resistor, or preferably by select-on-test fixed resistors. Several switched resistors can even be substituted for use with different RF heads, and the individual ones can even be built into the heads themselves, with a standard multiway connector connection used to carry the DC voltages, allowing plug-and-play swapping.

There is no setup on the PIC Module; after switch on and the introductory message, the display jumps straight to that shown:



```
F/R 3120 0412 mV
VSWR: 01.30
Pwr:  0158 W
P Cal 0275
```

The top line reads shows the input voltage to FWD and REF channels (in mV) with the second line indicating the VSWR calculated from these. In the example shown VSWR = (3120 + 412) / (3120 - 412) = 1.30

The bottom line shows the A/D converter output on channel 3, the power calibration input, on a scale of 0 – 1023. The power calculation is formed from the forward voltage reading and this calibration as described below.

SWR and RF Power Display Software Details

Refer to the source code listing in *SwrMeter.asm* . Inside the main loop the FWD and REF values are read and converted to mV for the first line of the display, in a manner similar to that for the voltmeter described above, except that the A/D is just read once with no averaging.

The power calibration voltage is read and multiplied by a constant value of 16 (stored in EEProm to allow for a later change) to give a Power calculation constant in the range 0 – 16368

The ReadPwr routine is then called which makes 16 consecutive readings of the FWD voltage and sums them to generate a 14 bit number in the range 0 – 16383. This is multiplied by itself (squared) and the lowest byte of the result thrown away (divide by 256) to generate a value from 0 – 4095 which is multiplied by the Power cal constant above. The resulting number, is interpreted as Watts * 65536. So the lowest two bytes are thrown away (divide by 65536) and the result in the range 0 – 1022 is interpreted as Watts, converted to BCD and displayed.

This gives an upper bound to the power display. If the channel 2 cal voltage is at full scale, 4.095V, then a FWD voltage of 4.095V will correspond to 1022 watts. By changing the Cal voltage input, this power level can be made any value. Once calibrated for a known FWD voltage / power combination, provided the Cal voltage remains fixed, and change in FWD will result in a correctly tracking power reading. (See the note at the end about diode drops and linearity)

In the ReadSWR routine, FWD and REF voltages are again read as 16 summed and averaged values. The FWD reading is multiplied by 2^{16} by appending two zero low-order bytes and FWD divided by REF in the routine Divide32x16 The output from this is the reflection coefficient, Rho' , as a value from 0 – 65535. Any spurious or erroneous values are checked for (like REF greater than FWD – it can happen with badly adjusted SWR heads) and substitute default values inserted.

The calculation 100* (64K + Rho') / (64K – Rho') is then made which is converted to BCD, a decimal point inserted and sent to the display as VSWR in the range 0 to 99.99 Refer to the .ASM listing for full details, as the promotion and scaling needed for the integer maths gets a bit involved. Hopefully the comments in the assembler listing explain the processes satisfactorily.

Linearity and Diode Drops

The SWR and power calculations assume the voltage in is linearly related to RF power which is not the case for diode power detectors. While it would be possible to insert a fixed value of diode drop into the PIC calculation, or even a first or second order approximation without too much effort it was decided this would complicate things too much. If the SWR head is designed to give 10 – 15V for full output power, then the effect of the diode drop on the resulting SWR and Power values is minimal. Accuracy worsens at lower power levels as voltage drops, but reading accuracy, especially SWR, are in most practical cases of less importance here.

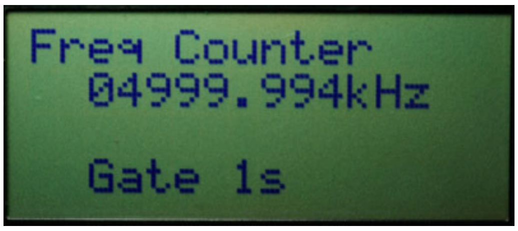
As an exercise in programming, try adding a fixed 'diode drop' voltage to the FWD and REF readings. Assume say 0.65V constant drop, and work this out in terms of a 4.096V reference with a 10 bit A/D (all right, its equal to 162). Adding this to each raw A/D reading is the simplest solution.

Frequency Counter

The ***FreqMeter*** software implements a frequency counter with selectable gate times of 10ms, 100ms and 1s allowing 100Hz, 10Hz and 1Hz frequency readout resolution respectively.

The 16F628 PIC is used in this design (although the 16F819 has all the necessary peripherals there are a few subtle differences in setting up and EEProm reading). A useful peripheral function inside this PIC is a 16 bit counter / timer – TIMER1 - that can be clocked at several tens of MHz from an external source, and read by the PIC . The vagaries of allocation of I/O pins means the input has to be made via PORTB, 6 which is one of those brought out to the 4-pin header otherwise used for in-circuit programming . The one labelled C. This has a Schmitt trigger input, so add an external pair of resistors of around 1kΩ each to bias at Vcc / 2 and AC couple the RF onto this pin though a capacitor of 100nF. A RF input of 4dBm or more in then enough to operate the counter.

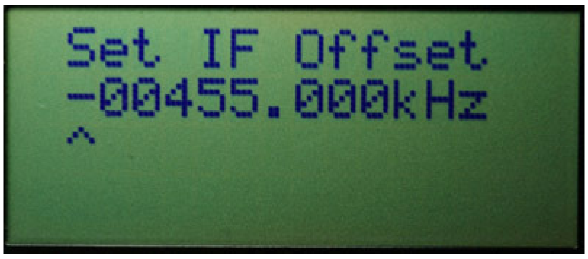
The data sheet is a bit vague about the maximum speed capability of this counter – suggesting it is 20MHz, the same as the processor clock. However, actual tests (on a not-very-statistically-valid set of two chips from the same batch) show that it will run at frequencies up to about 80MHz; so to play safe lets just call it 50MHz maximum.



For normal operation with the stored IF offset, turn on and after an introductory message the display shown will appear. To toggle gate times press the button to shift though 10/100/1000ms. On the 1s gate time setting, the button may have to be held pressed for up to 1s before it is detected.

IF Offset

A user defined IF offset which can be added to, or subtracted from, the input frequency can be stored in the PIC in EEProm. To change the stored value, hold down the pushbutton while switching on, or resetting the controller. Keep pressing the button until the introductory message goes and the display goes blank. On releasing the button, the IF offset setup display will appear;



Press the button briefly to move through the digit to be changed, as indicated by the cursor. The rotary control is then used to set that digit. When under the sign, the rotary control is used to toggle between plus '+' or minus '-' offset. TO complete the entry, press and hold the button for 2 seconds, whereupon the new value will be stored to EEProm and normal operation resumes.

Negative Frequencies

The stored IF offset is added to, or subtracted from the input frequency. If a negative IF offset is stored that is significantly greater than the input frequency, the resulting display will correctly show a negative frequency. This capability has been included for upconverting receivers where the IF lies above the input band. In this situation , $F_{IF} = F_{RF} + F_{LO}$, so the wanted indicated frequency (F_{RF}) is obtained by measuring F_{LO} and subtracting the higher value of F_{IF} to give a negative result.

Frequency Meter Software details

Refer to the source code listing *FreqMeter.asm*. As the TIMER1 counter is only 16 bits in length, it has to be continually tested for overflow by testing the flag in the interrupt enable register. Each time the counter overflows, another 16 bit counter made up of the two registers FreqCountHi/Lo are incremented, giving a 32 bit count overall.

The main loop first calls the critical timing part of this software, the routine MeasureFreq Inside this routine, the combined 32 bit counter is first cleared and TIMER1 enabled to start it counting from the input signal. Another 16 bit counter GVHi/Lo is preloaded with a value defining the gate period. A loop is organised so is continually testing for overflow and decrementing a loop counter for each pass though with all branches and tests being carefully equalised to have constant length (number of clock cycles) by padding out with strategically placed nop statements. A constant delay of 16 clocks is achieved for each pass though. After the gate defining counter has decremented to zero after the required number of passes though the loop, TIMER1 is stopped and the resulting value frozen. The value to be prestored in Gv in therefore equal to the gate period in seconds, multiplied by the clock frequency of 1MHz and divided by the 16 clocks in the loop. 1s requires a value of 62500 (which just fits into a 16 bit count), 100ms gate period needs GV = 6250, and 10ms GV = 625

The 32 bit count now holds the number of input cycles in either 10ms, 100ms or 1s gate time and needs to be corrected by the IF offset before display. The IF offset is stored in EEProm in units of

1Hz, so at the lower gate times, the stored value has to be first divided by 10 or 100 before being added to or subtracted from the measured count. The result of the calculation is then converted to BCD, a decimal point inserted at the appropriate place for showing the frequency in kHz, and sent to the display. The main loop also tests for button presses, and when detected, changes the gate value accordingly, restarting the counter.

At start up the button is tested and if pressed the PIC enters the UserSetup block of code. This monitors for short or long button presses, and reads the rotary encoder for count-up or count-down pulses. The rotary encoder is read as an interrupt driven routine which also makes use of the Timer0 peripheral in the PIC device for contact debouncing. See *FreqMeter.asm* for more details.

Accuracy and Resolution

Ultimate frequency measurement accuracy depends on the crystal oscillator being set to 4MHz for a resulting 1MHz clock rate. A trimmer capacitor can be installed on the PCB to facilitate this. Use a separate receiver or frequency counter to set this as accurately as possible. Try to use RF leakage only to detect the signal for setting purposes. Any probe capacitance on the PICs osc pins will shift the frequency leading to inaccurate setting.

All frequency counters have an inherent counting uncertainty of one count, but there is another annoying error in this particular application that is not easy to correct for. In order to read the TIMER1 register, the PICs internal load has to be synchronised with an edge of its 1MHz clock. This results in an additional delay on the gating period of ‘about’ half a clock cycle, or around 0.5us. At 1s gate period, this corresponds to 0.5 parts per million loss in accuracy, so introduces 5Hz error on a 10MHz signal. At shorter gate periods the error is less noticeable as it is now below the inherent LSB resolution) Since the error introduced is now related to the input frequency it can’t be corrected by subtracting a constant offset. It also can’t be corrected by tweaking either the clock frequency, or the loop delays as the tweak would have to be changed for each gate period. So I ignore it and just accept a reading a bit on the low side at maximum resolution. This is one reason, (amongst several others) that a 10s gate time is not made available.

As a more advanced programming exercise, change the PIC crystal frequency (anything up to 20MHz is OK, and in practice even higher values can work). Identify the values in the software to change to correct the display. Cope with the GV counter needing to have a higher value than can fit into a 16 bit counter . A higher clock will reduce the effect of the annoying synchronisation error . Note that certain delays are needed within the LCD writing routines, so the other programmed delay routines (such as DelayNms) will need to be changed

Synthesiser Controller

The software ***mfgctl_02*** is of more specialist nature. It will control two independent synthesiser chips of the TSA5505, SP5505, or U6239 type that use an I2C two-wire bus for programming. Such devices are in widespread use for TV, satellite and cable TV tuners, and many of the 2.4GHz Rx and Tx modules. This design was specifically intended for control of a pair of G1MFG FM TV tuner modules, allowing, for example, independent tuning for transmit and receive.

Ports A0-3 are configured for digital operation (use a low value resistor in series with each for protection) and connected to the synthesiser chip SCL and SDA programming lines according to Table 1. On the MFG TV modules, the existing PIC is removed so the connections from the controller can conveniently be made pins 1 and 2 on the now-empty IC socket.

Table 1 Connections for dual I2C bus synthesizer chip programming.

Controller Conn	Synth Unit	I2C Bus Line	G1MFG Module Connection
A0	1	SDA 1	Module A Pin 1
A1	1	SCL 1	Module A Pin 2
A2	2	SDA 2	Module B Pin 1
A3	2	SCL 2	Module B Pin 2

Freq 1 or Freq2 are selected in turn, and may be altered in 125kHz or 4MHz steps. Selection is made by repeatedly pressing the pushbutton to cycle though each frequency tuning option.

Please note that contrary to the statement in the RadCom article, IF Offsets and tuning limits cannot be changed using the rotary control / pushbuttons. This was too complicated to implement at the time of writing. Instead, these values are stored as constants in the PIC and can only be changed with a PIC programmer. See *mfgctl_02.asm* .

The relevant code appears at the start of the listing, and is repeated here. This version shows an IF offset of 479.5MHz included within the Rx module. All values are stored as integer multiples of 125kHz. Note the ability of the assembler to perform simple (integer) maths to make the storing of variables and constants a bit more easily understood.

```
;Rx Module
  IFOFFSET1     =    d'479' * 8 + 4      ;479.5MHz IF, arithmetic can't do 0.5MHz !
  DMAX1         =    d'2700' * 8 - IFOFFSET1    ;Determined by measurements
  DMIN1         =    d'2250' * 8 - IFOFFSET1    ;  max/min lock range
  STARTFREQ1    =    d'2500' * 8 - IFOFFSET1    ;Stored starting freq after assembly
;Tx Module
  IFOFFSET2     =    0
  DMAX2         =    d'2650' * 8 - IFOFFSET2    ;Determined by measurements
  DMIN2         =    d'1950' * 8 - IFOFFSET2    ;  max/min lock range
  STARTFREQ2    =    d'2500' * 8 - IFOFFSET2    ;Stored starting freq after assembly
```

As an advanced programming exercise, first arrange for the IF offsets and tuning limits to all be stored in EEPROM rather than being called up as constants. Then work out a way of adding the facility to alter these with the rotary encoder – perhaps by going into a setting routine while holding the pushbutton down at switch on.

When you have managed this, please let me know and consider adding your code, (with full credits of course) to this website.

A Bit More on (my) Assembly File Structure

Over the 16 or so years I have been programming PICs, the resulting assembly code has fallen into a common structure which aids readability and understanding.

1) It starts at Org 0 and usually jumps straight to startup.
2) Then comes any interrupt service routines (which have to lie at prog-memory location 4). Even if no interrupts are used, a retfie instruction is usually placed here "just in case" some catastrophe happens later down the line and interrupts do appear.
3) Then the *Startup* section, jumped into from the reset point at location 0. This is where all initialisation of ports, peripherals and variables is performed and startup variables initialised.
4) Followed by a MainLoop which the code loops round repeatedly, or looks for user interaction, or... or....
5) Below this are all the subroutines, all given meaningful names

6) Any code that requires either computed *goto* statements or lookup tables are located at the end of programme memory, usually in page 0x700 for the devices used here . This is so I can keep control of the high order byte of the programme counter in such calls.

7) Tables always use the assembler abbreviation dt (for define table). This offers far more flexibility in laying out tables and text etc in a sensible way than a huge list of *retlw* commands.

8) Extensive use is made of *#define* statements to give meaningful names to I/O lines. Annoyingly these have to be placed at the start of the listing

9) Variables are defined in a *cblock* section at the end. Contrary to many other PIC programmers listings , each variable does **not** have to be individually assigned a value, and certainly not using the '*equ*' statement. Also, register names can be put on a single line separated by commas, which certainly helps readability. Not a lot of other programmers seem to know this.

10) I don't like multiple return statements in subroutines, and certainly not in interrupt handlers. Although they do just occasionally make life easier. More often, a jump to an 'out of routine' label is preferred. And always in interrupts.

In the assembly listing, separators are included as comments at all strategic points. The characters making up the lines show what is being separated:

;-------------- is used to separate independent subroutines from each other, and also areas of the main code that perform differing tasks, and reached by separate routes. Normal programme flow rarely crosses this boundary – one exception is sometimes the transition from startup to main areas.

;========= serves as a boundary between completely separate areas of code. Interrupt service, Main, subroutines, and table areas get this break type. Programme flow never crosses it; always a call or goto command

;............. is used to separate short chunks the programme flow will never pass <u>over</u> . They will always be reached by a break in the direct flow, for example *goto* statements and skipped instructions.

Single line gaps separate small chunks of code doing each task.

Timing critical code usually has its clock cycle count spelled out.

PICs often have quite a lot of oddities in their startup and initialisation and register addressing. In many cases comments in the assembly listing explain these – such comments are usually themselves usually because I copy and paste my code repeatedly, so some comments could be 16 years old.

These rules are flexible and mostly, but not always rigidly, applied.

....