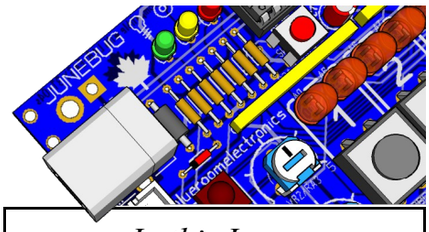
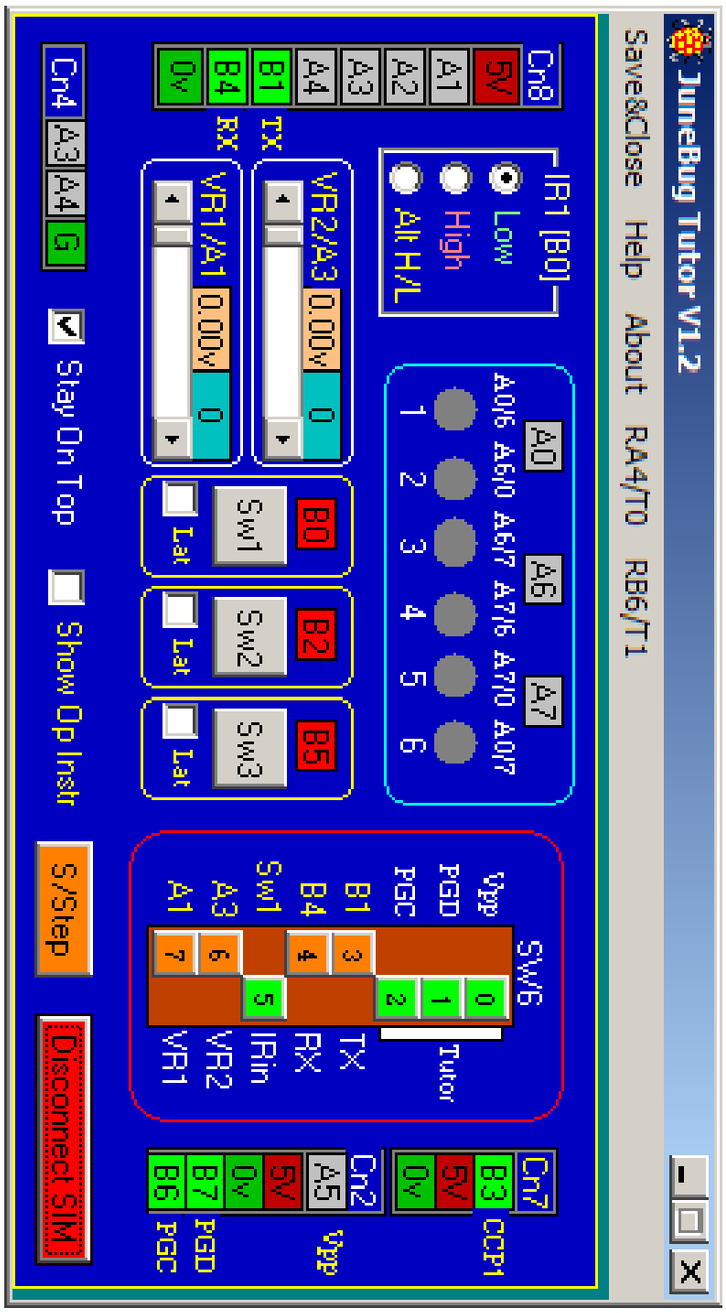
THE JPUG





Eric Gibbs amazing Junebug module for use with the OshonSoft PIC18F simulator

In this Issue

Tips & Tricks for the 18F

#1	W is an SFR
#2	BRA vs GOTO
#3	Even Numbered Branching
#4	2 Cycle delay
#5	200,000 Cycle delay
#6	Shifting bits with multiply
#7	LAT vs PORT
#8	TABLRD vs RETLW

Published by
blueroomelectronics

All articles are generously provided by PIC enthusiasts like you, have you got an idea for a project or have an article to contribute to JPUG?
Drop us a line at
jpugnews@gmail.com

JPUG is written and published using OpenOffice 2.4

Cover Story: Junebug Simulator

(OshonSoft PIC18F Simulator) by Eric Gibbs

When debugging firmware I traditionally start work by creating an External Module for the OshonSoft PIC18 Simulator IDE.

Junebug Sim 1.2 can be downloaded from
http://www.blueroomelectronics.com/download.html

The zipped file contains the external module program, which is named JunebugExt1.exe. and a number of programs written in OshonSoft Basic. The Basic programs are there to demonstrate the different features of the module.

Visual Basic 5 Runtime files are required, these files can be downloaded and installed, free from the web.

The OshonSoft PIC18Simulator IDE must be installed, for a trial version of the IDE, visit www.oshonsoft.com

Create and name a folder on the hard drive named Junebug1

With the above files installed in the Junebug1 folder, double clicking the JunebugExt1.exe will run the OshonSoft PIC18 IDE and the External module.

It would be a good idea to create a desktop shortcut to the JunebugExt1.exe, for future use.

Module Features.

The layout and the component identification of the module has been done to closely represent the physical layout of the actual Junebug Tutor PCB.

In addition to the components represented on the module there are control buttons and check boxes used to control the module.

Operation.

When ALL the 'required files' have been installed, double click (run) `JunebugExt1.exe`

After clicking the exe file the PIC18 IDE and module will appear on the PC monitor.

From the IDE menu's select the 18F1320 PIC type, crystal frequency and the Basic Compiler

Using the IDE File menu, load a Basic demonstration file from the examples provided, into the Compiler window.

Compile the file using the Compiler menu, the hex file is automatically loaded into the IDE.

Select the 'Rate' using the IDE menu, the Rate selected determines the rate at which the simulator runs. Being a PC based simulator it will not run at real time PIC speeds.

After compiling the program, select 'Start' from the IDE menu, the program will run and depending on which example program you have compiled, the module will display the operation of the program.

On the module menu bar is a 'Help' option, on selecting this option, a help text window is displayed. The text explains the operation of the module.

To close and exit the module use the 'Save & Exit' option on the module menu bar, this will save various settings of the module. The saved settings will be recalled next time the module is run.

For convenience, the 'Help' text of the module has been reprinted below:

The Junebug External Module has been designed to work in conjunction with the OshonSoft PIC18 Simulator IDE.

The 'Connect to Sim' button on the module enables the PORT pin read/write to the IDE.

The state of the PIC pins can be set High [Red] or Low (green] by clicking the left mouse button on the pin label. [eg: A0]

With the PIC Sim IDE started, the 'Connect to Sim' button enabled and the program running, the IDE will control the state of the module pins, the colour of the pin indicates the pin state, high or low.

When multiplexing the LED's 1 thru 6, a PIC input is White

SW6 on the module can be set by a left mouse click on the switch bit.

If VR1 or VR2 slider control is selected, by using SW6, then the VR1 and VR2 analog control voltage will be sent to the IDE.

To see any change in the voltage at the PIC in the IDE the running program must configure PORTA to suit analog input.

The input voltages to the ADC and conversion value are displayed on the module. SW1 button on the module shares the B0 pin with the IR1 input.
Use SW6 to select the SW1 or IR1 input to B0.
When the IR1 input is selected the state of the IR1 can be set Low, High or Alternating High/Low

On the Module menu bar is the option to set the `A4 [TMR0]` and `B6 [TMR1]` inputs "toggling", this feature is useful for checking Interrupts, when working on external clock inputs.

Normally the SW1 thru SW3 buttons on the module are momentary, to latch the SW1 thru SW3 switches, check the "Lat" box, this make the switch latching.

The Charlieplexed LED's, numbered 1 thru 6, expect the program in the IDE to be written to support multiplexing. Use the example Basic program included in the zip.

The LED's will indicate the multiplexed state of the `A0, A6` and `A7` pins.

When multiplexing, the LED's 1 thru 6, a PIC input pin state is coloured White .

By checking the Show Op Instr box the Program Counter, the Last Program Instruction will be displayed in the text box.

While the program is running in the IDE, pressing the 'S/Step' button on the module will set the IDE in the STEP mode.

NOTE: after the S/Step button has been pressed, the program will Stop and await further presses of the modules S/Step button.

Each press will STEP the program onto the next instruction.

Used in combination with the 'Show Op Instruction', the user can observe the instructions and the states of the module.

While the 'S/Step' button has the focus, the keyboard 'Enter' key can be used to STEP thru the program or to give a short burst RUN , if the key is held down.

To RUN or RESUME the program at any other rate than STEP, the IDE 'Rate' menu must be used.

To keep the module on top of other windows check the 'Stay On Top' box.

Keypad Using One A/D Input

(MPASM & Just BASIC) by Jason Lopez

Here's a method for reading a row of keys (in this example 12 push-buttons). The method is Included is a simple program for your PC that helps choose resistor values for the actual hardware (you'll need to connect this keypad externally via the Junebug's CON3 connector.)

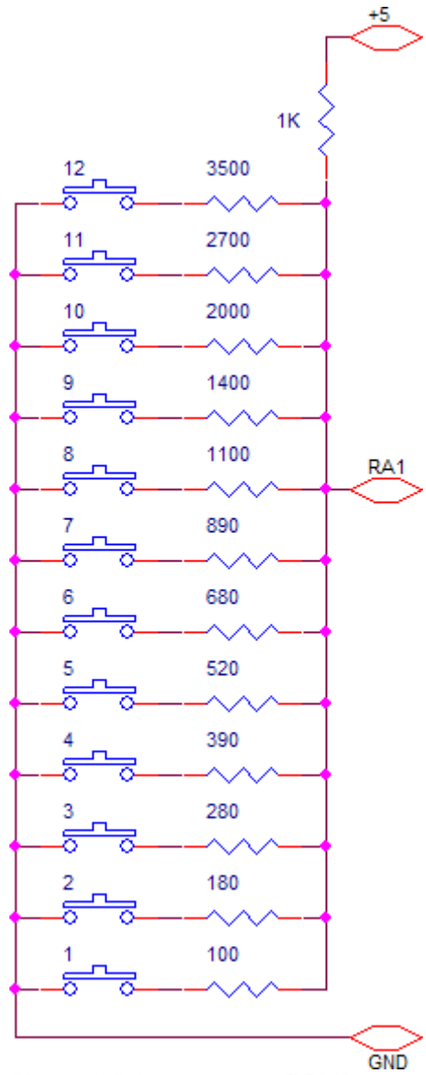
```
RPu = 1000  ' pullup top half of voltage
divider
ADbits = 4  ' A/D resolution in bits
For Count = 0 to 2^ADbits -2
  Vlow = Count / 2^ADbits
  Vhigh = (1+Count) / 2^ADbits
  Rmin = RPu / (1-Vlow) * Vlow
  Rmax = RPu / (1-Vhigh) * Vhigh
  Rideal = (Rmax - Rmin)/2 + Rmin
  If RPu + Rideal > 10000 then exit For
  Print ADbits;"bit A/D Result ";
  Print Count,"Rmin ";int(Rmin)," Rmax ";
  Print int(Rmax),"Ideal ";int(Rideal)
Next Count
END
```

Just BASIC program for finding Rx values

Running the program with a pullup resistor of 1K produced the following table

```
4bit A/D Result 0  Rmin 0    Rmax 66   Ideal 33
4bit A/D Result 1  Rmin 66   Rmax 142  Ideal 104
4bit A/D Result 2  Rmin 142  Rmax 230  Ideal 186
4bit A/D Result 3  Rmin 230  Rmax 333  Ideal 282
4bit A/D Result 4  Rmin 333  Rmax 454  Ideal 393
4bit A/D Result 5  Rmin 454  Rmax 600  Ideal 527
4bit A/D Result 6  Rmin 600  Rmax 777  Ideal 688
4bit A/D Result 7  Rmin 777  Rmax 1000 Ideal 888
4bit A/D Result 8  Rmin 1000 Rmax 1285 Ideal 1142
4bit A/D Result 9  Rmin 1285 Rmax 1666 Ideal 1476
4bit A/D Result 10 Rmin 1666 Rmax 2200 Ideal 1933
4bit A/D Result 11 Rmin 2200 Rmax 3000 Ideal 2600
4bit A/D Result 12 Rmin 3000 Rmax 4333 Ideal 3666
4bit A/D Result 13 Rmin 4333 Rmax 7000 Ideal 5666
```

You may have noticed it calculates the results for a total of 14 keys, depending on the A/D resolution you choose even more keys (or less) are possible.



Keypad Schematic for CON3

I've rounded the resistor values to as close to commonly available values and 5% resistors are fine. The reason for the particular resistor choices make it almost effortless to convert the A/D values into their respective pushbuttons. The ADbits = 4 means we're going to truncate *throw away* the lower 6 bits (it's a 10bit A/D on the 18F1320). The beauty is no lookup tables are used to convert the A/D result.

The program fragment presented below was designed for use with the MPLAB PICkit2 (Junebug) debugger and will return the result in the W register. Remember a value of `0x0F` in W would indicate a **no key pressed** condition.

```
        list        p=18F1320
        include     <p18F1320.inc>
        CONFIG OSC=INTIO2,WDT=OFF,LVP=OFF
        org   0x00          ; reset vector
Init    movlw 0x62          ; 4MHz OSC
        movwf OSCCON
        movlw b'00000101'   ; A/D on, AN1
        movwf ADCON0        ; left justify
        movlw b'00110011'   ; conversion speed
        movwf ADCON2
Main    rcall GetKey        ; watch WREG and
        bra   Main   ; set breakpoint here
; WREG will return 15 (no key) or 1-12 (key)
GetKey bsf    ADCON0,GO     ; start A/D
        btfsc ADCON0,DONE   ; conversion
        bra   $-2           ; loop till done
        swapf ADRESH,w      ; swap nibbles
        andlw   0x0F        ; mask off top 4bits
        return        ; exit with result in W
END
```

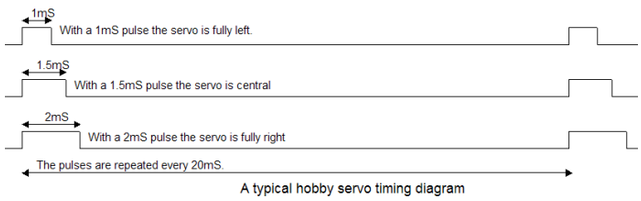
Since the result is only 4bit it's a simple matter of left justifying the A/D result, swapping the high/low nibbles and masking off the upper nibble (4 bits). If you're using results other than 4 bits you'll need to shift the results around; check out tip #6 for a pretty nifty way of shifting bits left or right. Of course if you're only shifting a single bit then the **rotate** instruction is probably your best choice.

Possible PCB layout for the project

Controlling a Hobby Servo Motor

(C18) by Mike Webb

Hobby Servos are a popular and inexpensive way to connect our electronics to the outside world. To control a hobby servo you have to send it pulses. A short pulse makes it go fully left and a long pulse fully right, anything in between and the servo goes to the proportional position.



A typical hobby servo timing diagram

We can use our Junebug to produce these pulses and hence control a hobby servo. To produce accurate pulses with a PIC we need to use one of the hardware timers. I chose to use timer 1 as this is a 16 bit timer and when combined with the capture compare module (CCP) can be reset at a predetermined count. To accomplish this we setup the CCP module in special events trigger mode. In this mode the timer is reset when the timer count is equal to the value in CCPR1. When these values match two things happen, timer 1 gets reset to zero and the CCP interrupt flag (CCP1IF) will get set. We can use this flag to accurately turn our servo pulse on and off.

Now we have a way to generate the pulses we need someway to alter their length. We do this by reading the value of the preset resistor VR1. To do this we setup the analogue to digital converter (ADC) to convert the voltage from VR1 into a number that varies between 0 and 1023 (10 bit) and we use this value to alter the length of the pulses.

```
#include <p18f1320.h>
#pragma config WDT = OFF, LVP = OFF, OSC = INTIO2
#define ServoPin LATBbits.LATB3
#define ServoTris TRISBbits.TRISB3

void main(void){
int ServoPos;
  OSCCON=0x70;        //Osc=8MHz
  ADCON0=0b00000101; //A2D on & select AN1
  ADCON1=0x7d;        //A1 = analogue
  ADCON2=0b10110101; //Right justify - Fosc/16
  ServoTris=0;        //make servo pin output
  ServoPin=0;         //Servo output off
  CCP1CON=0b00001011; //Special event trigger
  T1CON=0b10010001;  //Timer 1 on with Presc=2
  ServoPos=1500;      //set servo to mid position
  while(1){
    CCPR1=20000-ServoPos;   //20mS - Servo Time
    ADCON0bits.GO=1;        //start conversion
    while(ADCON0bits.GO);   //Wait till complete
    ServoPos=ADRES+1000;    //1mS to 2.023mS
    while(!PIR1bits.CCP1IF);//wait for IF
    ServoPin=1;             //start pulse
    CCPR1=ServoPos;         //Servo time in uS
    PIR1bits.CCP1IF=0;      //clear int flag
    while(!PIR1bits.CCP1IF);//wait for CCP IF
    ServoPin=0;       //end pulse
  PIR1bits.CCP1IF=0; //clear int flag
  }
}
```

The Complete Servo program

You need to connect a servo to con5 and ensure it is connected the right way around.
*Editors note: your USB port may not have enough current to reliably run a servo motor, see the **Standalone Power** article at the end of this article*

If you are familiar with MPLAB and the C18 compiler then the above code is all you need. If you're a newcomer to all this then, here's a step-by-step guide to get you going. Although it seems like a lot of steps, once you've done it a couple of times it'll be second nature.

1. Run MPLAB and select the program wizard from the project menu.
2. Click next to get to stage 1.
3. Select PIC18F1320 from the drop down box and click next.
4. Select "MPLAB C18 C Compiler (mcc18.exe)" from the list and click next. If this isn't in the list then you need to download it from Microchip and install it.
5. Create a new project by clicking browse and navigate to a folder where you want your project. Type servo as the file name, click save and then click next.
6. Skip stage 4 by clicking next.
7. Click finish.
8. If you are using the latest version of MPLAB you will now have a blank page. Select View-Project from the menu bar.
9. Select Project-Add new file to project. Navigate to your project folder and type servo.c as the file name. Note, there is also add file to project – you don't want this one.
10. You now have a blank window called servo.c into which you can copy and paste the above code.
11. In the projects window (servo.mcw) right click on the linker script folder and select add files. Navigate to C:\MCC18\LKR (or wherever you installed the C18 compiler) and select file 18f1320i.lkr
12. Connect your Junebug and turn on switches 1, 2, 3 & 8
13. From the debugger menu select tool PICkit 2.
14. Press F10 and you should have a compiled project.
15. Have fun.

Standalone Power for the Junebug

Thanks to portable MP3 players it's possible to find inexpensive 5V regulated power adapters. They're particularly handy for running your Junebug projects without your PC, plus the added benefit of typically higher current (1A-2A) then a computer USB port (100mA – 500mA)
Another option is a powered USB hub, these inexpensive devices will supply power (500mA to 1A typical) to the ports even when the host PC is disconnected and they can also power those high current projects like servo motors.

CSV 4 Channel A/D Recorder

(Swordfish BASIC) by William Richardson

This 4 channel A/D Recorder will continuously output data from AN1 (RA1) to AN4(RA4) via the PICkit2 UART Tool at a rate of one channel per second. It also formats the results for CSV (Comma Separated Value) which means the results can be directly loaded into almost any spreadsheet program such as OpenOffice Calc or Excel.

	A	B	C	D
1	Junebug CSV 10bit 4 Channel Recorder			
2	AN1	AN2	AN3	AN4
3	61	97	1016	911
4	60	64	1016	883
5	61	91	1016	888
6	60	58	1016	935

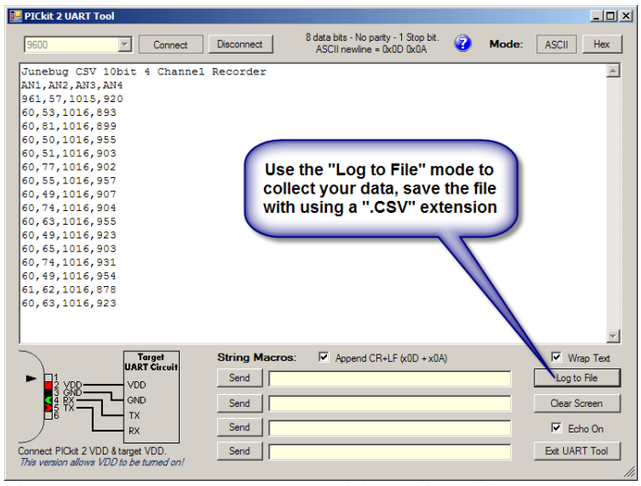
CSV data as seen by OpenOffice Calc, of course Microsoft Excel results would be similar

```
Device = 18F1320
Clock = 4
Config OSC = INTIO2, WDT = OFF, LVP = OFF
Include "USART.bas"
Include "convert.bas"
Include "Junebug.bas"
Dim ADSel As Byte

Public Function ADGO(ADCS As Byte) As Word
  ADCON1 = (0 << ADCS) ' select analog input
  ADCON0 = (ADCS << 2) Or $03  ' Channel & GO
    While(ADCON0.1 = 1) ' wait till done
    Wend
  ADGO=(ADRESH<<8)+ADRESL
End Function

SetBaudrate(br9600)
OSCCON = $62          ' 4MHz OSC
ADCON2 = %10100001  ' A/D speed, Right justify
USART.Write("Junebug CSV 10bit 4 Channel
Recorder", 13, 10)
USART.Write("AN1, AN2, AN3, AN4", 13, 10)
  While true
  For ADSel = 1 To 4
    USART.Write(DecToStr(ADGO(ADSel)))
    DelayMS(1000)
    LED(ADSel)
    If ADSel < 4 Then  ' comma for CSV
      USART.Write(",") ' formatting
      EndIf
    Next
```

```
  USART.Write(13,10)   ' CR/LF
  Wend
End
```

The Data Recorder program, if you need the JUNEBUG.BAS module you can find it in JPUG issue #1 or download it from the Junebug page at www.blueroomelectronics.com

Once you've compiled and programmed your Junebug you'll need to exit the Swordfish IDE and run the standalone PICkit2 software. You'll also need to turn OFF DIP switches 1,2&3 and turn ON switch #4 (TX). LEDs 1 thru 4 should flash sequentially at one-second intervals. From the PICkit2 software choose the UART Tool, set the baud rate to 9600 and press Connect. You should see data coming in right away. If you want to make a logger you can press the Junebugs RESET button and this will restart the recorder and also put the headers fields in if you decide to capture the data in a CSV spreadsheet format.



You can log your data with the PICkit2 UART Tool, don't forget to set DIP SW-4 (TX) to ON to see the data

Note: Technically this program is not a standalone data recorder as it requires your computer to log the data. Perhaps a standalone EEPROM version would make for an interesting future project for JPUG...

Tips & Tricks for the 18F

(MPASM) Various Sources on the Net

#1 W is an SFR

W is an SFR, what does that mean? With the new 16bit core of the PIC18Fxxxx you can access the W (working) register as you would any other file register (SFR). This opens up some interesting programming possibilities especially for the byte frugal programmers out there.

For example; a simple delay routine commonly would look something like this

```
        decfsz      DELAY,f
        bra         $-2
```

Could be written on an 18Fxxxx series PIC like this

```
        decfsz      WREG,f
        bra         $-2
```

The difference is you don't have to define the DELAY variable which uses one byte of RAM. This might seem trivial to many, but to a microcontoller programmer making every byte count.
This leads us to tip #2

#2 BRA vs GOTO

Another new instruction to the 18F series is the `BRA` instruction. `BRA` (branch unconditionally) takes only one instruction space vs `GOTO` which requires two instruction spaces. The limitation of bra is it can only jump short distances (`-1024 to +1024`) but hey a byte is a byte.

#3 Even Numbered Branching

All 18Fxxxx program memory is 16bits *one word* wide (the 16F is 14bits), MPASM allows access to every byte but all instructions begin on even memory address. Branching backwards *(will jump to the NOP)* by one instruction would look something like this

```
        NOP
        BRA         $-2
```

Where $ is an instruction to MPASM that inserts the current program location during assembly. The -2 will branch backwards by two bytes (or 1 instruction) Of course using a label would make it easier but where's the fun.

```
JumpHere    NOP
            BRA   JumpHere
```

#4 2 Cycle Delay using BRA $+2

Using a `BRA $+2` is a common method to create a two cycle delay with only a single instruction. It has the same result as using two `NOP` instructions in a row. You may have seen this routine used with the old PIC16F when you see `GOTO $+1` in a program.

#5 200,000 Cycle Delay

Myke Predko has some terrific books on PIC programming and his web site http://www.myke.com/basic.htm has many tips & tricks for the PIC16Fxxxx One of those tips (#26) is a 200,000 cycle delay routine. Myke also notes this is handy routine for a 200ms (1/5 sec) delay when the oscillator is 4MHz.

```
        clrf        DlayCount
        clrf        DlayCount + 1
        decfsz      DlayCount, f
```

```
        goto        $ - 1
        decfsz      DlayCount + 1, f
        goto        $ - 3
```
Original code as seen on Myke's website

I've modified Myke's routine for the PIC18Fxxxx

```
        clrf        WREG
        clrf        DelayCount
        decfsz      WREG, f
        bra         $ - 2
        decfsz      DelayCount, f
        bra         $ - 6
```
Modified PIC18F version

Running the code through the MPLAB simulator's stopwatch the 200,000 cycle delay is exactly 197,121 cycles just in case you're wondering.

#6 Shifting Bits with 8x8 Multiply

Something I found over in the Microchip Forums. A really neat way to shift bits left or right.

Shift the value in W 3 bits to the left:

```
        mullw       .1<<3       ; 3 = #bits
        movf        PRODL,w
```

Shift the value in W 3 bits to the right:

```
        mullw       .256>>3     ; 3 = #bits
        movf        PRODH,w
```

#7 LAT vs PORT

One new 18F instructions is `LATx` (latch) this new instruction addressed the old RMW (Read Modify Write) problem when writing a bit to a port that could be a real nightmare for programmers who were not looking out for it. The `PORTx` instruction is still used for reading an input port but you should use `LATx` when writing to an output port.

```
        btfss PORTB,2     ; test bit RB.2
        movf  PORTA,w     ; read byte RA
        bsf   LATB,2      ; write bit RB.2
        movwf LATA        ; write byte RA
```

#8 TABLRD vs RETLW

Anyone familiar with the 16F has probably used tables and the traditional method is using RETLW Below I've illustrated a typical example

```
; W holds the offset into the table
            call    Table
 Table      addwf   PCL
            dt      "Hello"
```

dt is the same as RETLW <data> but there are a couple of important differences vs the 16F counterpart.

- The table requires increments of two bytes
- Every value takes two bytes of memory

The first fix is easy just multiply the WREG x 2, the easiest way to do this in assembly is a simple rotate left without carry. So the routine would now look like this.

```
            rlncf   WREG
            call    Table
 Table      addwf   PCL
            dt      "Hello"
```

Does the 18F instruction set off a better solution, yes the TABLE instruction and the DB command.

```
            movlw   Table
            movwf   TBLPTRL
            TBLRD
Table       db      "Hello"
```

The W register is not used but instead the TABLAT register holds the result. You may have noticed the W offset is missing, instead you use the TBLPTRx registers. Since the TBLPTR is not limited to 256 bytes it's a good idea to load the 16 bit address before using the table. Also as an added bonus it's possible to have the pointer automatically increment if desired using the TBLPTR*+ instruction.

```
        movlw   High(Table)
        movwf   TBLPTRH
        movlw   Low(Table)
        movwf   TBLPTRL
        TBLRD*+
Table   db      "Hello"
```

Junebug Naming & VDD Calibration

(MPASM) by William Richardson

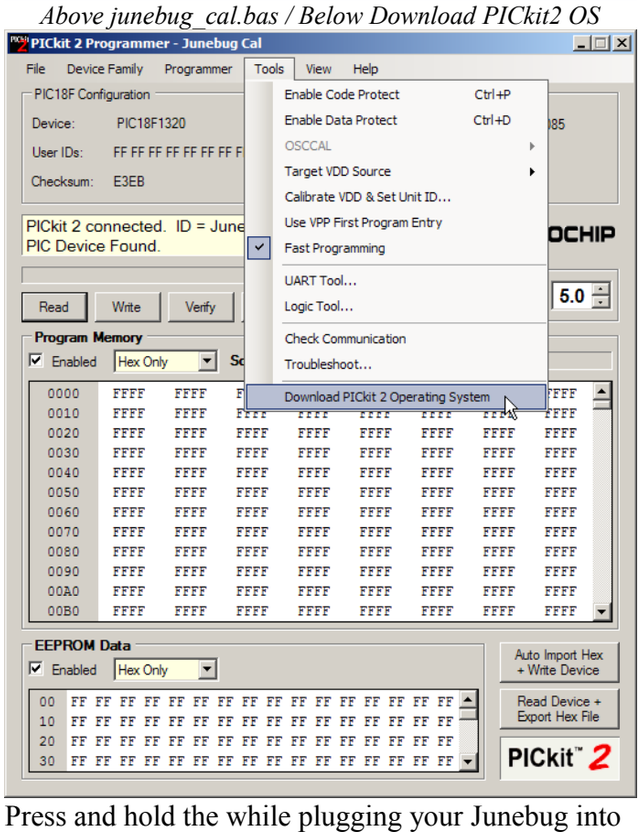
The PICkit2 2.50 software has a feature that allows you to name and calibrate your Junebug, but Junebug does not have an adjustable VDD so calibration cannot be performed and any existing calibration value will be lost. Calibration is only necessary if you plan on using 3.3V PICs.
The following program is designed to load into the Junebugs 18F2550 via the PICkit2 bootloader.

Save the following program as Junebug_Cal

```
; Change only the ID define below
; must be 15 characters or less
#define ID    "Junebug Cal"
#define Vcal  0xFA81        ; 4.2V Calibration
      list    p=18F1320
      include <p18F1320.inc>
      org     0x2000        ; bootloader vector
Start clrf    INTCON        ; interrupts OFF
      movlw   low(Name)
      movwf   TBLPTRL
      movlw   high(Name)
      movwf   TBLPTRH       ; point to Name table
      movlw   0xF0          ; EEPROM Name address
      movwf   EEADR
      clrf    EECON1
```

```
        movlw  '#'          ; EEPROM 0xF0 = '#'
        rcall  WriteEE
Loop    tblrd*+       ; read table and increment
        movf   EEADR,W
        bz     WR_Cal
        movf   TABLAT,W     ; W = TABLAT
        rcall  WriteEE
        bra    Loop
WR_Cal clrf   EEADR  ; EEPROM 0x00 = 0101FA81
        movlw  0x01         ; first two bytes
        rcall  WriteEE      ; are 0101
        movlw  0x01
        rcall  WriteEE
        movlw  high(VCal)   ; save calibration
        rcall  WriteEE      ; values into EEPROM
        movlw  low(VCal)
        rcall  WriteEE
        bra    $            ; wait for reset
Name    data   MyJune
        data   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

; Sub WriteEE W = Data, auto increments EEADR
WriteEE bcf   PIR2,EEIF     ; clear EE flag
        movwf  EEDATA       ; EEDATA = W
        bsf    EECON1,WREN  ; write enable
        movlw  0x55
        movwf  EECON2
        movlw  0xAA
        movwf  EECON2
        bsf    EECON1,WR    ; begin write
        nop
        btfss  PIR2,EEIF    ; wait till write
        bra    $-2          ; completes
        bcf    EECON1,WREN  ; write disable
        incf   EEADR        ; next address
        return
        END
```

Above junebug_cal.bas / Below Download PICkit2 OS



Press and hold the while plugging your Junebug into the USB port on your computer. The red busy LED should flash. Run the PICkit2 software and select `Download PICkit 2 Operating System` then locate and open `junebug_cal.hex` after it's loaded it will automatically run and update the 18F2550s internal EEPROM. The PICkit2 software will reset the Junebug but you'll need to manually run the bootloader again by holding down the boot button.

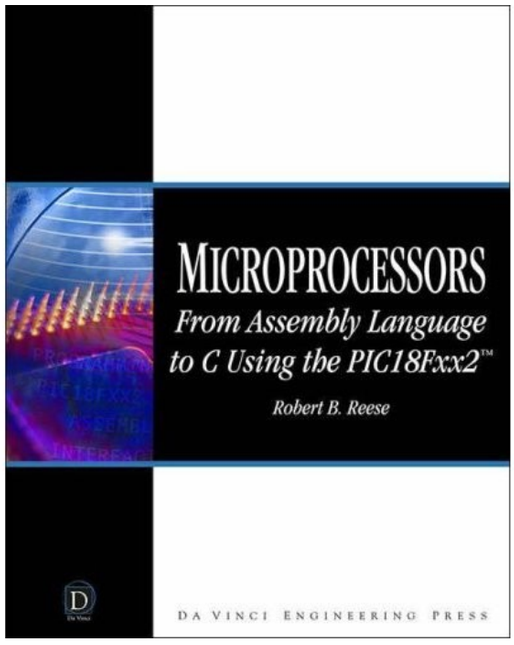
Hello World and the UART tool

(Swordfish BASIC) by William Richardson

Traditionally your first program is often called Hello World here's a small program that will send “Hello World” to the UART Tool

```
Device = 18F1320
Clock = 8           // define the clock speed
for the compiler
Config OSC = INTIO2, WDT = OFF, LVP = OFF
Include "USART.bas"
OSCCON = $72        // select 8MHz internal
oscillator
SetBaudrate(br9600)
While true
 USART.Write("Hello World",13,10)
 DelayMS(1000)
Wend
End
```

Book of the month



Microprocessors: From Assembly Language to C Using the PICI8FXX2 by Robet B Reese.

- **Hardcover:** 652 pages
- **ISBN-10:** 1584503785
- **ISBN-13:** 978-1584503781