Introduction to PICL and PICLab

Introduction

Assembling and running your first program:

1)Enter assembly code in the PICL text window.

2)Press the 'Build' button to assemble your code and check for syntax errors.
  Open the 'Assembler output' window from the 'View' menu to monitor the
  assembly and view informational messages.

  If the assembly was successful and PICLab/PICL are serially connected,
  the assembled code is uploaded to the PIC and executed. Otherwise,
  your code execution is simulated in software. You can switch between these
  two modes of operation by clicking the 'Connect' icon in the PICLab window.

  Press the 'Step' button to simulate the execution of each instruction;
  single stepping is not available if your code is being executed by the PIC.

  Press the 'Run' button to execute the code currently stored in the PIC
  or to run the simulation.

You can monitor the operation of your program and the changes it makes to the
PIC by opening diagnostic windows from the 'View' menu. These windows are
updated after your program completes or after the execution of a 'call Break'
instruction or the equivalent short form '!':

```
'PICLab output'    displays user serial output from the PICLab board;
'Processor status' tracks changes to the W register and status flags;
'File registers'   displays the contents of all the PIC file registers;
'EEPROM memory'    displays the contents of the PIC EEPROM memory;
'Program memory'   displays the contents of the PIC Flash program memory;
'Watch window'     tracks changes to user specified file registers.
```

If your program hangs or is executing an infinite loop, PICL will not
respond to user input. Reset the PICLab board to restore to normal the
operation of PICLab and regain control of the PICL software.

Macroassembler data type conventions:

```
hexadecimal             0x20, 0xFF, 0xad
decimal                 123, 1.23E4, 3.14159
8-bit binary            %01001110 (or B'01001110', include 8 bits)
ASCII character         'A, 'a, \0x20, (space) '
 (newline)
expression              int(($ALPHA+10)*2.5)
```

The $ sign preceding a label is only required inside an expression,
hence 'addlw LABEL' and 'addlw $LABEL' have the same meaning. Note that
the meaning of a label is determined by the instruction using it:

```
addlw LABEL ;here LABEL is interpreted as an 8-bit literal constant
addwf LABEL ;here LABEL is the 7-bit address of a file register
call LABEL ;here LABEL is the 11-bit address of a subroutine
```

Labels and variable names can only include A-Z, a-z, _ and 0-9. Some label names are used by PICL and will cause a 'label cannot be modified' error if an attempt is made to change them.

Data memory:

File registers:

There are 512 file registers implemented on the PIC16F877 microcontroller. The contents of this volatile RAM are lost when the power is removed. These registers are paged and not all available at the same time. The STATUS register contains the bits that are used to switch between these pages.

Special function registers, such as input/output ports, timers and other PIC control registers occupy locations 0-0x1F, 0x80-0x9F, 0x100-0x110, 0x180-0x18F.

When operations involving a file register other than INDF are used, the register space is arranged as four pages of 128 bytes: 0-0x7F, 0x80-0xFF, 0x100-0x17F, 0x180-0x1FF. The STATUS register bits RP1 and RP0 select the page currently accessed.

For programming efficiency, the INDF, PCL, STATUS, FSR, PCLATH, INTCON registers as well as the upper 16 bytes in page 0 (0x70-0x7F) are imaged to all four pages. The TMR0, OPTION_REG, PORTB and TRISB are imaged only to their other even or odd page.

The PICL system registers WH, WL, CH, CL, Flags, Hours, Minutes, Seconds and Count5ms reside in (0x70-0x7F) and are hence always accessible to the user. Other system registers occupy (0x60-0x6f) and are only available from page0.

```
Example:        bsf     STATUS,RP0
                bcf     STATUS,RP1      ;select page 1
                movwf   0x20            ;write to 0x20 in page 1, actually 0xA0
```

The file register memory can also be accessed indirectly using the FSR register as an index pointer. The INDF register represents the indexed data that can be operated upon as any other file register. When using indirect addressing, the register space is organized as two 256-byte pages 0-0xFF, 0x100-0x1FF that are switched by the STATUS register IRP bit.

```
Example1:       movlw   0x20            ;increment indirectly memory 0x20
                movwf FSR              ;on page selected by IRP bit
                incf INDF

Example2:       bcf     STATUS,IRP      ;copy data from an address at indirect
                movf INDF,W           ;page 0 t0 the same address at indirect
                bsf STATUS,IRP       ;page 1, the FSR points to both pages
                movwf INDF
```

In summary, the following file register locations are available to the user:

64 at 0x20-0x5F, 80 at 0xA0-0xEF, 96 at x 0x110-0x16F, 96 at 0x190-0x1EF

Note: the direct and indirect addressing modes operate independently; the thoughtful assignment of a data memory map can significantly improve the execution speed of a user program, decrease the need for memory bank switching, and reduce the size of the program code.

EEPROM memory:

Ther are 256 bytes of non-volatile EEPROM memory available for the storage of user settings and calibration data that is retained if power is lost to the PIC. The EEADDR (0x10D) register is used as an index to this memory space. PICL includes several routines that use the FSR register to index into the EEPROM array and read/write this memory space. Use the #eeprom directive to conveniently initialize the EEPROM memory with data.

The EEPROM memory can be used to store small data tables of 8-bit data; for storing larger data tables of 14-bit data, use program memory.

A write to EEPROM memory requires around 4ms to complete. the PIC continues to execute instructions during this time, but will wait if a previous write is in progress.

Program memory:

The PIC16F877 Flash program memory consists of 8192 14-bit words in the range 0-0x1FFF. This memory can be used to store program code or user data. The full range of available memory can be accessed using the EEADRH:EEADDR (0x10F,0x10D) registers as an index pointer. PICL occupies program memory locations 0-0x3FF and includes various routines to read/write and to implement a data table in Flash memory.

The program counter is used to step through the program code. It is 13-bits wide and can directly address all of program memory. The PCL register contains the lower 8-bits of the program counter. When the PCL register is used as the destination of an instruction (such as addwf PCL,F), the upper 5-bits of the program counter are loaded with the contents of the 5-bit PCLATH register. The PCL and PCLATH registers are cleared on Reset.

The Call instruction uses a subroutine return stack to store the incremented value of the 13-bit program counter before branching to the subroutine code. The Return, Retlw instructions restore to the program counter the value from the stack so that execution continues with the instruction after the call. The stack can nest a maximum depth of eight subroutine calls.

Call and Goto instructions use an 11-bit address and hence can only access a 2048 word memory range (0-0x7FF). Bits 4 and 3 of PCLATH are then used to set the program memory page that the program counter will branch to. If your code is no longer than 1024 instructions, this paging can be ignored.

If W is added to PCL and the result placed in PCL, a program counter relative branch takes place within the current page. This can be used to implement jump tables and data tables. The code cannot overlap page boundaries and the PCLATH register contents must match the page of the table and program counter.

PICL initializes PCLATH to page 4 before a user program is executed. The start of user program memory is at 0x400. If you place your table code in the memory range 0x401 to 0x4FF, you need not modify PCLATH as shown in the examples. Begin your program with a Goto instruction to skip past the table code.

Example1: jump table, selects from a series of branch targets (case or switch)

```
        movlw   int($pc/256)    ;set PCLATH if table NOT in page 4
        movwf   PCLATH          ;check that page bounds not exceeded

        movlw   offset          ;put table index is in W
        addwf   PCL,F           ;add W to current pc low 8-bits
        goto    branch0         ;pc+0
        goto    branch1         ;pc+1
        goto    branch2         ;pc+2
        goto    branch3         ;pc+3
        ----
```

Example2: data table, returns in W data based on table index

```
start   goto    main            ;skip past table to main program

Power2  addwf   PCL,F           ;add W to current pc low 8-bits
        retlw   1               ;return powers of 2 based on index
        retlw   2
        retlw   4
        retlw   8
        -----

main    movlw   offset          ;put table index in W
        call    Power2          ;raise to power of 2 value in W
        -----
```

A data table of 14-bit values can be stored in the program memory from the end of the user program to the end of available memory. Utility routines are available to perform this task. A write to the data table requires around 4ms to complete. During this time, the PIC stops executing instructions, but the peripherals (timers, USART, etc) continue running. Any interrupts that take place during this time are queued until the write completes.
MPLAB file compatibility:

For the most part, PICL can assemble MPLAB files. However, you will likely need to perform some edits to the file, or at least be aware of the following issues:

1) MPLAB numeric data differs from the format typically used elsewhere.
   In MPLAB, the number 16 represents a hexadecimal value as does 0x16;

decimal values use a leading decimal point, i.e. .16 is decimal 16.
In PICL, 16 represents a decimal number, 0x16 a hexadecimal number,
and .16 a non-integer value that causes an 'expected integer ...'
error message if used as the operand of an instruction.

2) In MPLAB, alphanumeric characters are represented by 'A' or "A".
The space character is given by ' ' or " ".
PICL understands this format but the double quotes will produce an
error and must be replaced with single quotes. An exception is the
space ' ' which it interprets as an empty string, causing a missing
argument error. To specify the space character, use the hex ASCII
value e.g. movlw 0x20.

3) MPLAB identifies the current program counter value by the $ symbol. In
PICL, the program counter is stored in the variable pc. For example,
an MPLAB instruction that uses the program counter to perform a relative
branch such as 'goto $-1' will yield a "couldn't use non-numeric
string as part of '-' " error and should be changed to read
'goto $pc-1'.

4) MPLAB files generally begin by executing some assembler configuration
commands and an include file of label definitions specific to the type
of microcontroller chip used.
Upon encountering an unimplemented command, PICL displays a message
that the command was ignored and continues the assembly. Unknown commands
assemble as labels. The include file directive however has to be deleted
or commented out since the file called is not likely to be present,
resulting in a missing file error.
PICL is specific to the 16F8xx series of microcontrollers and the
label definitions are included as part of the macroassembler.

5) Some MPLAB files include mutually exclusive sections of code that are
enclosed in some structure such as an IF...ELSE...ENDIF statement.
Since PICL does not support conditional compilation you will need to
selectively comment out or remove the conditional statements and the code
sections that are not required.

The PIC Simulator:

can be used to debug user code when a PICLab board is not available or is
not connected to the PICL software. While the PICLab program execution can
only be stopped with the insertion of 'Call Break' instructions into the user
code, a simulated execution provides the following advantages:

- the user can single-step the program and monitor the state of the PIC
  after each instruction. The user program is not altered with Call Breaks;

- With the Virtual PIC window open, the current instruction can be further
  analysed to reveal the internal PIC data flow and program sequencing
  operations that take place during each clock cycle of the instruction cycle;

- the user program can be exected at various speeds to dynamically modify the
  simulated environment. Use the slider to change the speed of the simulation.

- the program execution can be stopped when one or more conditions are met, such as when the contents of file registers match a certain value or the program counter has reached a specific memory location. Match phrases must precede labels with the $ sign and can use the following opeartors: == (equal), && (and), || (or). Example: $WL==0x10 && $pc==$here will stop the simulation when WL is equal to 0x10 and pc has reached the address 'here'.

The PIC simulator includes software simulations for the 7-segment display, the LCD display and the keypad. Also included are sliders used to represent virtual analog voltage inputs on pins RA1-RA3. These simulations duplicate the functionality of the PIC hardware with the following exceptions:

- the user interrupt Refresh routine is simulated, however, other UserISRon routines will not run as a background process;

- the simulation does not execute in real time so that time-dependent code may not behave as expected. Software delays (Delay1s, Wait) are ignored;

- all the preprogrammed subroutines are simulated and generate the same output as when executed on the PIC but execute in one instruction cycle. Also,there may be a value mismatch with some of the registers that are listed as modified by the subroutine. Typically, a simulated register value remains unchanged while the hardware register would have been changed to some unpredictable value;

- writes to PORTD and PORTB(0-3) will correctly modify the LED display, however all other hardware functionality such as port reads/writes, timer and serial port operation, are not supported and will yield unexpexted results.

Opcodes

Arithmetic operations

Mnemonic		Description	Flags affected
addlw	k	Add W and literal	C,D,Z
addwf	f d	Add W and F	C,D,Z
clrf	f	Clear f	Z
clrw	-	Clear W	Z
decf	f d	Decrement f	Z
incf	f d	Increment f	Z
rlf	f d	Rotate left through Carry	C
rrf	f d	Rotate right through Carry	C
sublw	k	Subtract W from literal	C,D,Z
subwf	f d	Subtract W from f	C,D,Z

f = file register address 0x00-0x7F
k = literal constant value 0x00-0xFF
d = W: result placed in W register
d = F or none: result placed in f register
C = carry flag: set if f,k - W >= 0, f,k + W > 0xFF
      : set if bit shifted into C is 1
Z = zero flag: set if result = 0

DC = digit carry: set if carry from the low order nibble

These operations execute in one instruction cycle, or four clock cycles.
If PCL is the destination, then two instruction cycles are required.

Logical operations

Mnemonic		Description	Flags affected
andlw	k	And W with literal	Z
andwf	f d	And W with f	Z
comf	f d	Complement f	Z
iorlw	k	Inclusive or W with literal	Z
iorwf	f d	Inclusive or W with f	Z
xorlw	k	Exclusive or W with literal	Z
xorwf	f d	Exclusive or W with f	Z

f = file register address 0x00-0x7F
k = literal constant value 0x00-0xFF
d = W: result placed in W register
d = F or none: result placed in f register
Z = zero flag: set if result = 0

These operations execute in one instruction cycle, or four clock cycles.
If PCL is the destination, then two instruction cycles are required.

Load/Store operations

Mnemonic		Description	Flags affected
bcf	f b	Clear bit b in f	
bsf	f b	Set bit b in f	
movf	f d	Move f to destination	Z
movlw	k	Move literal to W	
movwf	f	Move W to f	
nop		No operation	
swapf	f d	Swap nibbles in f	
clrwdt		Clear watchdog timer (1)	TO,PD
sleep		Go into standby mode (1)	TO,PD

f = file register address 0x00-0x7F
b = register bit value 0x00-0x07
k = literal constant value 0x00-0xFF
d = W: result placed in W register
d = F or none: result placed in f register
Z = zero flag: set if result = 0
(1) these are power management operations, TO,PD are control bits

These operations execute in one instruction cycle, or four clock cycles.
If PCL is the destination, then two instruction cycles are required.

Program flow control operations

```
Mnemonic        Description                     Instr.
                                                Cycles
btfsc   f b     Bit test f, skip if clear       2 (1)
btfss   f b     Bit test f, skip if set         2 (1)
call    a       Subroutine call at addr         2
decfsz  f d     Decrement f, skip if Z=1        2 (1)
goto    a       Unconditional branch to addr    2
incfsz  f d     Increment f, skip if Z=1        2 (1)
retfie          Return and enable interrupts    2
retlw   k       Return with literal in W        2
return          Return from subroutine          2

f = file register address 0x00-0x7F
b = register bit value 0x00-0x07
k = literal constant value 0x00-0xFF
d = W: result placed in W register
d = F or none: result placed in f register
a = absolute branch address 0x00-0x7FF

These operations do not affect any flags.
Conditional branches skip the next instruction if test is true and
then require two instruction cycles or eight clock cycles to execute,
otherwise one instruction cycle is required.
```

System variables

```
These labels load if the 'Options','Load Defaults' box is checked:

File registers 0x60-0x7F are used by the PICLab operating system.
You can read/write these locations but do not modify registers 0x77-7B, 0x7E-7F.
The following labels load prior to assembly:

WH       equ    0x7D   ( MSB/LSB of a two-byte scratch space, used to pass)
WL       equ    0x7C   ( parameters to/from utility subroutines)
CH       equ    0x76   ( MSB/LSB of a two-byte software counter/scratch space)
CL       equ    0x75   ( used in Wait and other timing related subroutines)
Flags    equ    0x74   ( PICLab operating system flags)
Count5ms equ    0x73   ( variable updated every 5 ms by PICLab)

The Flags register contains the following system flags:

Flags(7=SIO)            ;serial output: 1=RS232, 0=LCD
Flags(6=LCDE)           ;LCD enable strobe state: falling edge latches data
Flags(5=LCDRW)          ;LCD display R/W state: 1=read, 0=write
Flags(4=LCDRS)          ;LCD display RS state: 1=data, 0=instruction
Flags(3=PGMWREN)        ;write to loader memory (0-3FF): 1=enable 0=disable
Flags(2=USERISR)        ;user ISR: 1=enabled, 0=disabled
Flags(1=SIGNED)         ;arithmetic: 1=signed, 0=unsigned
Flags(0=SIGN)           ;current sign: 1=negative, 0=positive

Three flags are available if the LCD display is not implemented. DO NOT modify
```

the PGMWREM bit; this could lead to the permanent modification of the loader code and cause the board to fail. Good programming practice requires that only bit operations be used to change the contents of this register.

The current value of the program counter during assembly is stored in the variable 'pc'. Microchip's MPLAB assembler uses the symbol '$' for this purpose. For example, a non-labeled branch can be made relative to the current program counter as follows:

```
        btfss   register bit
        goto    $pc-1           ('goto $-1' in Microchip's MPLAB)
```

will loop the test operation until the bit is set.

These labels load if the 'Options','Load Defaults' box is checked:

```
Seconds  equ    0x72   ( Real time clock seconds counter)
Minutes  equ    0x71   ( Real time clock minutes counter)
Hours    equ    0x70   ( Real time clock hours counter)

DPPtr    equ    0x6F   ( current decimal point position for LED, LCD/TCL)
DigitPtr equ    0x6E   ( points to the digit currently displayed, 0-3)
Digit3   equ    0x6D   ( Digit3 is the most significant LED digit.)
Digit2   equ    0x6C   ( Digit0 is the least significant digit, while)
Digit1   equ    0x6B   ( variables to the four seven-segment displays)
Digit0   equ    0x6A   ( 'call Refresh' outputs the contents of these)

temp     equ    0x69   ( general purpose temporary storage)
RTChi    equ    0x68   ( Real Time clock coarse frequency adjust)
RTClo    equ    0x67   ( Real Time clock fine frequency adjust)
Ticksh   equ    0x66   ( RTC time as 16-bit value spans 24h in 2s ticks)
Ticksl   equ    0x65   ( Use Seconds bit-0 to get 1s resolution)

Dec4     equ    0x64   ( 5 byte buffer for ASCII data transmission to LCD)
Dec3     equ    0x63   ( or PIC output window. Stores the result of binary)
Dec2     equ    0x62   ( to BCD conversions of 16-bit value in WH:WL)
Dec1     equ    0x61   ( Dec4 is most significant byte 0-9 of conversion and
Dec0     equ    0x60   ( Dec0 is the least significant byte)
```

An attempt to redefine these labels once they are loaded will cause a 'Label cannot be modified' error.

System subroutines

These routines load if the 'Options','Load Defaults' box is checked:

```
Break     suspend program execution, view processor status
          the symbol '!' can be used instead of a 'call Break' instruction
          modifies: none
Wait      software delay; the value in W times 775us@4MHz or 155us@20MHz
          modifies: CH, CL, W
```

```
Delay1s   hardware delay uses Count5ms; seconds value in W
          modifies: CH, CL, Count5ms, W

ReadAD    read ADC 10-bit value to WH:WL, channel 0-4 in W, ADC channels
          correspond to PIC pins: 0=RA0, 1=RA1, 2=RA2, 3=RA3, 4=RA5
          maximum conversion rate is 20000 samples/s
          modifies: CH, WH, WL ,W
Getkey    read keypad to W; returns 2-6 or 7 if no key pressed
          modifies: CH, WH, WL, W

Bin2BCD   convert 16-bit value in WH:WL to 5 bcd digits in Dec0-Dec4
          if Flags(SIGNED)=1, value is interpreted as 15-bit number plus sign
          and sign bit WH(7) is copied to Flags(SIGN)
          modifies: CL, WH, WL, W, FSR, Dec0-Dec4

LedTable  convert value 0-F in W to seven-segment bit pattern in W
          modifies: W
Refresh   output to 7-segment display next digit of Digit0-Digit3
          modifies: DigitPtr, W, PORTD, PORTB(3-0)
BCD2LED   convert value in Dec0-Dec3 to 7-segment bit patterns and store in
          Digit0-Digit3, decimal point included if DPPtr={0..3}. Digit3='E'
          if -1999>value>9999 according to Flags(SIGNED) and Flags(SIGN).
          modifies: Digit0-3, W

These Flash/EEPROM routines load if the 'Options','Load Defaults'
box is checked; These routines return pointing to file register direct page 0.

ReadEE    read byte from EEPROM memory to W; address in FSR
          modifies: EEADR, EEDATA, W
WriteEE   write byte in W to EEPROM memory; address in FSR
          the write requires around 4ms, the PIC continues to run but will
          wait until a previous write completes.
          modifies: EEADR, EEDATA, W

FlashSet  set current Flash memory address to value in WH:WL
          modifies: EEADRH, EEADR, W
FlashInc  increment current Flash memory address
          modifies: EEADRH, EEADR
FlashRd   load WH:WL with currently addressed 14 bit Flash memory data
          modifies: WH, WL, W
FlashWr   write to currently addressed Flash memory 14 bit contents of WH:WL
          during the write, the PIC stops executing instructions for around 4ms
          until the write completes; peripherals continue to run.
          modifies: EEDATH, EEDATA, W

A table of 14-bit data can be stored in flash program memory. The table begins
at the end of the user program and can occupy the rest of available flash
memory. This feature is useful for remote data acquisition when PICLab is not
connected to the host running PICL. After all writes to the data table are
made, CloseDT must be executed to properly store the data table length in
program memory. These routines return pointing to file register direct page 0.

A write to the data table requires around 4ms. The PIC does not execute
```

```
instructions during this time. Peripherals continue to run and interrupts
events are queued until the write completes.

LoadDP   load pointer to first word DT(0) of data table in EEADRH:EEADDR.
          modifies: W, WH, WL, EEADRH, EEADDR
OpenDT   LoadDP then store initial length of 2 to first table element DT(0)
          and a format word to second table element DT(1). Pointer is then set
          to first empty data word DT(2). Therefore a data table of N entries
          has a word length of N+2 and an empty data table has a length of 2.
          W is stored in low byte of 14-bit word at DT(1) and the upper six
          bits store the decimal point position in DPPtr ANDed with 7.
          Valid DPPtr value of 0-7 is applied to all the Y data.
          W formats the 'PICLab output' data into 8 character columns:
            if W = 0x0n, first column is X data, n-1 columns are Y data
            if W = 0x1n, X column generated by PICL as 1,2,... and n are Y data
          modifies: EEADRH, EEADR, W, WH, WL, temp
WriteDT   write 14-bit word in WH:WL to data table and increment the data table
          pointer stored in EEADRH:EEADDR to next empty element
          modifies: EEADRH, EEADR, EEDATH, EEDATA, W
ReadDT    read 14-bit word in data table to WH:WL and increment the data table
          pointer stored in EEADRH:EEADDR to next empty element
          modifies: EEADRH, EEADR, WH, WL, W
CloseDT   calculate data table length and store as first table element DT(0)
          modifies: EEADRH, EEADR, W, WL, WL, CH, CL

To read the contents of the data table connect to PICL then open the
'PICLab output' window and press the 'Read PIC' button. PICL will format the
incoming data into columns suitable for graphical output or display a message
if the data table is empty.

Note(1):  The address pointer to Flash memory is stored in EEADRH:EEADDR.
          FlashRd and FlashWr use the memory address currently stored in
          these registers. FlashInc increments this pointer and FlashSet sets
          this pointer to a specific value. The data word read from or written
          to Flash memory is stored in EEDATH:EEDATA.

Note(2):  The address pointer to EEPROM memory is stored in EEADDR. ReadEE and
          WriteEE load this register with the value in FSR. The data byte is
          transferred via the EEDATA register.


These serial PICL/PICLab routines load if the 'Options',
'Load Defaults' box is checked:

RxByte    receive to W byte from serial port, loop until ready
          modifies: W
TxByte    transmit byte in W to serial port, loop until ready
          modifies: none
TxDigit   add 0x30 to W and TxByte, ASCII decimal digit conversion
          modifies: W
Reg2TCL   send byte in W as two ASCII hex digits 00-FF plus newline
          modifies: WH, WL, W
Hex2TCL   send byte in W as two ASCII hex digits 00-FF
```

```
          modifies: WH, WL, W
Dec2TCL   send byte in W as two ASCII decimal digits 00-99, no limit check
          modifies: WH, WL, W
BCD2TCL   convert Dec0-Dec4 to ASCII decimal and display from lsd
          digit count in W={1..5}, invalid W values default to 5
          if DPPtr(7)=0, a decimal point DPPtr={0..4} is displayed before
          corresponding digit value, if Flags(SIGNED)=1, '-' sign or blank
          is sent as per Flags(SIGN), value overflow outputs a field of '*',
          field width is W plus sign if used plus decimal point if used
          modifies: FSR, temp, W

RTC2TCL   send current PICLab real time clock value in ASCII format hh:mm:ss
          modifies: WH, WL, W

To use, load values then execute a call instruction. The above routines can
be used to send data or commands from a user program to the PICL software.
Transmitted ASCII characters are stored in a serial receive buffer until
a newline character is received. The string of characters is then processed
by PICL as follows:

ASCII characters 0x20-0xFF are written to the PICLab output window;
ASCII characters 0x00-0x1F represent control characters:

0x0A :newline character causes PICL to process serial data string
0x1C :refresh gnuplot window if PICLab output window is open
0x1B :clear PICLab output data if PICLab output window is open

Example: refresh the gnuplot graphics window from the user program

movlw   0x1b    ;command to clear data in PICLab window
call    TxByte  ;send ASCII byte to PICL serial buffer
movlw   '\n     ;newline character
call    TxByte  ;send newline, execute clear command

your code here loads the piclab window with data columns.
Generate a line of data by using calls to BCD2TCL or Dec2TCL
separated by a space character 0x20 and terminate each line
with a newline character

movlw   0x1c    ;command to redraw gnuplot window
call    TxByte  ;send ASCII byte to PICL serial buffer
movlw   '\n     ;newline character
call    TxByte  ;send newline, execute draw command

These LCD control routines load if the 'Options','Load Defaults'
box is checked:

LCDinit   initialize LCD interface, clear display and set address 0
          modifies: CH, CL, PORTD, W
LCDstat   read to W current LCD address/ready status
          modifies: CH, CL, PORTD, W
LCDdata   read to W LCD display data at current RAM location
          modifies: CH, CL, PORTD, W
```

```
LCDcmd    write to LCD command/character data stored in W
          modifies: CH, CL, PORTD, W

LCDset    set LCD display write address, value in W
          modifies: CH, CL, PORTD, W
ASC2LCD   display on LCD byte in W as ASCII character
          modifies: CH, CL, PORTD, W
Hex2LCD   display on LCD byte in W as two ASCII hex digits 00-FF
          modifies: CH, CL, PORTD, WH, WL, W
Dec2LCD   display on LCD byte in W as two ASCII decimal digits 00-99,
          no range check is performed
          modifies: CH, CL, PORTD, WH, WL, W
BCD2LCD   convert Dec0-Dec4 to ASCII decimal and display from lsd
          digit count in W={1..5}, invalid W values default to 5
          if DPPtr(7)=0, a decimal point DPPtr={0..4} is displayed before
          corresponding digit value, if Flags(SIGNED)=1, '-' sign or blank
          is sent as per Flags(SIGN), value overflow outputs a field of '*',
          field width is W plus sign if used plus decimal point if used
          modifies: FSR, temp, W
RTC2LCD   display on LCD real time clock value in ASCII format hh:mm:ss
          modifies: CH, CL, PORTD, WH, WL, W

reads/writes to the LCD display post-increment the display address.
The display memory is organized as two lines of 40 characters. Note that the
first character of the second line is at memory address 0x40. You can scroll
the visible 16x2 display along the 40x2 character memory, program special
charachers and change the display behaviour. (see HD44780 manual)

To use, load values then execute a call instruction.
```

Assembler directives

```
equ, set directives

Syntax: <label> equ <expression>

Assign a value to a label. The expression can be a previously
defined label or a TCL expression, in which a label name is
preceded by a $ sign.

An equ label must evaluate to an integer and cannot be redefined.
The assembly message displays the value in hexadecimal format.

Examples:

hex_number        equ     0x20
decimal_number    equ     123
ASCII_character   equ     'A
8_bit_binary      equ     %01001110 (or B'01001110', include 8 bits)
Integer_expr      equ     int(($ALPHA+10)*2.5)

Use the set directive to define a variable label that will accept
```

non-integer values. The assembly message displays a decimal value.

Examples:

```
PI      set     3.14159
```

macro directive

Syntax: <label> macro [<var, ..., var>] [<inst, ..., inst>] endm

A macro is a set of instructions that is inserted into the
assembly code using a single macro call. It can also be used
to redefine opcode names. A macro may call itself or another
macro and may include comments. End a macro definition with the endm
instruction.

Example 1: opcode redefinition

```
SkipCS  macro
        btfss   STATUS C
        endm
```

Example 2: register to register move

```
Move    macro   srcreg dstreg
        movf    srcreg W
        movwf   dstreg
        endm
```

Example 3: register negate, twos complement

```
Negate  macro   reg
        comf    reg     ;ones complement register
        incf    reg     ;twos complement register
        endm
```

cblock directive

Syntax: cblock <expression> <label>[:incr] [,<label>[:incr]] endc

Assign incremental values to a list of labels.
The first label is assigned the value of <expression>.
The increment value is optional and defaults to one if none is given.

Used to label non-byte data or to specify file register labels without
the need for a series of equ sattements. The list may include comments.

Example 1:

```
cblock 0x20             ;user file registers begin at 0x20

byte1                   ;1 byte  : byte1=0x20
byte2                   ;1 byte  : byte2=0x21
```

```
word:0  wordh,wordl     ;2 bytes : word=0x22, wordh=0x22, wordl=0x23
buffer:0x10             ;16 bytes: buffer, 0x24 to 0x33
byte3:1                 ;1 byte  : byte3=0x34
queue:QUEUESIZE         ;reserve QUEUESIZE registers at queue=0x35

encd                    ;end of constant label block
```

Example 2:

```
cblock 0x20 alpha beta delta gamma endc ;compact cblock definition
```

#eeprom directive

syntax: #eeprom <start_address> [D] {<expr> <expr>, ..., <expr>}

Load EEPROM memory with byte/word data starting at at start_address
0 to 0xFF. The data is space delimited and enclosed in curly brackets.
An error occurs if a data write is attempted beyond address 0xFF.
If the start_address is followed by 'D', then each data element is
interpreted as a sixteen-bit value and written high/low byte into two
adjacent memory locations.
Open the 'EEPROM memory' window to view the current contents of
the EEPROM memory.

Example 1: load string of 8-bit constants to EEPROM at 0x10 and 0x20

```
#eeprom 0x10 { 0 1 2 3 4 5 6 7 8 }
#eeprom 0x20 { 'H 'e 'l 'l 'o \0x20 'W 'o 'r 'l 'd }
```

Example 2: load string of 16-bit constants to EEPROM at 0x30

```
#eeprom 0x30 D {65535 0 -1 0xFFFF }
```

#include directive

Syntax: #include <pathname>

Include an external file as part of the assembly. The assembly code is
inserted at the position of the include statement. This directive
is used to access libraries of frequently used macros and labels.

At the start of each assembly, PICL loads a list of register name
definitions. This avoids a read to the hard drive and hence executes faster.
If the 'load defaults' box in the Options menu is checked, a list of
utility routine and variable names is also assembled.

#UserISRon command

Syntax: #UserISRon <label>

Defines the named routine as the current user interrupt service routine.
The routine will be run as part of the 5ms interrupt cycle while the board
is executing user code. The ISR vector is stored in program memory and will

persist until changed by another #UserISRon command.

The user ISR is enabled by the #UserISRon command, however it is good practice to explicitly enable the user ISR with a 'bsf Flags,USERISR' instruction. This must be done if a user program is set to execute after a hardware reset. A 'bcf Flags,USERISR' instruction disables the user ISR.

The user ISR routine will increase the execution time of the user program; routines requiring over 5ms to complete will disrupt the timing of the 5ms interrupt loop.

Example:

```
#UserISRon Refresh      ;enable background scanning of LED display

begin   bsf     Flags,USERISR   ;optional instruction
        ;include code here to update display buffer
        goto    begin
```

#debug directive

Syntax: #debug <value>

The assembly of the user code halts on the first error encountered. The offending line is highlighted in the user code and an error message is displayed. This is intended to make the debugging of the user code easier and avoid long lists of error messages, mostly due to some previous error.

The assembly of the user code may be monitored using informational messages. This output is presented as part of the Assembler output. A message includes feedback on label, variable and macro definitions. The display of messages can be toggled using the Show messages button in the Options menu or by using the #debug command:

#debug 0: - no messages displayed in Assembler output;
#debug 1: - messages are displayed in the Assembler output.
#debug 2: - messages output if 'Options, Show Messages' box is checked

This command can be used to selectively display messages during the assembly of the code. For example, there may be a long list of proven code that is being imported by an #include statement and you may not wish to display all the resulting message output:

```
#debug 0
#include filename
#debug 1
your code
```

#RTCsync command

Syntax: #RTCsync

Synchronize the PIC real time clock with the current time of the host
running the PICL script. The PICLab board implements a real time clock
via the 5ms interrupt that updates the variables Count5ms, Seconds, Minutes
and Hours. At power up or after a reset these variables are set to zero.
The real time clock then counts time since power-up.

The RTCsync command loads the current PICL time to the PICLab board and
can be used to periodically reload the current time in order to minimize
timing drift.

Alternately, if the PICLab real time clock is running either fast or slow,
the 5ms timing loop can be fine tuned by adjusting the contents of file
registers RTChi and RTClo. This facility can be used to compensate for
board specific variations in crystal and oscillator frequency.

#gset, #gplot commands

These commands control the Gnuplot display and data format.
Embedded in the user code, these selections override the default PICL
Gnuplot settings and hence can generate a graphical interface that is
specific to the needs of a user program.

Use #gset to execute Gnuplot commands such as set and unset:

```
  #gset " set pointsize 15; unset key; unset xtics; unset ytics "
  #gset " set xrange [.5:8.5]; set yrange [.5:7.5] "
```

The #gplot command behaves like the Gnuplot plot command. Note that
each data set requires a #gplot statement:

```
  #gplot " '-' using 1:2 with points pt 7"
  #gplot " '-' using 1:3 with points pt 7"
  #gplot " '-' using 1:4 with points pt 6"
```

PICL determines the number of data sets to plot by counting the number of
data points on the first line of the PICLab output window. The number of
#gplot statements should match this number of data sets.

Hardware

PICLab boot sequence:

After a reset, PICLab initializes the PIC port pins as follows:

```
PORTA: RA0-RA3 as analog inputs 0-3, RA5 as analog input 4, RA4 as input;
PORTB: RB0-RB3 as outputs, connected to LED display digit drivers;
       RB4=RTS_OUT and RB5=CTS_IN as handshake lines of serial port;
       RB6-RB7 as outputs, connected to ICSP socket;
PORTC: RC0-RC5 as inputs, RC6=Tx and RC7=Rx as data lines of serial port;
PORTD: RD0-RD7 as outputs, connected to LED display segments, LCD control;
PORTE: RE0-RE2 as inputs.
```

The PIC timer TMR0 is programmed to generate an interrupt every 5ms. The user interrupt service routine is disabled. The real time clock is reset.

PICLab then attempts to auto-detect the LCD display. If the probe fails, the LED display is flashed to check that all digits operate, then only digit 0 is enabled. The LED and LCD displays both use PORTD and hence their presence on the PICLab board is mutually exclusive.

Next, a mode flag stored in Flash memory is checked. This flag determines whether PICLab will execute the loader program or a stored user program.

If the loader program is to be executed, PICLab sends a release character to PICL in case the user program was aborted with a reset, then waits for user input.

If the user program is to be run, PICLab jumps to the start of the user code. Note: A user ISR set with the #UserISRon command can be enabled at anytime by executing a 'bsf Flags,USERISR' instruction as part of the user code.

PICLab reset options:

If a keypad button is pressed during the reset sequence, PICLab enters a loop that allows the user to change the operation of the PICLab board. To enter the loop, press and hold the appropriate keypad button(s), then momentarily press the RESET button. The current options are as follows:

```
LEFT+UP: toggle run mode flag to execute loader/user program after a reset
UP     : run keypad diagnostic routine
LEFT   : run keypad diagnostic routine
ENTER  : run keypad diagnostic routine
RIGHT  : run keypad diagnostic routine
DOWN   : run keypad diagnostic routine
```

To exit the loop, reset the PICLab board.

Shown below is the arrangement and identification of the PICLab keys and the value returned by the Getkey subroutine when a key is pressed:

```
                                        RESET

                                  UP(2)

                        LEFT(3)    ENTER(4)    RIGHT(5)

                                  DOWN(6)
```

PICLab/PICL serial interface

PICLab and PICL communicate via a serial port. This port is used to upload assembled code and commands from PICL to PICLab and to exchange data between PICL and the running user program. The PICLab serial port is configured as:

baud rate: 57600

```
data bits: 8
stop bits: 1
parity:         none
handshake: RTS/CTS. ('hardware' in Windows)
```

These settings should be matched on the serial port used by PICL.
On a Windows operating system, disable buffering of the local port.
PICLab must be in program mode to accept loader commands.

On startup, PICL should automatically connect to PICLab if the port settings are correct and the PICLab serial cable is attached. If there is no connection, the icon on the left of the 'Host:' box will display a single disconnected plug. You can manually make the connection as follows:

If the serial port that PICLab is connected to is on the machine running the PICL software, enter the name of the local port (i.e. com1 or /dev/ttyS0) in the 'Port:' entry box and leave the 'Host:' box empty.

If the serial port that PICLab is connected to is not on the machine running the PICL software, set up a network connection to the remote host by entering that IP address in the 'Host:' entry box and specifying 'Port:' 87.

Once these selections are made, click on the icon. The icon will display a connected plug/socket if the connection was successful.

You can test the serial connection from the PICLab board by executing an instruction that checks if PICL is ready to accept data:

```
        btfsc   PORTB,CTS_IN    ;CTS_IN is cleared if PICL is ready
        goto    no_connection
        goto    connected
```

LED display connections:

```
  PORTD    7 6 5 4 3 2 1 0      PORTB  3 2 1 0
  segment  p g f e d c b a      digit  3 2 1 0


      f g a b 3 2 f a b   f g a b 1 0 f a b              a
    x 0 0 0 0 0 0 0 0 0 x 0 0 0 0 0 0 0 0 0             ---
                                                     f| g |b
    40-pin display socket viewed from above            ---
                                                     e| d |c
    x 0 0 0 0 0 0 0 0 0 x 0 0 0 0 0 0 0 0 0             --- .p
      e d c p e d g c p   e d c p e d g c p
```

Corresponding segments of the four displays are wired in parallel. Each segment connects to a PORTD bit via a 470 Ohm current limiting resistor. The common cathode of each display is switched to ground via a NPN transistor controlled by a PORTD bit. To turn on a display segment, the PORTD segment bit and the PORTB digit bit connected to the common cathode must both be set to a high logic state.

Interrupts:

After a reset, PICLab initializes Timer0 (TMR0) to generate a hardware interrupt (IRQ) every 5ms interval. When an IRQ takes place, further IRQs are disabled and remain pending. The called routine saves the W and STATUS registers, optionally executes a user ISR and then increments the Count5ms variable (0-199) and updates the real time clock. It then restores W and STATUS, enables IRQs and exits. To setup a user ISR, consult the #UserISRon directive help menu.

You can change the interrupt rate of TMR0 by selecting different TMR0 prescaler and divisor values. The OPTION_REG bits 0-2 are loaded with a value N corresponding to a prescale factor of 2^(N+1). RTChi and RTClo are predefined user variables. The Timer0 interrupt frequency is given by:

F(irq) = F(osc)/(4*prescaler*(255-RTChi))

In our case, F(osc)=20MHz, prescaler=128 and RTChi=58

You can increase the precision of F(irq) by varying the contents of RTClo. For reference, the IRQ code that takes care of this fine tuning is:

```
        movf    RTClo,W         ;trim seconds by varying TMR0 divisor over
        subwf   Count5ms,W      ;200 Count5ms cycles:
        movf    RTChi,W         ;Count5ms >= RTClo, load TMR0 with RTChi+1
        SkipCC                  ;Count5ms <  RTClo, load TMR0 with RTChi
        addlw   1
        movwf   TMR0            ;load TMR0 and start timer
        incf    Count5ms,F      ;increment 5ms software timer
```

Note that changing these TMR0 values will disrupt the operation of the real time clock. To execute an ISR in response to a IRQ form another of the many PIC hardware interrupt sources and leave the 5ms timing unaffected, you need to initialize and enable that IRQ source in your main program and test for the appropriate IRQ in your ISR code:

```
MyISR   btfss   irqreg,bit      ;your IRQ sets bit in irqreg ...
        return                  ;not your IRQ, so exit and execute TMR0 ISR

        your ISR code           ;interrupts are disabled and W and STATUS
        goes here               ;are saved before MyISR is called, but you
        ----                    ;may need to save and restore other registers
        ----                    ;modified by MyISR
        ----
        return                  ;updates TMR0 if TMR0 IRQ is pending, then
                                ;restores W and STATUS, enables IRQs
```