NAME (PRINT)	STUDENT ID	LAB DATE	GRADE

Experiment 8

PICLab programming

Brock's own PICLab is the development board package that will be used in several experiments in this lab. The microcontroller used in PICLab is the Microchip PIC16F877-20 or PIC16F887. With a 20-MHz oscillator, most instructions require 200 ns (4 clock cycles) to execute. In addition to the PIC itself, the PICLab board contains power supply, display, and interface circuits necessary to communicate with the board via a USB port of a Linux workstation or PC.

PICLab bootloader

A small bootstrap utility program has been pre-loaded into the memory of your PICLab, and a computer program called `picl` has been written to provide transparent communications with the PICLab board.

`picl` IDE

`picl` is an Integrated Development Environment of the PICLab board. It allows you to write assembler programs, compile and download them to the memory of the PIC, and to examine the state of the PIC memory or registers during the debugging process. `picl` can also provide a real-time text and graphical display of data sent by your running program.

`picl` also includes a PICLab simulator. Implemented in software is most of the functionality of PICLab, including the internal hardware of the microcontroller and the external hardware elements such as the LED display, keypad, LCD display and serial port. The user subroutines that are preloaded on PICLab are also simulated in software. Hence, your code developed with the simulator should run as expected on the real PICLab.

With the simulator you can easily single step through your code and monitor the outcome of each instruction. Further, Virtual Pic allows you to view how an instruction is executed inside the PIC and the path that your data follows on every cycle of the PIC clock.

You can easily switch execution of your code between the simulator and the PICLab board by making or breaking the PICLab connection.

PICLab schematic

Fig. 8.1 provides an overall block diagram of the PICLab board, showing the essential connections between the PIC itself and the other components of the PICLab board. Together with the `picl` help menus and the PIC reference documentation (see the References section of the class website) this information should be sufficient for you to get your PICs to work.

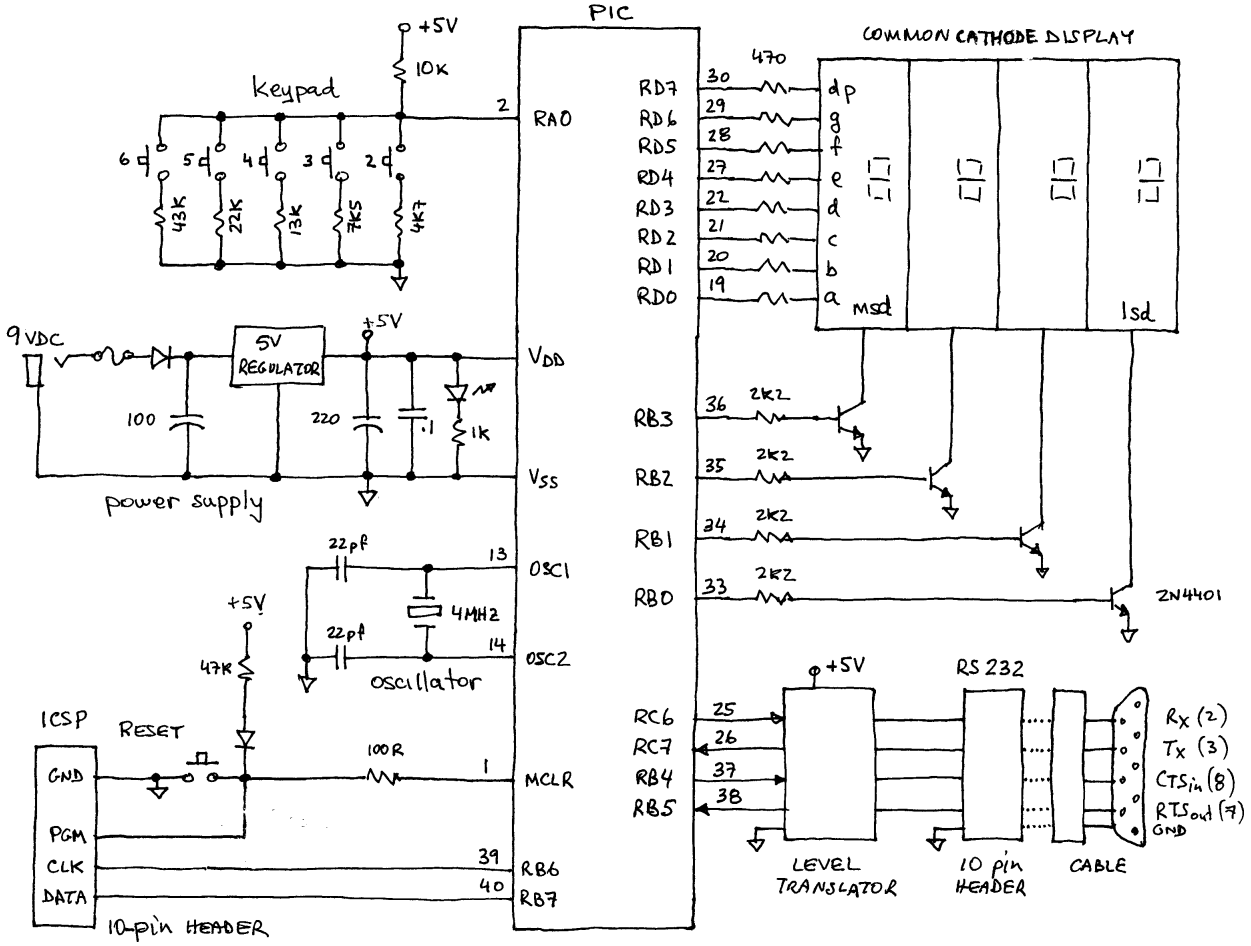

Figure 8.1: Block diagram of a PICLab board

Connecting PICLab

By default, `picl` enters the PIC Simulator mode on start-up with the 'Connect' icon showing a single plug meaning that PICLab is not currently connected to PICL. With your PICLab board connected to the USB port of your Linux workstation and the port set to `/dev/piclab`, click the connection icon. The icon changes to two connected plugs, a message 'Connected to PICLab at 57600 baud" appears in the status box, and `picl` is ready to communicate with PICLab. Check the 'Auto connect' box in the 'Settings' menu to detect and connect to PICLab on start-up. [1]

PICLab interface

On your Linux workstation, invoke the graphical user interface to PICLab by typing:

```
picl &
```

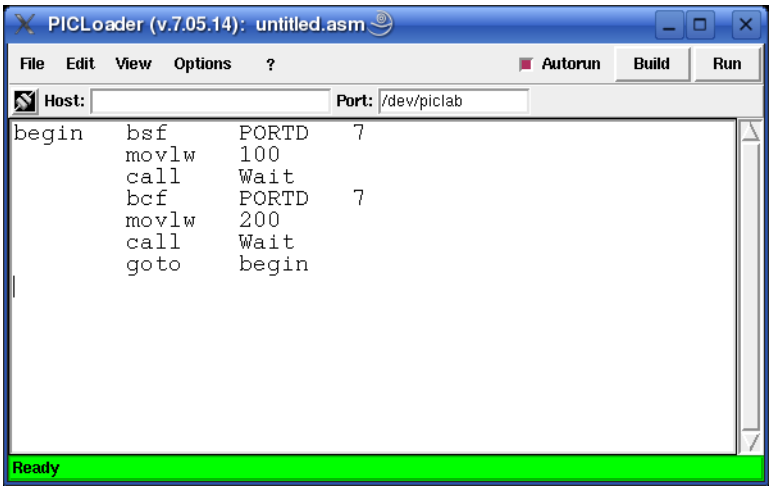

Figure 8.2: `picl` application window, connected to PICLab

Fig. 8.2 shows what `picl` window should look like on your screen. Typically, you enter one or more opcodes in the entry window and click **Build**. If the assembly completes without errors, PICLab loads the given instruction(s) to the Flash program memory and executes them.[2] This code will not erase if the PIC is reset or the power is turned off. You can execute the program currently stored in the PIC memory by clicking the **Run** button.

As the program runs, PICLab sends a variety of data back to the user. You can open several windows in the **View** menu to keep track of how the values in the PIC registers and memory change as a result of your instructions being executed.

Click the **?** button for help on the PIC opcodes, assembler directive commands and a list of the utility subroutines pre-written for you to use in your programs. Click 'Exit' in the 'File' menui when done, your working environment will be saved.

[1] To program PICLab from your laptop or PC, you need `picl` and the Tcl/Tk 8.4 interpreter installed; it is freely available for all platforms at `www.activestate.com`. You may also need to specify the connection: e.g. /dev/piclab or /dev/ttyUSBn under Linux or COMn under Windows, where n is the desired port. Under Windows, a USB serial device is identified as a COM port.

[2] By default, the loader chooses 0x0400 as the starting address of the user program, out of the total program address space of PIC of 0x0000–0x1FFF; the loader itself is using 0x0000–0x03FF.

8.1 Assembler instructions and code development

You are now ready to begin programming. As you progress through the exercises, be sure to understand the function of each instruction (see the help menu) and the overall logic of the program code. Familiarity with the PIC instruction set and with these basic technoques of interacting with PICLab will make programming the PICLab board a more pleasurable experience.

(!) Start the `picl` application. Connect to PICLab. In the following steps, the comments shown in brackets do not need to be entered; the text can be formatted using the tab key.

Begin by sending to the PIC an instruction to turn on bit 7 of its Port D. This pin is connected to the decimal point of the seven-segment displays. In the entry window, input the following opcode:

```
        bsf     PORTD,7     ; set bit 7 of file register PORTD
```

Click the **Build** button. The LED attached to bit 7 of Port D on the PIC should turn on. Modify the above instruction to read as follows:

```
        bcf     PORTD,7     ; clear bit 7 of file register PORTD
```

Clicking **Build** should turn off the LED.

(!) You will now implement a loop. Enter the following code:

```
begin   bsf     PORTD,7     ; set bit 7 of file register PORTD
        bcf     PORTD,7     ; clear bit 7 of file register PORTD
        goto    begin       ; branch to label called "begin"
```

Since the program is toggling the Port D bit on and off a couple of times every microsecond, the LED should appear continuously lit, but somewhat dimmer. The PICLab board is now executing an infinite loop and will not respond to commands.

Connect an oscilloscope to pin 7 of `PORTD` on the expansion connector. Sketch and label the output waveform.

[?] Explain the timing in terms of the PIC instruction execution times. Is the timing in agreement with your expectations?

(!) Press the **Reset** button on the PICLab board to interrupt the program and regain control of the hardware.

You can slow down the LED flashing rate by introducing a long, in PIC terms, delay after each of the bit operations. A utility subroutine called `Wait` is available to implement such a delay. The `W` register is loaded with a delay value to be passed to the subroutine. Try the following code:

```
begin   bsf     PORTD,7     ; set bit 7 of file register PORTD
        movlw   250         ; pass delay count to Wait subroutine
        call    Wait        ; execute a delay of 250*150us
        bcf     PORTD,7     ; clear bit 7 of file register PORTD
        movlw   250         ; delay value for Wait subroutine
        call    Wait        ; execute a delay of 250*150us
        goto    begin       ; branch to label called "begin"
```

The flashing of the LED is now clearly noticeable. Vary the value in the `movlw` instructions to observe how the flashing rate varies.

8.2 Loops, conditional branching, and calls to subroutines

The next step in our exploration of PIC programming is to add some flow control to the program's execution. One possibility is to have the algorithm flash the LED a set number of times, then terminate and return control to the user. You can use labels to make the program more readable. The `equ` directive assigns a value to a label. The following code will flash the LED `COUNT` times and terminate.

```
; Flash.asm: Program to flash LED on and off a specified number of times
COUNT_REG equ       0x20         ; use register 0x20 to count, 0..1F are reserved
COUNT     equ       0x10         ; count value to be put into the count register
DELAY     equ       0xff         ; "Wait" this many tics, ~150us ea

          movlw     COUNT        ; put a count value into the accumulator
          movwf     COUNT_REG    ; put accumulator into the count register

flash     bsf       PORTD,7      ; turn on LED segment
          movlw     DELAY        ; pass DELAY count to function Wait
          call      Wait
          bcf       PORTD,7      ; turn off LED segment
          movlw     DELAY        ; pass DELAY count to function Wait
          call      Wait
          decfsz    COUNT_REG    ; decrement count, skip over the next...
          goto      flash        ; ...instruction when file register=0
```

Another way to terminate a loop is to check for a certain condition and run until it is satisfied, such as when the user presses a button. The `Getkey` subroutine reads the keypad and returns in `W` a value of 2-6 if a button is pressed, otherwise a value of 7 is returned. The `STATUS` register maintains the state of several flags that can be tested to alter the program flow. The zero flag `Z` is set when the result of an operation is zero, and is cleared otherwise.

(!) Document the following code and test the algorithm by running the program. What does the program do? Does the program behave as expected? Consult the help menu (?) to obtain more information on the PIC opcodes and utility subroutines that are available for you to use.

```
; Showkey.asm:      Program to ......................................
KEYSAVE   equ       0x20         ; ......................................
          clrf      PORTD        ; ......................................

readkey   call      Getkey       ; ......................................
          movwf     KEYSAVE      ; ......................................
          sublw     7            ; ......................................
          btfsc     STATUS,Z     ; ......................................
          goto      readkey      ; ......................................

          movf      KEYSAVE,W    ; ......................................
          movwf     PORTD        ; ......................................
          sublw     2            ; ......................................
          btfss     STATUS,Z     ; ......................................
          goto      readkey      ; ......................................
          return                 ; required if code follows main program .
```

In the above example, a simple return from a subroutine (instruction `return`) is being used. In the `picl` convention, the entire user code is assumed to be a subroutine of the PICLab loader, and so at the very end of the code, an automatic return is always inserted for you. This is why your one-line "programs" like `bsf PORTD 7` worked just fine even though they did not have a `return` operation. However, you must insert an explicit return at the end of every subroutine that you yourself write, and at the end of your main program if any code follows it.

An extra `return` somewhere in the middle of the code can also be used as a simple debugging tool. Upon encountering such a "premature" return, the program will terminate, pass the control to the PICLab loader, and it in turn will update all of the open windows of `picl` with the current values of various registers, memory contents, *etc.* You will then be able to examine the current status of your PIC and decide if the code you wrote is doing exactly what you intended it to do.

You may also execute a `call Break` instruction to update `picl` with the recent values from the PIC, and to continue program execution. This subroutine is not a part of the PIC instruction set, but is made available through the utility loader function set. You can use the `!` symbol in a blank line as a short form for `call Break`.

Note: The `call Break` and `!` instructions may cause unexpected program behaviour when placed in your code following a conditional branch instruction or as part of a jump table.

In addition to the simple returns, the PIC instruction set has other flow control instructions that allow one to take some programming shortcuts. For example, `addwf PCL F` instruction increments the current program counter (PC) by a value stored in the `W` register before proceeding to the instruction stored at that location. In this way, an indexed goto statement is implemented. The `retlw` is a combined load-and-return instruction; it first loads the `W` register with a literal value and then exits by executing a return-from-subroutine instruction.

? You can use a lookup table to convert binary data into bit patterns that corresponds to a decimal digit on the seven-segment display. Determine from the PICLab schematic the mapping of `PORTD` pins to the display segments and write down the bit patterns that represent decimal digits 0 through 7 on the seven-segment display.

! Append the following code to the above program (that is why you needed the explicit `return` at the end of it) and convert your keypad data by calling the `Convert` subroutine at an appropriate time in your program. Explain the program flow of this code. What is the purpose of the `andlw` instruction?

```
Convert   andlw     7           ; ....................................
          addwf     PCL,F       ; ....................................
          retlw     %00111111   ; seven-segment bit pattern for digit "0"
          retlw     %________   ; ....................................
          retlw     %________   ; ....................................
          retlw     %________   ; ....................................
          retlw     %________   ; ....................................
          retlw     %________   ; ....................................
          retlw     %________   ; ....................................
          retlw     %________   ; ....................................
```

Each seven-segment display consists of eight LEDs connected together at the cathode (-). The `PORTD` pins connect to the individual LED anodes (+). With the common cathode pins of each display connected to ground, it would require 32 bits to control the four displays of the PICLab board.

A more efficient use of resources employs the technique of time multiplexing to generate an output on the four displays. Here, the digits are displayed sequentially with only one of the four digits enabled at anytime. If the switching between digits is sufficiently rapid, the persistence of the human eye creates the illusion that all the digits are on at the same time.

Four output pins (`PORTB` pins 0–3) control the voltage at the display cathodes via current-driving transistors. The corresponding anodes of the four displays are connected in parallel (refer to the PICLab schematic, Fig. 8.1). A software loop then enables each of the displays in turn while the bit pattern corresponding to that digit is presented on the `PORTD` pins. With multiplexing, the pin count has been reduced to 12 from 32, a saving of 20 input/output pins.

(!) Develop a flowchart to multiplex four different digits of data on the PICLab display. Convert the flowchart to PIC instructions and test your code.

(!) Vary the loop timing to determine the minimum refresh rate necessary to prevent the display from flickering.

8.3 Macros and subroutines

A Macro is a group of instructions that are referred to as a single new instruction. During assembly, every time a Macro instruction is encountered, the original group of instructions is assembled into the program.

A subroutine consists of a group of instructions within the user program that begin with a subroutine name tt label and end with a `return` statement. A `call label` instruction branches the program to `label` and executes the subroutine code until the `return` instruction restores the program flow to the instruction following the call.

The following code incorporates some practical programming techniques to implement a display multiplexing scheme. A macro definition is shown as well as an efficient method of implementing a jump table to select one of several branch possibilities.

(!) Analyse and document the code; explain clearly the functionality of the subroutines, then compare the functionality of this program with your version. You may want to save your code into a file (say, `Show4.asm`) before you build and run it. Be sure to add some of your own code to provide meaningful values to `7seg_0 ... 7seg_3`, then run the program.

```
; ..... Show4.asm: multiplex the four-digit seven-segment display ........

7seg_0    equ       0x20         ; least significant display digit .......
7seg_1    equ       0x21
7seg_2    equ       0x22
7seg_3    equ       0x23         ; most significant display digit ........
7seg_ptr  equ       0x24         ; pointer to current digit displayed.....

Move      macro     src,dst      ; register to register move, modifies W
          movf      src,W        ; the macro defines a new Move
          movwf     dst          ; instruction that is not available
          endm                   ; as part of the PIC instruction set

;         your code goes here, with a call to Scan7seg
```

```
          return

Scan7seg                          ; ....................................
          incf      7seg_ptr,F    ; ....................................
          movlw     3             ; ....................................
          andwf     7seg_ptr,F    ; ....................................
          movlw     0xf0          ; ....................................
          andwf     PORTB,F       ; ....................................
          call      Show7seg      ; ....................................
          iorwf     PORTB,F       ; ....................................
          return                  ; ....................................

Show7seg                          ; ....................................
          movf      7seg_ptr,W    ; ....................................
          addwf     PCL,F         ; ....................................
          goto      Show0         ; ....................................
          goto      Show1         ; ....................................
          goto      Show2         ; ....................................
          goto      Show3         ; ....................................
Show0     Move      7seg_0,PORTD  ; ....................................
          retlw     %00000001     ; bit 0 selects display 0, active high ..
Show1     Move      7seg_1,PORTD  ; ....................................
          retlw     %00000010     ; ....................................
Show2     Move      7seg_2,PORTD  ; ....................................
          retlw     %00000100     ; ....................................
Show3     Move      7seg_3,PORTD  ; ....................................
          retlw     %00001000     ; ....................................
```

The bits of a port are generally assigned various functions so you must take care to modify only the pertinent bits when using byte size instructions. This masking process requires reading the current port value, modifying only specific bits, then writing back the data to the port. Bit manipulation instructions are not useful when the bit to be modified varies, as in the selection of the digit to be displayed.

8.4 Interrupts

You will note that depending on the code that you added, the above program will initialize the digit values and display them indefinitely, or you will have implemented a loop that modifies the digit variables and calls the `Scan7seg` subroutine. In the first case, the PIC is fully occupied scanning the display and can perform no other function; in the second case the refresh rate is determined by repeated calls to a subroutine.

The ideal way to execute a periodic sequence of events is to use an interrupt. A hardware timer on the PIC interrupts the program flow every 5ms and executes a call to a user interrupt service routine (ISR). The ISR code runs in the background independently of the user program. You can, with an ISR, update the display variables or wait for input while the `Scan7seg` ISR scans the display at a constant rate. Note that the execution time of the ISR must be less than the time between interrupts or your program will hang. With this in mind, your ISR is disabled when the PIC is reset.

To define an interrupt service subroutine that will be remembered by the PIC until redefined, add the `#UserISRon` directive following your ISR subroutine code:

```
#UserISRon Scan7seg       ;set Scan7seg as user ISR and enable interrupt
```

The ISR routine will only execute while your program is running. To test some ISR code, you can program a one-line instruction (e.g. `here  goto here`) to execute an infinite loop; the ISR routine will execute until the PICLab board is reset. The user ISR routine can be turned on and off from within your program with the following instructions:

```
bcf       Flags,USERISR   ;reset user ISR flag, disable user ISR
bsf       Flags,USERISR   ;set user ISR flag, enable user ISR
```

8.5 Analog-to-digital conversion

The PIC can sample one of eight input channels with a 10-bit resolution. To perform an analog-to-digital conversion (ADC), an input channel is selected. A delay follows, to allow the input voltage to be sampled. A start of conversion flag is set to begin the ADC and another flag is set when the conversion is completed. The data is then ready to be used.

The `ReadAD` subroutine performs all of the above tasks. Load the `W` register with the number of the input channel and call the routine. After 50μs, the lower eight bits of data are returned in the `WL` file register and the two most significant bits are in `WH`. The `Getkey` routine reads A/D channel 0 and uses the three most significant bits of the converted value to determine which key was pressed.

(!) `picl` has predefined pointers to the 7-segment LED displays called `Digit0..Digit3`, a subroutine equivalent to `Scan7seg` called `Refresh` and a routine `LedTable`, similar to your `Convert` routine, that converts a value 0-0x0F contained in `W` to the 7-segment pattern for the corresponding hex digit. For convenience and to make your code more compact, use these as part of your programs.

The following code reads a 10-bit value from the A/D converter channel connected to the keypad and displays it as three hexadecimal digits on the LED display. Complete the missing code and verify that the program functions as expected:

```
          #UserISRon  Refresh       ;define LED display scanning routine as user ISR

          bsf       Flags,USERISR   ;enable execution of user interrupt routine

begin     .....     .....           ;select the keypad channel
          .....     .....           ;read 10-bit A/D value, store in WH:WL
          .....     .....           ;place lower 8 bits of 10-bit A/D value in W
          .....     .....           ;set bits 4-7 to zero, bits 0-3 are A/D bits 0-3
          call      LedTable        ;convert value in W to 7-segment hex digit
          movwf     Digit0          ;display hex digit for A/D bits 0-3
          swapf     WL,W            ;place swapped nibbles (hex digits) from WL into W
          .....     .....           ;set bits 4-7 to zero, bits 0-3 are A/D bits 4-7
          .....     .....           ;convert value in W to 7-segment hex digit
          .....     .....           ;display hex digit for A/D bits 4-7
          .....     .....           ;place upper 2 bits of 10-bit A/D value in W
          .....     .....           ;convert value in W to 7-segment hex digit
          .....     .....           ;display hex digit for A/D bits 8-9
          .....     .....           ;loop code
```

8.6 Utility subroutines and data output

There are several pre-loaded subroutines available for use as part of your programs. Click on the help menu '?', and browse the 'Routines' subdirectories. You should become familiar with these routines; they are bug-free and will make your programming task much easier.

Several of these routines make the output and conversion of data a simple matter. For example, the `Bin2BCD` routine takes the 16-bit binary value stored in the file registers `WH` and `WL` and converts it to a five-digit signed or unsigned decimal value stored in registers `Dec0-Dec4`. The `BCD2LED` routine converts and outputs this result to the 7-segment display. Alternately, the contents of `Dec0-Dec4` can be sent to the 'PICLab output' window in `picl` as an ASCII [3] string by calling the `BCD2TCL` routine, or to the LCD display by calling the `BCD2LCD` routine. The PICLab LCD display uses the ASCII character set. These routines require parameters to be set prior to execution.

To send to `picl` a single ASCII character stored in `W`, use the `TxByte` routine. To send a value in `W` in the range of 0-9 as the corresponding ASCII decimal character, use the `TxDigit` routine. The `Hex2TCL` routine outputs the value in `W` as a 2-character hexadecimal string, while the `Dec2TCL` routine outputs the value in `W` in the valid range of 0-99 as a 2-character decimal string.

The characters sent to `picl` are stored in a buffer until a newline character ASCII=0x0A, is received. This is handy when several columns of data need to be sent; they will be displayed on the same line until terminated by a newline character. To separate your data values with a space, send the 'space' character, ASCII=0x20.

The following code segment outputs the value in `WH:WL` as a decimal string to the PICLab output window:

```
call      Bin2BCD   ;convert 16-bit value in WH:WL to decimal string
movlw     6         ;set field width for BCD2TCL to 6 characters
call      BCD2TCL   ;send string to TCL buffer
movlw     '\n       ;load W register with ASCII newline character
call      TxByte    ;send character, flush buffer to display contents
```

For the following exercises, begin by sketching a flowchart of the logical steps required to perform the given task, then convert each step to one or more PIC instructions that will define your program. Be sure to thoroughly document your code. Test your code initially on the PIC simulator, then execute your program on PICLab:

1. write a program that reads the keypad channel and displays the result as a decimal value 0-1023 on the 7-segment LED display. The value should change as the various keypad switches are pressed;

2. write a program that outputs three pairs of coordinate points (1,2), (2,4), (3,8) to the PICLab output window. The data should appear as an array of three rows by two columns. Click the **Graph** button to generate a Gnuplot graph of your data. Check the Lines box to interpolate the data with line segments;

3. write programs that implement in software the summing and shift/add algorithms used to multiply two 4-bit numbers that were implemented in hardware as part of Experiment 6. Begin by reviewing the arithmetic instructions available to the PIC and their effect on the carry `C` and zero `Z` flags contained in the `STATUS` register.

[3] ASCII refers to the American Standard Code for Information Interchange, where an 8-bit value is used to represent the character set of alphanumeric characters a-z, A-Z, 0-9 as well as other symbols typically found on a keyboard, such as ?, +, !. Coded in the range of ASCII=0-0x1F, are non-printed control characters.