Embedded C Language Development Kit

For the PICmicro® MCU

EXERCISE BOOK

Custom Computer Services, Inc.
Brookfield, Wisconsin, USA
**262-797-0455**

1 Unpacking and Installation

Inventory

' Use of this kit requires a PC with Windows 95,98,ME,NT,2000 or XP. The PC must have a spare 9 pin serial port, a CD ROM drive and 5 meg of disk space.
' The diagram on the following page show each component in this kit. Make sure all items are present.

Software

' Insert the CD into the computer and wait for the install program to start. If your computer is not set up to autorun CDs then select **START > RUN** and enter: **D:\SETUP1.EXE** where D: is the drive letter for your CD drive.
' Click on **Install** and use the default settings for all subsequent prompts. Click NEXT, OK, CONTINUE... as required.
' Identify a directory to be used for the programs in this booklet. The install program will have created an empty directory **c:\program files\picc\projects** that may be used for this purpose.
' Select the compiler icon on the desktop. In the PCW IDE click **Help > About** and verify a version number is shown for the IDE and/or PCM. This ensures the software is installed properly. Exit the software.

Hardware

' Connect the PC to the ICD-S (6) using the 9 pin cable (8) and connect the prototype board (10) to the ICD-S using the modular cable (5). Plug in the AC adaptor (9) and plug it into the prototype board (10). See the diagram in section 3 for help.
' The LED on the ICD-S should be on.
' Run the program at **Start > Programs > PIC-C > ICD-S**. If the program reports a communication error, select a different COM port until you find the port connected to the ICD-S.
' Select **Test Port**, then **Test ICD-S**, then **Test Target**. If all tests pass, then the hardware is installed properly.
' Disconnect the hardware until you are ready for Exercise 3. Always disconnect the power to the prototype board before connecting / disconnecting the ICD-S or changing the jumper wires to the prototype board.
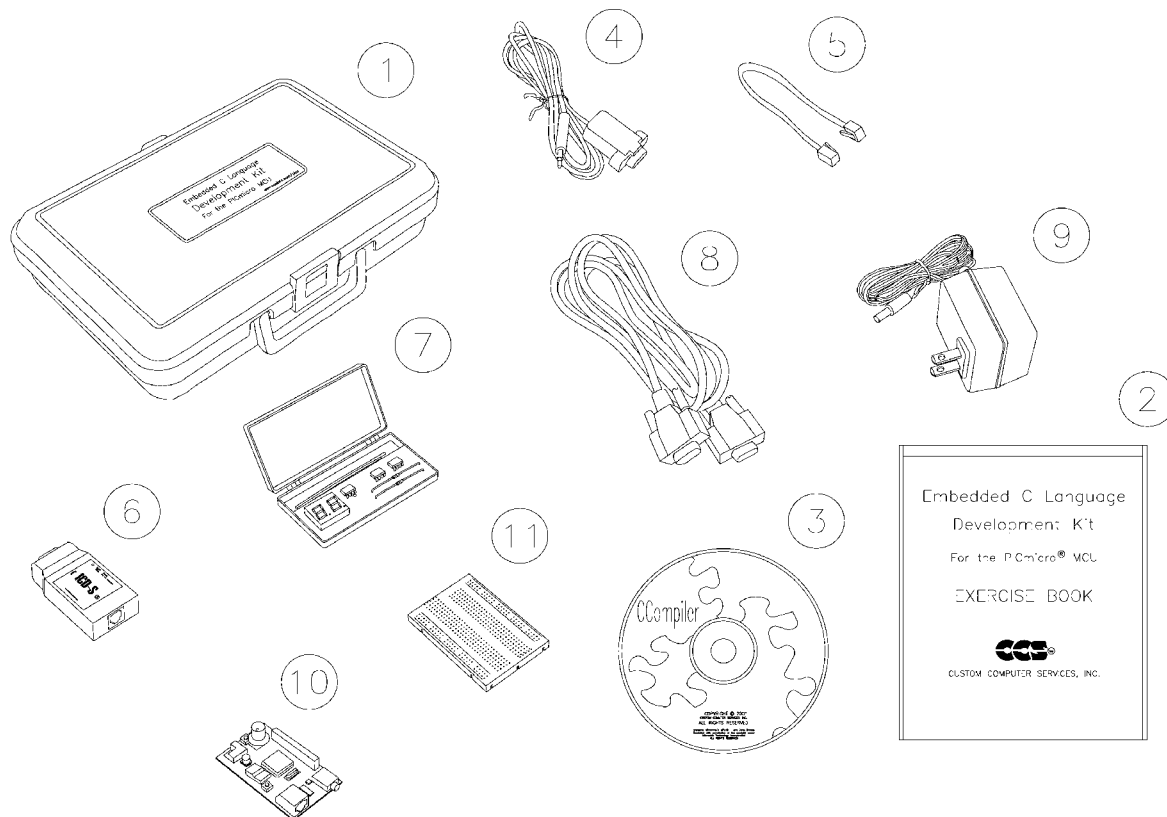
(1) Carrying case.

(2) This exercise booklet.

(3) CD with software including the C compiler.

(4) Serial PC to Prototype board cable

(5) Modular cable (ICD-S to Prototyping board)

(6) ICD-S unit allows programming and debugging Flash parts from a PC.

(7) Parts box with:

  93LC56 serial EEPROM chip
  DS1631 digital thermometer chip
  NJU6355 real time clock chip with attached 32 kHz crystal
  Two digit 7 segment LED module
  Two 1K resistors
  Jumpers to connect the prototyping board to the breadboard

(8) Serial cable, 9 pin Male to Female. (ICD-S to PC)

(9) AC Adaptor (12VDC) to power the prototyping board.

(10) Prototyping board with a PIC16F877 processor chip.
  See inside cover for details on the board layout and schematic.

(11) Breadboard for prototyping circuits.

2 Using the Integrated Development Environment (IDE)

Editor

' Open the PCW IDE. If any files are open, click **File > Close All**

' Click **File > Open**. Select the file: **c:\program files\picc\examples\ex_stwt.c**

' Scroll down to the bottom of this file. Notice the editor shows comments, pre-processor directives and C keywords in different colors.

' Move the cursor over the **Set_timer0** and click. Press the F1 key. Notice a help file description for set_timer0 appears. The cursor may be placed on any keyword or built-in function and F1 will find help for the item.

' Move the cursor to **do** and press the right arrow key until the cursor reaches {. Notice the editor highlights the matching }. This is also done for (and) to ensure the open/close brackets are right.

' Review the editor special functions by clicking on **Edit.** The IDE allows various standard cut, paste and copy functions along with setting bookmarks and various C specific functions.

' Review the editor option settings by clicking on **Options.** The IDE allows selection of the tab size, editor colors, font and icons that should appear on the toolbar.

Compiler

' Use the white box on the toolbar to select the compiler. CCS offers different compilers for each family of Microchip parts. All the exercise in this booklet are for the PIC16F877 chip, a 14-bit opcode part. Make sure **14 bit** is selected in the white box.

' The main program compiled is always shown in the lower right corner of the IDE. If this is not the file you want to compile, then click on the tab of the file you want to compile. Right click to editor and select **Make file project**.

' Click **Options > Include Dirs...** and review the list of directories the compiler uses to search for include files. The install program should have put two directories in this list to point to the device .h files and the device drivers.

' Normally the file formats need not be changed and global defines are not used in these exercises. To review these settings, click **Options > File Formats** and **Options > Global Defines**.

' Click **Compile > Compile** or the compile icon to compile. Notice the compilation box shows the files created and the amount of ROM and RAM used by this program. Press any key to remove the compilation box.
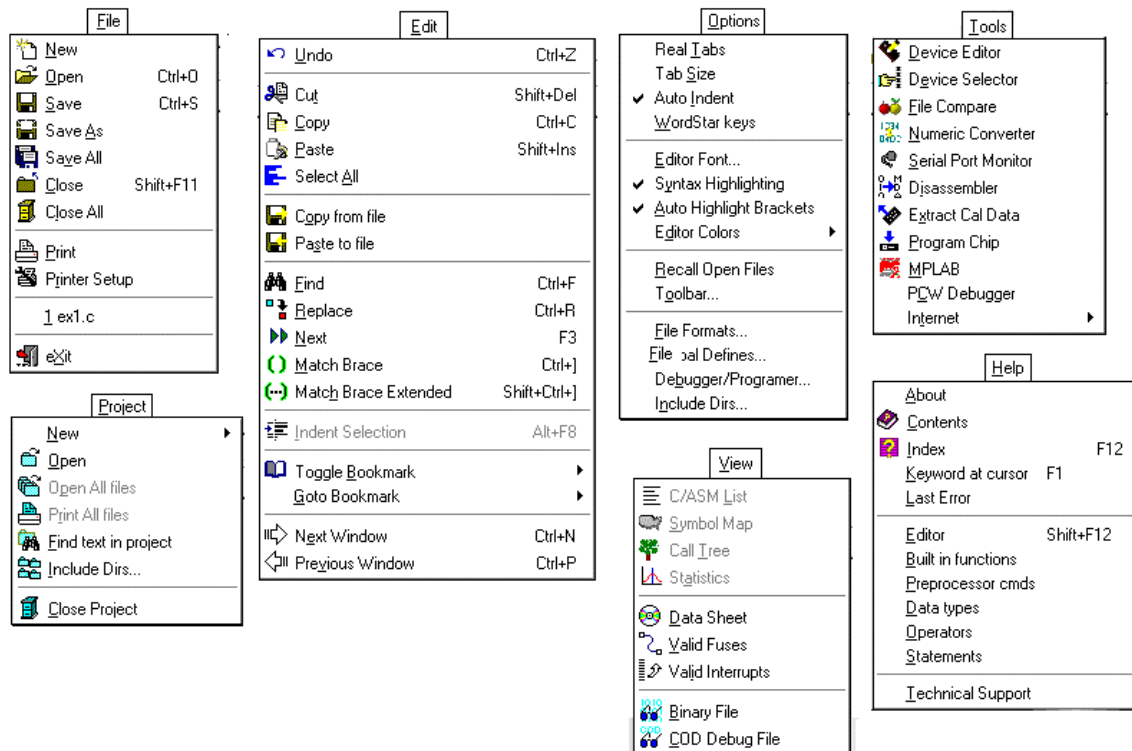
Viewers

'   Click **View > Symbol Map**. This file shows how the RAM in the microcontroller is used. Identifiers that start with @ are compiler generated variables. Notice some locations are used by more than one item. This is because those variables are not active at the same time.

'   Click **View > C/ASM list**. This file shows the original C code and the assembly code generated for the C. Scroll down to the line:
        int_count=INTS_PER_SECOND;
Notice there are two assembly instructions generated. The first loads 4C into the W register.   INTS_PER_SECOND is #defined in the file to 76. 4C hex is 76 decimal. The second instruction moves W into a memory location. Switch to the Symbol Map to find the memory location is where int_count is located.

'   Click **View > Data Sheet** , then OK. This brings up the Microchip data sheet for the microprocessor being used in the current project.

File	
New	
Open	Ctrl+O
Save	Ctrl+S
Save As	
Save All	
Close	Shift+F11
Close All	
Print	
Printer Setup	
1 ex1.c	
eXit	

Project	
New	▶
Open	
Open All files	
Print All files	
Find text in project	
Include Dirs...	
Close Project	

Edit	
Undo	Ctrl+Z
Cut	Shift+Del
Copy	Ctrl+C
Paste	Shift+Ins
Select All	
Copy from file	
Paste to file	
Find	Ctrl+F
Replace	Ctrl+R
Next	F3
( ) Match Brace	Ctrl+]
(∙∙) Match Brace Extended	Shift+Ctrl+]
Indent Selection	Alt+F8
Toggle Bookmark	▶
Goto Bookmark	▶
Next Window	Ctrl+N
Previous Window	Ctrl+P

Options	
Real Tabs	
Tab Size	
✔ Auto Indent	
WordStar keys	
Editor Font...	
✔ Syntax Highlighting	
✔ Auto Highlight Brackets	
Editor Colors	▶
Recall Open Files	
Toolbar...	
File Formats...	
File al Defines...	
Debugger/Programer...	
Include Dirs...	

View
C/ASM List
Symbol Map
Call Tree
Statistics
Data Sheet
Valid Fuses
Valid Interrupts
Binary File
COD Debug File

Tools	
Device Editor	
Device Selector	
File Compare	
Numeric Converter	
Serial Port Monitor	
Disassembler	
Extract Cal Data	
Program Chip	
MPLAB	
PCW Debugger	
Internet	▶

Help	
About	
Contents	
Index	F12
Keyword at cursor	F1
Last Error	
Editor	Shift+F12
Built in functions	
Preprocessor cmds	
Data types	
Operators	
Statements	
Technical Support	

③ Compiling and Running a Program

' Open the PCW IDE. If any files are open, click **File > Close All**
' Click **File > New** and enter the filename **EX3.C**
' Type in the following program and **Compile**

```
#include <16f877.h>
#device ICD=TRUE
#fuses  HS,NOLVP,NOWDT,PUT
#use delay(clock=20000000)

#define GREEN_LED PIN_A5

main() {

    while(TRUE) {
        output_low(GREEN_LED);
        delay_ms(1000);
        output_high(GREEN_LED);
        delay_ms(1000);
    }
}
```

L   Notes:

- The first four lines of this program define the basic hardware environment. The chip being used is the PIC16F877, running at 20 Mhz with the ICD debugger.

- The #define is used to enhance readability by referring to GREEN_LED in the program instead of PIN_A5.

- The "while (TRUE)" is a simple way to create a loop that never stops.

- Note that the "output_low" turns the LED on because the other end of the LED is +5V. This is done because the chip can tolerate more current when a pin is low than when it is high.

- The "delay_ms(1000)" is a one second delay (1000 milliseconds).
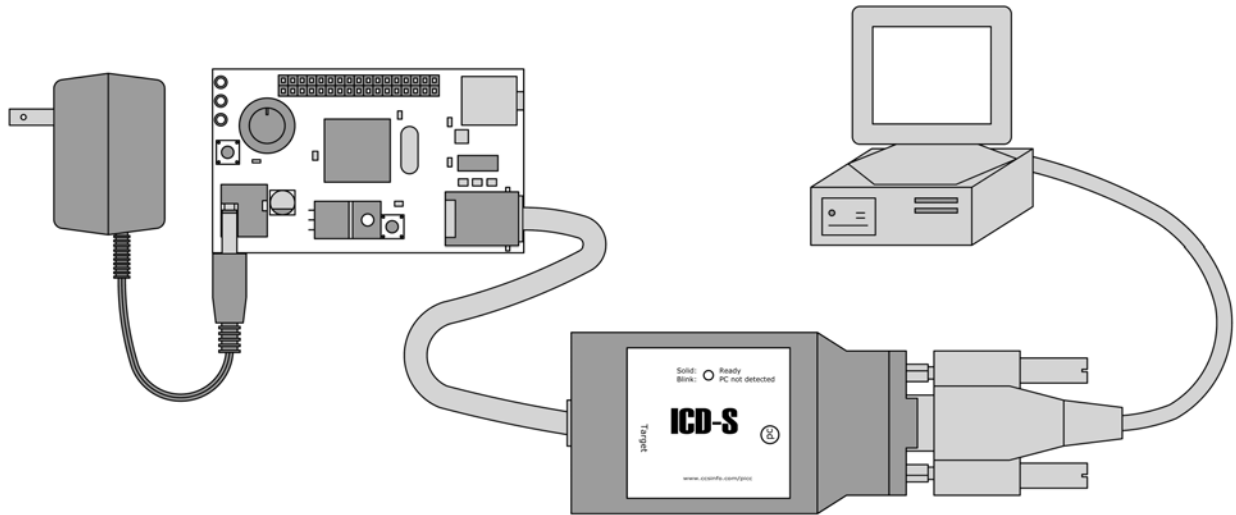
' Connect the ICD to the PC and the proto-board to the ICD. Power up the proto-board and verify the LED on the ICD is flashing.

' Click **Debugger > Enable** and wait for the program to load.

' Click the green go icon: GO

Expect the debugger window status block to turn yellow indicating the program is running.

The green LED on the proto board should be flashing. One second on and one second off.

The program can be stopped by clicking on the stop icon: S

Further Study

A Modify the program to light the green LED for 5 seconds, then the yellow for 1 second and the red for 5 seconds.

B Add to the program a #define macro called "delay_seconds" so the delay_ms(1000) can be replaced with: delay_seconds(1); and delay_ms(5000) can be: delay_seconds(5);.

Note: Name these new programs EX3A.c and EX3B.c and follow the same naming convention throughout this booklet.

4 Handling Input

' Type in the following program, named **EX4.C**, **Compile** and **Run:**

```
#include <16f877.h>
#device ICD=TRUE
#fuses  HS,NOLVP,NOWDT,PUT
#use delay(clock=20000000)

#define GREEN_LED PIN_A5
#define YELLOW_LED PIN_B4
#define RED_LED PIN_B5
#define PUSH_BUTTON PIN_A4

light_one_led(int led) {
    output_high(GREEN_LED);
    output_high(YELLOW_LED);
    output_high(RED_LED);
    switch(led) {
     case 1 : output_low(GREEN_LED);  break;
     case 2 : output_low(YELLOW_LED); break;
     case 3 : output_low(RED_LED);    break;
    }
}

wait_for_one_press() {
    while(input(PUSH_BUTTON)) ;
    while(!input(PUSH_BUTTON)) ;
}

main() {
    while(TRUE) {
        light_one_led(1);
        wait_for_one_press();
        light_one_led(2);
        wait_for_one_press();
        light_one_led(3);
        wait_for_one_press();
    }
}
```

' As can be seen from the program, the green LED should come on. Press the button and the yellow LED should light and then the red LED when pressed again.

L   Notes:

- The prototyping board has one momentary pushbutton that may be used as an input to the program. The input pin is connected to a 10K pull-up resistor to +5V. The button, when pressed, shorts the input pin to ground. The pin is normally high while in this configuration, but it is low while the button is pressed.

- This program shows how to use simple C functions. The function wait_for_one_press() will first get stuck in a loop while the input pin is high (not pressed). It then waits in another loop while the pin is low. The function returns as soon as the pin goes high again. Note that the loops, since they do not do anything while waiting, do not look like much– they are a simple ; (do nothing).

- When the button is pressed once, is it common for several very quick connect/disconnect cycles to occur. This can cause the LEDs to advance more than once for each press. A simple debounce algorithm can fix the problem. Add the following line between the two while loops: **delay_ms(100);** The following scope picture of a button press depicts the problem:
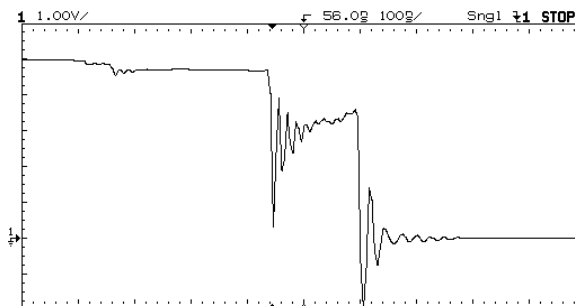
Further Study

A Add the following new type:

```
typedef enum {GREEN,YELLOW,RED} colors;
```

Then change the parameter to light_one_led to colors instead of int. Then change the 1,2,3 in the call to GREEN,YELLOW,RED.

B Modify the program so that while the button is held down the LEDs alternate as fast as possible. When the button is not pressed the LED state freezes. This creates a random color program.
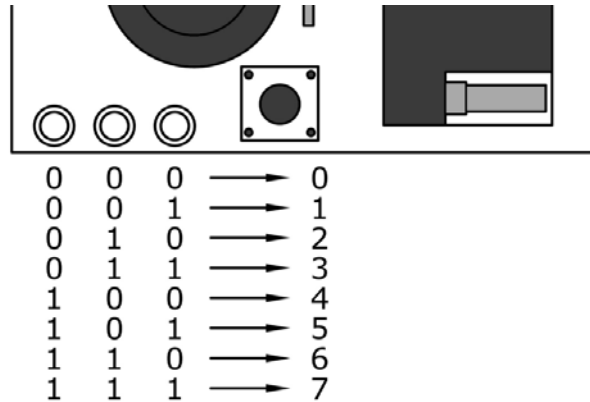
5️⃣ Program Structure

' It is a good practice to put all the hardware definitions for a given design into a common file that can be reused by all programs for that board. Open EX4.C and drag the cursor over (highlight) the first 9 lines of the file. **Click Edit > Paste to file** and give it the name **prototype.h**.

' It is also helpful to collect a library of utility functions to use as needed for future programs. Note that just because a function is part of a program does not mean it takes up memory. The compiler deletes functions that are not used. Highlight the wait_for_one_press() function and the typedef line (if you added it) and save that to a file named **utility.c**. Open utility.c and add the following new function to the file:

```
show_binary_on_leds(int n) {
    output_high(GREEN_LED);
    output_high(YELLOW_LED);
    output_high(RED_LED);
    if( bit_test(n,0) )
        output_low(GREEN_LED);
    if( bit_test(n,1) )
        output_low(YELLOW_LED);
    if( bit_test(n,2) )
        output_low(RED_LED);
}
```

' Close all files and start a new file EX5.C as follows:

```
#include <prototype.h>
#include <utility.c>

main() {
    int count = 0;

    while(TRUE) {
        show_binary_on_leds(count);
        wait_for_one_press();
        count++;
    }
}
```

' Compile and Run the program. Check that with each button press the LEDs increment a binary number 0-7 as shown in the diagram on the next page.

```
0  0  0  ────▶  0
0  0  1  ────▶  1
0  1  0  ────▶  2
0  1  1  ────▶  3
1  0  0  ────▶  4
1  0  1  ────▶  5
1  1  0  ────▶  6
1  1  1  ────▶  7
```

L Notes:

- In C, a function must either appear in the input stream before it is used OR it must have a prototype. A prototype is the part of the function definition before the {. In a program where main calls function A and function A calls function B the order in the file must be B, A, MAIN. As an alternative, have Ap, Bp, MAIN, A, B where Ap and Bp are prototypes. Frequently, prototypes are put in a header file with a .h extension.

- The scope, initialization and life of C variables depend on where and how they are declared. The following is a non-inclusive summary of the common variable scopes. Note that if a variable has an initializer (like **int a=1;**) the assignment happens each time the variable comes to life.

Where it is defined:	Can be access from:	Life of the variable:
Inside a function	Only in that function	While function is active
Inside a function with STATIC	Only in that function	During the entire run of the program
Outside all functions	In any function defined afterwards in the file	During the entire run of the program
After a { inside a function	Only between the { and corresponding }	Only up to the corresponding }

Further Study

A Modify the program to increment the binary number by 1 every second (the button is not used).

B Instead of the built-in function BIT_TEST use the standard C operators (such as & and ==) to test the bits.

❻ Debugging

' Open EX5.C and start the debugger **Debugger > Enable**.

' Click the reset icon: 🔘 to ensure the target is ready.

' Click the ⬚ icon twice. This is the step over command. Each click causes a line of C code to be executed. The highlighted line has not been executed, but the line about to be executed.

' Step over the `Show_binary_on_leds(count);` line and notice that one click executed the entire function. This is the way step over works. Click again and notice the debugger does not stop since it is now stuck in `wait_for_one_press();`. Press the prototype button and notice the debugger now stops since the function terminates.

' Click the **Watch** tab, then the ➕ icon to add a watch. Enter **count,** then click OK. Notice the value shown. Continue to step over through the loop a few more times (press the button as required) and notice the count watch increments.

' Step over until the call to `show_binary_on_leds(count);` is highlighted. This time, instead of step over, use the standard step icon: 👣 several times and notice the debugger is now stepping into the function.

' Click the GO icon to allow the program to run. Press the prototype button a couple of times to verify that the program is running normally. Click the stop icon Ⓢ to halt execution. Notice the C source line that the program stopped on. This is the line where the program is waiting for a button press.

' In the editor, click on `show_binary_on_leds(count);` to move the editor cursor to that line. Then click the Breaks tab and click the ➕ icon to set a breakpoint. The debugger will now stop every time that line is reached in the code. Click the GO icon and then press the prototype button. The debugger should now stop on the breakpoint. Repeat this a couple of times to see how the breakpoint works. Note that the ICD-S with PIC16 chips only allow one breakpoint at a time.

' Click **View > C/ASM list**. Scroll down to the highlighted line. Notice that one assembly instruction was already executed for the next line. This is another side effect of the ICD-S debugger. Sometimes breakpoints slip by one ASM instruction.

' Click the step over icon a few times and note that when the list file is the selected window, the debugger has executed one assembly instruction per click instead of one entire C line.

' Close all files and start a new file **EX6.C** as follows:

```
#include <prototype.h>
#include <utility.c>

main() {
    int a,b,c;

    a=11;
    b=5;
    c=a+b;
    c=b-a;
    while(TRUE);
}
```

' Compile the program and step over until the c=a+b is executed. Add a watch for c and the expected value is 16.

' Step over the subtraction and notice the value of c. The int data type by default is not signed, so c cannot be the expected -6. The modular arithmetic works like a car odometer when the car is in reverse only in binary. For example, 00000001 minus 1 is 00000000, subtract another 1 and you get 11111111.

' Reset and again step up to the c=a+b. Click the **Eval** tab. This pane allows a one time expression evaluation. Type in a+b and click **Eval** to see the debugger calculate the result. The complete expression may also be put in the watch pane as well. Now enter b=10 and click **Eval**. This expression will actually change the value of B. Step over the addition line and click the **Watch** tab to observe the c value was calculated with the new value of b.

Further Study

A Modify the program to include the following C operators to see how they work: *   /   %   & | ^
Then, with b=2 try these operators: >>   <<
Finally, try the unary complement operator with: c=~a;

B Design a program to test the results of the relational operators:
   <   >   ==   !=
by exercising them with b as 10,11 and 12.
Then, try the logical operators || and && with the four combinations of a=0,1 and b=0,1.
Finally, try the unary not operator with: c=!a; when a is 0 and 1.

Analog to Digital Conversion

' The PIC16F877 chip has 8 pins that may be used to read an analog voltage. These 8 pins can be configured to certain combinations of analog input and digital pins, but not all combinations are possible. The following is a simple program (**EX7.c**) to read one analog pin.

```c
#include <prototype.h>
#include <utility.c>

#define cutoff 128        // 2.5 Volts
#define neutral_zone 25   // 0.5 Volts

main() {
    int reading;

    setup_adc_ports( RA0_ANALOG );
    setup_adc( ADC_CLOCK_INTERNAL );
    set_adc_channel( 0 );

    while(TRUE) {
        reading = read_adc();
        if(reading<(cutoff-neutral_zone/2))
          light_one_led(GREEN);
        else if (reading>(cutoff+neutral_zone/2))
          light_one_led(RED);
        else
          light_one_led(YELLOW);
    }
}
```
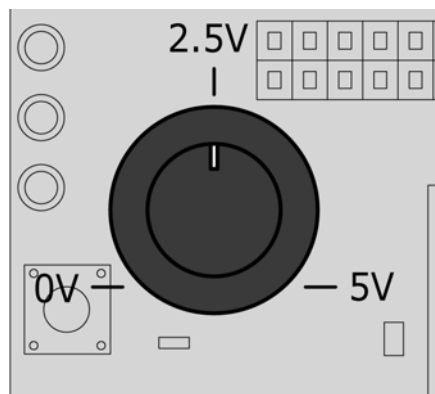
' Compile and Run the program. Verify that the prototype knob (A0) is turned so the green LED is on when it is low, the red LED when high and the yellow LED for a small region in the center.

L   Notes:

- By default, the analog to digital converter is 8 bits. Thus, a range of 0 to 5 volts analog is represented by the numbers 0-255. The A/D reading can be converted to volts by the formula:

    Volts = reading * (5.0/255)

- The set_adc_ports function call determine what pins are set to be analog inputs. The setup_adc function call determines how fast the conversion is done. The internal clock option uses an internal RC clock. Although the timing is not exact, it is long enough for a accurate conversion. The time can be based off the instruction clock for more precise timing.

- The set_adc_channel function sets the A/D converter to channel 0 (AN0 or A0). This switches an internal mux in the part but does not start an A/D conversion. Even though a conversion has not started, there is a small capacitor in the chip that must charge up after the port switch and before the voltage is read. This is fast with a low impedance input, but for a higher impedance input, a small delay should be put in after the channel is changed.

- The call to read_adc starts a conversion, waits for it to complete and returns the result. The conversion time is around 20 us.

Further Study

A Modify the program to use a long variable for reading. Add this line after the include for protoype.h:

    #device   ADC=16

This will set the range to 0-65535, however, since this part only has a 10 bit A/D converter the actual range is 0-65472. Change the constants in the program to reflect the new range.

B Write a timer program that will light the green LED for x seconds when pressing the button. x should be 0-25 depending on the setting of the analog knob.

8 Arrays and Analog Filtering

' The following EX8.c program illustrates how to change the EX7.c program such that the value used to light the LED is the average voltage over the previous 10 seconds.

```c
#include <prototype.h>
#include <utility.c>

#define cutoff 128        // 2.5 Volts
#define neutral_zone 25   // 0.5 Volts

main() {
    int history[10],i;
    int history_ptr = 0;
    long reading;
    int count=0;

    setup_adc_ports( RA0_ANALOG );
    setup_adc( ADC_CLOCK_INTERNAL );
    set_adc_channel( 0 );

    while(TRUE) {
        delay_ms(1000);
        history[history_ptr++] = read_adc();
        if(history_ptr==10) {
            history_ptr=0;
            count = 10;
        } else
            if(count<10)
              count=history_ptr;
        for(i=0;i<count;i++)
          reading += history[i];
        reading /= count;
        if(reading<(cutoff-neutral_zone/2))
          light_one_led(GREEN);
        else if (reading>(cutoff+neutral_zone/2))
          light_one_led(RED);
        else
          light_one_led(YELLOW);
    }
}
```

’ Run the new program and confirm movement of the knob takes 10 seconds to appear on the LEDs. Furthermore, confirm that a quick movement of the knob from high to low makes no difference in the LEDs.

L   Notes:

- This program uses several of the C shortcut operators. For example the `reading += history[i]` statement is the same as
  ```
  reading = reading + history[i]
  ```
  and `history[history_ptr++] = read_adc();` is the same as
  ```
  history[history_ptr] = read_adc();
  history_ptr = history_ptr+1;
  ```

- A C array declared history[10] means the valid subscripts are history[0] through history[9].

- The reading variable needs to be a long (16 bits) because the largest value 255*10 is larger than a 8 bit int..

- The history variable can be placed in the watch list and then when the program is halted, the debugger will show all the points in history being used to make up the filtered reading.

Further Study

A Modify the program to keep all LEDs off until 10 samples are obtained.

B Modify the program to handle the LEDs differently on even and odd cycles as follows:
   Even: Show the actual last reading on the LED (not filtered)
   Odd: If the last reading is the same as the filtered reading, show this on the LEDs. Otherwise, turn all LEDs off.
   The LED flashes after a change, and when the reading is stable, the LED will be solid.

9 Stand-alone Programs and EEPROM

′ Execution of the EX5.c program always begins counting at 0. This can be modified by creating **EX9.c** that continues counting where it left off when restarted. This is done by saving the count value in the PIC16F877 internal data EEPROM. This memory retains the data even when the power is removed.

′ Create the file **EX9.c** as follows:

```
#include <prototype.h>
#include <utility.c>

main() {
    int count;

    count = read_eeprom(0);
    while(TRUE) {
        show_binary_on_leds(count);
        wait_for_one_press();
        count++;
        write_eeprom(0,count);
    }
}
```

′ Compile and Run the program. Verify when the program is halted, reset and restarted that the count continues where left off.

L   Notes:

● The first argument to read/write_eeprom is the address in the EEPROM to write the byte to. The PIC16F877 part ranges from 0 to 255 and allowing 256 bytes to be saved.

● There is a limit as to how many times a given location in the data EEPROM can be written to. For example, the PIC16F877 chip allows 100,000 times and the A version of this chip may allow 10 times that amount. For this reason, a program should be designed not to write any more often than is necessary. For example, if the volume setting for a TV is being saved, one might wait until there are no changes for 5 seconds before saving a new value to EEPROM. Some system designs can give an early warning on power down and the program can only save to EEPROM at power down.

' Copy the prototype.h file to a new file **protoalone.h**. Remove from this new file the line: #device ICD=TRUE
This makes a program that uses the new include file a stand alone program which does not need the ICD to run.

' Modify EX9.c to use **protoalone.h**. Compile the program and make sure the debug window is opened so the file gets loaded to the prototype.

' Disconnect the power from the proto board, then disconnect the ICD from the proto board.

' Power up just the proto board and verify the program runs correctly.

' Press the reset button on the proto board and release. The LEDs should go off while in reset then the program will restart.
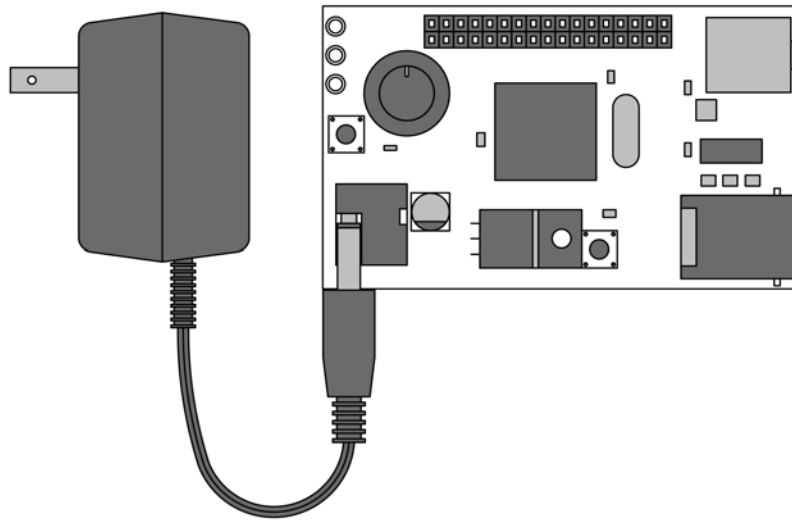
Further Study

A Modify EX7.c so the cut-off point is a variable and that variable is kept in EEPROM location 100. Establish a new cut-off point whenever the pushbutton is pressed to wherever the knob is set to. Be careful to only write the EEPROM once per press.

B Modify the EX9.c program so that 10 EEPROM locations are used and each time the button is pressed only one of the 10 locations is written to and the location changes with each press. This will extend the life of this unit by 10 times if it were a real product.
Hint: The count value could be the sum of all 10 locations % 8.

10 Using an RS-232 Port

' RS-232 is a popular serial communications standard used on most PCs and many embedded systems. Two wires are used (in addition to ground), one for outgoing data and one for incoming data. The PIC16F877 chip has built-in hardware to buffer the serial data if pins C6 and C7 are used. The compiler will allow any pins to be used and will take advantage of the built-in hardware if you pick those pins. Add the following line to the end of the **protoalone.c** file:

```
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)
```

' Create the file **EX10.c** as follows:

```c
#include <protoalone.h>
#include <utility.c>
#include <input.c>

main() {
   long a,b,result;
   char opr;

   setup_timer_0(RTCC_INTERNAL);
   while(TRUE) {
     printf("\r\nEnter the first number: ");
     a=get_long();

     do {
        printf("\r\nEnter the operator (+-*/): ");
        opr=getc();
     } while(!isamoung(opr,"+-*/"));

     printf("\r\nEnter the second number: ");
     b=get_long();

     switch(opr) {
      case '+' : result= a+b;    break;
      case '-' : result= a-b;    break;
      case '*' : result= a*b;    break;
      case '/' : result= a/b;    break;
     }

     printf("\r\nThe result is %lu ",result);
   }
}
```
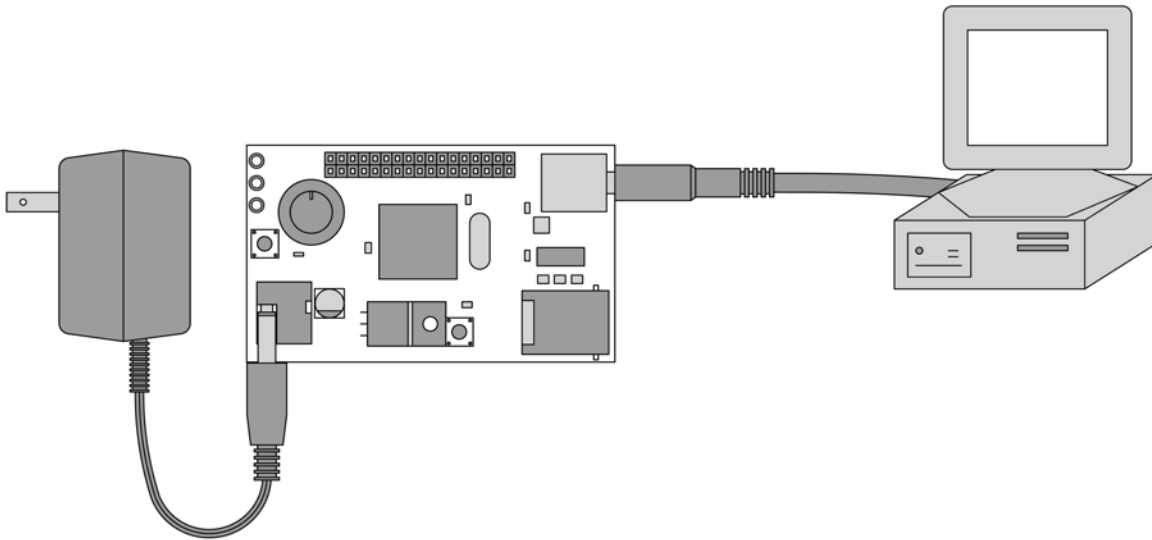
' Compile and load the program into the proto board.

' Connect the proto board to the PC as shown on the following page.
' At the PC, close the debugger window and start the program **Tools > Serial port monitor.** Set the correct COM port if necessary.
' Power up the proto board and a prompt at the PC should appear. Enter a number followed by the enter key, a operator (like +) and another number followed by enter. Verify the result is shown correctly.

L   Notes:

- The basic functions for RS-232 are puc() and getc(). printf calls putc() a bunch of times to output a whole string and format numbers if requested. get_long() is a function in input.c to read a long number by calling getc() a bunch of times. See input.c for other functions such as get_int() and get_string().

- The % in the printf indicates another parameter is included in the printf call and it should be formatted as requested. %lu indicates to format as an unsigned long.

- getc() will cause the program to stop and wait for a character to come in before it returns.

.

Further Study

A Modify to add the operators: % | & ^

B Modify to use float instead of long. You will need to do get_float() instead of get_long() and use the format specifier %9.4f to get 4 digits after the decimal place.

11 More RS-232 and Using Structures

' Create the file **EX11.c** as follows:

```c
#include <protoalone.h>
#include <utility.c>

struct animal { char code;
                char name[8];
                int count; };

#define MAX 3

struct animal animals[MAX] = {{'A',"Ant",0},
                              {'B',"Bird",0},
                              {'C',"Cat",0}};

int find(char code, int & index) {
    for(index=0;index<MAX;index++)
      if(animals[index].code==code)
        return(TRUE);
    return(FALSE);
}

main() {
    int index;
    char code;

    while(TRUE) {
      printf("\r\nEnter the code: ");
      code=getc();
      if(find(code,index)) {
       animals[index].count++;
        printf("\r\nYou now have %u %s's\r\n",
               animals[index].count,
               animals[index].name);
      } else
        printf("\r\nCode not valid\r\n");
    }
}
```
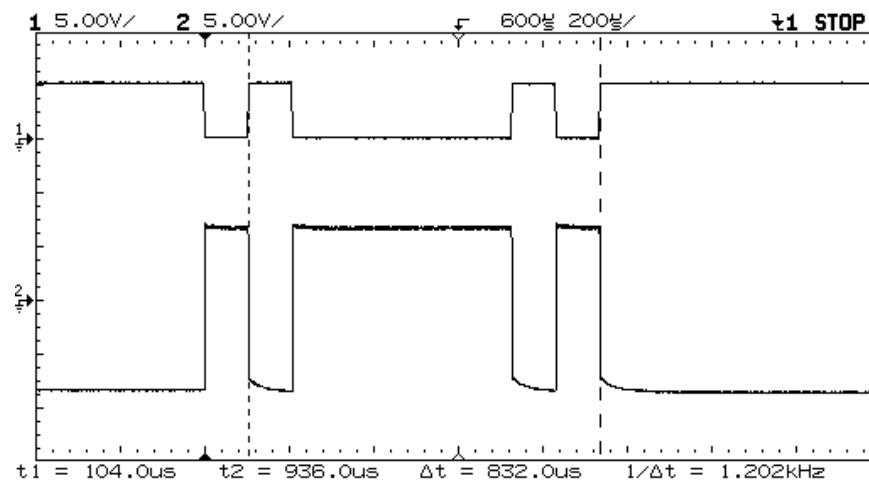
' Compile and load the program into the proto board.
' Power up the board and use the serial port monitor to test.

L   Notes:

- The `int & index` is our first example with an output parameter to a function. The & indicates the value is returned to the caller (actually the callers copy of the variable is used).
- At a hardware level, RS-232 sends a series of bits. The baud= option specifies how many bits are sent per second. The bit stream, as specified above, is a start bit (always 0), 8 data bits (lsb first) and a stop bit (always 1). The line then remains at the 1 level. The number of bits may be changed with a bits= option and a parity bit can be added before the stop bit with a parity= option. A 0 is represented as a positive voltage (+3 to +12V) and a 1 is represented as a negative voltage (-3 to -12V). Since the PIC16F877 outputs only 0V and 5V a level converter is required to interface to standard RS-232 devices such as a PC. A popular chip that does this is the MAX232. See the schematic in the back cover for details. The following diagram shows a single character A (01000001) as sent at 9600 baud. The top is from the PIC16F877, the bottom is from the MAX232, the 8 data bits are between the dotted lines. Each bit is 104us.


```
 1  5.00V/    2  5.00V/              600ǵ 200ǵ/           ↑1  STOP


 t1 = 104.0us    t2 = 936.0us    Δt = 832.0us    1/Δt = 1.202kHz
```

Further Study

A Modify the program to keep the counts in a separate (from the structure) array. Then add the CONST keyword before animals to put this data in ROM instead of RAM. Compare the memory usage for both programs.

B If there is room, modify to add new entries when the code is not found. Set MAX to 6 and add a new variable to keep track of the number of entries.

12 Advanced Debugging

'   RS-232 printf statements can be a good tool to help debug a program. It does, however, require an extra hardware setup to use. If the ICD is being used as a debug tool, the compiler can direct putc() and getc() through the debugger interface to the debugger screen. Add the following line to the end of the **prototype.h** file:

```
#use rs232(DEBUGGER)
```

'   Modify EX10.c to create **EX12-1.c** by changing protoalone.h to **prototype.h**.
'   Compile and load the program into the proto board.
'   Click GO, then click the **Monitor** tab.
'   A prompt should appear. Enter some data to confirm that the program is working.

'   Stop and reset the program.
'   In PCW click **Project > Open all files** as an easy way to get all the project files open in the IDE.
'   Click the **stdlib.h** tab, and set a breakpoint in the atol() function on the line:
     result = 10*result + © - '0');
    This function is called from get_long() to convert a string to a number. This line is executed for each character in the string.
'   Click the debugger **Log** tab, check the LOG box, set the breakpoint as 1 and expression as **result**. Result is the value of the number being converted.
'   Click GO, then click the **Monitor** tab and enter **1234 enter**.
'   Click the **Log** tab and notice that each time the breakpoint was hit the value of the **result** variable was logged. In this case the breakpoint did not cause a full stop of the program, it just logged the value of the requested expression and kept on going.

'   Delete the breakpoint by selecting the breakpoint and click on the ▬ icon.
'   Uncheck the LOG box under the log tab.
'   Set a breakpoint on the last printf() in the program.
'   Enter watches for **a**, **b** and **result**.
'   Click GO and enter two numbers and +.
'   When the break is reached click on the snapshot icon: 📷
'   Check **Time** and **Watches**, uncheck everything else.
'   If a printer is connected to the PC select **Printer** otherwise select **Unique file**.
'   Click on the **Now** button.
'   Notice the requested data (time and watches) are either printed or written to a file as requested.

' Click on the snapshot icon again and this time select **Append to file**, put in a filename of EX12.TXT and check **After each single step**.
' Check **Last C line** in addition to the **Time** and **Watch** selected already and close the snapshot window.
' Reset and then Step Over until the final printf() is executed. Enter the data when requested.
' Use File > Open to find the file EX12.TXT (by default in the Debugger Profiles directory) after setting the file type to all files.
' Notice you have a log of what happened with each step over command.

' Uncheck the **After each single step** in the snapshot window.
' Clear the breakpoints and set a breakpoint on the **switch.**
' Click Reset then GO and enter the requested data using the + operator.
' When the break is reached click on the **Peripherals** tab and select Timer 0.
' Shown will be the registers associated with timer 0. Although this program does not use timer 0 the timer is always running so there is a value in the **TMR0** register. Write this value down.
' Clear the breakpoints and set a breakpoint on the final **printf().**
' Click GO.
' Check the TMR0 register again. If the new value is higher than the previous value then subtract the previous value from the current value. Otherwise add 256 to the current value and then subtract the previous value (because the timer flipped over).
' The number we now have is the number of click ticks it took to execute the switch and addition. A clock tick by default is 0.2us. Multiply your number of ticks by 0.2 to find the time in us. Note that the timers (and all peripherals) are frozen as soon as the program stops running.

Further Study

A The debugger **Eval** tab can be used to evaluate a C expression. This includes assignments. Set a break before the switch statement and use the Eval window to change the operator being used. For example type a + but change it to a - before the switch.

B Set a break on the switch statement and when reached change to the C/ASM view and single step through the switch statement. Look up the instructions executed in the PIC16F877 data sheet to see how the switch statement is implemented. This implementation is dependent on the case items being close to each other. Change * to ~ and then see how the implementation changes.

13 Timers

'   The PIC16F877 has three built-in timers. Each timer has a different set of features. The following example will use Timer #1 to measure the time it takes to execute some C code.
'   Create the file **EX13.c** as follows:

```
#include <prototype.h>

main() {
    long time;

    setup_timer_1(T1_INTERNAL | T1_DIV_BY_1);
    set_timer1(0);

    time = get_timer1();
    printf("Time in ticks is %lu\r\n",time);
}
```

'   Compile and Run the program. Check the monitor tab to see the result.
'   This number is the number of timer ticks that it took to set and read the timer. The T1_INTERNAL indicates the instruction clock is the source for the timer. The instruction clock is the oscillator divided by 4, or in our case, 0.2 us. This time represents the overhead of our timer code and may now be used in a more useful example.
'   Modify the program as follows and replace the ??? with the number of ticks determined in the above program.

```
#include <prototype.h>

main() {
    long time;
    long a,b,c;

    setup_timer_1(T1_INTERNAL | T1_DIV_BY_1);
    set_timer1(0);
    a=b*c;
    time = get_timer1();
    time -= ???; // subtract overhead
    printf("Time is %lu microseconds.\r\n",
            (time+2)/5);
}
```

L   Notes:

- Since "time" represents the number of 0.2 microsecond ticks that it takes to do "a=b*c", then time/5 is the number of microseconds it takes to do that one line of C code. Use (time+2)/5 to round instead of truncating.

- All the timers on the PIC16F877 count up and when the maximum value is reached, the timer restarts at 0. The set_timer1(0) resets the timer to 0. Timer 1 is 16 bits and the range is 0 to 65535. This means it will overflow every 13107.2 us. This is the largest time the program will be able to measure.

- If using T1_EXTERNAL instead of INTERNAL, then the timer would increment every time pin C0 cycled. This makes it more of a counter

- If using T1_DIV_BY_2 instead of BY_1, then the timer would increment once for every 2 instruction clocks. This makes the timer tick 0.4 us and the range of the timer is now 26214.4 us.

- The following is a summary of the timers on the PIC16F877 chip:

#0	Input is Instruction Clock or external pin Range is 0-255 Input can be divided by 1,2,4,6,8,16,32,64,128,256 Can generate interrupt on each overflow
#1	Input is Instruction Clock or external pin Range is 0-65535 Input can be divided by 1,2,4,8 Can generate interrupt on each overflow
#2	Input is Instruction Clock only Range can be programmed from 0-1 to 0-255 Input can be divided by 1,4,16 Can generate interrupt on 1-16 overflows

Further Study

A Time the actual time for a delay_us(200) to see how accurate the compiler is

B Make a program to time the addition operator for 8 bit, 16 bit, 32 bit and floating point. Instead of **int** the compiler allows the use of **int8, int16** and **int32** to specify the number of bits in a integer variable.

14 Interrupts

'   An interrupt is a specific event that causes the normal program execution to be suspended wherever it is and an interrupt function is executed. Normal program execution continues when the interrupt function returns. The PIC16F877 has a number of interrupt sources such as a timer overflow, an incoming RS232 character or a change on a pin.

'   In this exercise, the timer 1 overflow interrupt will be used to extend the timer 1 timer from 16 bits to 32 bits by counting the number of times the timer overflows. Create the file **EX14.c** as follows:

```c
#include <prototype.h>

int16 overflow_count;

#int_timer1
timer1_isr() {
    overflow_count++;
}

main() {
    int32 time;

    setup_timer_1(T1_INTERNAL | T1_DIV_BY_1);
    enable_interrupts(int_timer1);
    while(TRUE) {
        enable_interrupts(global);

        while(input(PUSH_BUTTON));// Wait for press
        set_timer1(0);
        overflow_count=0;

        while(!input(PUSH_BUTTON));//Wait for release
        disable_interrupts(global);
        time = get_timer1();
        time = time + ((int32)overflow_count<<16);
        time -= 15; // subtract overhead
        printf("Time is %lu.%06u seconds.\r\n",
               time/5000000, (time/5)%1000000);
    }
}
```

'   Compile and Run the program. Press the button, release and note the time it was held down is shown to 6 decimal places in the Monitor pane.

L   Notes:

- The interrupt function is designated by preceding it with #INT_TIMER1. A number of interrupt functions can be specified by preceding each with the proper directive like #INT_EXT for the external interrupt pin (B0) or #INT_RDA for a incoming RS232 character.

- An interrupt must be specifically enabled (via enable interrupts) and interrupts must be globally enabled (via enable_interrupts(GLOBAL)). The GLOBAL enable/disable control whether any interrupts are serviced.

- Notice interrupts are disabled before the timer is read and combined with the overflow count. This is done to prevent the following situation:

    The timer value is read and it is 65535
    The overflow interrupt happens and the counter is inc'ed to 1
    The program continues and reads the counter as 1
    The time is assumed to be 65536+65535 when in fact
       The correct time is 65535

- If interrupts are disabled and an interrupt event happens, then the interrupt function will be called when interrupts are enabled. If multiple interrupt events of the same type happen while interrupts are disabled, then the interrupt function is called only once when interrupts are enabled.

- The %06lu format specifier is the same as %6lu except leading zeros are printed.

Further Study

A Make a version of this program that prints in the format:
                MM:SS.FFFFFF
Where MM is minutes, SS is seconds and FFFFFF is fractions of a second.

B Add a second interrupt using timer 0 to interrupt every 13.1 ms. In the interrupt routine count interrupts and when 76 interrupts have happened do a putc('.');. This should display a period every second while interrupts are enabled.

15 Using the Breadboard

' The breadboard may be used to prototype circuits with components not on the prototyping board. The black terminal block is used to connect jumpers from the prototype board (see inside front cover for pinout) to the breadboard. The breadboard has two columns on either side where all the holes in the column are connected. These are usually used for +5V and Ground. The inside of the breadboard is split down the center and the holes on ether side are connected within the row. The gray shading on the right diagram shows how the holes are electrically connected.
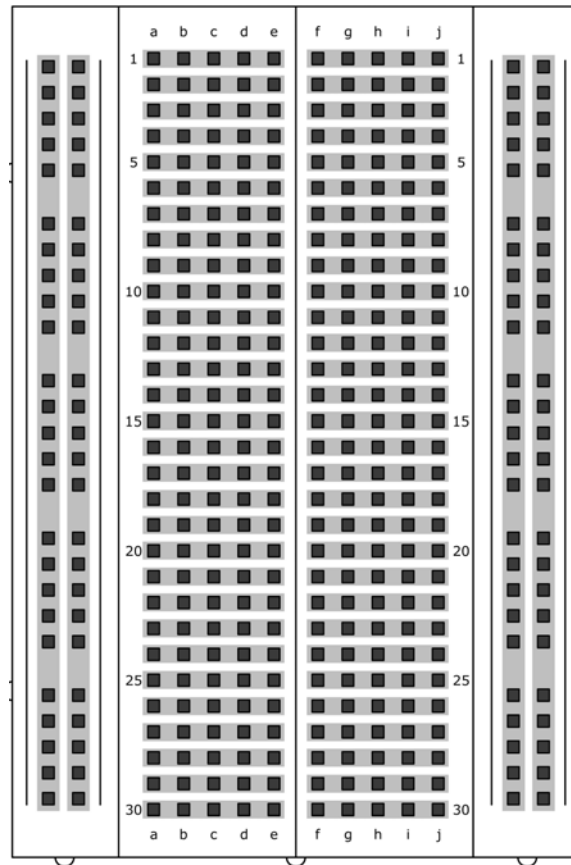
' This exercise will use an external serial EEPROM chip. Create the file **EX15.c**, a variation of EX9.C as follows:

```c
#include <protoalone.h>
#include <utility.c>
#include <9356.c>

main() {
    int count;

    init_ext_eeprom();
    count = read_ext_eeprom(0);
    while(TRUE) {
        show_binary_on_leds(count);
        wait_for_one_press();
        count++;
        write_ext_eeprom(0,count);
    }
}
```

' Wire up the circuit as shown in the following diagram. The diagram on the left shows how to route the wires. The schematic view on the right is how the remaining exercises in this booklet will show the wiring.
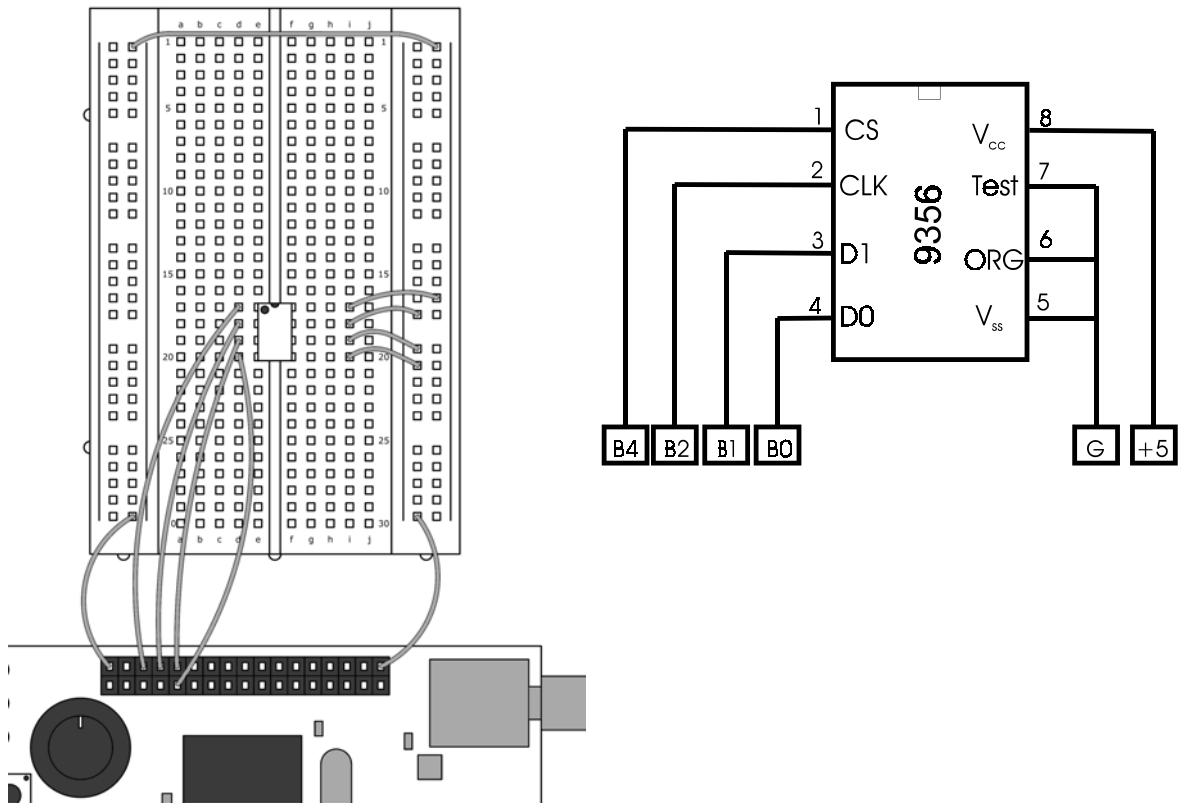
' Compile and Run the program. Test it like EX9.

Further Study

A To show your understanding of the breadboard, rewire this same circuit to use 3 less jumper wires.

B The functions to communicate with the 9356 chip are in 9356.c. Since the data in and data out pins are not used at the same time they can be the same pin. Modify the circuit and the 9356.c code so both data in and data out use the same pin on the PIC16F877.

16 Clocking Data In and Out

' Using the circuit from exercise 15 enter, compile and test the following program to more easily test the serial EEPROM:

```
#include <protoalone.h>
#include <input.c>
#include <9356.c>

main() {
    int cmd,address,data;

    init_ext_eeprom();
    while(TRUE) {
        printf("\r\nEnter address: ");
        cmd = get_int();

        do {
            printf("Read or Write (R,W): ");
            cmd = toupper(getc());
        } while ((cmd!='R') && (cmd!='W'));

        if(cmd=='R')
          printf("\r\nValue is %u\r\n",
                    read_ext_eeprom(address));
        else if(cmd=='W') {
            printf("Enter data: ");
            write_ext_eeprom(address,get_int());
        }
    }
}
```

' The 9356 device being used has 4 I/O connections as shown in this timing diagram. A diagram like this is typically found in the devices datasheet. The transmission starts with CS going high on this chip. Then the chip expects the DI to change while the CLK line is low; and it is safe to be read while the CLK is high. Note that the clock pulse before CS goes high is not required. Open the 93LC56 data sheet using **Tools > View Data Sheet**.
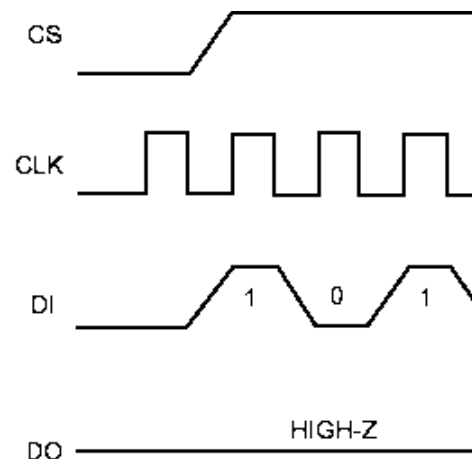
L Notes:

- Table 2-1 in the data sheet outlines the command format. All commands start with a 1 are followed by a 2 or 3 bit command. Depending on the command there may then be 8 address bits and 8 data bits. Commands in total are either 12 or 20 bits.

- The following code makes the CS high, then clocks 20 bits of data out in accordance with the previous timing diagram. The data is kept in a three byte array called cmd.

```
output_high(CS);
for(i=1;i<=20;++i) {
   output_bit(DI, shift_left(cmd,3,0));
   output_high(CLK);
   output_low(CLK);
}
```

  The shift_left function shifts all the bits in cmd one position to the left. The 3 tell the function how many bytes are in cmd and the 0 is the bit value to shift into the lowest position. The bit shifted out of the highest position is returned from the function, and in this case then passed to output_bit. output_bit() is like output_high/low except that the high/low is determined at run time by the second parameter.

- Open the 9356.c file in the drivers directory. Reviewing the code in write_ext_eeprom() you will see the code to issue a write command. When the shift is left, the first byte transmitted is cmd[2]. Since there are 24 bits in cmd and we only need 20, there is an initial shift of 4 bits to line up before transmission.

- Figure 3-1 in the data sheet shows the required times between events. For example, the time from CS going high to CLK going high is labeled as Tcss. Table 1-2 shows Tcss needs to be at least 50 ns. Not a problem in the code.

Further Study

A Add a new command to the EX17 program to erase the entire EEPROM. Add a function erase_ext_eeprom() that uses the chips ERAL command.

B The write_ext_eeprom function in 9356.c has a 11ms delay at the end to wait for the write to complete. Make a new version of this function that instead of a fixed delay uses the busy check feature described in the data sheet section 3.8

17 Using an I²C Temperature Sensor

' The previous exercise used 3-4 wires to communicate serially to a chip. That method is generally referred to as SPI (Serial Port Interface). A popular 2 wire communications bus that allows multiple devices to use the same two wires was developed by Phillips and is called I²C. This exercise uses a temperature sensor that communicates via I²C. Enter the following simple program to use the temperature sensor:

```
#include <prototype.h>

#include <ds1621.c>

main() {
    byte value;

    init_temp();

    do {
        value = read_temp();
        printf("%u\r\n",value);
        delay_ms(1000);
    } while (TRUE);
```

' Because multiple devices can use the same two wires, no device ever drives the wires high. Pull-up resistors on the wires allow them to float high and devices may ground the wire to make them low. This way a device will not be driving a line high at the same time another is driving it low. The two wires are designated SDA and SCL. Using the DS1631 chip wire up the following circuit:
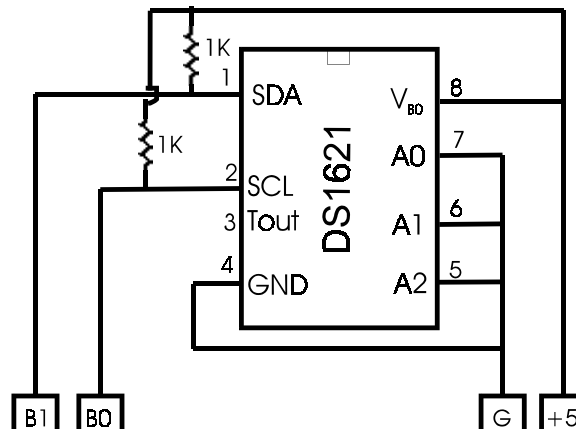
' Compile and Run the program. The monitor window should display the temperature every second. Hold your finger on the chip to raise the temperature reading.

L Notes:

- Since multiple devices are using the same two wires each device on the bus has a 7 bit address. For the DS1631, four of the address bits are fixed and the other 3 may be set via the A0, A1 and A2 pins. In our case we set them all to 0. This means up to 8 of these chips could be on the same two wire bus.

- Data is transferred over the bus by first sending a unique pattern on the pins called a start condition. This is always followed by the 7 bit address and a bit to designate if data is to transfer to or from the master. The master in our case is the PIC16F877. This byte is followed by any number of data bytes and a stop condition. Some devices allow the data direction to change by sending a new start, address and direction bit without a stop condition. The DS1631 requires a read command to be sent to it and then the data direction changes and two bytes are read from it. The following is an extract of the code from DS1631.C
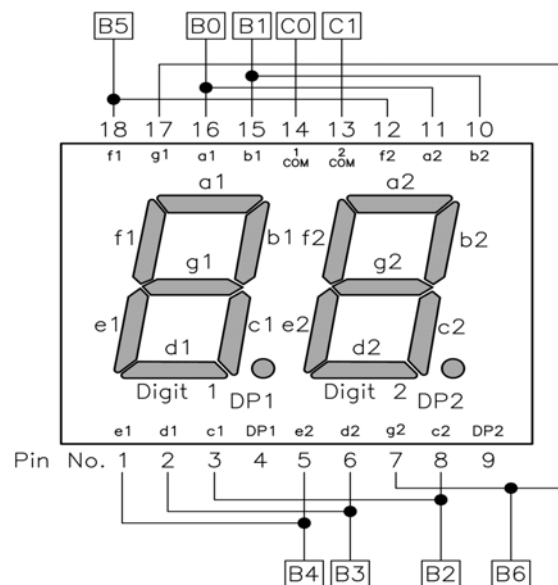
```
i2c_start();
i2c_write(0x90);   // Address and direction
i2c_write(0xaa);   // DS1631 command to read
i2c_start();
i2c_write(0x91);   // Address and direction
datah=i2c_read();
datal=i2c_read(0);
i2c_stop();
```

Further Study

A Make a version of this program that lights the green LED if the temperature is at or below 75, the yellow LED if it is 76-79 and the red LED when it is 80 and up.

B Each byte on the bus is acknowledged. I2C_WRITE returns a 0 if the byte was accepted. Make a new version of read_temp() that checks the result of the first I2C_write() and displays an error if the byte was not accepted. Then change the address from 1001000x to 1001010x and note that the new error is displayed. Now change the H/W circuit to use the new address.

18 Driving a 7 Segment LED Display

' 7 Segment LED units are used as an easy way to display numbers. Each of the 7 segments is a individual LED that is can be turned on just as the LEDs on the prototyping board. In order to save on I/O pins, it is common to multiplex the LED segments. In this case, there are two digits. Each segment is connected to the corresponding segment on the other digit. Only one digit is energized with the correct segments set. After a short period of time, that digit goes off and the other digit is lit with its segments on. By alternating very quickly, it will appear that both digits are on all the time and 9 I/O pins are used instead of 16. Connect up the following circuit to accomplish this goal:

∟ Notes:

- Each segment is identified with a designator like a1. Segment a is the top LED and 1 is the first digit. To light this digit, power up pins 16 and 14. To light the same segment on digit 2 (a2) then power up pin 16 and 13.

- This example does not use the decimal points (DP1 and DP2).

- Unlike the onboard LEDs, there is no built-in current limiting resistor on these units. Many applications will require a series resistor on each of the segments. In this case, we use a high current LED and know that the PIC16F877 drive capability will ensure the LED segments are not damaged.

' The following program will count from 1 to 99. Enter and run the program:

```
#include <protoalone.h>

byte CONST LED_MAP[10] =
{0x3f,0x06,0x5b,0x4f,0x6b,0x6d,0x7d,0x07,0x7f,0x67};

void display_number( int n ) {
   output_b(LED_MAP[n/10]);
   output_low(PIN_C0);
   delay_ms(2);
   output_high(PIN_C0);
   output_b(LED_MAP[n%10]);
   output_low(PIN_C1);
   delay_ms(2);
   output_high(PIN_C1);
}

void main() {
 int count=1,i;

 while(TRUE) {
    for(i=0;i<=200;i++)
       display_number(count);
    count = (count==99) ? 1 : count+1;
 }
}
```

L   Notes:

- Unlike the previous programs, this program writes to all I/O pins on one port at the same time using **output_b()**.

- The **LED_MAP** array specifies for each number which segments need to be on. The first entry (zero) needs segments a,b,c,d,e,f on. This is B0-B5 or in hex 0x3F.

- The ? : notation was used to make it easy to change the increment value and range with just this one line.

Further Study

A Make a version of this program that counts from 00-FF in hex. The hex digits A,b,C,d,E,F can be displayed using 7 segments.

B Use the DS1631 and this display to make a digital thermometer.

19 Real Time Clock Interfacing and BCD Numbers

' Computers work most effectively with binary numbers and humans like to use decimal numbers. A compromise number system is popular with some devices called Binary Coded Decimal (BCD). BCD uses 4 binary bits to encode each decimal digit. The numbers in memory are then saved, such that each decimal digit is exactly 4 bits. Twenty three in decimal is 23 and in binary is 00010111. In BCD it would be 00100011. Notice in hex, the binary number is 23. In one byte, although a binary number can hold 0 to 255 and in BCD a byte can hold only 0 to 99 (4 bits per decimal digit). To begin, create a file BCD.C that can be used to input and output BCD numbers as follows:

```c
byte get_bcd() {
  char first,second;

  do {
    first=getc();
  } while ((first<'0') || (first>'9'));
  putc(first);

  do {
    second=getc();
  } while (((second<'0') || (second>'9')) && (second!='\r'));
  putc(second);

  if(second=='\r')
    return(first-'0');
  else
    return(((first-'0')<<4)|(second-'0'));
}


void display_bcd( byte n ) {
  putc( (n/16)+'0' );
  putc( (n%16)+'0' );
}
```
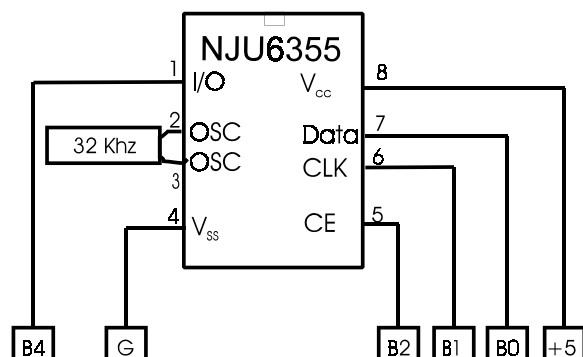
' Connect up the circuit to the right using the NJU6355 real time clock chip. This chip can keep accurate date and time using the small crystal attached to the part. The interface to the PIC16F877 is an SPI interface. Internally the date/time is kept in the chip in BCD.

' Enter and run the following program to demonstrate the real time clock:

```
#include <protoalone.h>
#include <nju6355.c>
#include <BCD.c>

void set_time() {
    int hour,min;

    printf("\r\nHour: ");   hour=get_bcd();
    printf("\r\nMin: ");     min=get_bcd();
    rtc_set_datetime(0,0,0,0,hour,min);
}

void main() {
    int hour,min,sec,last_sec;

    rtc_init();
    while (TRUE) {
        rtc_get_time( hour, min, sec );
        if(sec!=last_sec) {
            printf("\r\n");
            display_bcd(hour); putc(':');
            display_bcd(min);  putc(':');
            display_bcd(sec);
            last_sec=sec;
        }
        if(kbhit()&&(getc()=='S')
          set_time();
    }
}
```

L Notes:

- The **kbhit()** function returns true if a character comes in the RS232 port. The && operator evaluates right to left. This means if the **kbhit()** returns false, then the **getc()** is never called. If it were called, the program would hang here (because there was no character). This effect is referred to as a short circuit. Not all languages guarantee this, however, ANSI C does.

Further Study

A Make a version of this program that starts by asking for an alarm hour and minute. When the alarm time is reached light the red LED.

B Update the program to do both date and time.   See the NJU6355.c file for help on the parameters to **rtc_setdatetime()**.

20 Migrating to Your Own Hardware

The following diagram is a somewhat minimal circuit for a PIC16F877.

- Notice this chip has two +5V and ground connections. Some chips have only one of each. A 0.1uf capacitor mounted near the chip is a good idea and one on either side of the chip is even better. This will reduce noise both to and from the chip.
- The clock circuit here uses a crystal. With many high speed crystals the resistor on clock out prevents the crystal from being over driven. The capacitor values used are generally specified in the crystal specification sheets. The PIC16F877 data sheet will show a number of other options for the clock.
- The MCLR pin must be in a high state for the chip to run. Note the prototype board schematic uses a pushbutton to ground this pin and to reset the chip.
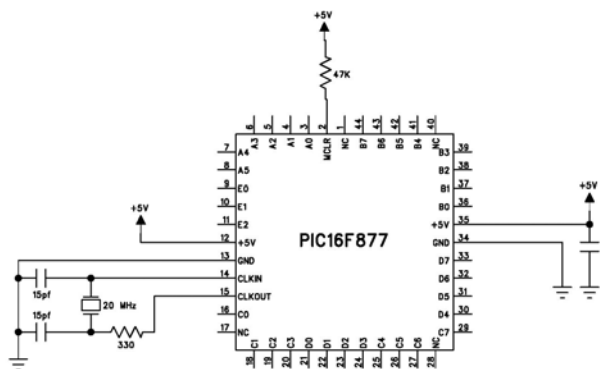

Troubleshooting.

- Most problems involve the clock. Make sure the configuration fuses are set to the proper oscillator setting. In the above case, for a 20 MHz crystal HS (High Speed) is the proper setting. In the above circuit, the size of the resistor may need adjustment depending upon the crystal.
- If the program does not seem to be running, verify 5 Volts on the MCLR pin and the two power pins.
- Isolate hardware problems from firmware problems by running a program with the following at the start of main() and check B0 with a logic probe or scope:

```
while(TRUE) {
    output_low(PIN_B0);
    delay_ms(1000);
    output_high(PIN_B0);
    delay_ms(1000);
}
```

The in circuit programming/debugging interface.

- To program and/or debug in circuit, two I/O pins (B6,B7) are reserved. If debugging is not to be done, then these pins may also be used in the target circuit. However, care must be taken to ensure the target circuit has high impedance during programming.
- The MCLR pin is also used by the programmer and for debugging. Note that during programming, the voltage on this pin is 13 volts. The 47K resistor to 5V is sufficient isolation for the 12V. However, if anything else is connected to the MCLR pin be sure the 13V will not damage or interfere.
- The ICD unit requires 5V from the target. It is easiest to power up the target normally and then connect the target board 5V to the ICD for power. An alternative would be to power the ICD from a special cable. In that case, there is no need to connect the 5V to the target unless the target is to be powered from the same source.
- The B3 pin is optional and is not used for programming. However, the Monitor feature of the debugger does use B3. It is possible to program and debug (without monitor) and allocate B3 to the target hardware. In this case do not connect B3 to the ICD connector.

Target ICD connector, looking into the connector

1 2 3 4 5 6
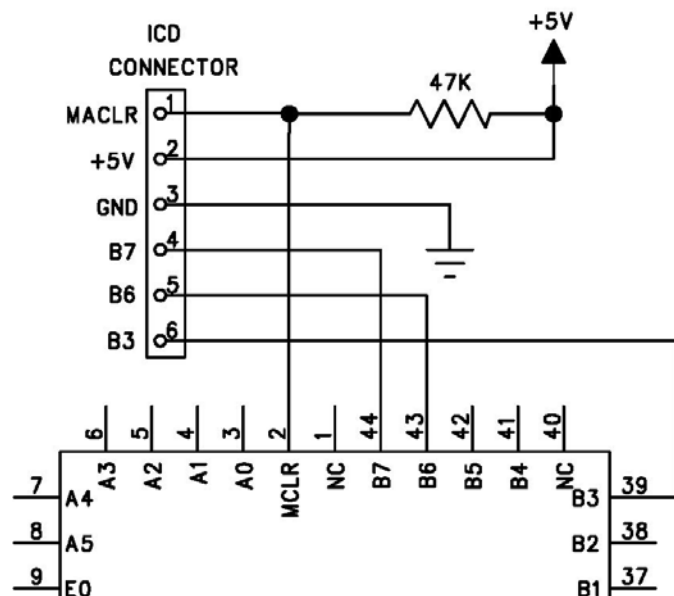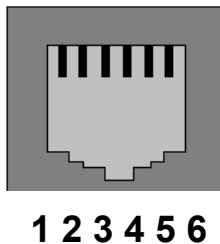
Note that the ICD to target cable reverses the pins so the MACLR signal is ICD pin 6 and that connects to the target pin 1.

References

This booklet is not intended to be a tutorial for the PIC16F877 or the C programming language. It does attempt to cover the basic use and operation of the development tools. There are some helpful tips and techniques covered, however, this is far from complete instruction on C programming. For the reader not using this as a part of a class and without prior C experience the following references should help.

Exercise	PICmciro® MCU C - An introduction to programming the Microchip PIC in CCS C By Nigel Gardner	The C Programming Language By Brian W. Kernighan and Dennis M. Ritchie (Second Edition)
3	1.1 The Structure of C Programs 1.2 Components of a C program 1.4 main() 1.5 #include 1.8 Constants 1.11 Macros 1.13 Hardware Compatibility 5.5 while Loop 9.1 Inputs and Outputs	1.1 Getting Started 1.4 Symbolic Constants 3.1 Statements and blocks 3.5 Loops 4.11 The C Preprocessor
4	1.7 Variables 1.10 Functions 2.1 Data types 2.2 Variable Declaration 2.3 Variable Assignment 2.4 Enumeration 3.1 Functions 3.4 Using Function Arguments 4.2 Relational Operators 5.7 Nesting Program Control Statements 5.10 switch Statement	1.2 Variables and Arithmetic Expr 2.1 Variable names 2.2 Data Types and sizes 2.3 Constants 2.4 Declarations 2.6 Relational and Logical Operators 3.4 Switch 1.7 Functions 1.8 Arguments - Call by Value 4.1 Basics of functions
5	4.3 Logical Operators 4.4 Bitwise Operators 4.5 Increment and Decrement 5.1 if Statement 5.2 if-else Statements 9.3 Advanced BIT Manipulation	3.2 If-Else 2.8 Increment and Decrement Ops 2.9 Bitwise Operators
6	4.1 Arithmetic Operators	2.5 Arithmetic Operators
7	9.5 A/D Conversion	3.3 Else

8	5.4 for Loop 6.1 One-Dimensional Arrays	1.3 The For statement 1.6 Arrays 2.10 Assignment Operators and Exp
10	1.6 printf function 9.6 Data Comms / RS232	1.5 Character Input and Output 3.6 Loops - Do-while 7.1 Standard Input and Output 7.2 Formatted Output - Printf
11	6.2 Strings 6.4 Initializing Arrays 8.1 Introduction to Structures	1.9 Character Arrays 6.1 Basics of Structures 6.3 Arrays of Structures
13	9.4 Timers	
14	2.6 type Conversion 9.11 Interrupts	2.7 Type Conversions
16	9.8 SPI Communications	
17	9.7 I2C Communications	
18	5.2 ? Operator	2.11 Conditional Expresions
19	4.6 Precedence of operators	2.12 Precedence and Order of Eval

On The Web

Comprehensive list of PICmicro development tools and information:	www.pic-c.com/links
Microchip home page:	www.microchip.com
CCS compiler/tools home page:	www.ccsinfo.com/picc
CCS compiler/tools software update page:	www.ccsinfo.com/download.shtml
C Compiler user message exchange board:	www.pic-c.com
Data sheet links	www.ccsinfo.com/data.html
C Compiler Technical Support	support@ccsinfo.com

Other Development Tools

EMULATORS

The ICD used in this booklet uses two I/O pins on the chip to communicate with a small debug program in the chip. This is a basic debug tool that takes up some of the chips resources (I/O pins and memory). An emulator replaces the chip with a special connector that connects to a unit that emulates the chip. The debugging works in a simulator manner except that the chip has all of its normal resources, the debugger runs faster and there are more debug features. For example an emulator typically will allow any number of breakpoints. Some emulators can break on an external event like some signal on the target board changing. Some emulators can show you the instructions that were executed before a breakpoint was reached. Emulators cost between $500 and $3000 depending on the chips they cover and the features.

DEVICE PROGRAMMERS

The ICD can be used to program FLASH chips as was done in these exercises. A stand alone device programmer may be used to program all the chips. These programmers will use the .HEX file output from the compiler to do the programming. Many standard EEPROM programmers do know how to program the Microchip parts. There are a large number of Microchip only device programmers in the $100-$200 price range. Note that some chips can only be programmed once (OTP) and some parts need to be erased under a UV light before they can be re-programmed (Windowed).

PROTOTYPING BOARDS

There a large number of prototyping boards available from a number of sources. Some have an ICD interface and other simply have a socket for a chip that is externally programmed. Some boards have some advanced functionality on the board to help design complex software. For example CCS has a prototyping board with a full 56K modem on board and a TCP/IP stack chip ready to run internet applications such as a e-mail sending program or a mini web server. Another prototyping board from CCS has a USB interface chip making it easy to start developing a USB application program.

SIMULATORS

A simulator is a program that runs on the PC and pretends to be a microcontroller chip. A simulator offers all the normal debug capability such as single stepping and looking at variables, however there is no interaction with real hardware. This works well if you want to test a math function but not so good if you want to test an interface to another chip. With the availability of low cost tools such as the ICD-S in this kit there is less interest in simulators. Microchip offers a free simulator that can be downloaded from their web site. Some other vendors offer simulators as a part of their development packages.