Project 1: C Functions and Programs
MAE 5483

Charles O'Neill

September 19, 2004

Introduction

This project's purpose is to become familiar with C programming on the PIC 16F876 micro-controller. Each of the four programs demonstrates a particular aspect of micro-controller C programming: mathematical operations, functions, simple data input, and pointers.

The micro-controller is a 28 pin DIP PIC16F876 manufactured by Microchip. The PIC voltage input is +5 volts DC via a μA7805 dc/dc voltage regulator. The compiler is the CCS C compiler (v. 3.207) for 14 bit PIC chips. C compilation occurs on a x86 based PC. Data transfer between the PIC and the PC is through a 9 pin serial cable. All programming and experiments were performed by Charles O'Neill.

1 Math Operations

This part investigates mathematical operations in C on the PIC. The objective is to learn about basic mathematical operations and data sizes.

1.1 Math Calculations and Sizes

This program investigates basic mathematical calculations and data type sizes. The C code is given in mathops.c Code Listings section (p. 12). Functions are used for each 'part' of this program. This allows for a smaller memory footprint since memory locations can be reused. Otherwise, this program exceeds the PIC's memory because the floating point numbers require extra storage and extra calculation operations.

Browsing through the limit.h and float.h header files gives some insight without running the PIC. First, limit.h gives the representation limits for integers of various types. For example, the maximum representable unsigned integer is 255. For an unsigned long integer, the maximum is 65535. These numbers can be determined within the compiler by including limit.h. Also, float.h gives the properties for floating point numbers. Interestingly, the maximum and minimum float values are 3.4e38 and 1.17e-38 —not symmetrical. The most important number is $\epsilon = 1.19e{-}7$, the smallest representable non-dimensional change. The mathops.c program will attempt to get an approximation for ϵ. Additionally, the float.h file verifies that float and double are treated as equivalent for this compiler.

Basic Calculations The program's first part is the basic calculations function, mathops(). The objective is to test integer and floating point arithmetic.

Integer arithmetic output is shown below. Addition, subtraction, etc. are tested. Surprisingly, $2/2 = 0$, which wasn't expected. A past-maximum cast was attempted —it can't possibly work right, but the returned value, 0, is interesting. Hex arithmetic works correctly. Operator precedence is tested.

```
Mathematical Operations:
Integer:
 1      = 1
 1 + 1  = 2
 1 - 1  = 0
 1 - 2  = -1
 2 * 3  = 6
 2 / 2  = 0
 2 / 3  = 0
 3 / 2  = 1
 14 mod 4  = 2
 (int) 3.001 / (int) 1.999 = 3
 (int) 1e200      = 0
 0xaaaa - 0xaaa9 = 1
 2 * 3 / 2 + 1 = ((2 * 3) / 2) +1= 4
```

Next, a corresponding floating point arithmetic test was performed. The output is shown below. The various arithmetic operations work as expected. The past-maximum output is the maximum floating point number... not Inf

as expected. The most interesting part is the smallest float and double change test. The program attempts to find the smallest changable number[1] which gives an indication of how many digits are kept. Interestingly, the compiler given $\epsilon = 1.19e-7$ is near the mathops.c *simple and fast* test of $\epsilon = 1.46e-7$. Again, float and double are verified as the same.

```
Float:
 1.0      = 1.000000
 1.0 + 1.0  = 2.000000
 1.0 - 1.0  =  .000000
 1.0 - 2.0  = -1.000000
 2.0 * 3.0  = 6.000000
 2.0 / 2.0  =  .666666
 3.0 / 2.0  = 1.500000
 3.001 / 1.999 = 1.501250
 1e400      = 3.402821E+38
 2.0 * 3.0 / 2.0 + 1.0 = ((2.0 * 3.0) / 2.0) + 1.0= 4.000000
 Smallest float change= 1.464457E-07
 Smallest double change = 1.464457E-07
```

Data Type Sizes This part determines the memory sizes of the data types. The primary tool is the sizeof() function. The data types are as given in the manual.

```
Data Sizes on the PIC 16F876
Data Type      Definition   Bytes

Integer        int            1
Character      char           1
Short          short          1  Note!
Float          float          4
Double         double         4
Long Integer   long           2
1bit Integer   int1           1  Note!
8bit Integer   int8           1
16bit Integer  int16          2
32bit Integer  int32          4
Integer String int[10]       10
```

[1] Non-dimensionalized to 1.0. ϵ is *not* the smallest representable number!

Interestingly, int1 has a sizeof()=1 byte. The actual size is 1 bit as seen with the compiler memory map. The int1 variables i1–i16 are each 1 bit; however, the compiler fits 8 int1 values in one byte. The memory address is 03F in this particular program. Thus, the size of an int1 is 1 bit and not one byte. The same situation occurs for the short int.

```
03F.0   datasizes.i7
03F.1   datasizes.i8
03F.2   datasizes.i9
03F.3   datasizes.i10
03F.4   datasizes.i11
03F.5   datasizes.i12
03F.6   datasizes.i13
03F.7   datasizes.i14
```

Also, the int1 has some restrictions. `int1 stringint1[8]` fails with "arrays of bits are not permitted." Of course, most programmers would use a regular int and mask for each bit, but still the failure of an array of int1 isn't intuitive.

1.2 Powers x^y

This program investigates possible power functions x^y in C. The intuitive specification is the caret symbol (∧); however, this symbol is reserved for bitwise XOR. A proper C power function is defined in math.h as both `pwr()` and `pow()`. From math.h, the code for `pow()` is:

```
////////////////////////////////
//      float pow(float x,float y)
////////////////////////////////
// Description : returns the value (x^y)
// Date : N/A
//
float pow(float x,float y)
{
   if(x>=0)
     return(  exp(y*log(x)) );
   else
     return(  -exp(y*log(-x)) );
}
```

Notice that the method is:

$$x^y = \exp(\ln(x^y)) = \exp(y \ln x)$$

Near $x = 0$, the method approaches a singularity at $\ln(0)$, so sufficiently small x values will give poor numerical results.

A program was written to investigate the XOR ($\wedge$) operator and `pow()` function. The C code is given in the Code Listings section (power.c). The default mathematical operation was $2^3 = 8$. The output is given below.

```
power.c:
   Experiments with powers: x^{y}
   x=2, y=3
```

The first part tests the XOR operator. Converting to binary, the operation is: $0010 \wedge 0011 = 0001$, which is 1 in decimal. XOR is a bitwise operator, so C is expecting integers. The float input returns a unique but totally incorrect float.

```
Intuitive caret symbol ^:
Integer 2^3 = 1
Float   2.0^3.0 = 1.175494E-38
```

Next, the math.h defined pow() function is tested. Interestingly, C automatically casts integers to floats for `pow(2,3)`. The float input `pow(2.0,3.0)` gives the expected float 8.0.

```
Math.h pow() function:
Integer pow(2,3) = 8
Float   pow(2.0,3.0) = 8.000000
```

The last test is for operating limitations. From the math.h code, the `pow()` function is expected to have difficulties when ln is poorly defined near $x = 0$. The first test is 0^0, which is considered undefined but should practically give 1.0. As expected from the source exp(0) yields 1.0, even though the underlying log(0.0) calculation failed!

A number to the power one should give that number. For $10^1 = 10$,the PIC correctly returns 10.0. Increasing to $1e200^1$ fails; although $1e200^0$ is correct. This behavior shows that powers of 0.0 are always correct, but do not reflect the *real* accuracy of the `pow()` function for non-zero powers (ie. x^y where $y \approx 0.0$).

```
pow() Limits:
pow(0.0,0.0) = 1.000000E+00
log(0.0) = -8.802969E+01
pow(10.0,1.0) = 1.000000E+01
pow(1e200.0,1.0) = 0.000000E00
pow(1e200.0,0.0) = 1.000000E+00

Finished
```

This program shows that the intuitive power operator is not available in C; `pow()` in math.h must be used instead. The `pow()` function has operating limitations for x=0 based on the mathematical method.

2 Is-Among Function

This section creates an `isamong` function which tests for the presence of a particular character in a string. The function requires a character and a string argument, returns 1 if the character is contained in the string, and returns 0 otherwise. The function is:

```
/* Among function: Tests for presence of <character> in <string> */
int cro_among(char character, char* string ){
   // Step through string elements
   while(*(string++)){
      if(*(string)==character) return(1); // Characters match
   }
   return(0); // End of string with no match
}
```

The function steps through each character in the string `while(*(string++))` until a null character is found and 0 is returned. If the characters match `if(*(string)==character)`, the function returns 1. Unterminated strings will cause over-reads and likely failure. The following outputs were generated with the supporting program given in the cro_among.c Code Listings section (p. 16).

Validation of the `cro_isamong` function requires showing the two cases: among and not-among. The string will be: `The quick BROWN fox JUMPED over the LAZY dog.`. The test letters are 'q', among, and 'Q', not-among. The program output for the among case is:

```
Isamong:
```

```
    Tests a string for the presence of a character.

    Type your string and press <enter>:
       The quick BROWN fox JUMPED over the LAZY dog.
    Would you like to remove the string termination? (y/n)
    Type your search character:
       q

 The character 'q' is among "The quick BROWN fox JUMPED over the
 LAZY dog.".
```

and for the not-among case:

```
 Isamong:
    Tests a string for the presence of a character.

    Type your string and press <enter>:
       The quick BROWN fox JUMPED over the LAZY dog.
    Would you like to remove the string termination? (y/n)
    Type your search character:
       Q

 The character 'Q' is not among "The quick BROWN fox JUMPED over
 the LAZY dog.".
```

The program is working properly. However, the next output concerns the sadistic case of no terminating null character.

```
 Isamong:
    Tests a string for the presence of a character.

    Type your string and press <enter>:
       Bad_string
    Would you like to remove the string termination? (y/n)
    Type your search character:
       b

 The character 'b' is among "Bad_string#hat is behind an untermi
nated string?!".
```

Although this program used the same string for the 'over read', the general case could allow for reading arbitrary or dangerous memory locations. The given among function can do nothing about this type of error.

3 Vowel Count

This section creates a C program which counts the number of characters of vowels, consonants, and non-letters in a string. Vowel search characters are "aeiouAEIOU". Consonants are the letters 'a' through 'z' and 'A' through 'Z' except for vowels. Non-letters are anything else: numbers, symbols, etc. The C source code is given in the vowelcount.c Code Listings section (p. 18).

The first validation test is with a lowercase alphabet string, which has 5 vowels, 21 consonants, and no other characters. The program output is:

```
Vowelcount:
   Counts the number of vowels
   input from the keyboard.

   Type your string and press <enter>:
       abcdefghijklmnopqrstuvwxyz

There are 5 vowels, 21 consonants, and 0 non-letters in:
abcdefghijklmnopqrstuvwxyz

Finished!
```

The output is correct. Next, a more complex case is tested.

```
Vowelcount:
   Counts the number of vowels
   input from the keyboard.

   Type your string and press <enter>:
       12345^&*()AleTTEroRtWOO 1 2;

There are 5 vowels, 7 consonants, and 16 non-letters in:
12345^&*()AleTTEroRtWOO 1 2;

Finished!
```

Again the program is correct: 5 vowels (AeEoO), 7 consonants (lTTrRtW), and 16 non-letters. The vowelcount program works correctly.

4 Pointer Demonstration

This section investigates pointers and memory locations. The objective is to create, find, and operate with memory addresses. A program was created to perform a series of pointer demonstrations. The C source code is given in the pointer_demo.c Code Listings section (p. 20).

The compiled program's memory map is given below:

```
Address Variable
021     main.i
022     main.a
023     main.b
024     main.c
025-029 main.d
02A     main.p1
02B     main.p2
02C     main.p3
02D     main.p4
02E     main.p5
02F-033 main.p6
034-037 main.x
038-03B main.y
03C-03F main.z
040     main.px
041     main.py
042     main.pz
043     main.pzz
044-067 main.string
068     main.element
```

Now, the program is started. From the memory map, a is at address 022 and b is at 023. The float x is at 034. The pointer p1 points to 'a'. The pointer pz points to py, which is address 0. From the program output:

```
Pointer Demonstration

Variable Value and Location
'a' is 3 at 22
'x' is 3.1E+00 at 34
p1 points to 22 and dereferences to 3
One byte past 22 is 23, which happens to be b of value 16 = 16
```

```
pz points to py which points to y. *pz=38


Pointer  Address  Map  Dereference
  p1        22       022    3
  p2        23       023    16
  p3        23       023    16
  p5        25       025    1
  px        34       034    3.1E+00
  py        38       038    5.7E+01
  pz        41       041     38
```

Next, a cycling stack is created by printing the value referenced by a moving pointer —with the mod operator % to allow for cycling.

```
Cycling Stack value:  1  5  3  4  2  1  5  3  4  2
```

Next, a function call is made with pointers. The `void print_backwards(char* characters, int* size)` function requires a pointer for a character array and an address for the size of the array. The function reverses an input string. An attempt to change the address of a string fails... variable addresses are fixed during the compile as seen from the memory map.

```
The string is: " Quick, Watson, the game is afoot! "
at address 44
Shifting string by +7 yields: "Watson, the game is afoot!"
at address 4b
Changing memory location FAILS: string=string+7;

Reverse the string with a function.
The string is now: "!toofa si emag eht ,nostaW ,kciuQ"
at address 44.
```

Finally, pointers were used to write out a portion of the PIC's memory. The output contents contain the expected values at the appropriate memory address. For example, address 023 has a value of 10 hex or 16 decimal. Also, the pointer address format is clear from address 02a, which is pointer p1.

```
Contents of an arbitrary memory range
 Address   Contents(hex)
 21      00
```

```
22      03
23      10
24      ff
25      01
26      05
27      03
28      04
29      02
2a      21

Finished!
```

This program did have a problem. Unfortunately, a standard method for making pointers to pointers (**p) wasn't understood by the compiler. An attempt to fool the complier didn't work well: `float *pzz=&pz`. The unfortunate problem is that pointers to pointers are often useful for multi-dimensional arrays. A further search will look into how the CCS compiler handles pointers to pointers.

Conclusions

Four sets of programs were created to become familiar with C programming on the PIC16F876. The programs experimented with mathematical operations, functions, strings, input, memory addressing and pointers. The programs are given in the Code Listings section.

Code Listings

mathops.c

```
/*      Charles O'Neill
        MAE 5483
        Project 1.1a
*/

#include <16F876.h>
#use delay(clock=20000000)
#fuses HS,NOWDT
#use rs232(baud=19200,parity=N,xmit=PIN_C6,rcv=PIN_C7)
#include <stdio.h>
#include <math.h>

#define MAXPOWER 300

/* Declarations */
void mathops(void);
void datasizes(void);

void main(void){

   // Functions for each 'part' of this program. Allows for
   // a smaller memory footprint since memory locations can
   // be reused.
   mathops();
   datasizes();

   // Inform that the program is complete (not hung on calc)
   printf("\r\n\r\n  Finished! ");

}

/* Determine data sizes on PIC */
void datasizes(void){

   int a_int, string[10];
```

```
    char a_char;
    short a_short;
    float a_float;
    double a_double;

    long a_long;
    int1 a_int1; int8 a_int8; int16 a_int16; int32 a_int32;
//    int1 stringint1[8]; FAILS... arrays of bits are not permitted

    // Determine if the compiler flattens multiple 1 bits -> 1 byte
    // Look at Memory map to see
    int1 i1, i2,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12,i13,i14,i15,i16;

    printf("\r\n\r\nData Sizes on the PIC 16F876\r\n");
    printf("Data Type       Definition    Bytes\r\n\r\n");
    printf("Integer         int           %3d\r\n", sizeof(a_int));
    printf("Character       char          %3d\r\n", sizeof(a_char));
    printf("Short           short         %3d  Note!\r\n", sizeof(
       a_short));
    printf("Float           float         %3d\r\n", sizeof(a_float));
    printf("Double          double        %3d\r\n", sizeof(a_double));

    printf("Long Integer    long          %3d\r\n", sizeof(a_long));
    printf("1bit Integer    int1          %3d  Note!\r\n", sizeof(
       a_int1));
    printf("8bit Integer    int8          %3d\r\n", sizeof(a_int8));
    printf("16bit Integer   int16         %3d\r\n", sizeof(a_int16));
    printf("32bit Integer   int32         %3d\r\n", sizeof(a_int32));
    printf("Integer String int[10]        %3d\r\n", sizeof(string));

}

/* Experiment with integer and float calculations */
void mathops(void){

    int i;
    float a_float;
    double a_double;

    printf("\r\nMathematical Operations:");
```

```
printf("\r\nInteger:\r\n");
printf("␣1␣␣␣␣␣␣=␣%d␣\r\n", 1);    // No Operation
printf("␣1␣+␣1␣␣␣=␣%d␣\r\n", 1+1);    // Addition
printf("␣1␣-␣1␣␣␣=␣%d␣\r\n", 1-1);    // Subtraction
printf("␣1␣-␣2␣␣␣=␣%d␣\r\n", 1-2);    // Subtraction
printf("␣2␣*␣3␣␣␣=␣%d␣\r\n", 2*3);    // Multiplication
printf("␣2␣/␣2␣␣␣=␣%d␣\r\n", 2 / 3);    // Division
printf("␣2␣/␣3␣␣␣=␣%d␣\r\n", 2 / 3);    // Division
printf("␣3␣/␣2␣␣␣=␣%d␣\r\n", 3 / 2);    // Division
printf("␣14␣mod␣4␣␣␣=␣%d␣\r\n", 14 % 4);    // Modulus
printf("␣(int)␣3.001␣/␣(int)␣1.999␣=␣%d␣\r\n", (int) 3.001 / (int
    ) 1.999);    // Division
printf("␣(int)␣1e200␣␣␣␣␣␣=␣%d␣\r\n", (int) 1e200);    // No
    Operation Overflow
printf("␣0xaaaa␣-␣0xaaa9␣=␣%d␣\r\n", 0xaaaa - 0xaaa9  );    //
    Addition
printf("␣2␣*␣3␣/␣2␣+␣1␣=␣((2␣*␣3)␣/␣2)␣+1=␣%d␣\r\n"
    , 2 * 3 / 2 +1);    // Multiplication

/* Experiment with integer and float calculations */
printf("\r\nFloat:\r\n");
printf("␣1.0␣␣␣␣␣␣=␣%f␣\r\n", 1.0);    // No Operation
printf("␣1.0␣+␣1.0␣␣␣=␣%f␣\r\n", 1.0+1.0);    // Addition
printf("␣1.0␣-␣1.0␣␣␣=␣%f␣\r\n", 1.0-1.0);    // Subtraction
printf("␣1.0␣-␣2.0␣␣␣=␣%f␣\r\n", 1.0-2.0);    // Subtraction
printf("␣2.0␣*␣3.0␣␣␣=␣%f␣\r\n", 2.0*3.0);    // Multiplication
printf("␣2.0␣/␣2.0␣␣␣=␣%f␣\r\n", 2.0 / 3.0);    // Division
printf("␣3.0␣/␣2.0␣␣␣=␣%f␣\r\n", 3.0 / 2.0);    // Division
printf("␣3.001␣/␣1.999␣=␣%f␣\r\n", 3.001 / 1.999);    // Division
printf("␣1e400␣␣␣␣␣␣=␣%e␣\r\n", 1e400);    // No Operation
    Overflow
printf("␣2.0␣*␣3.0␣/␣2.0␣+␣1.0␣=␣((2.0␣*␣3.0)␣/␣2.0)␣+␣1.0=␣%f␣\r
    \n", 2.0 * 3.0 / 2.0 + 1.0);    // Multiplication

// Find smallest representable number for float and double
for(i=1; i<MAXPOWER; i++){
    if((1) == (1+pow(10,-i))){
    printf("␣Smallest␣float␣change=␣%e␣\r\n", pow(10,-i+1));
    break;
```

```
        }
    }
    for(i=1; i<MAXPOWER; i++){
        if((1) == (1+pow(10,-i))){
         printf("␣Smallest␣double␣change␣=␣%e␣\r\n", pow(10,-i+1));
         break;
        }
    }

}
```

power.c

```
/*
*       power.c: Experiments with powers $x^{y}$
*
*       Charles O'Neill
*      MAE 5483
*       Project 1.1b
*/

#include <16F876.h>
#use delay(clock=20000000)
#fuses HS,NOWDT
#use rs232(baud=19200,parity=N,xmit=PIN_C6,rcv=PIN_C7)

#include <stdio.h>
#include <math.h>

void main(void){

    int     x=2, y=3;
    float  a=2.0, b=3.0;

    // Inform the user about the program's purpose
    printf("\r\n␣power.c:");
    printf("\r\n␣␣␣␣Experiments␣with␣powers:␣x^{y}");
    printf("\r\n␣␣␣␣x=2,␣y=3\r\n");
```

```
    // Experiment with intuitive caret symbol
    printf("\r\n\r\n  Intuitive caret symbol ^:");
    printf("\r\n  Integer 2^3 = %d", x^y);
    printf("\r\n  Float   2.0^3.0 = %e", 2.0^3.0);

    // Experiment with math.h defined pow() function
    printf("\r\n\r\n  Math.h pow() function:");
    printf("\r\n  Integer pow(2,3) = %d", (int) pow(x,y));
    printf("\r\n  Float   pow(2.0,3.0) = %f", pow(a,b));

    // Limits of pow() function
    printf("\r\n\r\n  pow() Limits:");
    printf("\r\n  pow(0.0,0.0) = %e  ", pow(0.0,0.0));
    printf("\r\n  log(0.0) = %e  ", log(0.0));
    printf("\r\n  pow(10.0,1.0) = %e  ", pow(10.0,1.0));
    printf("\r\n  pow(1e200.0,1.0) = %e  ", pow(1.0e200,1.0));
    printf("\r\n  pow(1e200.0,0.0) = %e  ", pow(1.0e200,0.0));

    // Inform that the program is complete (not hung on calc)
    printf("\r\n\r\n  Finished! ");
}
```

cro_among.c

```
/*
*       cro_among.c
*       Determines if a specific character is in a string
*
*       Charles O'Neill
*      MAE 5483
*       Project 1.2
*/

// Default PIC Initilization
#include <16F876.h>
#use delay(clock=20000000)
#fuses HS,NOWDT
#use rs232(baud=19200,parity=N,xmit=PIN_C6,rcv=PIN_C7)

// Program Specific Initilizations
```

```
#include <input.c>
#include <string.h>
#include <stdio.h>
#define MAX_LENGTH 50

// Function Declarations
int cro_among(char character, char* string);

void main(void){

   // Initilization
   int i, j, vowels=0, consonants=0, len=0;
   short presence;
   char string[MAX_LENGTH+1], not[4];
   char search;

   // Fill string for an eventual 'unterminated string' overrun test
      !
   strcpy(string,"Who knows what is behind an unterminated string?!"
      );

   // Inform the user about the program's purpose and expected input
   printf("\r\n Isamong:");
   printf("\r\n      Tests a string for the presence of a character.\
      r\n");
   printf("\r\n      Type your string and press <enter>: \r\n\t");

   // Read a string from the keyboard
   get_string(string, MAX_LENGTH+1);

   // Ask the user about an 'unterminated string' overrun test
   printf("\r\n      Would you like to remove the string termination
      ? (y/n)");
   if(getchar()=='y') (*(string+strlen(string))='#');

   // Read a character from the keyboard and echo it back
   printf("\r\n      Type your search character: \r\n\t");
   search=getchar();
   printf("%c",search); // Echo
```

```
    // Call the 'among' function to test for <search> in <string>.
    presence=cro_among(search, string);

    // Inform the user if search is in string.
    if(presence){
        printf("\r\n\r\n The character \'%c\' is among \"%s\".   ",
            search, string);
    } else {
        printf("\r\n\r\n The character \'%c\' is not among \"%s\".   "
            , search, string);
    }

}


/* Among function: Tests for presence of <character> in <string> */
int cro_among(char character, char* string ){

    // Step through string elements
    while(*(string++)){
        if(*(string)==character) return(1); // Characters match
    }
    return(0); // End of string with no match
}
```

vowelcount.c

```
/*
*       vowelcount.c
*       Counts the number of vowels in a given string
*
*       Charles O'Neill
*       MAE 5483
*       Project 1.3
*
*/

// Default PIC Initilization
#include <16F876.h>
#use delay(clock=20000000)
```

```
#fuses HS,NOWDT
#use rs232(baud=19200,parity=N,xmit=PIN_C6,rcv=PIN_C7)

// Program Specific Initilizations
#include <input.c>
#include <stdio.h>
#define MAX_LENGTH 80
#define NUM_VOWELS 10

void main(void){

   // Initilization
   int    i, j, vowels=0, consonants=0, len=0;
   char   string[MAX_LENGTH+1];
   char   search[NUM_VOWELS+1];

   // Define search characteristics
   strcpy(search,"aeiouAEIOU");

   // Inform the user about the program's purpose and expected input
   printf("\r\n Vowelcount:");
   printf("\r\n     Counts the number of vowels");
   printf("\r\n     input from the keyboard.\r\n");
   printf("\r\n     Type your string and press <enter>: \r\n\t");

   // Read a string from the keyboard
   get_string(string, MAX_LENGTH+1);
   printf("\r\n");

   // Determine the string's length
   for(i=0; i<=MAX_LENGTH; i++){
      if(*(string+i) == '\0'){
         len=i;
         break;
      }
   }

   // Step through string elements
   for(i=0; i<len; i++){
```

```
        // Check for letters
        if(   (('A'<=string[i]) && (string[i]<='Z')) ||   // A through Z
              (('a'<=string[i]) && (string[i]<='z')) ){   // a through z

           // Initial consonant increment... decrement later if
              determined to be a vowel
           consonants++;

           // Check and count vowel at the current string location
           for(j=0; j<=NUM_VOWELS; j++){
              // Compare current character with search letters
              if(*(string+i)==*(search+j)){
                 // Found a vowel; Increment vowel and decrement
                    consonant
                 vowels++; consonants--;
                 break;
              }
           }
        }

     }
     // Output total vowels
     printf("\r\n There are %d vowels, %d consonants, and %d non-
        letters in:\r\n %s \r\n", vowels, consonants, (len-vowels-
        consonants), string);
     printf("\r\n Finished!\r\n");

}
```

pointer_demo.c

```
/*
*      pointer_demo.c
*      Pointer Functionality Demonstration
*
*      Charles O'Neill
*      MAE 5483
*      Project 1.4
*/
```

```
// Default PIC Initilization
#include <16F876.h>
#use delay(clock=20000000)
#fuses HS,NOWDT
#use rs232(baud=19200,parity=N,xmit=PIN_C6,rcv=PIN_C7)

// Program Specific Initilizations
#include <stdio.h>
#include <string.h>
#define MAX_LENGTH 35
#define ELEMENTS 5
#define CYCLES 2

void print_backwards(char* characters, int* size);

void main(void){

   // Initilization
   int    i;
   int    a=3, b=16, c=255, d[ELEMENTS]={1,5,3,4,2};
   int    *p1, *p2, *p3, *p4, *p5, *p6[ELEMENTS];
   float  x=3.1415926, y=57.3, z=0.517;
   float  *px, *py;

   // Pointer to Pointer
   // float **pz; didn't work!!!
   float *pz; float *pzz;

   char string[MAX_LENGTH+1];
   char *element;

   // Inform the user about the program's purpose and expected input
   printf("\n\r\n\r Pointer Demonstration\n\r");

   // Setup pointers
   p1 = &a;    p2 = &b;    p3 = p2;    p4 = &c;    p5 = d;
   px = &x;    py = &y;

   // Pointer to Pointer
   pz = &py;
```

```
// Look at some memory locations and values
printf("\n\r Variable Value and Location");
printf("\n\r 'a' is %d at %x", a, &a);
printf("\n\r 'x' is %e at %x", x, &x);
printf("\n\r p1 points to %x and dereferences to %d", p1, *p1);
printf("\n\r One byte past %x is %x, which happens to be b of
    value %d = %d ", p1, p1+1, *(p1+1), b);

// Pointer to Pointer
printf("\n\r pz points to py which points to y. *pz=%x", *pz);

// Pointers and Addresses
printf("\n\r\n\r");
printf("\n\r Pointer  Address  Map  Dereference");
printf("\n\r   p1         %x       022    %d", p1, *p1);
printf("\n\r   p2         %x       023    %d", p2, *p2);
printf("\n\r   p3         %x       023    %d", p3, *p3);
printf("\n\r   p5         %x       025    %d", p5, *p5);
printf("\n\r   px         %x       034    %e", px, *px);
printf("\n\r   py         %x       038    %e", py, *py);

// Pointer to Pointer
printf("\n\r   pz         %x       041    %x", pz, *pz);

// Create a cycling stack with array d
printf("\n\r\n\r Cycling Stack value: ");
for(i=0; i< (CYCLES*ELEMENTS); i++){
    printf(" %d ", *(d+(i % ELEMENTS)));
}

// String Manipulation
strcpy(string, "Quick, Watson, the game is afoot!");
element=string;
printf("\n\r\n\r The string is: \" %s \" \n\r at address %x",
    string, string);
printf("\n\r Shifting string by +7 yields: \"%s\" \n\r at address
    %x", string+7, string+7);
printf("\n\r Changing memory location FAILS: string=string+7;");
    //string=string+7;
```

```
    // Use a Pointer in a Function
    printf("\n\r\n\r Reverse the string with a function.");
    i=strlen(string);
    // Call function to reverse and print string
    print_backwards(string, &i );
    printf("\n\r The string is now: \"%s\" \n\r at address %x.\r\n",
        string, string);

    // Write out contents of an arbitrary memory range
    p1=&i;
    printf("\n\r\n\r Contents of an arbitrary memory range");
    printf("\n\r   Address   Contents(hex)");
    for(i=0; i<10; i++){
        printf("\n\r  %x       %x", p1+i , *(p1+i));
    }

    // Inform that the program is complete
    printf("\r\n\r\n  Finished!  ");
}

// Reverses a String of Characters of length size. Palindromes
    Excepted :)
void print_backwards(char* characters, int* size){
    int index;
    char temp_string[MAX_LENGTH+1];

    //Save a copy of the string
    strcpy(temp_string, characters);

    /* Loop backwards but DON'T move the null character */
    for(index=0; index<=*size; index++){
        *(characters+index)=*(temp_string+*size-1-index); //flip
            characters
    }
    *(characters+*size)=0; //Make sure the null character is at the
        end!
}
```

EOF