The Hitchhiker's Guide to C Programming on the PIC

David Money Harris
E155 Fall 2008

This guide introduces C programming with emphasis on the PIC C compiler. It assumes familiarity with another high-level programming language such as Java. Parts of this tutorial are adapted from Nathaniel Pinckney's C Microtoys tutorial. More information is available in the Microchip MPLAB C18 C Compiler manuals.

Using the MPLAB C Compiler

You must configure MPLAB with the locations of the C18 programs. The C18 compiler should already be installed in the directory `C:\mcc18`. To configure MPLAB with the locations of the C18 programs, first go go to the `Project Menu` → `Set Language Tool Locations`... In the dialog that appears, under `Registered Tools` expand `Microchip C18 Toolsuite` and set `Executables` → `MPLAB C18 C Compiler (mcc18.exe)` location to `C:\mcc18\bin\mcc18.exe`, as shown in Figure 1. Next, expand `Default Search Paths & Directories`, set `Library Search path, $(LIBDIR)'s` location to `C:\mcc18\lib` and `Linker-Script Search Path, $(LKRDIR)`'s location to `C:\mcc18\lkr`. Lastly, click `OK` to close the dialog box and save the changes.

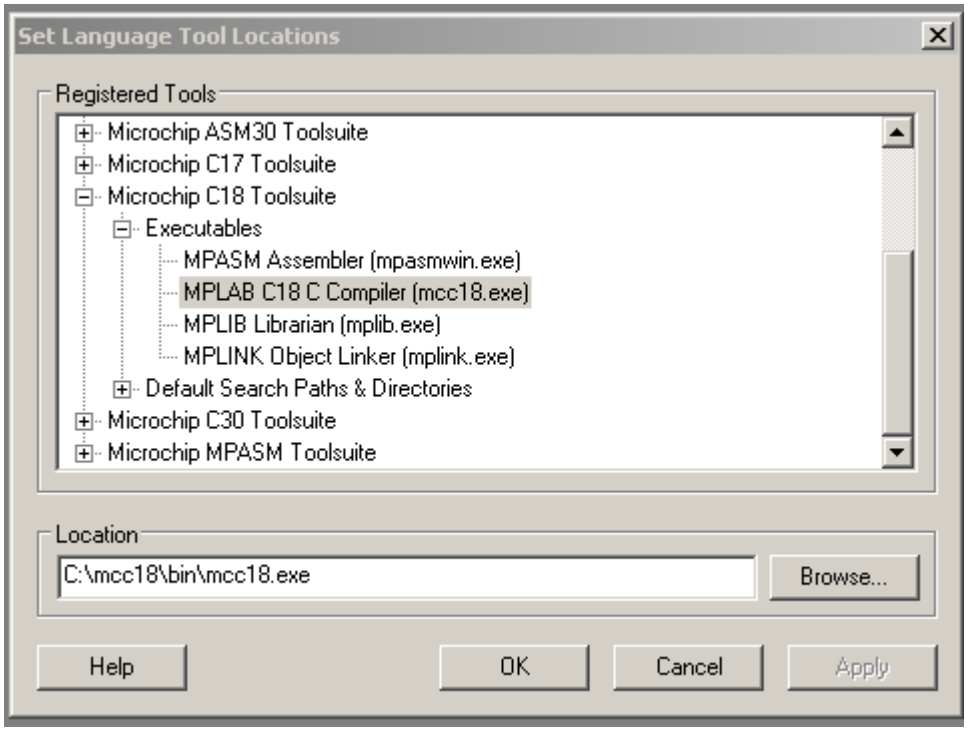

Figure 1: Setting the Language Tool Locations in MPLAB

Next, you want to create a project to use the C18 compiler. Select `Project` → `New` and enter a suitable name and location for your project. Note that MPLAB is picky about

project names and locations. You must preface the location with a drive letter; the \\server notation for remote directories is not supported. After creating the project, set the project toolsuite to C18, go to `Project` → `Select Language Toolsuite`. Change `Active Toolset` to `Microchip C18 Toolsuite`. Verify that all of the `Toolsuite Contents` values have associated `Locations`. Click `OK`. Next, go to `Build Options…` → `Project` and click the button `Suite Defaults`. **(Multiple people have overlooked this step, to their chagrin.)** The Library and Linker-Script paths should automatically be set from the values you entered earlier. Click `OK`.

Projects can have multiple source files. When compiling, C source files (.c) are translated and assembled into object files (.o), after which a linker combines the object files into a single binary file (.cof), suitable for programming onto a PIC's EEPROM. To create a new source file go to `File` → `New`. Save the file by selecting `File` → `Save As…`, navigating to the project's directory, entering a filename ending in .c, and clicking `Save`. Lastly, add the file to the project by right-clicking on `Source Files` in the project window and selecting `Add Files…` Be sure to begin your C source file with `#include <p18f452.h>` to be able to access common named special function registers, which will be discussed later.

Lastly, you must specify a linker script for the PIC you are using. Linker scripts are PIC specific and are used to generate an appropriate binary data image for the PIC you are using. Right-click on `Linker Scripts` in the project window and select `Add Files…` Navigate and select `C:\mcc18\lkr\18f452.lkr` as the linker script.

A Test Program

Try out the compiler using a simple test program:

```
/* ledtest.c */

#include <p18f452.h>

void main(void) {
	TRISD=0;
	PORTD=0x4A;
}
```

Build All, and then use the debugger to test the code in simulation and on your PIC. When you single step through, note that the C compiler adds some initialization code before it jumps to your main function.

If Build All complains that it can't find files such as c018i.o, clib.lib, or p18f452.lib, then you probably neglected to click Suite Defaults.

Data Types

C has a number of built-in data types available. The size of each primitive data type, except pointers which are covered later, along with the minimum and maximum numeric

values, is shown in Table 1. Unless declared with the qualifier *unsigned*, data types are signed (can hold positive and negative values) by default. Remember that the 18F452 is an 8-bit microcontroller. Using data types of more than 8-bits takes multiple instructions, so use 8-bit data types (char or unsigned char) where possible. Double-precision floating point numbers do not comply with the IEEE standard 64-bit size, but instead are identical to 32-bit floats. When byte-addressing data types, keep in mind that data on the PIC is stored in little endian format (the least-significant byte is stored at a lower address in memory than the most-significant byte). It is good practice to isolate code that depends on byte-ordering in memory, to make porting to other platforms easier.

Type	**Size**	**Minimum**	**Maximum**
char	8 bits	-2^-7 = -128	2^7 -1 = 127
unsigned char	8 bits	0	2^8 -1 = 255
int	16 bits	-2^15 = -32,768	2^15 – 1 = 32,767
unsigned int	16 bits	0	2^16 – 1 = 65,535
short	16 bits	-2^15 = -32,768	2^15 – 1 = 32,767
unsigned short	16 bits	0	2^16 – 1 = 65,535
short long	24 bits	-2^23 = -8,388,608	2^23 – 1 = 8,388,607
unsigned short long	24 bits	0	2^24 – 1 = 16,777,215
long	32 bits	-2^31 = -2,147,483,648	2^31 – 1 = 2,147,483,647
unsigned long	32 bits	0	2^32 – 1 = 4,294,967,295
float, double	32 bits	2^-126	2^128 * (2-2^-15)

Table 1: C18 Primitive Data Types and Sizes

Since PIC's are very limited in data memory, economical use of data memory is important. The PIC18F452 includes more program memory (32KB) than data memory (1.75KB), therefore the compiler defaults to storing constants, including string constants, in program memory.

Comments

Comments can begin with /* and end with */. They can span multiple lines. Comments can also begin with // and terminate at the end of the line.

```
// this is an example of a one-line comment.

/* this is an example
   Of a multi-line comment */
```

Operators

C supports the following operators. They are listed by category in order of precedence. For example, multiplication operators take precedence over addition operators. Within the same category, operators are evaluated in the order that they appear in the program.

Category	Name	Symbol
Monadic	Post-increment	++

	Post-decrement	--
	Address	&
	Bitwise NOT	~
	Type cast	(type)
	Logical NOT	!
	Negation	-
	Pre-increment	++
	Pre-decrement	--
	Size of data	sizeof()
Multiplicative	Modulus	%
	Multiply	*
	Divide	/
Additive	Add	+
	Subtract	-
Bitwise Shift	Shift left	<<
	Shift right	>>
Relational	Less than	<
	Less than or equal	<=
	Greater than	>
	Greater than or equal	>
	Equal to	==
	Not equal to	!=
Bitwise	AND	&
	XOR	^
	OR	|
Logical	AND	&&
	OR	||
Ternary	Conditional	? :
Assignment	Equal	=
	Other arithmetic	+=, -=, *=, /=, %=
	Other shift	>>=, <<=
	Other bitwise	&=, |=, ^=

Global and Local Variables

A global variable is declared outside of all the functions (normally at the top of a program). It can be accessed by all functions. A local variable is declared inside a function and can only be used by that function. Therefore, two functions could have local variables with the same names without interfering with each other. Use global variables sparingly because they make large programs more difficult to read.

Language Constructs

If//else

```
int bigger(int a, int b)
{
	if (a > b) return a;
	else return b;
}
```

While loops

```
void main(void)
{
	int i = 0, sum = 0;

	// add the numbers from 0 to 9
	while (i < 10) { // while loops check condition before executing body
		sum = sum + i;
		i++;
	}
}
```

Do loops

```
void main(void)
{
	int i = 0, sum = 0;

	// add the numbers from 0 to 9
	do {
		sum = sum + i;
		i++;
	} while (i < 10); // do loops check condition after executing body
}
```

For loops

```
void main(void)
{
	int i;
	int sum = 0;

	// add the numbers from 0 to 9
	for (i=0; i<10; i++) {
		sum += i;
	}
}
```

Pointers

A pointer is the address of a variable. Normally, pointers used in PIC programming are 16-bit numbers specifying an address in RAM or ROM.

For example, suppose you declare the following variables:

```
unsigned long salary1, salary2;	// 32-bit numbers
unsigned long *ptr;		/* a 16-bit pointer specifying the address of an
				   unsigned long variable */
```

The compiler will assign arbitrary locations in RAM for these variables. For the sake of concreteness, suppose `salary1` is at addresses 0x40-43, `salary2` is at addresses 0x44-47 and `ptr` is at 0x48-49.

In a variable declaration, a * before a variable name indicates that the variable is a pointer to the declared type. In a variable use, the * operator dereferences a pointer, returning the

value at the given address. The & operator is read “address of,” giving the address of the variable being referenced.

```
salary1 = 67500;              // assign salary1 to be $67500 = 0x000107AC
ptr = &salary1;               // assign ptr to be 0x0040, the address of salary1
salary2 = *ptr + 1000;        /* dereference ptr to give the contents of address 40 =
                                 67500, then add $1000 and set salary2 to $68500 */
```

The C18 compiler uses little-endian data storage, so the least significant byte of a multi-byte variable is stored at the lowest address. Thus, at the end of this program, the memory contains:

RAM Address	Contents	Notes
0x040	AC	LSB of salary1
0x041	07	
0x042	01	
0x043	00	MSB of salary1
0x044	94	LSB of salary2
0x045	0B	
0x046	01	
0x047	00	MSB of salary2
0x048	40	LSB of ptr
0x049	00	MSB of ptr

The C18 compiler supports a few qualifiers for pointers. Since memory on a PIC is not in a flat-addressing space, but split between program ROM and data RAM, a pointer must be labeled as either *rom* or *ram*, the two types are not compatible. Additionally, there are two sizes of pointers available, *near* and *far*. *Near* is a pointer within the first 64KB of ROM or in access RAM (i.e. a bank switch is not required). *Far* refers to any location in RAM or ROM. Since the PIC18F452 only includes 32KB of program memory, only a *near rom* pointer is needed to access a location in program memory. By default a pointer is a *far ram* pointer. *Ram* pointers are always 16-bits in size. *Near rom* pointers are 16-bits in size and *far rom* pointers are 24-bits in size.

Arrays

An array is a group of variables stored in consecutive addresses in memory. The elements are numbered starting at 0. In C, the array is referred to by the address of the 0^{th} element. It is the programmer’s responsibility not to access elements beyond the end of the array; the code will compile fine, but will stomp on other parts of memory.

When an array is declared, the length should be defined so that the compiler can allocate memory. When the array is passed to a function, the length need not be defined because the function only cares about the address of the first entry.

In the following example, suppose we have an array indicating how many wombats crossed the road each hour for each of the past 10 hours. Suppose the 0^{th} element is stored at address 0x20.

```
int wombats[10];       // array of 10 2-byte quantities stored at 0x20-0x33.
int *wombptr;          // a pointer to an integer

wombats[0] = 342;      // store 342 in addresses 0x20-21
wombats[1] = 9;        // store 9 in addresses 0x22-23
wombptr = &wombats[0]; // wombptr = 0x020
*(wombptr+4) = 7;      /* offset of 4 elements, or 8 bytes.  Thus addresses 0x28-29 = 7,
                          so this is another way to write wombats[4] = 7. */
```

The last example shows that `*(array+k)` is equivalent to `array[k]`.

```
int sort(int *vals, int numvals) // no need to declare the size of vals explicitly
{
        int i, j, temp;

        for (i=0; i<numvals; i++) { // numvals indicates the number of values
                for (j=i+1; j<numvals; j++) {
                        if (vals[i] > vals[j]) {
                                temp = vals[i];
                                vals[i] = vals[j];
                                vals[j] = temp;
                        }
                }
        }
}
```

By default, data structures stored in RAM cannot exceed one page (256 bytes). This is a problem for things like large arrays. See page 104 of the C Compiler Getting Started manual for how to override this.

Arrays of constants stored in program memory should be declared as *rom* (e.g. `rom int notes[] = {0x1234, 0x5678, …};`). This avoids the one page limitation for RAM arrays.

Characters

A character is an 8-bit variable. It can be viewed either as a number between -128 and 127 or as an ASCII code for a letter, digit, symbol, or so forth. Characters can be specified as a numeric value (in decimal, hexadecimal, etc.), or as a printable character enclosed in single quotes.

For example, the letter A has the ASCII code 0x41, B=0x42, etc. Thus `'A' + 3` is 0x44, or `'D'`.

Special characters include:
`'\r'`: carriage return (when you press the enter key)
`'\n'`: new line
`'\t'`: tab

Most C environments use \n alone to go to the beginning of a new line. However, some terminals expect both \n and \r to advance to the beginning of the next line. In particular, the PIC should send both characters when writing to a HyperTerm terminal.

The character 0 is called the null character. It terminates a string.

Strings

A string is an array of characters. Each character is a byte representing the ASCII code for that letter, number, or symbol. The array has a size, specifying the maximum length of the string, but the actual length of the string could be shorter. In C, the length of the string is determined by looking for a NULL character (0x00) at the end of the string.

```
void strcpy(char *src, char *dst)
{
	int i = 0;

	do {
		dst[i] = src[i];	// copy characters one byte at a time
	} while (src[i++] != NULL);	// until the NULL terminator is found
}
```

Inline Assembly

Occasionally you might want to include assembly code in parts of your program. It is possible with the C18 compiler to include inline assembly, but you should use it sparingly as the compiler will not optimize your code. Begin a block of assembly code with *_asm* and end with *_endasm*.

The syntax inline assembly code is similar to the MPLAB assembler, with some differences:

- No assembler directive support
- Comments should be in C notation – // instead of ;
- No default values for operands – Optional operands such as *d* (controls to/from WREG) for *movf* (and others) are required to be specified.
- Literals must use C radix notation – For hex, 0xABCD and not h'ABCD'.
- Labels must include colons – Always include : in label:

An example of calling a C function from inline assembly is given in the interrupts section.

Direct Hardware Access

You may access most special function registers, by the names given in the PIC's datasheet, by including the header file "p18f452.h". Some registers are included in other header files, for example, timer registers (TMR0, T0CON, etc) are in "timers.h". Individual bits of a special function register may be accessed through a C structure named by appending "bits" to the special function register's name. The individual bits share the name given in the PIC's datasheet. For example, all of INTCON, the interrupt control register, may be accessed (as an unsigned char) by just using "INTCON" or individual bits may be accessed (as boolean variables) like INTCONbits.TMR0IF.

```
TRISD = 0;
PORTD = 0xA5; // turn on four of the LEDs
```

Libraries

The C18 compiler comes bundled with C libraries to make a PIC programmer's life easier. Refer to the *MPLAB C18 C Compiler Libraries* manual for information about the different library functions included with C18; they would not all fit in this documentation. Header files might need to be included depending on what functions you are using. A sampling of the categories of functions included:

- Basic PIC features: A/D Converter, Input Capture, I^2C, I/O Port, Microwire, Pulse-Width Modulation, SPI, Timer, USART.
- External LCD
- Data Conversion – Including string to integer conversions, pseudo-random number generation, lower/upper-case ASCII conversion.
- Memory and String Manipulation – Copy data from different memory locations, compare strings, determine length of strings, tokenize strings, etc.
- Delay Functions
- Reset Functions – Determine if reset was caused by a brown-out, low voltage, MCLR pin, watchdog timer, or wake-up.
- Character Output – printf, fprintf, putc, and other conventional string and character output functions.
- 32-bit Floating Point – Routines implementing functions to manipulate IEEE-754 floating point numbers.
- C Standard Library Math Functions – sin, cos, log, sqrt, and much more.

printf Statements

The `printf` statement normally displays to a console. The PIC doesn't have a screen attached, so it instead sends characters over the serial port. If the serial port is properly connected to a PC running a terminal program such as HyperTerm, printf will display the text in the terminal window.

To make this connection, the PIC USART must be configured for serial output at a specific baud rate with no parity. Write to the appropriate registers to do this configuration. The PIC can connect to a PC over a serial cable or wireless link. To drive a serial cable, a 3.3-V compatible RS232 transceiver is required to change the voltage levels to the RS232 standard (+/- 5-15 V). Lab 6 explains how to connect wirelessly using a BlueSMiRF module attached to the PIC and a Bluetooth dongle on a PC.

`printf` takes a string containing text and optional commands to print variables. For example,

```
#include <stdio.h>
int num = 42;
printf("The answer to the ultimate question about life, the universe, and
        everything is %d\r\n", num);
```

`%d` tells `printf` to print the next variable as an integer (in decimal).

`printf` requires the `stdio.h` library. More formatting commands are described in the PIC libraries manual. Unfortunately, the library does not support printing floating point variables.

The C18 compiler gives a warning when the string is provided directly in the `printf` statement, but generates correct code anyway. The warning can be circumvented by defining the string separately:

```
#include <stdio.h>
int num = 42;
char fmtstr[] = "The answer is %d\r\n";
printf(fmtstr, num);
```

Interrupts

```
/* example of using a timer interrupt */
#include <p18f452.h>

/* Function Prototypes */
void main(void);
void isr(void);

// The #pragma tells the compiler to start a code section, named
// high_vector at the program memory address of 0x08.  This is the
// interrupt vector address.
#pragma code high_vector = 0x08
void high_interrupt(void) {
        _asm
                GOTO isr
        _endasm
}

// Now start the main code section.
#pragma code

void main(void) {
        // Set up timer interrupt
        T0CON = 0xC0;  // Timer on. 8-bit.
        INTCON = 0xA0; // Enable interrupts.  Interrupt on TMR0 overflow.

        while(1) {
                // do stuff that should normally happen
        }

}

// The #pragma lets compiler know isr() is an interrupt
// handler.  The compiled code will return correctly
// (with retfie instead of retlw) and save registers.
#pragma interrupt isr
void isr(void) {
        INTCONbits.TMR0IF = 0;        // clear interrupt flag
        // do stuff that should happen when the interrupt occurs
}
```

Example

```
/* lab4.c:  Implements E155 Lab 4 in C.
 * Author: Nathaniel Pinckney <npinckney@hmc.edu>
 * Date: March 2005
```

```
*/

#include <p18f452.h>

#define FINDMAX_NUM 5
#define SORT_NUM 12

/* Function Prototypes */
void main(void);
char findmax(char []);
void sort(char sortarr[]);

char maxarr[FINDMAX_NUM] = {-100,0,15,-20,-30};
char sortarr[SORT_NUM] = {0xc,0xb,0xa,0x9,0x8,0x7,0x6,0x5,
                          0x4,0x3,0x2,0x1};

void main(void) {
       TRISD = 0;
       PORTD = findmax(maxarr);
       sort(sortarr);
}

char findmax(char max[]) {
       char i=0, curr;
       curr = max[0];
       for(i = 1; i < FINDMAX_NUM; i++)
              if(max[i] > curr) curr = max[i];
       return curr;
}

// Simple bubble sort.
void sort(char sort[]) {
       char i, j;
       for(i = 0; i < (SORT_NUM-1); i++) {
              for(j = (i+1); j < SORT_NUM; j++) {
                     if(sort[j] < sort[i]) {
                            // Swap elements if [j] < [i]
                            sort[j] ^= sort[i];    ; Swap elements
                            sort[i] ^= sort[j];    ; by xoring them
                            sort[j] ^= sort[i];    ; repeatedly.
                     }
              }
       }
```