# Introduction to C Language for Team Ford FIRST

Chris Teslak

# Why C?

- Modular
- Powerful
- Industry standard
- Fast and efficient

# FUNCTIONS

- A segment of code that you write once; use many times in many different places in your program.

```
returnType functionName( parameterType parameter,
    …)
{
    do something;
    return valueToReturn;
}
```

- Functions must either be defined before main( ) or defined somewhere else and provide a function "prototype" before main. This gives the function global scope.
- Functions cannot be "nested" inside of each other.
- Functions can "call" each other.

# FUNCTIONS

- “Arguments” are values “passed” into a function. “Parameters” are the variables defined in the called function to hold the argument values.
- Starting point for the C program is the function known as “main”. Every C program must have a function with that name.

```
void  main (void)
{
  return;
}
```

- The keyword “void” is used in the example to indicate that no arguments are expected to be passed into this function and no return value is to be sent back to the calling function.

# VARIABLES

Variables are names that represent storage containers for values you can modify.

## NAMING RULES:

- Variable names can be any series letters, digits, and underscores but cannot begin with a digit or be keywords.
- Variable names are case sensitive and only the first 31 characters are recognized in C.
- Variable names should be descriptive of the values they represent.

# VARIABLE ATTRIBUTES

- **Types**
  - The variable “type” determines how much memory is allocated for values to be stored. It also determines how the value is to be interpreted (signed, etc.)
- **Initialization**
  - In general, unless you specify an initial value, the value of a variable is indeterminate until it is assigned a new value.
- **Scope**
  - Defines how far reaching the variable is known. The scope is usually determined by where in the code the variable is defined. There are modifiers that can broaden or limit a variable’s scope.
- **Lifetime**
  - Duration of a variable, how long the variable will persist. In other words, the memory allocated for the variable will be “freed” (reallocated for another) when the lifetime is expired.
- **Storage Class**
  - Affects the “linkage” of a variable. Examples are “extern” and “static”. They affect scope and lifetime. They can provide access to variables in areas of the software outside the normal scope. They can also limit access to variables from other parts of the code.
- **Modifiers**
  - Change how the compiler treats the variables. Two examples are “const” and “volatile”.

# VARIABLE TYPES

| Type | Size | Minimum | Maximum |
|---|---|---|---|
| **char** | **8 bits** | **-128** | **127** |
| signed char | 8 bits | -128 | 127 |
| **unsigned char** | **8 bits** | **0** | **255** |
| **int** | **16 bits** | **-32768** | **32767** |
| **unsigned int** | **16 bits** | **0** | **65535** |
| short | 16 bits | -32768 | 32767 |
| unsigned short | 16 bits | 0 | 65535 |
| short long | 24 bits | -8,388,608 | 8,388,607 |
| unsigned short long | 24 bits | 0 | 16,777,215 |
| long | 32 bits | -2,147,483,648 | 2,147,483,647 |
| unsigned long | 32 bits | 0 | 4,294,967,295 |
| float | 32 bits | $2^{-126}$ | $2^{128}$ |
| double | 32 bits | $2^{-126}$ | $2^{128}$ |

# VARIABLE INITIALIZATION

It is good practice to initialize all variables with some default value.

- Assign variables using the assignment operator: “=“
- **unsigned char** and **unsigned int** are the typical variable types used in the default code.
- The cast operator can be used to convert from one type to another but it is better not to mix types.

```
/* This is a comment.  It is ignored by the
      compiler but it helps us humans explain
      things. */

#include “printf_lib.h”
unsigned char pwm02 = 254;
void main( )
{
      unsigned char  pwm01 = 127;
      pwm01 = pwm01 + 1;

      /* Displays the value of pwm01 in the IFI
      terminal window */
      printf(“pwm01 equals %d \n”, (int)pwm01);

} /* end of main – program ends here */
```

# VARIABLE SCOPE

- **Where in the code you declare the variable will typically define the “scope” or “visibility” of the variable.**
- **If you declare a variable inside a function, the variable is only visible or known inside that function.**
- **If you declare a variable outside of any function, it is visible to any function in that file that is below the declaration of the variable.**
- **If you declare the variable with a storage class modifier, the scope can change.**

# VARIABLE LIFETIME

- **If you declare a variable inside a function, the variable only persists while the function is executing.**
- **If you declare a variable outside of any function, it persists as long as the entire program is still running.**
- **Adding a storage class modifier such as "static" to the declaration can change the variable's lifetime.**

# STORAGE CLASS

- **Typical storage class modifiers are “static” and “extern”.**
- **Typical use of “extern” is to make a file scope (“global”) variable in one file available to be used by functions in another file.**
- **Typical use of “static” is to make a function scope (“local”) variable persist for the duration of the program and not “go away” when the function is done executing.**

# VARIABLE MODIFIERS

- **Typical variable modifiers are "const" and "volatile".**
- **Typical use of "const" is to make a variable constant so that no code in the program will be allowed to change its value.**
- **Typical use of "volatile" is to tell the compiler that the variable may change but not necessarily by the code itself. For example, if you declare a variable to store the value of a digital input. The value stored will be determined by something external to the robot controller. You don't want the "smart" compiler to eliminate the variable because it doesn't see code writing to it.**

# MANIFEST CONSTANTS

- **These are constant (code can't change them) values with scope determined by where they are defined but they are not variables, they have no storage associated with them.**
- **Think of them as "macros." The compiler does a global search-and-replace (within the scope) of the left string of text with the right string of text.**

Examples:

```
#define p4_sw_aux2        rxdata.oi_swB_byte.bitselect.bit7
#define FIELD_WIDTH       125
#define FIELD_LENGTH      300
#define FIELD_PERIM       FIELD_WIDTH*2 + FIELD_LENGTH*2
```

# PREPROCESSOR DIRECTIVES

The # symbol denotes a pre-processor directive that is evaluated before the compiler takes over. The # symbol must be in the first column of a line. Some typical ones used in the default code are as follows:

- **# include <filename.h>**
  - Inserts the given file name into the current file. Often used for function prototypes and #define constants listed in a .h file. Allows functions and #define constants to be edited from one place (file) for an entire project
- **#define**
  - Replaces a variable name with with some other string. Also know as a macro. Used in the default code to create “aliases”.
- **#ifdef and #ifndef … #endif**
  - Conditional compilation of statements between the #ifndef and #endif
- **#pragma**
  - Compiler specific preprocessor directives. Refer to compiler documentation description of available pragma directives.

# PREPROCESSOR DIRECTIVES

Instead of putting a bunch of macros at the top of your code,

```
#define p1_y         rxdata.oi_analog01
#define p2_y         rxdata.oi_analog02
#define p3_y         rxdata.oi_analog03
#define p4_y         rxdata.oi_analog04
#define p1_x         rxdata.oi_analog05
#define p2_x         rxdata.oi_analog06
#define p3_x         rxdata.oi_analog07
#define p4_x         rxdata.oi_analog08
```

You can put them in a header file and #include the header at the top of your code.

```
#include "ifi_aliases.h"
```

This is much neater. Now you can put this one line at the top of all your C files and they will all have access to the macros (aliases).

# PREPROCESSOR DIRECTIVES

**Here are some really useful things you can do with preprocessor directives:**

```
#define MY_DEBUG  1

#if MY_DEBUG
  printf("Port1 Y %3d, \r", (int)p1_y);
  execute_regular_robot_code();
#else
  execute_regular_robot_code();
#endif
```

**The robot controller doesn't evaluate the "#if" at run time. The compiler looks at the "#if" condition and will compile the code depending on whether or not the condition is true. In the above case, the condition is true so the "printf" line will get compiled and become part of the software that is loaded into the robot controller. If the MY_DEBUG was set to 0 and the code was compiled again, the "printf" statement would not get compiled and not be part of the code downloaded into the robot controller.**

# ASSIGNMENTS

**No, this is not homework.  This is the technical term for the process of evaluating the expression to the right of a single equals sign (called the "assignment operator") and storing the result into the storage specified by the name to the left of the equals sign.**

Examples:

```
pwm08 = 127;

#define INPUT 1
digital_io_01 = digital_io_04 = INPUT;

pwm11 = Limit_Mix(2000 + p1_y + p1_x - 127);
```

# DECIMAL, BINARY, & HEX

$1 * 2^0 = 1$

$0 * 2^1 = 0$

$0 * 2^2 = 0$

$1 * 2^3 = 8$

$0 * 2^4 = 0$

$0 * 2^5 = 0$

$1 * 2^6 = 64$

$+ \; 1 * 2^7 = 128$

**201 Decimal**

**Binary**

**11001001**

| Binary | Hex |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

**Hex**

**C9**

# UNARY OPERATORS

| | |
|---|---|
| & | Address operator |
| * | Indirection operator |
| + | Unary plus operator |
| - | Unary minus operator |
| ~ | Bitwise Complement (1's complement)<br>(simply flips all the bits from 0 to 1 or 1 to 0) |
| ! | Logical negation ("NOT")<br>(turns a 0 into 1, turns non-zero value into 0) |
| ++ | Auto-increment |
| -- | Auto-decrement |

**Examples:**

```
relay8_fwd = !rc_dig_in18;  /* Power pump only if switch is off. */
trcnt ++; /* Same as "trcnt = trcnt + 1;" */
```

# BINARY OPERATORS

+, -, *, /, %    Add, subtract, multiply, divide, mod

<<, >>    Shift left, shift right

&, ^, |    Bitwise AND, XOR, OR

&&, ||    Logical AND, OR

<, >    Less than, greater than

<=, >=    Less than or equal to, greater than or equal to

==, !=    Equal to, not equal to

There are others but they're for more advanced C programming.

Note that some of the operators are made of two characters. The individual characters must not be separated by space or any other character otherwise it won't have the same meaning anymore.

# CONDITIONALS

| A | B | A \|\| B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | T | 1 |
| T | 0 | 1 |
| T | T | 1 |

| A | B | A && B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | T | 0 |
| T | 0 | 0 |
| T | T | 1 |

The conditions used in the if clause or the else clause must be expressions that can evaluate to a single number. The condition is “true” if it evaluates to any non-zero value. The condition is “false” if the expression evaluates to zero.

In the truth tables to the left the “T” represents any non-zero value.

When C evaluates the “A && B” expression, for example, the result is either a 1 or 0.

```
if (CONDITION 1)
{
    DO THIS;
}
else if (CONDITION 2)
{
    DO THIS;
}
else
{
    DO THIS BY DEFAULT;
}
```

# SWITCH - CASE

**The switch - case statement is a special case of an "if-else" structure that is commonly used. What if you were to have a large number of "else if" conditions based on a common variable? See how the switch-case helps.**

```
if (x == 0)
   function0();
else if (x == 1)
   function1();
else if (x == 2)
   function2();
else if (x == 3)
   function3();
else if (x == 4)
   function4();
else
   default_function();
```

**These two segments of code implement the same logic.**

```
switch (X)
{
      case 0 :
         function0();   break;
      case 1 :
         function1();   break;
      case 2 :
         function2();   break;
      case 3 :
         function3();   break;
      case 4 :
         function4();   break;
      default:
         default_function();   break;
}
```

# LOOPING: while

Generic while Loop

```
while ( condition == true )
{
    repetative_task;
 }
```

Infinite Loop

```
while ( 1 )
{
    endless_task;
 }
```

Wait for Something to Happen

```
while (!rc_dig_in01)
{
    /*  do nothing */
 }
```

- Statements between the while loop braces will be executed until the logical expression in parenthesis proves false.
- Avoid infinite loops caused by logical expression that never become false.  This will cause your program to “freeze”.
- The robot controller will also “freeze” if a section of code takes longer then 26 ms
- There are times when you’ll want to create an infinite loop on purpose.  Look at main() in the default code.
- You can use a while to wait for an event.  In the bottom example to the left, it’s doing nothing but continually checking the condition which of course will change only when digital input #1 sees 5 volts, possibly from a contact switch.

# LOOPING: for

Generic for Loop

```
for   ( counter = initial value;    counter <= someNumber;    ++counter )
{
     do_something_a_certain_number_of_times() ;
}
```

Real Example

```
 for  (i=1;   i<29;   i++)
{
  if (Breaker_Tripped(i))
       User_Byte1 = i;
 }
```

Wait for Something to Happen

But don’t wait too long

```
for (i=0;   i<20000;   i++ )
   if (data_rdy!=0 )   return 1;
```

A **for** loop handles all the details of counter-controlled repetition automatically.

1. The first thing the computer does when it enters a for loop is evaluate the first expressions “counter = initial value”
2. Then the loop condition “counter <= someNumber” is evaluate. If false the for loop exits.
3. The expression “++counter” is evaluated last before the loop condition is checked again

# DEBUGGING

**After you finish writing your logic to calculate the exact number to send to the wheels, you download it and test it on the robot. As soon as you give the robot some power it starts moving without joystick input. How can you find out why?**

**Well, you need to determine whether the problem is electrical or software.**

**You can add a line in your code to display the value being sent to the motors. If the value is not 127, then the software is telling the motors to turn. If the value displays as 127 (or very close to it), then you need to check the wiring or something else.**

**Another possibility is that there is another part of the program that is assigning a value to the motor that executes after your code essentially overwriting your value. Use the search feature of your text editor to find all the places in the code (in all C and H files) where anything writes to the motor.**

# DEBUGGING

**There's no monitor on the robot. How exactly do you display the values?**

**After you make the modifications shown below, download the program into the robot controller but leave the programming cable attached. The IFI_Loader has a terminal window that pops up. The "printf" statements in the code will send the data through the programming cable and display the values in the terminal window. Cool, huh?**

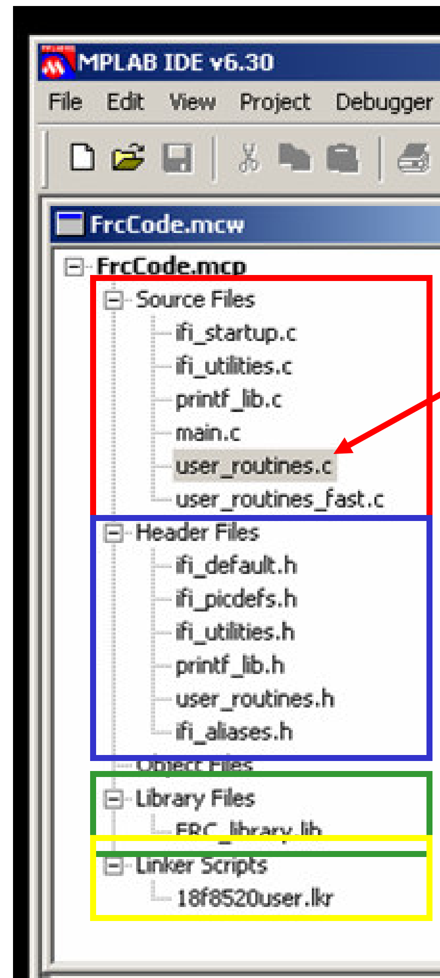
**Open the file, user_routines.c and find the end of the**

`Process_Data_From_Master_uP()` **function.**

**Just above the** `Putdata(&txdata)` **function call, place the following statement:**

```
printf("Port1 Y %3d, X %3d, \r", (int)p1_y, (int)p1_x);
```

# One project, many files…




- .c -Source Files
- .h -Header Files
- .lib -Library Files
- .lkr -Linker Scripts
- Edit the .c and .h files
- Compile
- Link

→ FrcCode.hex

# Files of the Default Code

- **“C” Files**
  - **These are the files containing the C-Language instructions. You will spend the bulk of your effort looking at and modifying these files.**
- **“H” Files**
  - **Also called, “header” files or “include” files, these files usually contain definitions and “function prototypes.”**
- **“Lib” Files**
  - **These are library files which contain software for frequently used functions, precompiled and assembled into a neat package. These files are not meant to be modified but linked with the rest of the files at compile time.**
- **“Lkr” Files**
  - **These are “linker” files. They include instructions for the linker so that code (instructions) and data (constants and initial variable values) gets placed in the flash memory in pre-determined, desired locations.**
- **“Hex” Files**
  - **These are the “image” files produced by the compiler/linker. Every time you “build” your software, if there are no errors, a single hex file will be produced. It will contain the machine language version of all your code in the order that the code needs to be in the flash memory. It is in a (text) format that makes it easy to download.**

# 18f8520user.lkr

The flash memory, where the program resides in the user processor, has 32 K (32768) addresses, or 8-bit storage locations . Each is identified by the processor by its address (a number from 0 to 32767). The linker file gives a name to different ranges of these addresses, identified by a START address and END address. In the C files, there are instructions that direct sections of code or data to belong to one of these named groups. As the compiler converts the C to machine language, every instruction or piece of data must reside at a particular address in the flash. Since only a name is known to the compiler, not a numerical address, it is up to the linker and the linker file to assign the addresses.

```
// $Id: 18f8520i.lkr,v 1.4 2003/03/13 05:02:23 sealep Exp $
// File: 18f8520i.lkr
// Sample linker script for the PIC18F8520 processor

LIBPATH .

//FILES c018i.o
FILES clib.lib
FILES p18f8520.lib

CODEPAGE   NAME=vectors   START=0x0        END=0x7ff       PROTECTED
CODEPAGE   NAME=page      START=0x800      END=0x7FFF
CODEPAGE   NAME=idlocs    START=0x200000   END=0x200007    PROTECTED
CODEPAGE   NAME=config    START=0x300000   END=0x30000D    PROTECTED
CODEPAGE   NAME=devid     START=0x3FFFFE   END=0x3FFFFF    PROTECTED
CODEPAGE   NAME=eedata    START=0xF00000   END=0xF003FF    PROTECTED

ACCESSBANK NAME=accessram  START=0x0       END=0x5F
DATABANK   NAME=gpr0      START=0x80       END=0xFF        PROTECTED
DATABANK   NAME=gpr1      START=0x100      END=0x1FF
DATABANK   NAME=gpr2      START=0x200      END=0x2FF
DATABANK   NAME=gpr3      START=0x300      END=0x3FF
DATABANK   NAME=gpr4      START=0x400      END=0x4FF
DATABANK   NAME=gpr5      START=0x500      END=0x5FF
DATABANK   NAME=gpr6      START=0x600      END=0x6FF
DATABANK   NAME=gpr7      START=0x700      END=0x7F3
DATABANK   NAME=dbgspr    START=0x7F4      END=0x7FF       PROTECTED
ACCESSBANK NAME=accesssfr  START=0xF60     END=0xFFF       PROTECTED

SECTION    NAME=CONFIG    ROM=config

STACK SIZE=0x100 RAM=gpr6
```

# FRC_default.hex

The format of this file is a standard one from Intel. It is used in many places all over the world in many industries. Loader programs understand the format and are sure to put the code and data in the correct locations within the flash memory.

All lines in the file begin with a colon. Most lines follow that with an address, then some code or data. Then at the end of each line is a checksum for making sure that the line of data received by the flash is the same one transmitted.

```
:020000040000FA
:0608000097EF1AF0120050
:020806000634B6
:060808006FEF14F0120076
:02080E00C250D6
:08081000020B01E0010E1200D1
:06081800A7EF18F012002A
:02081E000200D6
:100820008E350000C8070000040000000608000024
:08083000CE07000002000000E9
:0C08D400D9CFE6FFE1CFD9FFFD0EDB50CD
:1008E000AD6EBBEC16F0E552E7CFD9FF1200D9CFC1
:1008F000E6FFE1CFD9FF080EE126DE6ADD6A200EB1
:10090000F36E020EF3CFDBFF030EDB6AFB0EDBCFD1
:100910002BF0FC0EDBCF2CF02C50000AE8AE02D0FE
:100920002C3405D0000ED8802B54000E2C545AE2E3
:10093000040EDB6A050EDB6AFD0EDBCF1DF0FE0E3A
:10094000DBCF1EF0060E1DC0DBFF070E1EC0DBFF57
:10095000060EDBCFE9FF070EDBCFEAFFEF500BE01F
:10096000040EDB2A050E01E3DB2A060EDB2A070E46
:1009700001E3DB2AEDD7040EDBCF2BF0050EDBCF36
:100980002CF0FB0EDBCF2DF0FC0EDBCF2EF02C502D
:100990002E18E8AE02D02E3404D02D502B5C2E50F1
:1009A0002C5805E3FB0EDB6AFC0EDB6A10D0040E4C
:1009B000DBCF2BF0050EDBCF2CF0D950FB0FE96E0F
:1009C000FF0EDA20EA6E2B50EE5E2C50ED5AF90E37
```

# Building the Download File

MPLAB "Build" Button does all this in one step.

This is done with the separate program called, IFI_Loader

**Source Code Editor** → **Preprocessor** → **Compiler** → **Linker** → **Post-processor** → **Loader**

- **Source Code Editor**: Modify the contents of the source code files.
  - **user_routines.h**
  - **user_routines.c**
  - **user_routines_fast.c**
- **Preprocessor**: Modifies the version of the C files the compiler will see by expanding all macros, and generally executing all the pre-processor directives:
  - #define
  - #include
  - #if
  - #else
  - #endif
  - #ifdef
  - #ifndef
  - #pragma
- **Compiler**: Translate the human-readable program instructions into a form that is machine readable and executable by the target processor.
- **Linker**: Include pre-compiled code for library functions called from the source code. Assign specific addresses to all instructions and variables and functions.
  - **18f8520user.lkr**
- **Post-processor**: Convert the output code generated by the compiler/linker into Intel Hex 32 format for the downloader.
  - FRC_default.hex
- **Loader**: Download the software (in Intel Hex 32 format) to the flash memory of the robot controller's User MCU.

Files used by the Preprocessor and Compiler:

delays.h
ifi_aliases.h
ifi_default.h
ifi_picdefs.h
ifi_utilities.h
printf_lib.h
**user_routines.h**

ifi_startup.c
ifi_utilities.c
main.c
printf_lib.c
**user_routines.c**
**user_routines_fast.c**

**FRC_library.lib**
(if using the Generate_Pwms() function)

(or **FRC_alltimers.lib** if using Timer 0 separately from PWM generator)

# C LANGUAGE REFERENCES


Deital & Deital, **C How to Program 3rd Ed**. New Jersey: Prentice Hall, 2001.

David Maxwell, **First_C_2004_Full_Presentation** Online seminar, 2004. http://www.usfirst.org/robotics/C_help.htm

Innovation First, "**2004_Programming_Reference_Guide_10-29-2003.pdf**.", 2004. www.innovationfirst.com.

MicroChip, Inc., "**MPLAB_C18_Users_Guide_51288b.pdf**.", 2003. MPLAB_CBOT CD.

Brian W. Kernighan and Dennis M. Ritchie. **The C Programming Language, Second Edition**, http://cm.bell-labs.com/cm/cs/cbook/, Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).

Dennis M. Ritchie. **The Development of the C Language**, Bell Labs/Lucent Technologies,Murray Hill, NJ 07974 USA http://cm.bell-labs.com/cm/cs/who/dmr/chist.html

**C Language Tutorial**, http://www.physics.drexel.edu/courses/Comp_Phys/General/C_basics/c_tutorial.html

Eric Huss, **The C Library Reference Guide,** Release 1, 1997, http://www.acm.uiuc.edu/webmonke ys/book/c_guide/

# C LANGUAGE WEB SITES

**Borland C++ 5.5**

Download site to get the Borland C Compiler for free (for Windows).

http://www.borland.com/products/downloads/download_cbuilder.html

**C Programming**

Good "how to" guide (on-line) for C programming from Univ. of Strathclyde Computer Centre

http://www.strath.ac.uk/IT/Docs/Ccourse/

**Free C/C++ Compilers and Interpreters**

This page lists numerous free C and C++ compilers, cross-compilers and interpreters for a wide variety of operating systems.

http://www.thefreecountry.com/compilers/cpp.shtml

**Programming in C**

Programming in C with a bent toward UNIX users.

http://www.cs.cf.ac.uk/Dave/C/CE.html

**Programming in C: A Tutorial**

Written by Brian W. Kernighan, one of the inventors of the C language

http://www.lysator.liu.se/c/bwk-tutor.html

# OTHER REFERENCES

**From IFI Robotics (http://www.ifirobotics.com/)**

- **Control System Overview:** http://www.ifirobotics.com/docs/legacy/control-system-overview-2004-01-07.pdf
- **FRC Quick Start Guide:** http://www.ifirobotics.com/docs/legacy/frc_system_quick_start_2004-1-14.pdf
- **Robot Controller Reference Guide:** http://www.ifirobotics.com/docs/rc-ref-guide-01-31-2005.pdf
- **Operator Interface Reference Guide:** http://www.ifirobotics.com/docs/oi-ref-guide-1-25-05.pdf

**From Microchip (http://www.microchip.com/):**

**Summary of the PIC18F8520:**

http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1335&dDocName=en010319

**PIC18F8520 Datasheet:**

http://ww1.microchip.com/downloads/en/DeviceDoc/39609b.pdf

**Implementing a PID Controller Using a PIC18 MCU:**

http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en020434

**PIC18Fxx20 FLASH Microcontroller Programming Spec.:**

http://ww1.microchip.com/downloads/en/DeviceDoc/39583b.pdf

**PICmicro ® 18C MCU Family Reference Manual:**

http://ww1.microchip.com/downloads/en/DeviceDoc/39500a.pdf

**MPLAB IDE v7.20 and v7.22:**

http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469

**MPLAB C18 C Compiler (v3.0) and documentation:**

http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010014

# ROBOTICS REFERENCES

**Chief Delphi Forums**

Discussion group with a wide audience and recommended by Dave Lavery of NASA at the kickoff.

http://www.chiefdelphi.com/forums/

**FIRST Forums**

Online community for peer-to-peer Q&A

http://jive.ilearning.com/index.jsp

**FIRST Homepage**

Robotics, lego league, etc. program information

http://www.usfirst.org/

**Robot Documentation**

Control system, mechanical, master code, default code, programming tools, speed controllers, relay modules, and dashboard viewer

http://www.innovationfirst.com/FIRSTRobotics/documentation.htm

**Robot Documents (alternate site)**

Contains all the files FIRST provided, as well as the MPLab compiler (for a limited time), and ALL the videos that were available for download.

http://www.davedelong.com/first/

**Team Ford FIRST Homepage**

News, events, pictures, past presentations, discussion board, etc.

http://www.teamfordfirst.org/index.htm

**FIRST Robotics Gallery**

Pictorial history of past robots

http://firstrobotics.net/index.html

**Innovation FIRST Homepage**

Products, documents, white papers

http://www.ifirobotics.com/

**The 2004 Championship Event, Atlanta, GA**

Footage and photos from the 2004 national championship event

http://www.soap108.com/2004/events/cmp/index.cfm

# Introduction to C Language for Team Ford FIRST

Chris Teslak