

Bullbots Programming Training Session Dec 4th, 2007

Mentor - Randy Steiner
randy.steiner@eccs-com.net

Agenda

- Basic “C” Programming
- “C” programming Gotchas and Best Practices
- First Robotics Control Module Layout
- FRC Build Environment
- Sample Build and Load to Robot

Basic “C” Programming

Quick Tutorial on “C”

<http://www.loirak.com/prog/ctutor.php>

Good “C” Book

Practical C Programming by Steve Oualline
O'Reilly & Associates, Inc.

Basic “C” Programming - Basic

Computer Basic Definitions

Bit – a digital “1” or “0”. All a computer knows is a bit. Other concepts are “abstracted” over the top of this basic principle.

Byte – one of the abstract types. A byte is made up of 8 bits. (a “nibble” is 4 bits)

CPU – Central Processing Unit

Word – Another abstract type. Typically 2 bytes in size (16 bits)

Basic “C” Programming - Syntax

Data Types

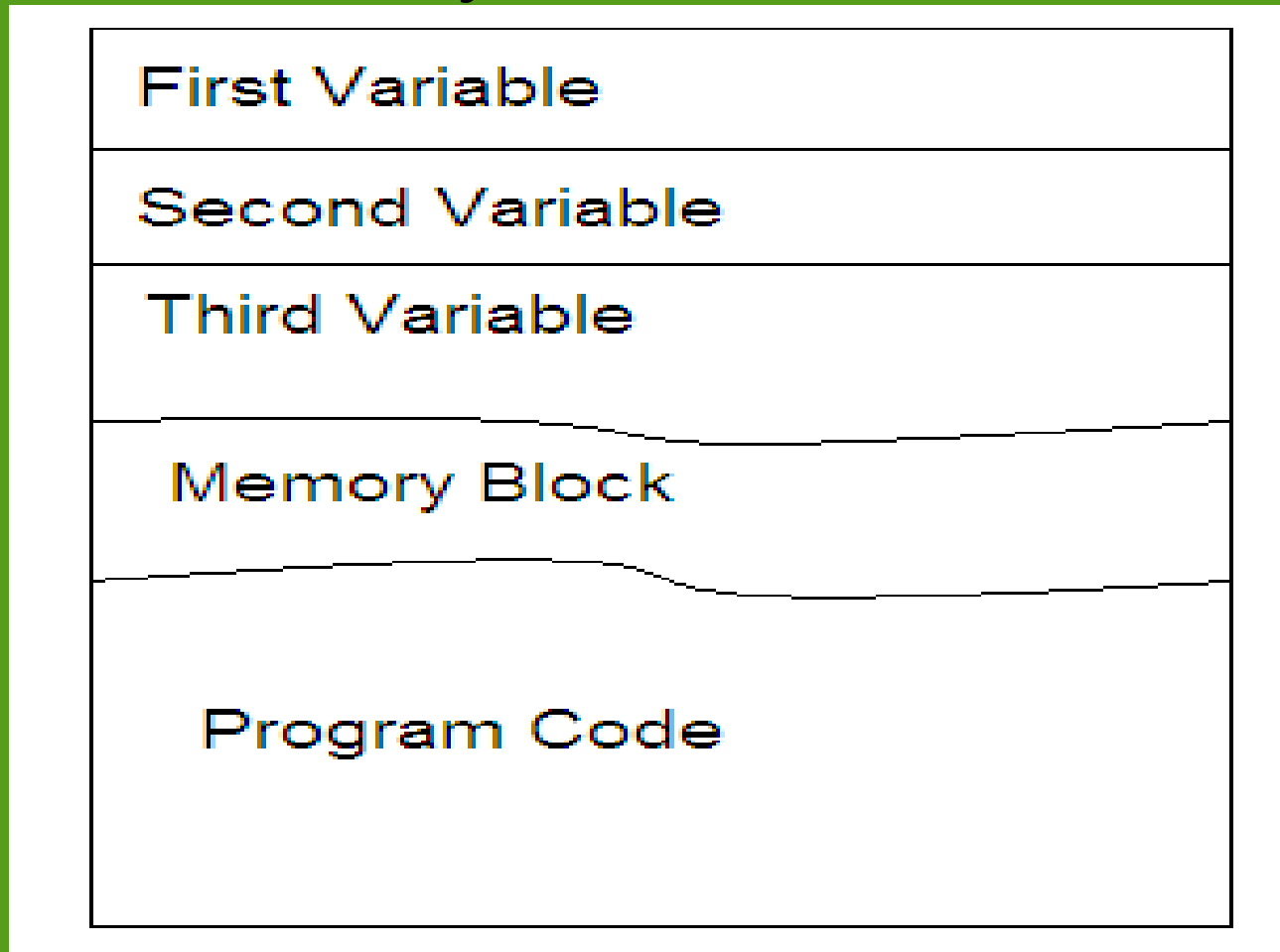
Data types tell the “C” compiler what type of data is being stored in a variable.

Variables are a place in the computers memory to store information.

Think of a variable as a mailbox. It can hold a certain amount of information, and it may be able to only hold a specific type of information.

Basic “C” Programming - Syntax

Variables in Memory



Basic “C” Programming - Syntax

Data Types - Continued

int – An Integer. This is a non-decimal number with a limited range. Based on the CPU architecture, may only be 2 bytes wide with a range of -32,768 to 32,767 or 0 to 65,535 (2^{16})

short – A “short” integer to force the size of the variable to exactly 2 bytes

long – A long integer to force the size of the variable to exactly 4 bytes with a range of -2,147,483,648 to 2,147,483,647 or 0 to 4,294,967,295 (2^{32})

Basic “C” Programming - Syntax

Data Types - Continued

A signed variable allows negative numbers.

An unsigned variable does not allow negative numbers.

NOTE!!! If you do this

```
int counter = 0;
while(1) {
    counter = counter + 1
}
```

counter will go from 0 to 32,767 and the next +1 will take it to -32,768. This is a risk to keep in mind when incrementing variables

Basic “C” Programming - Syntax

Variable “Scope”

Scope defines what parts of the program can “see” a variable.

Functions always have a global scope, but the compiler may not “see” the function if it is not defined in an include file.

Variables can have a “local” scope, or a “global” scope.

Basic “C” Programming - Syntax

Scope - Example

```
int counter = 1;
```

```
int my_function( int param ) {
```

```
    int local_counter = 1;  
    counter = counter + local_counter;
```

```
}
```

```
int function_two( void ) {
```

```
    int other_variable = 1;  
    counter = other_variable + local_counter;
```

```
}
```

This is an error. local_counter only exists in the scope of the my_func function

Basic “C” Programming - Syntax

Basic “C” File Structure

(includes)

(local function prototypes not in the header file)

(global definitions)

(local module definitions not in the header file)

(functions)

Basic “C” Programming - Syntax

Basic File Structure - Sample

```
#include <stdio.h>
```

```
void main( void );
```

```
#define VERSION_NUMBER 1;
```

```
int counter = 1;
```

```
void main( void ) {
```

```
    printf( “Hello World ---- version %d\n”, VERSION_NUMBER );
```

```
    return;
```

```
}
```

Basic “C” Programming - Syntax

Includes (also known as header files)

- Includes are other files “pulled into” this file and processed by the compiler before the rest of the files are
- Used to have a common definition of “parts” of the program
- Used to help with modules of a larger more complex program

Basic “C” Programming - Syntax

Include Example in Code File

```
/* comment at top of file – author, purpose, rev history*/
```

```
#include <usart.h>
```

```
#include <spi.h>
```

```
#include <adc.h>
```

```
#include <capture.h>
```

```
#include <timers.h>
```

```
#include <string.h>
```

```
#include <pwm.h>
```

```
#include "delays.h"      /*defined locally*/
```

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

```
#include "ifi_default.h"
```

```
#include "ifi_utilities.h"
```

```
#include "user_routines.h"
```

<filename.h> has a different meaning than “filename.h”. The <> mean it is a standard include file (means it's location is usually in a fixed location. The “” mean the include file is provided by the user, or is in a nonstandard location.

Any include file you create will be referenced by using “”. Any more details on this are outside of the scope of this training and if more information is desired, please reference many of the “C” tutorials on the Internet

Basic “C” Programming - Syntax

Included file – Example

```
/* comment at top of file – author, purpose, rev history*/
```

```
#ifndef __ifi_default_h_  
#define __ifi_default_h_
```

Compiler Directives
Covered Later
in this Training

```
#define MAX_SPIN_RATE 14  
#define OTHER_CONSTANTS 5
```

```
extern int spin_counter;  
  
int calc_spin_rate( int, int );
```

Extern is a special
keyword used to
expose a module's
“global” variables to
other modules

```
#endif
```

Basic “C” Programming - Syntax

Function Prototypes

Used by the compiler so it knows how to check for the correct usage by the program.

Checks the return value and the parameters so make sure the right data types are passed in.

Basic “C” Programming - Syntax

Function Prototype Examples

```
void main( void );
```

No return value and no parameters

```
int calc_spin_rate( int, int );
```

Returns an integer and takes 2 integer parameters

```
void set_PWM_rate( int, int, int, int );
```

No return value and takes 4 integer parameters

Basic “C” Programming - Syntax

Local and Global Definition Examples

Global Definitions are typically put in include files, unless it is a variable then use the “extern” special key word in the include file.

```
#define SPIN_LIMIT 30
```

```
int global_counter = 0;
```

```
long big_value_to_track = 0;
```

```
unsigned long really_big_value = 0;
```

Basic “C” Programming - Syntax

Functions Example

```
int calc_spine_rate( int spin_counter, int spin_time ) {  
    return spin_counter / spin_time;  
}
```

Why use functions? A function is used where a common “task” in the program is done more than once. This simplifies the code and allows for one place to make changes when the “task” needs to be changed. If it was in 5 places in the code and not in a function, then there is more risk for missing a change in the 5 locations the “task” exists.

Basic “C” Programming - Syntax

Assignments

The target on the left side of the “=” is where the result of the right hand side operations is stored

```
count = 0;
```

```
count = count + 1;
```

```
count = mycounter( 1, 15 );
```

Only can have a single target on the left hand side

Basic “C” Programming - Syntax

Conditional Statements

```
if( test condition ) {  
  
}  
else if ( test condition ) {  
  
}  
else {  
  
}
```

Basic “C” Programming - Syntax

Conditional Statements - continued

The test condition must result to a “boolean”. In C any nonzero value is a true, and a zero is a false.

```
if ( count > 0 ) { ... }
```

```
if ( result == 2 ) { ... }
```

Basic “C” Programming - Syntax

Conditional Statements - continued

NOTE!!! Assignments “=” and equates “==” are not the same thing!

```
if ( count = 1 ) { ... }
```

This will always be “true” as the assignment is putting a nonzero value in count and then the test is run.

```
if ( count == 1 ) { ... } is the correct way
```

Basic “C” Programming - Syntax

Conditional Statements - continued

switch is another conditional test. Used instead of several else if statements.

```
switch ( count ) {  
  
    case 10 : operation_a();  
             break;  
    case 20 : operation_b();  
             break;  
    default  : condition_z();  
             break;  
}
```

Basic “C” Programming - Syntax

Loop Structures

for(initialize; test; control)

for(j = 0; j < 100; j++) { ... } will execute the loop 100 times.

all three parts of the for loop are optional

for(; ; j++) { ... } is legal as well (must use a “break” statement to exit the loop.

Basic “C” Programming - Syntax

Loop Structures - continued

```
while ( condition ) { ... }
```

will execute a loop as long as the condition is true.
May never execute the loop structure if the condition is false to start with.

```
while( count < 100 ) { ... }
```

Basic “C” Programming - Syntax

Loop Structures - continued

```
do { ... } while ( condition )
```

Will execute a loop at least once and then test the condition at the end of the loop. Used when you want to execute at least one time.

```
do { ... } while ( count < 100 )
```

Basic “C” Programming - Syntax

Structures, Pointers, Pass by Reference and Pass by Value

These are more advanced concepts and may not be needed for the First Competition. If discovered to be needed we can cover these topics in a later session. Structures may need to be reviewed.

Basic “C” Programming

Build Process

Each “C” file is compiled to an “object” file. This is machine instructions.

After all of those are done, the build process “links” all of the object files together into the final target image. In this case it will be a HEX file for the First Robotics Control module.

Libraries used in the link process contain several pre-built object files.

Basic “C” Programming

Compiler Directives

These are special instructions for the compiler.

`#ifdef` is like an if statement. Only if the condition is true will the compiler include the code in the compile.

`#ifdef` can be used to “hide” test code in the file build of the project.

```
#define DEBUG 1
```

```
#ifdef DEBUG  
    printf( “This is a debug statement\n” );  
#endif
```

Basic “C” Programming

Compiler Directives - Continued

`#define` set a substitution for the compiler. This is a way to set a “constant” or rename a special function. If the compiler sees the text on the left, it replaces it with the text on the right.

```
#define SPIN_RATE 10
```

```
if( spin_speed == SPIN_RATE ) { ... }
```

The compiler turns into during the first pass

```
if( spin_speed == 10 ) { ... }
```

Gotchas and Best Practices

Mixing up “=” and “==”. This is a very easy error to make and can take a while to find. If things are not working right the expected way, look for errors like these.

Case. “C” is case sensitive. In other words “Counter” and “counter” would be seen as different variable names. This can lead to frustrating compile time errors with “undefined variable name” errors.

Gotchas and Best Practices

Style

Define a coding style for the team.

Does “{” appear at the end of a line, or on the next line?

```
while( test ) {
```

or

```
while( test )  
{
```

A decision the coding team should make

Gotchas and Best Practices

Style - continued

Anytime a “{” appears, or a complex concept, use indentation to make it clearer.

```
if( test ) {  
    activate_claw();  
    move_forward();  
}
```

Text is indented “4” space, or a “tab”

Gotchas and Best Practices

Style – Comments

- Code comments are critical!!! Be descriptive of what you are doing!!!
- Recommend you write comments first (pseudo code) then the final code. Allows to think about the flow, not the syntax.
- Pick a comment block style a stick to it. Use the First code as an example.
- “/*” begins a comment block and “*/” ends it. Then can not be nested.

```
/* set the control variable */
```

```
/* test if the robot has moved forward 10 feet */
```

First Robotics Control Module

Links

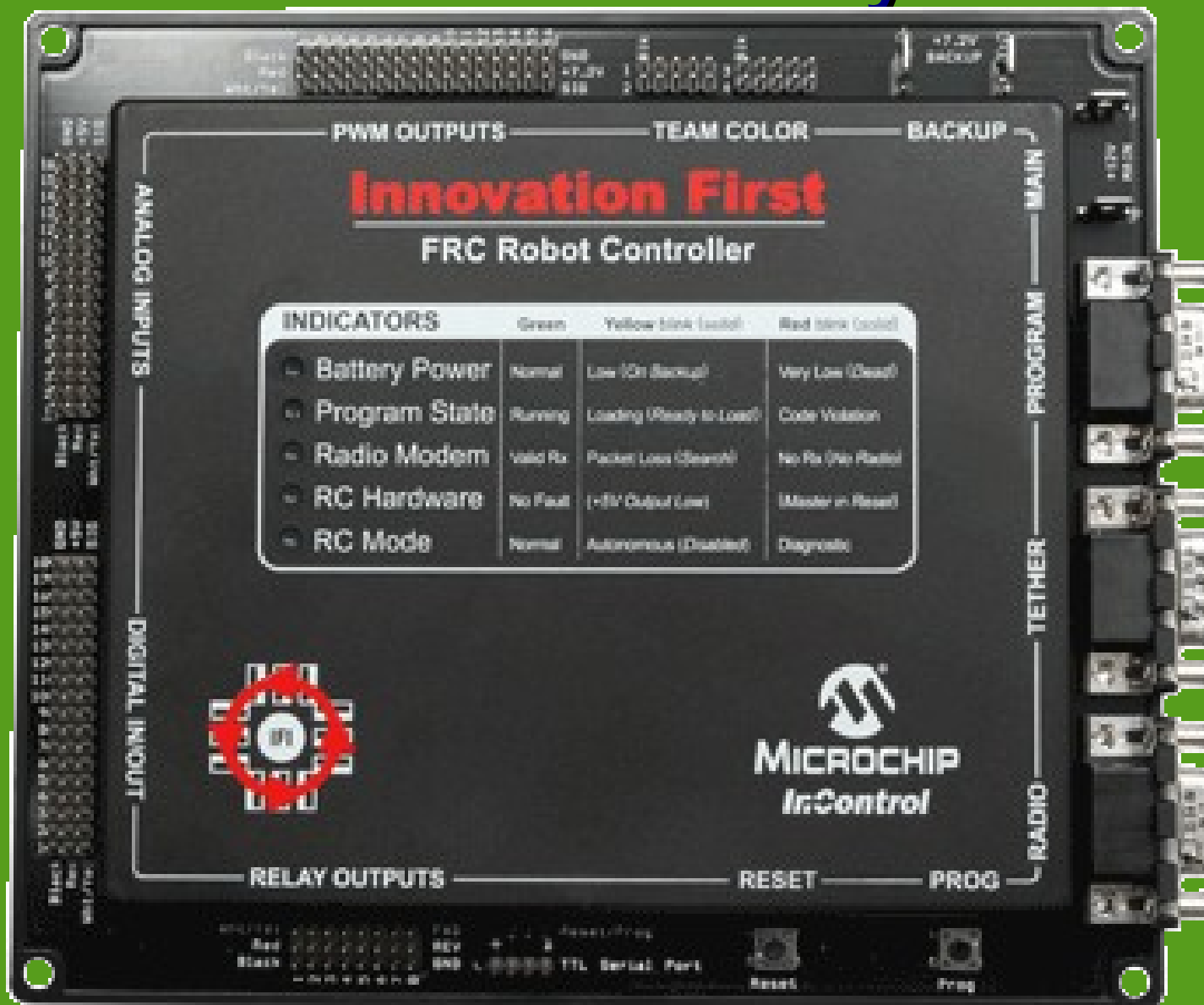
The Controller and default code links

<http://www.ifirobotics.com/rc.shtml>

Description of the key user_routines.c file

<http://frc.wikidot.com/user-routines-c-2007>

FRC Module Layout



FRC Module Layout

Notes:

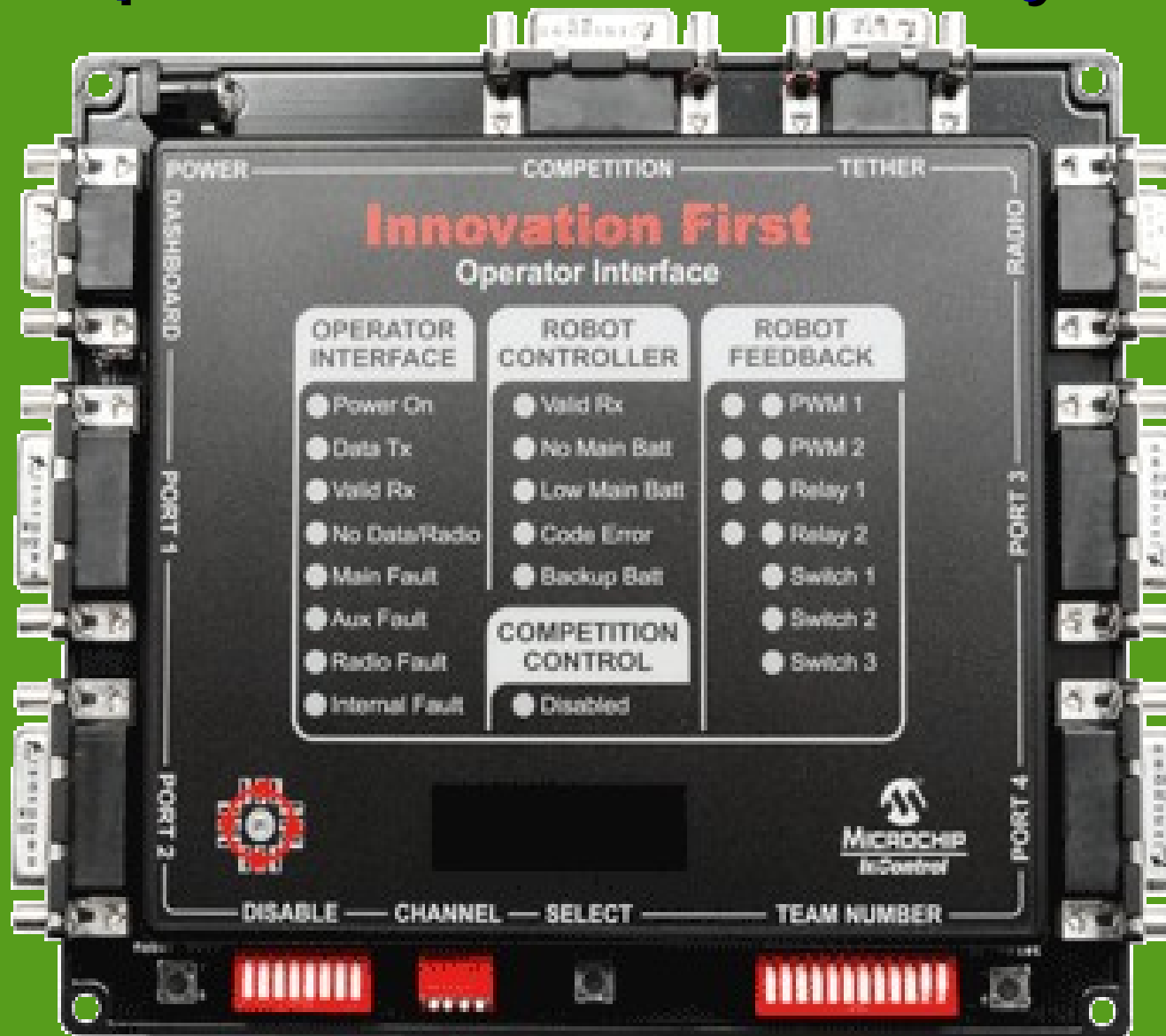
2 Microchip PIC CPUs

Master controls the Radio and several of the pins on the controller. Communicates internally with the user CPU. This is done with a packet exchange every 26.2ms. This is the fastest certain types of activities may run.

PWM Output 1-12 usually controlled by master

PWM Output 13-16 controlled by user CPU

Operator Interface Layout



FRC Module Layout

PWM

What is it?

How is it used?

(ref Robot Builder's Bonanza (Tab Electronics)
(Paperback) by Gordon McComb Chapter 20)

FRC Build Environment

Tools

MPLAB IDE v7.20

C18 Compiler version 2.40

IFI Loader V 1.1.0 (1.1.1 may work as well)

Can find these tools on the [jvex](http://www.jvex.com) site, or the [Microchip](http://www.microchip.com) site

<https://jvex-robotics.dev.java.net/ToolChainSetup.html>

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

Sample Build and Load to the Robot

Demo