WPILib

A Framework for Simplified Competition Robot Programming

Brad Miller

Worcester Polytechnic Institute
Robotics Resource Center

12 January 2008

Contents

Preface

WPILib was originally developed at Worcester Polytechnic Institute to make it easier for high school students to quickly learn robot programming on Innovation First Robovation and Radio Shack VEX controllers. The FRC (FIRST Robot Competition) version of the code was developed as a joint project between intelitek and WPI. This version is the base of the runtime system for intelitek easyC for FRC controllers.

Without the generous support from intelitek this project would not have been possible.

This software is designed to support three robot controllers developed by Innovation First – the FRC (FIRST Robot Competition) controller, the Innovation First VEX controller, and (hopefully soon) the Innovation First Robovation controller.

Introduction

What is WPILib?

WPILib is a library of functions to simplify programming of the family of Robot Controllers that have been developed by Innovation First. The goal in creating this library is to simplify a number of robot programming challenges:

- First and foremost, it's **easy to write programs** – just create a single file – your main program. As the complexity of your project grows, you can add additional files for other functions.
- The program structure presented by the standard **default code has been greatly simplified** and is not used to write programs with WPILib. There are no "fast" and "slow" loops and the master to user processor communications are completely done in the background for you (no GetData/PutData).
- WPILib is **completely modular**. Devices (gyros, accelerometers, etc.) can be added to the library without causing any modifications to user programs. New versions of the library don't require you to merge source code changes. You can share code with others.
- **WPILib is a library** – the only features included in your program are the features your program uses. When you link the program, the final .hex file only includes the pieces of the library that your program uses.
- **WPILib is portable** – the same library works with FRC (2005 and 2006), VEX, and Robovation controllers. This means you can prototype FRC robot algorithms on a VEX or Robovation controller and move the code with no changes.
- **WPILib is extensible** – the design philosophy is that experienced developers can create device drivers and modules that can be easily used by less experienced programmers.
- **WPILib includes common devices** – typical devices, such as those included in the FRC kit of parts or sensors sold by Radio Shack are supported "out of the box" in WPILib. There is no need to write additional code to use these devices.

WPILib lets you focus on making your robot do whatever it's supposed to do, and not on the implementation details of devices like camera and gyros.

What is easyC?

easyC is a graphical development environment for writing C programs that has been developed by intelitek. Using easyC inexperienced programmers have been able to successfully write C programs and learn programming concepts without getting bogged down with many of the details of C syntax.

It is highly recommended that less experienced programmers consider using easyC for their development. Using easyC you still get all the features in WPILib but in a very friendly environment.

WPILib is the runtime library that easyC uses to give the programs support of devices on the robots. The C programs that are created in easyC are automatically linked with WPILib.

A Simple Example

Let's start with a simple example: You want your robot to drive forward until it is about 2 feet from an object in front of it. We'll assume that this is a two motor drive robot with the left motor connected to PWM port 1 and the right motor connected to PWM port 2. There is also an ultrasonic rangefinder connected to ports 1 and 7. Here's the complete program to do this:

```
#include "BuiltIns.h"

void main(void)
{
	TwoWheelDrive(1, 2);			// define robot two wheels
	StartUltrasonic(1, 7);			// start ultrasonic rangefinder
	while (GetUltrasonic(1, 7) > 24)	// loop until about 2 feet away
	{
		Drive(100, 0);			// drive forward
	}
	StopUltrasonic(1, 7);			// turn off rangefinder
	Drive(0, 0);				// stop driving
}
```

That's all you have to do! Notice a few things:

- The `TwoWheelDrive` function defines the general architecture of the robot – two motors connected to ports 1 and 2. Once defined, the `Drive` function lets you drive and turn using a simplified model. Speeds range from -127 (full reverse) to 0 (stopped) to 127 (full forward). The program starts the robot driving. It continues driving at that speed until it is told to do something else, in this case stop at about 24".
- The ultrasonic rangefinder support is built-in; you don't have to do anything special to use it except turn it on, off and read values.
- There is only a single file – this is really everything you write, **no kidding**!

In the interest of full disclosure it should be pointed out that the robot will not really stop exactly 24" from the object. Depending on the speed of the robot, it will probably coast once the motors are stopped and overshoot the 24" target. But you get the idea.

Architecture

The microcontroller actually has two processors in it, a master processor where proprietary Innovation First code runs and a user processor which is, for the most part, reserved exclusively for user code. The two processor architecture exists for a number of reasons:

1. Tasks mostly dealing with communication with the operator interface for FRC and the PWM (radio) inputs on VEX are handled by the master processor. The master processor receives the data and passes it off to the user processor.
2. Handling of motors is done mostly by the master processor. Delegating this time sensitive work to the master processor improves the performance and reliability of

the program by offloading this work and it ensures that the motors will be stopped when the robot is disabled or there is no Operator Interface to control it.

The user to master communication happens periodically over a serial bus connecting the two processors. Every 26ms for FRC or 14ms for VEX the master processor sends the current state of the OI to the user processor and receives the current speeds for the PWM outputs to drive the motors. If this communication does not happen, the master processor will stop the user processor resulting in the flashing red code error light.

As you have seen, creating a simple program with WPILib is fairly straightforward. The parts of programs that are usually complex – interrupt and timed event handling is for the most part hidden in drivers.

WPILib is created in layers where each layer provides a higher level of abstraction to the layers below it.

1. IFI base code such as ifi_startup, ifi_library, ifi_utilities, etc.
2. WPILib timer and interrupt handling
3. WPILib included device drivers and field control
4. User written drivers
5. User written robot applications

WPILib is built as a library therefore only the components that are used in your programs are actually included thereby decreasing the size of the resultant hex file that is loaded into the robot.

Compiling and Linking Programs

WPILib has been used with easyC, MPLAB, and the Eclipse development environments. The only requirement is to include the library appropriate for the target architecture. The BuiltIns.h file includes definitions for most of the common devices that are available.

Microcontroller Support

WPILib is designed to work with all the PIC Micro based controllers designed by Innovation First. These include the 2005 and 2006 FRC controllers and the VEX controller. The same drivers are supported across all the controllers however there are differences in the port configuration on each.

Overall System Performance

With WPILib and easyC it becomes very simple to connect many sensors to your robot. However, you have to keep in mind that the processor inside the robot controller has limited performance and it is easy to overload the processor with many more tasks than it can handle.

This overloading of the processor will be seen when there are too many fast interrupting sensors connected *and operating* at the same time.

There are two factors that affect the ability of the system to handle interrupt loads.

1. The rate at which interrupts are generated. Devices that generate large numbers of interrupts will cause the system to fail. An interrupt service routine (the actual

code that handles an interrupt) takes a minimum of about 50us to handle an interrupt. This is due to the overhead imposed by the C compiler in saving and restoring the registers and other state. This translates to an upper limit of 20,000 interrupts per second.

2. The time required for an individual device. Quadrature encoders measure the direction by looking at the state of a second input channel. If the time from the actual hardware interrupt until the first line of the interrupt service routine is too long, then it won't be able to read the second channel in time and might incorrectly interpret the direction.

Here are some of the commonly used sensors with and their performance impact:

Device and condition	Demand on the robot controller
CMU Camera running at 115,200 baud (default)	11,520 interrupts/sec in bursts when it's sending packets
Wheel encoder (128 pulses/revolution, 6" wheel, robot moving at 10 ft/sec)	766 interrupts/sec/wheel
Gear tooth sensor	1 interrupt/tooth passage/sensor
Gyro (sampling rate is 50hz. Additional gyros split this time between all the running devices)	50 interrupts/sec
Accelerometer	
Compass	1000 interrupts/sec
Ultrasonic rangefinder	40 interrupts/sec

Table 1: Interrupt load for various sensors that can be connected to your robot.

What can you do to reduce the system load?

In general, try to reduce interrupt rates of all you devices and don't run them all at the same time. Try to turn off the camera when the other sensors are under heavy load. None of these devices generate any load on the system when they have been stopped. Each has a start and stop function that will retain any initialization data, but will suspend the device from operating. All devices are stopped after initialization.

Field Control

In an FRC competition when a robot is brought out to the field for competition its operation is partially controlled by the field state. For example, before the match starts the field is set to disabled and the autonomous state is false. In addition competitions often have autonomous periods, human player periods, and operator control periods.

With Vex competitions the robot operation is controlled by the transmitter. When the transmitter starts the robot runs the autonomous function followed by the operator control function.

WPILib makes it easier to develop software for competitions by reading the data sent to the robot controller and correctly responding to the environment. There are two ways of programming the robot for a competition:

1. WPILib controls the autonomous and operator control modes of the field by automatically running your code in one of three user created functions as the field conditions change.
2. The programmer takes complete responsibility for interrogating the field state and having his software work correctly.

In addition FRC and Vex each operate differently:

Period	Vex	FRC	What function runs
Placing robot on field before match starts	Robot is turned on, but the transmitter is disabled by the field	Robot is turned on, but the robot is in the disabled state.	Initialize()
Autonomous	Transmitter is turned on – timed autonomous period starts– Radio input is ignored by the robot	Field sets the robot to Autonomous mode through the OI - Radio input is ignored by the robot	Autonomous()
Operator control	Autonomous timed period ends and operator control starts	Field sets the robot to operator control mode	OperatorControl()
End of match	Either the timed operator period ends or the transmitters are again disabled by the field	The robots are disabled through the field controls and the OI	No code runs

Field Operating States

The field can either be in autonomous mode or operator control mode. Additionally there might be periods where the robot is temporarily disabled. In past games there was a short break between autonomous and operator control.

Programmer Field Control

In this type of program the programmer is responsible for knowing when the field is in autonomous mode and when it's in operator control mode. There are two functions that interrogate the information sent by the field controls:

```
unsigned char IsAutonomous(void);      // returns 1 if in autonomous mode
unsigned char IsEnabled(void);         // returns 1 if the robots are enabled
```

With these two functions the programmer can make the robot behave properly in all cases. You can consult the "Match progression table" table below to understand how the function results change at different points in the match.

WPILib Field Control

To use the WPILib field control the user simply creates three functions:

Function	What the function is expected to do
Initialize(void)	Called when the robot is first turned on but the field is in a disabled state. Typically before the start of a match the robot is placed on the field turned on. At this point the Initialization function will start running. The program should be initializing devices such as the gyro and camera so it finishes before the actual match starts. During this time the operator interface communication is running so initialization parameters can be set in switches on the OI that the robot can read. The initialization function must return when it is finished. *The program will not automatically enter the autonomous or operator functions until after the initialization function returns.*
Autonomous(void)	Called when the field is in autonomous mode and the robot is expected to run on its own. All the communications with the operator interface are disabled during this period. The robot can take advantage of all static data that was created during the initialization function.
OperatorControl(void)	Called when the field is in operator control mode. During this time the drivers are operating the robot and all communication is enabled between the robot controller and the operator interface.

Table 2: User written functions that are automatically called by WPILib as the field condidtions change.

Sample Competition Program

The following code is a sample competition program and can be used as a template for either Vex or FRC competitions. The only change to make it work for Vex is to replace the SetCompetitionMode() function with one that has two arguments, the autonomous time and the operator control time. You have to replace the contents of the Initialize, Autonomous and OperatorControl functions with code to run your robot.

```
#include "BuiltIns.h"

/*
* This function must be here for a competition project. It
* is automatically referenced by WPILib at startup and run.
* At that point the SetCompetitionMode function sets the
* competition mode. Basically, a mode of 0 is the default
* (without the IO_Initialization function) and runs a main
* function only.
* SetCompetitionMode(1) runs a competition project as shown.
*/
void IO_Initialization(void)
{
    SetCompetitionMode(1);
}

/*
* Initialize is run immediately when the robot is powered on
* regardless of the field mode.
*/
void Initialize(void)
{
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("Initialize %d\r", i);
        Wait(500);
    }
}

/*
* Autonomous is run as soon as the field controls enable the
* robot. At the end of the autonomous period, the Autonomous
* function will end (note: even if it is in an infinite loop
* as in the example, it will be stopped).
*/
void Autonomous(void)
{
    while (1)
    {
        printf("In autonomous\r");
        Wait(500);
    }
}

/*
* The OperatorControl function will be called when the field
* switches to operator mode. If the field ever switches back
* to autonomous, then OperatorControl will automatically exit
* and the program will transfer control to the Autonomous
* function.
*/
void OperatorControl(void)
{
    while (1)
    {
        printf("In OperatorControl\r");
        Wait(500);
    }
}

/*
* the main program is not used, but needs to be here to
* statisfy a reference to it. This requirement will probably
* go away in the next version of WPILib.
*/
void main(void)
{
}
```

Setting Competition Mode

To indicate that your program should run with the automatic sequencing of the above functions call the `SetCompetitionMode` function:

```
void SetCompetitionMode(unsigned char mode);   // FRC
void SetCompetitionMode(unsigned char autoTime, OperTime);  // VEX
```

The mode argument takes the following values:

Mode	Operation
0	Not a competition project. WPILib will not call your `Initialize`, `Autonomous`, or `OperatorControl` functions. Instead it will simply start your `main` function.
1 (FRC only)	Competition project. WPILib will use the inputs from the operator interface and field controls to determine which of `Initialize`, `Autonomous`, and `OperatorControl` functions should be called.
Number > 1	WPILib will treat this as a competition project, but instead of looking at the field or operator interface for the field state, it will simulate a tournament match by first running your `Initialize` function, then the `Autonomous` function will run for n seconds, where n is the argument to `SetCompetitionMode`, and then it will run your `OperatorControl` function.

On Vex projects the second parameter of the SetCompetitionMode function is the operator control time. If set, the robot will be disabled at the end of this period.

With Vex, a jumper in Interupt Port 5 will cause only the autonomous function to run and a jumper in Interrupt Port 6 will cause only the operator control function to run. This is used for testing these functions.

The `SetCompetitionMode` function **must** be called from a user created function named `IO_Initialization`. This function is automatically called by WPILib before either the main function or any of the competition functions. The following is an example of `IO_Initialization`:

```
Void IO_Initialization(void)
{
	SetCompetitionMode(1);		// set to do full field controls
}
```

If you don’t supply your own `IO_Initialization` function, one will be provided from the library that sets your program to non-competition mode.

Similarly an empty main() function must be supplied to satisfy a linker requirement. The sample program shown above has an example.

WPILib will constantly interrogate the state of the field and call the appropriate function. The programmer is not responsible for calling these functions. As an example, the following table describes what functions will be called in a match with a 15 second autonomous period, followed by a 15 second human player period followed by operator control.

What's going on?	Field Disable vs. Enable	Field Autonomous vs. Operator	Function called by WPILib
Robot placed on field and turned on	Enabled	Operator	`Initialize()`
Autonomous mode starts	Enabled	Autonomous	`Autonomous()`
Human player period	Disabled	Operator	`Operator()`
Operator control	Enabled	Operator	`Operator()`
Robot reset before match starts	Disabled	Operator	`Initialize()`
Robot reset (pressing robot reset on OI) while in operator mode	Enabled	Operator	`Initialize(); Operator()`

Table 3: Match progression table shows states of the field indicators and the user written functions that are called in each state.

The sequence of code (behind the scenes) inside WPILib for the match described above is essentially this:

```
Initialization();                 // WPILib will call your Init function
while (!IsAutonomous());          // WPILib will wait for autonomous to start
Autonomous();                     // WPILib will call your Autonomous function
                                  //   when the field starts autonomous mode
while (IsAutonomous());           // WPILib will wait until the autonomous
                                  //   period is over
OperatorControl();                // WPILib calls your operator function
While (1);                        // WPILib waits until the robot is shut off
                                  //   or reset
```

It's important to understand that the above sequence happens inside of WPILib in response to the field conditions changing. You do not need to write this code yourself. You do write the `Initialize`, `Autonomous`, and `OperatorControl` functions – those functions are called by WPILib.

When the field state changes WPILib will automatically take control out of the function that was running and call the next appropriate function. For example, if your `Autonomous` function doesn't return before the end of the 15 second autonomous period, WPILib will automatically redirect control to the `OperatorControl` function at that time.

Built-in Device Support

To help programmers get off to a fast start WPILib includes built-in support for a number of common devices. These devices are either available as accessories from Innovation First for the VEX Robot System, supplied by FIRST as part of the basic FRC kit of parts, or are commonly available components from robotics suppliers.

For these built-in devices the programmer does not need to be aware of the details of the device operation. This is handled by WPILib.

For devices not provided by WPILib advanced users can write their own device drivers by following the instructions in a later section of this document.

Accelerometer

The accelerometer can measure acceleration in milli-Gs (1/1000s of a G) in two axes.

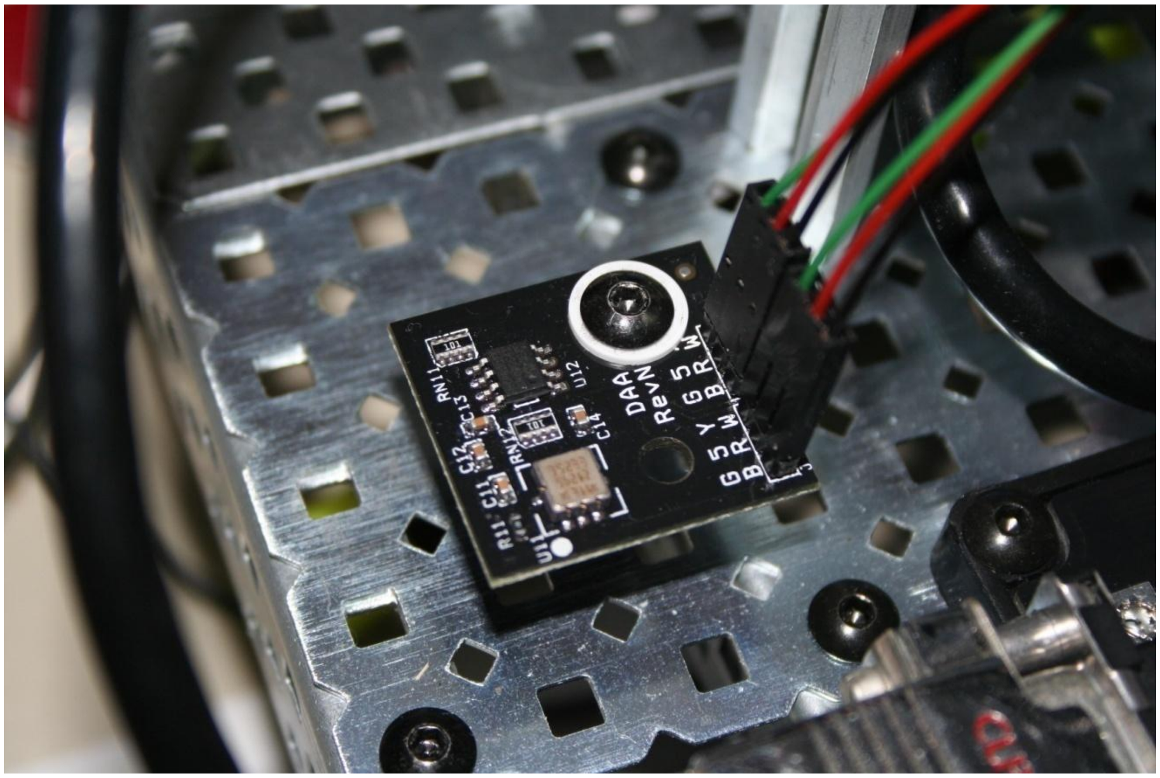


Figure 1: Accelerometer with connections for both the X and Y axis. Each axis is treated as a different accelerometer.

The accelerometer is treated as two devices, one for the X axis and the other for the Y axis. This is to get better performance if your application only needs to use one axis. The accelerometer can be used as a tilt sensor – actually measuring the acceleration of gravity. In this case, turning the device on the side would indicate 1000 milliGs or one G.

Functions

```
void InitAccelerometer(unsigned char port);
void StartAccelerometer(unsigned char port);
void StopAccelerometer(unsigned char port);
unsigned GetAcceleration(unsigned char port);
```

Before using the accelerometer is must be initialized. Initialization will compute bias for the device that will be used later when it is reading values. This is only done once.

Once initialized the program should call `StartAccelerometer` which begins sampling the analog input port that the accelerometer is connect to.

Calling `GetAcceleration` will return the most recent acceleration value that was retrieved from the device. The results are returned in milliGs. If the accelerometer was on its side, then it would indicate about 1000 representing one G.

Compressor and Pressure Switch

WPILib has built-in support for the FRC air compressor and pressure switch. The pressure switch is connected to the high pressure side of the pneumatic circuit and determines when the system is completely charged. WPILib reads the pressure switch every 500ms, and will turn a specified relay port to on/forward whenever the pressure is low.

Functions

```
void InitPressureSwitch(unsigned char pressureSwitchPort,
      unsigned char relayPort);
```

Once called (and it only needs to be called once – probably in the Initialization function), WPILib will continuously poll the pressure switch connected to `pressureSwitchPort` and set the relay output on `relayPort` to forward/on if the switch closes or off if the switch opens.

CMU Camera

The CMU Camera has an onboard microprocessor that helps it perform a number of functions. The color tracking function is directly supported by WPILib. To use the camera it must be initialized with all the parameters that define the object that is to be tracked. The camera supports over one hundred settings – only a small handful is supported by WPILib.

Once initialized the camera must be put into tracking mode by starting the camera. The camera will now continuously return a sequence of packets that define the position and size of the tracked region. Each of these parameters is returned as an unsigned byte.

Connecting the Camera to the Controller

WPILib assumes that the camera is connected to the TTL serial port on the robot controller. This connection is made using the supplied TTL serial adapter board supplied by Innovation First with the camera. It also requires power that should be supplied from an unused PWM port.

On FRC controllers the camera (PWM port) is powered from the backup battery. If the backup battery starts getting low the camera might behave erratically. Always be sure to use a fresh backup battery when the camera is being used.

Initialization

All the initialization parameters for the camera are stored in a `CameraInitializationData` structure. A pointer to an instance of this structure is passed to `InitializeCamera` and the parameters are sent to the camera.

```
void InitializeCamera(CameraInitializationData *data);
```

Getting Tracking Data

The tracking data is constantly being read from the camera in the background while the program is running. This data can be retrieved by copying the current record into an internal `TPacket` struct. To get the data call the function `CopyTrackingData` as shown:

```
TPacket *t = CopyTrackingData();
```

The most recent data will be copied from the incoming stream and a pointer to the `TPacket` struct will be returned. The struct contains the following data describing the tracked object:

Parameter	Description
mx	Middle of mass x value
my	Middle of mass y value
x1	Upper Left most corner x value
y1	Upper Left most corner y value

x2	Lower Right most corner x value
y2	Lower Right most corner y value
regionSize	The number of pixels in the tracked region, scaled and capped at 255: (pixels+4)/8
confidence	A measure of confidence in the tracked object: (# pixels / area) * 256. Where area is of the bounded rectangle
pan	Pan servo output value
tilt	Tilt servo output value

Table 4: Tracking packet (TPacket) return values as the CMU Camera is tracking color regions.

More detailed information on the initialization parameters and tracking results can be found in the CMU Camera user's guide.

Stop Tracking

The camera runs at 115,200 baud and generates approximately 11,500 characters per second while it is sending data. This high speed data stream may interfere with the operation of other sensors on your robot depending on the overall system load. It is therefore a good idea to stop the camera when the data is not being used. For example, if your robot picks up a heading to a target while it is stationary, then starts driving the camera could be stopped during the driving time. To stop the camera use the `StopCamera` function:

```
void StopCamera();
```

All data transmission between the robot and the camera will stop. You may also reinitialize the camera with new tracking data to change tracked targets.

You can turn on debugging output from the camera by calling the function:

```
void SetCameraDebugMode(unsigned char mode);
```

Setting the mode to a non-zero value causes all the camera initialization messages to be sent to the terminal window. This allows you to verify that the camera has initialized and that the hardware is connected properly and working.

Sample Program

The following sample program fragment will use the middle of mass x (centroid) value from the camera to drive a robot towards a target.

```
TwoWheelDrive(2, 1);            // two wheel drive robot – motors ports 1 & 2
InitCamera(&cameraInfo);        // address of CameraInitializationData struct
StartCamera();                  // start the camera tracking
while (1)
{
      int error;
      TPacket *t = CopyTrackingData(); // capture the tracking data
      error = ((int) 80) – t->mx;      // distance from center of frame
      if (t->confidence > 0)           // check if camera has target
            Drive(80, error);          // drive robot towards target
      else
            Drive(0, 0);               // stop the robot when no target seen
}
```

The program segment computes an error value that is positive or negative depending on the horizontal position of the center of the target. This value is used directly to generate a turn rate for `Drive`.

Compass

WPILib supports the CMPS03 digital compass which is available from a number of hobby robotics suppliers. Additional information on the device can be found from Robot Electronics at http://www.robot-electronics.co.uk/htm/cmps3doc.shtml. The compass is supported in PWM mode and not through the I2C interface. This device will return headings in degrees in the range 0-360. The PWM output signal from the compass must be connected to an interrupt port on the robot controller.

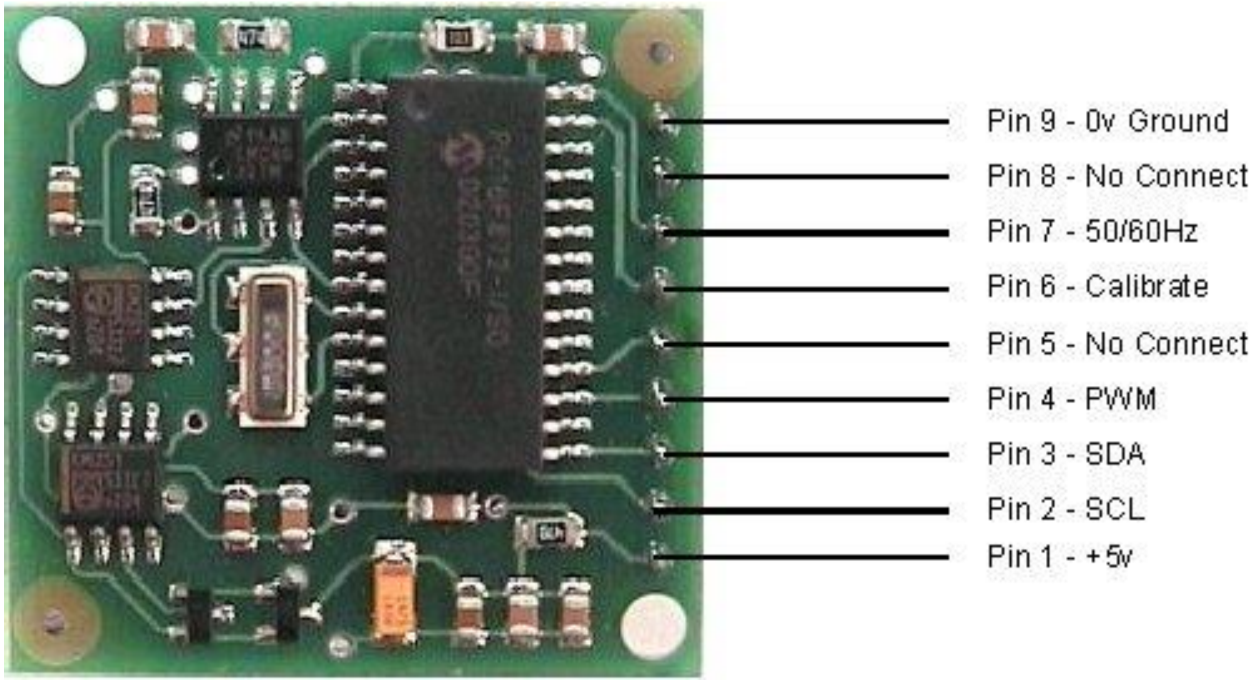


Figure 2: CMPS03 Compass drawing from the Robot-Electronics web site.

It is important to note that the compass uses the earth's weak magnetic field to compute the direction. You have to carefully place the compass on the robot and verify that it is not affected by motors and other electronics that might have their own much stronger local magnetic fields. It is possible that on a large FIRST robot there may not be any place on the robot that will give accurate results.

Functions

```
void InitializeCompass(unsigned char port);
unsigned GetCompassHeading(void);
```

The compass is initialized by calling `InitializeCompass` with the number of the interrupt port that it is connected to. To get headings, call `GetCompassHeading`. The resultant values are headings in the range 0-360 degrees.

Encoders

Background Information

Encoders are devices for measuring the rotation of a spinning shaft. Encoders are typically used to measure the distance a wheel has turned that can be translated into robot distance across the floor. Distance moved over a measured period of time represents the speed of the robot and is another common measurement for encoders. There are several types of encoders supported in WPILib.

Simple encoders	Single output encoders that provide a state change as the wheel is turned. With a single output there is no way of detecting the direction of rotation. The Innovation First VEX encoder is an example of this type of device.
Quadrature encoders	Quadrature encoders have two outputs typically referred to as the A channel and the B channel. The B channel is out of phase from the A channel. By measuring the relationship between the two inputs the software can determine the direction of rotation. The software looks for Rising Edge signals (ones where the input is transitioning from 0 to 1) on the A channel. When a rising edge is detected on the A channel, the B channel is read. If the encoder was turning clockwise, the B channel would be a low value and if the encoder was turning counterclockwise then the B channel would be a high value. The direction of rotation determines which rising edge of the A channel is detected, the left edge or the right edge.
Gear tooth sensor	This is a device supplied by FIRST as part of the FRC robot standard kit of parts. The gear tooth sensor is designed to monitor the rotation of a sprocket or gear that is part of a drive system. It uses a Hall-effect device to sense the teeth of the sprocket as they move past the sensor.

Table 5: Encoder types that are supported by WPILib

These types of encoders are described in the following sections.

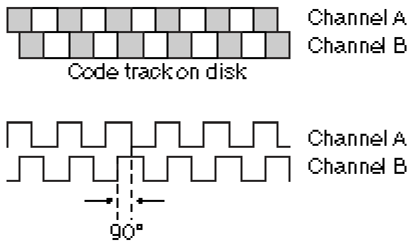


Figure 3: Quadrature encoder phase relationships between the two channels.

Performance

Encoders potentially have the ability to generate high interrupt rates while the motors are turned. It is important to attach them to your robot so that they turn at the minimum speed possible that will still give adequate resolution to meet your requirements. For example, a 128 count encoder returns 128 interrupts per encoder revolution. Suppose it is connected to the 6" wheel with no reduction. Each revolution of the wheel represents about 18.8" (π*6). Since there are 128 pulses per revolution then the resolution of these encoders is about 0.15" (18.8" / 128).

This resolution may be more than required for your application and it could have a profound effect on the performance of the program. Imagine if the robot can drive at 10 feet per second at full speed. That is about 6.4 wheel revolutions per second and 766 interrupts each second! Now imagine if you have encoders on all four wheels – that's 3064 interrupts each second for all the wheels (1 interrupt every 3ms).

How do you know if the encoder interrupt rate is too high?

There will be several possible errors:

- There might be missed counts – that is the calculated distances are less than the actual distances. This is because some counts never were registered.
- On quadrature encoders the returned motion of the robot direction might be backwards even though the robot is moving forwards. This is because the encoders are rotating so fast that by the time that the computer is reading the B channel on an A channel interrupt, it has already moved to the next transition.

What can you do about this?

Reduce the speed of the encoder by gearing it down or using some other method to reduce the encoder shaft speed. This will reduce the resolution of the encoder, but 0.1" accuracy is probably 10 times more than we need for our robots. Alternatively, use an encoder that generates fewer counts per revolution.

Quadrature Encoders

Two models of quadrature encoders have been tested although others should also work:

- Grayhill 63R128 – 128 pulses per revolution
- US Digital S2-1000-B – 1000 pulses per revolution

Each of these encoders has two outputs: an A channel and B channel. The A channel must be connected to digital input ports 1-6 (or interrupt port 1-6 for VEX) and the B channel can be connected to any digital input port 1-18 (1-16 for VEX).

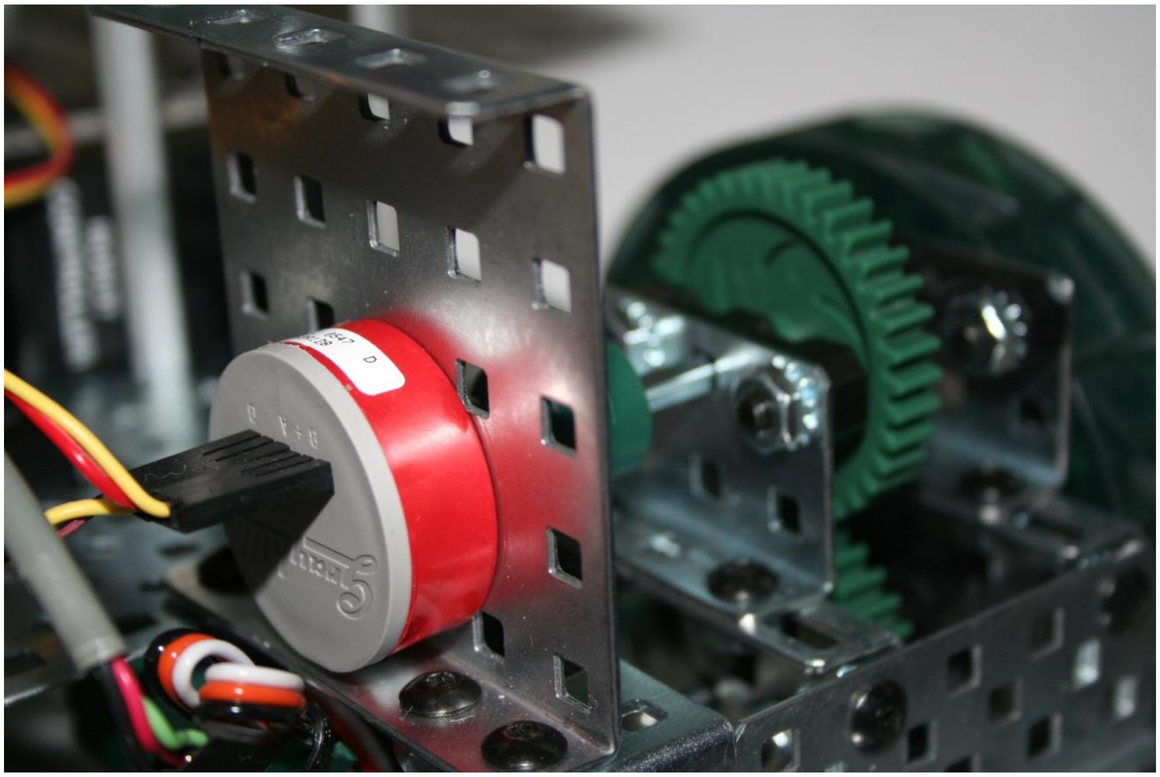


Figure 4: A Grayhill quadrature optical encoder mounted on a VEX robot chassis. Note the two connectors, one for the A Channel and the other for the B Channel.

Some quadrature encoders have an extra Index channel. This channel pulses once for each complete revolution of the encoder shaft. If counting the index channel is required for the application it can be done by connecting that channel to a simple encoder which has no direction information.

Functions

```
void StartQuadEncoder(unsigned char channelA, unsigned char channelB,
                      unsigned char invert);
void StartQuadEncoder(unsigned char channelA, unsigned char channelB);
void GetQuadEncoder(unsigned char channelA, unsigned char channelB);
void StopQuadEncoder(unsigned char channelA, unsigned char channelB);
void PresetQuadEncoder(unsigned char channelA, unsigned char channelB,
                       unsigned presetValue);
```

The encoder is first started – this starts sampling in the background. Each tick in the forward direction results in a count in the encoder. Each tick in the reverse direction subtracts one from the count. The results can be inverted (negated) by setting the invert argument in `StartEncoder` to 1.

Some encoders have an additional output called an index output. This creates a pulse every complete revolution of the encoder shaft. To use the index output create an additional simple encoder (VEX encoder) and connect it to the index output of your device. That value can then be read independently to get the number of complete revolutions.

Performance

It is strongly recommended to have the minimum number of counts per revolution as possible. Additional accuracy where it isn't needed will adversely affect the performance of the application. Additional interrupts might cause the program to miss other interrupts that are more important and give erroneous results.

Consider a typical 8" wheel and a robot traveling at 10 ft/min. The wheel has roughly a 24" circumference so at that speed the wheel is rotating 5 times per second. A 64 pulse / revolution encoder will a pulse for every 0.375" and about 5*64 or 320 pulses per second. That accuracy is more than adequate for most applications. A higher pulse count encoder or one that is on a faster moving part of the transmission is likely to cause problems with other parts of your software.

Gear Tooth Sensors

Gear tooth sensors are designed to be mounted adjacent to spinning ferrous gear or sprocket teeth and detect whenever a tooth passes. The gear tooth sensor is a Hall-effect device that uses a magnet and solid state device that can measure changes in the field caused by the tooth.

Due to limitations on the parts provided and the performance of the PIC microprocessor the gear tooth sensor direction output is not available.

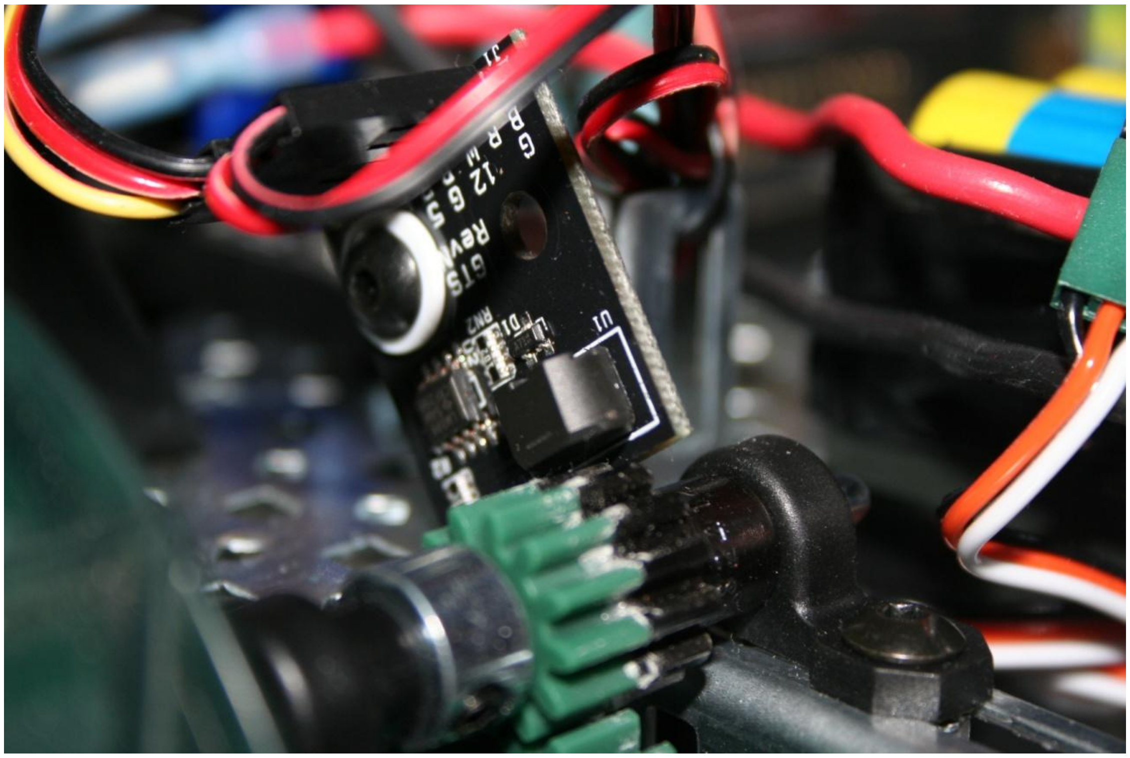


Figure 5: A gear tooth sensor mounted on a VEX robot chassis measuring a metal gear rotation. Notice that there is a metal gear attached to the plastic gear in this picture. The gear tooth Sensor needs a ferrous material passing by it to indicate rotation.

Using the Gear Tooth Sensor

The functions to use the gear tooth sensor are shown here:

```
void StartGTSensor(unsigned char port, unsigned char invert);
void StopGTSensor(unsigned char port);
long GetGTSensor(unsigned char port);
void PresetGTSensor(unsigned char port, long presetValue);
```

The gear tooth sensor is first started – this begins reading the sensor. The first time it is started the count will be zero. Subsequently starting and stopping the sensor does not reset the count. The initial value of the count may be set using the `PresetGTSensor` function.

Reading the value from the gear tooth sensor returns the number of ticks (positive for forwards and negative for backwards) accumulated. Each time a gear tooth passes in front of the sensor another count is added to the running total.

Note: the gear tooth sensor counts up regardless of the direction of rotation. This is significant when writing PID code that expects the sensor to show overshoot and correction of position.

Simple Encoders (Pulse Counter)

WPILib supports standard simple (non-directional) encoders such as the VEX encoders supplied by Innovation First. The VEX encoders cannot report direction of rotation so it is up to the programmer to determine this based on the direction the motors are rotating at any time.

Any other pulse generating device can be used with these functions. The functions simply increment the counter whenever there is a low to high (leading edge) signal on the specified input. A few examples of other devices that may be used are:

- The 2005 FRC kit included a gear tooth sensor that could be used with this device. With the appropriate logic it returns a pulse for every count.
- Counting closures of a limit switch is possible by connecting the limit switch so the input goes high on each closure.

Functions

```
void StartEncoder(unsigned char port);
void PresetEncoder(unsigned char port, unsigned long value);
void StartEncoder(unsigned char port);
unsigned long GetEncoder(unsigned char port);
```

The encoder (or other device) must be started first with the `StartEncoder` function. This function will set the count to zero on the first time. Subsequent stops and starts will not reset the count. To reset the count use the `PresetEncoder` function with a value that will be loaded into the encoder.

Gyro

A gyro, or yaw rate sensor, is a device that returns the instantaneous rate of rotation as an analog value. By integrating (summing) the rate over time the heading can be computed. The gyro driver runs in the background and is constantly summing the rate information and makes the current value of heading available whenever the program needs it. The Analog Devices line of MEMS gyros are supported by WPILib although others may work. An important consideration when using gyros is the maximum sensitivity. These gyros are rated in the maximum rate of turn that can be measured. The tradeoff is accuracy. Just as with a car speedometer, if the scale were to go to 300 miles/hour it would be harder to read small changes. If the scale only went to 20 miles/hour, there would be very good accuracy (easy to read small changes), but you would never know how fast the car was going beyond 20 miles/hour. It is the same with the gyro – a maximum sensitivity of, for example, 80 degrees/second will limit readings to that speed, i.e. a complete turn in about 4 seconds. Most robots can turn faster than that and would cause the gyro to lose track of its heading.

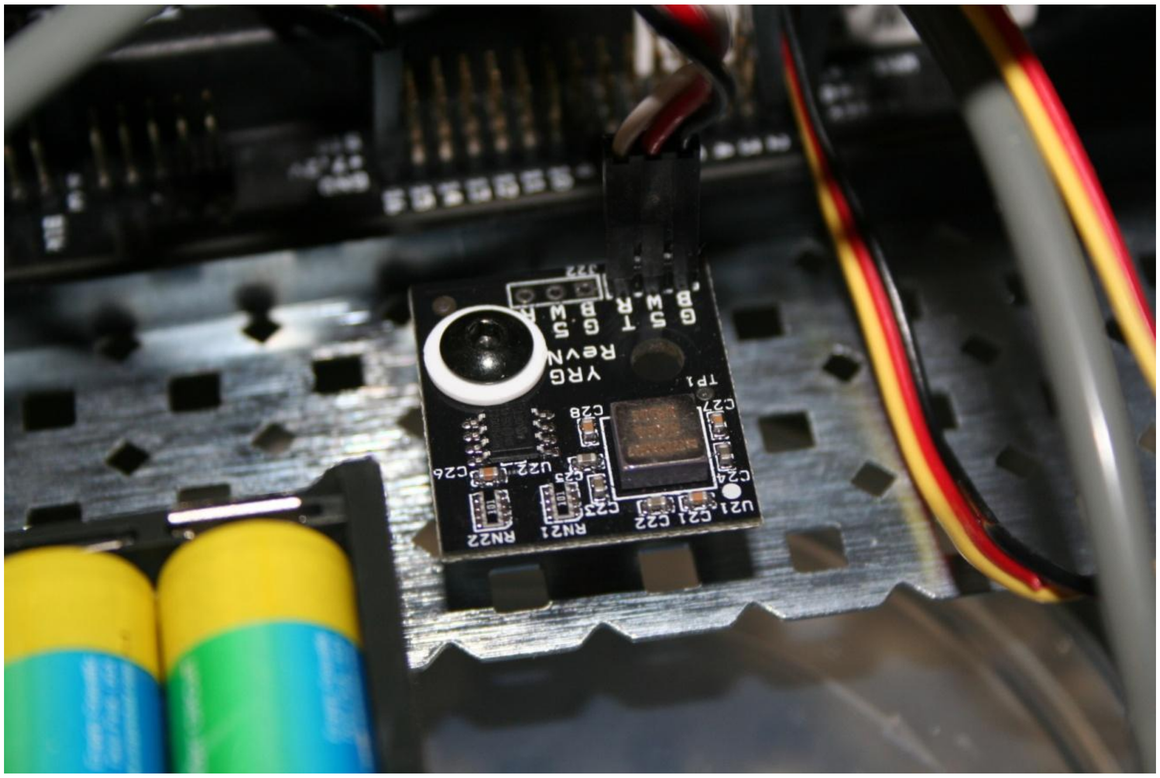


Figure 6: A gyro (yaw rate sensor) mounted near the center of rotation of a VEX robot chassis. The temperature compensation connections are unused.

To use the gyro it must first be initialized by calling the function `InitGyro`. This runs the gyro for 1 second and computes the offset or bias value for the gyro. The bias value is the zero point with no rotation that is subtracted from subsequent samples. During this time the robot must be held perfectly still. Normally this would be done as soon as the robot is turned on but before a match starts.

Once initialized the gyro is in a stopped state. To start the gyro the function `StartGyro` is called. Starting the gyro causes the rate value to be read periodically and the current heading to be calculated. A running gyro is constantly using the processor so even though the program may not be asking for headings, it is still taking processor time. The gyro can be stopped again if it is not being used by calling `StopGyro`. This will remove the load on the system but updated headings will not be available.

To get the current heading from a running gyro use the function `GetGyroAngle`. This function returns the heading, or angle of the robot in tenths of a degree. The heading is relative to the initial heading when the `StartGyro` function was called. Negative heading values indicate rotation to the left, positive values indicate rotation to the right.

Setting Gyro Attributes

You can set the gyro model by calling SetGyroType. Currently there are three common analog devices gyros supported. The values represent the device sensitivity so other devices might also work. The choices of types are listed in BuiltIns.h and are as follows:

```
#define ADXRS300    50
#define ADXRS150    125
#define ADXRS80     125
```

It is also possible to set the deadband for the gyro by calling SetGyroDeadband. These are are gyro values that are ignored by the software because they are considered too be noise. Increasing values of the deadband will decrease the drift but will decrease the accuracy. The default value is 3.

The gyro functions are shown below:

```
void InitGyro(unsigned char port);  // initialize gyro (1 second, no movement)
void StartGyro(unsigned char port); // start gyro background processing
void StopGyro(unsigned char port);  // stop gyro from background processing
int GetGyroAngle(unsigned char port);   // return angle in 0.1 degrees
void SetGyroType(unsigned char port, unsigned type);
void SetGyroDeadband(unsigned char port, char deadband);
```

Sample Program

The following program drives the robot in a straight line even if it is knocked off course as it is moving. If the robot is turned to the left or right the error will reflect that with a positive or negative value. The size of the error is proportional to amount the robot is off course. The correction is proportional to the error.

```
int error;                       // the error (actualHeading–desiredHeading)
TwoWheelDrive(2, 1);             // two motors, ports 1 and 2
InitGyro(1);                     // initialize gyro offset on port 1
StartGyro(1);                    // start sampling the gyro on port 1
while (1)
{
      error = GetGyroAngle(1);   // actual robot heading from gyro (0.1 deg)
      if (error < -70) error = -70;    // limit error to [-70,70]
      if (error > 70) error = 70;
      Drive(50, error);          // correction proportional to error term
}
```

Graphic Display Functions

In easyC V2 are set of Graphic Display (GD) functions that allow you much more control of the data "printed" in the terminal window. Gone are the days of being unable to read debugging output streaming by your eyes at high speed. Now you can draw output values at specific positions in the output window as well as drawing frames, or boxes, in which text can be displayed. You can even write programs that simulate bar graphs or display data in a position that's related to the values.

The functions to display data in the terminal window are shown below:

```
void ResetGD(void);
void ClearGD(unsigned char ucRow1, unsigned char ucCol1,
             unsigned char ucRow2, unsigned char ucCol2,
             unsigned char ucFrame);
void PrintTextToGD(unsigned char ucRow, unsigned char ucCol,
             unsigned long ulColor, rom const char *szText, ...);
void PrintFrameToGD(unsigned char ucRow1, unsigned char ucCol1,
             unsigned char ucRow2, unsigned char ucCol2,
             unsigned long ulColor);
```

To use the graphic display functions you must send the output using a robot tethered to a computer running the Intelitek easyC Pro terminal window set to graphics mode. When using the Graphic Display Functions the output must be displayed in the easyC terminal window. The IFI loader does not have the necessary code to interpret the graphic display output.

The Graphic Display functions tend to take a considerable amount of CPU time on the PC. You can slow down the display of each of the graphics display functions using the following function:

```
void SetGDWaitTime(unsigned time);
```

Setting a delay time sets causes each of the Graphical Display functions to pause before sending data.

Input and Output

All the supported controllers provide a number of digital I/O ports and analog input ports. Digital I/O ports operate on values that are logic level 0 or 1 (single bit input and output). Analog inputs read values from 0-5v and provide an output (0-1023) that is proportional to the voltage on the pin.

- On the VEX and Robovation controllers there are up to 16 I/O ports shared between digital and analog I/O. The 16 available ports are split between analog and digital – all ports with lower numbers than the split position are designated as analog inputs and all ports above the split position are for digital I/O.
- On the FRC controller there are 18 dedicated digital I/O ports and 16 dedicated analog input ports.

Digital Input and Output

```
void SetDigitalOutput(unsigned char port, unsigned char value);
unsigned char GetDigitalInput(unsigned char port);
void SetDirection(unsigned char port, unsigned char direction);
```

For all controllers it is necessary to set the direction for any ports used for digital I/O – identifying each as either used for input or output.

Analog Input

The robot controllers have up to 16 analog inputs.

- On the VEX controllers the analog inputs are shared with digital I/O pins and the program must select whether each of the pins is to be used for analog or digital data.
- On the FRC controller there are 16 independent analog inputs.

In all cases the analog inputs have 10 bits of precision (the values range from 0-1024) for input values from 0-5v.

```
unsigned int GetAnalogInput(unsigned char port);
```

Relays

The FRC controller has 8 relay outputs – each output has two independent outputs for forward and reverse signals.

```
void SetRelay(unsigned char port, char forward, char reverse);
```

Interrupts

Interrupts are built into the hardware of the PIC processor that is inside the robot controller. Devices connected to interrupt ports can cause a temporarily suspension of execution of the robot program and start executing a small piece of code that can respond to the interrupt. This code is called an **Interrupt Service Routine** (ISR). Interrupt Service Routines can be difficult to write due to critical timing and possible race conditions. Bugs associated with interrupt service routines are often very difficult to track down.

WPILib has two methods of handling interrupts to simplify coding:

Interrupt Watchers	Code built into the system that watches for state changes of hardware input ports and notifies the program that these values have changed. Interrupt watchers are easy to use and the preferable method of watching for quick changes in a device.
Interrupt Service Routines	Code that the user writes that is called whenever an interrupt on a designated port occurs. These are very difficult to write correctly and should be reserved for people with experience in real-time programming. Interrupt service routines are intended to help developers create device drivers that can be used by less experience programmers.

Interrupt Watchers

Often it is desirable to have the robot program notice when a device like a pneumatic position switch or limit switch changes state. The switch can be polled by testing its value every time through a loop of robot operation. If the switch changes state, and then changes back again very quickly it is possible to miss the state change while polling. One way of detecting this case is to connect the switch to an interrupt port. The problem with interrupt ports is that interrupt programming is difficult and can be error prone. WPILib introduces the concept of interrupt watchers. These are a set of functions that can watch for an interrupt on a particular port and latch the value so it can be read any time after the interrupt has occured.

You can have up to 6 interrupt watchers that can monitor devices connected on FRC Digital I/O ports 1-6 or VEX Interrupt ports 1-6.

Using an Interrupt Watcher

Here is an example of using an interrupt watcher:

```
// initialize interrupt watcher on port 1 looking for a rising edge
//    this is an input going from low to high (like a switch opening)
StartInterruptWatcher(1, RISING_EDGE);
while (1)
{
      // your program would be doing lots of work here and couldn’t afford
      // the time to poll the digital input port that was being watched
      //
      //
      // check if the value has changed since the last check
      if (GetInterruptWatcher(1) == 1)
      {
            printf(“Interrupt has occurred on port 1\r”);
      }
      //
      // program doing more work over here.
      //
}
```

It is important to note that in the sample program the robot is not doing anything besides waiting on the interrupt watcher. If that was really all that was happening it would be more efficient to just poll the input port waiting for the state to change. Normally there would be lots of computation going on in the loop that would prevent the program from just watching the port.

The interrupt watcher will latch when the value of the device connected to port 1 has changed from 0 to 1 (`RISING_EDGE`). This means that if the value changes from 0 to 1, then 1 to 0 before you could check, `GetInterruptWatcher(1)` will still show the latched value.

To start looking for falling edges (going from 1 to 0) instead, call `StartInterruptWatcher` again, but this time with `FALLING_EDGE` as the second parameter.

Motors

There are three sets of functions to drive the motors on your robot in WPILib.

1. Functions that drive the PWM outputs directly with the hardware values of 0-255
2. Functions that take a description of the robot and have a simplified interface with signed speed and direction values ranging from -127 to 127.
3. Composite functions that automatically take input from the operator interface and drive the robot in either tank or arcade mode.

Direct PWM Output

The PWM outputs can be used for controlling motors and servos. The range of values accepted by a PWM varies from 0 for full speed in one direction to 127 for stopped to 255 for full speed in the opposite direction.

To set PWM values use the SetPWM function with a PWM port number and a value.

```
void SetPWM(unsigned char port, unsigned char value);
```

Robot Based Driving

WPILib has a set of motor functions designed to operate simple robots using a more intuitive set of driving functions. These are designed primarily for autonomous operations but work equally well with operator controls. There are several changes from the traditional driving functions:

- The definition of the robot is supplied identifying whether it is two motor drive or four motor drive. The list of ports is retained by WPILib to make subsequent functions easier to write.
- The individual motor speeds are abstracted into a single robot speed value that sends the appropriate values to the left and right sets of motors. The speed values range from -127 for full reverse to 127 for full forward and 0 for stopped. WPILib takes care of the bookkeeping of reversing the values on one side for opposite mounted motors.
- Turning is handled by supplying a turn direction rather than having to compute the offsets to each motor although that is available also. The turn values range from -127 for full left turn to 127 for full right turn with 0 for no turn. These values are used on the Drive function described below.

Functions

Robot description functions are required before driving to set up WPILib for the configuration of your robot.

```
void TwoWheelDrive(unsigned char leftMotor, unsigned char rightMotor);
void FourWheelDrive(unsigned char leftMotor, unsigned char frontLeftMotor,
             unsigned char rightMotor, unsigned char frontRightMotor);
void SetInvertedMotor(unsigned char port);
```

These functions define the robot and the attachment of the motors. By defining the placement of the motors the programmer doesn’t have to remember later how the motors are connected to the robot controller.

`TwoWheelDrive` defines a robot with two drive motors, one on the left and the other on the right. `FourWheelDrive` defines a robot with two motors on each side. In four wheel drive, both the front and rear motors on a side are driven in the same direction.

Sometimes motors are connected with transmissions that reverse the operating direction of the motors. In this case, any motor can be inverted by calling the `SetInvertedMotor` function and specifying the port the motor is connected to.

Driving functions actually control the motors and are shown here:

```
void Drive(int speed, int direction);
```

The `Drive` function computes the left and right motor speed by adding the direction to one motor and subtracting it from the other. This is typically how mixing is done and is the same method as the arcade steering functions use for translating a single joystick x and y values into direction and speed.

`Drive` effectively looks like this:

```
Void Drive(int speed, int direction)
{
	Motors(speed + direction, speed – direction);
}
```

Notice that if the speed + direction or speed - direction exceeds 127 then the value will be greater than the speed controls can accept. To solve this problem, `Drive` limits the direction to

```
	128 - abs(speed)
```

If you supply a larger value for direction, `Drive` will limit it automatically. This might not be the desired behavior so programs should take care to not allow the values to get out of range.

```

void Motors(char leftMotorSpeed, char rightMotorSpeed);
```

The `Motors` function takes values for the left and right motors. These values are assumed to range from -127 to 127. If there are two motors on each side, they are both driven at the right speed. The `Motors` function takes into account the left/right bias of the motors therefore motors on the left side of the robot will turn in opposite directions from motors on the right. The `Motors` function is especially useful for implement tank driving algorithms.

```
void Motor(unsigned char port, char speed);
```

The `Motor` function drives a single motor at the specified speed. It will convert the -127 to 127 values to 0-255 for the hardware. Also a dead band calculation is applied to smooth out the motor speeds. It is assumed that speeds between -3 and +3 are stopped and will cause 127 to be sent to the motors. Also the motor speed range is ramped slightly to take into account poor slow speed performance.

Example Program

The `Drive` function is particularly useful for autonomous operation. Typically the program has a target speed or direction. Suppose we want to drive in a straight line using a gyro for correction. The difference between the actual heading (read from the gyro) and the desired heading is called the error term. Using the `Drive` function, this error term can be applied directly to the robot. Here is an example:

```
int error;                          // the error (actualHeading–desiredHeading)
TwoWheelDrive(2, 1);                // two motors, ports 1 and 2
InitGyro(1);                        // initialize gyro offset on port 1
StartGyro(1);                       // start sampling the gyro on port 1
while (1)
{
       error = GetGyroAngle(1);  // actual robot heading from gyro (0.1 deg)
       if (error < -70) error = -70;    // limit error to [-70,70]
       if (error > 70) error = 70;
       Drive(50, error);          // correction proportional to error term
}
```

The desired heading is 0 degrees since the goal is to drive straight ahead. Therefore the error is:

`GetGyroAngle() - 0` or just `GetGyroAngle()`.

There are two important things to notice about this simple example:

- Notice that if the gyro angle becomes negative, it indicates a turn in one direction and if it is positive it indicates a correction in the other direction. This is exactly what the `Drive` function takes for its direction argument.
- As the error increases, the rate of turn increases. This makes the robot correct more quickly as it gets further off track. This is simple proportional control and is often sufficient for operating robot motors.

Composite Operator Interface to Motor Driving

WPILib supports a number of functions to make setting up common driving configurations very simple to implement. With these functions the programmer maps joystick ports directly to motors.

There are two basic types of driving:

Tank steering	The left and right joystick y-axis control the speed (forward and reverse) of the left and right set of wheels. Tank steering has been much more commonly used for FIRST robots.
Arcade steering	Only a single joystick is used where the speed is mapped to the y-axis on the joystick and the turn rate is mixed in from the x-axis.

The method used is a driver preference choice – some people prefer tank steering and other prefer arcade steering. Arcade steering only uses a single joystick so it has an advantage in applications where there is limited space on the operator interface.

There are versions of the functions for two motor drive robots and four motor drive robots. The four motor versions simply send the same values to the front and rear motors on each side.

Functions

```
void Arcade2(unsigned char movePort, unsigned char moveChannel,
             unsigned char rotatePort, unsigned char rotateChannel,
             unsigned char leftPWM, unsigned char rightPWM,
             unsigned char leftInvert, unsigned char rightInvert);

void Arcade4(unsigned char MovePort, unsigned char ucMoveChannel,
             unsigned char RotatePort, unsigned char RotateChannel,
             unsigned char LeftfrontPWM, unsigned char RightfrontPWM,
             unsigned char LeftrearPWM, unsigned char RightrearPWM,
             unsigned char LeftfrontInvert, unsigned char RightfrontInvert,
             unsigned char LeftrearInvert, unsigned char RightrearInvert);

void Tank2(unsigned char leftPort, unsigned char leftChannel,
             unsigned char rightPort, unsigned char rightChannel,
             unsigned char leftPWM, unsigned char rightPWM,
             unsigned char leftInvert, unsigned char rightInvert);

void Tank4(unsigned char LeftPort, unsigned char LeftChannel,
             unsigned char RightPort, unsigned char RightChannel,
             unsigned char LeftfrontPWM, unsigned char RightfrontPWM,
             unsigned char LeftrearPWM, unsigned char ucRightrearPWM,
             unsigned char LeftfrontInvert, unsigned char RightfrontInvert,
             unsigned char LeftrearInvert, unsigned char RightrearInvert);
```

Each of these functions takes a set of joystick operator interface ports and a set of PWM ports that these are mapped to. In addition each motor (PWM) port can be inverted since motors may change directions as a result of transmissions and drive line designs. The arguments are:

movePort	The joystick port that controls forward and backward movement in arcade steering.
moveChannel	The channel on the move port that controls forward and backward movement (usually y-axis).
rotatePort	The joystick port for rotation in arcade steering
rotateChannel	The channel on the rotate port that controls left and right rotation. This value is mixed with the move port/channel.
leftPort leftChannel	The joystick port and channel that control the forward and backward movement in tank steering mode for the left side of the robot. This is generally the y-axis for the joystick.
rightPort rightChannel	The joystick port and channel that controls for forward and backward movement in tank steering mode for the right side of the robot. This is generally a y-axis of the joystick.
leftPWM	The motor or motors on the left side of the robot.
rightPWM	The motor or motors on the right side of the robot.
leftInvert	Value should be 1 if the motor or motors connected to the left PWMs will be inverted. The value sent to the motors will actually be 255 – value.
rightInvert	Value should be 1 if the motor or the motors connect to the right PWMs will be inverted. The value sent to the motors will actually be 255 – value.

Every time one of these functions is called, the values from the operator interface will be mapped to the appropriate motors. Therefore the function should be called repeatedly to get continuous operation. Calling the function only once will drive the motors with the OI values only once, and the motors won’t change as the robot is driven.

More complex drive configurations such as holonomic systems require custom code.

Operator Interface (Getting Driver Information)

The robots that are supported by WPILib can be remotely controlled. There are three types of controls:

- VEX robots come with a 6 channel transmitter (and can be extended to 12 channels by purchasing a second transmitter/receiver pair).
- FRC robots have a very sophisticated **operator interface** (OI) that uses bi-directional communications to send data to and from the robot controller. The OI has 4 input ports that can be connected to either analog joysticks or other user built devices. In addition the program running on the robot controller can send data back to several LED indicators on the OI.
- Robovation robots have PWM inputs that are compatible with standard ground RC receivers such as those used with model cars.

Field Interface

In many robot competitions there are time limits placed on a match. The matches are often started through controls external to the robots. For example, in FRC competitions there is a sophisticated field control system that sends signals to the robot controllers that can be read by the program that provide information about the state of the match and the robot – autonomous or tele-operated, and enabled or disabled.

WPILib has a set of functions to make this information available to the program.

Functions

```
unsigned char IsAutonomous(void);
unsigned char IsEnabled(void);
```

Operator Interface to Device Functions

In addition to the standard driving functions there is another set of functions for controlling devices directly from the Operator Interface. These functions take an OI value or a joystick switch and send its value directly to an output port.

Functions

```
void OIToRelay(unsigned char port,
             unsigned char function,
             unsigned char relayNumber,
             unsigned char direction);

void OIToPWM(unsigned char port,
             unsigned char function,
             unsigned char pwm,
             unsigned char invert);

void OIToDOutput(unsigned char port,
             unsigned char function,
             unsigned char dport);
```

The functions work as follows:

OIToRelay	Takes OI digital inputs (triggers and buttons) and controls the specified relay.
OIToPWM	OI joystick values are sent to the specified PWM port. This is useful for operations like driving an arm or other mechanism that is motor powered and controlled with a joystick.
OIToDOutput	OI digital inputs (triggers and buttons) control digital outputs. Operating the buttons will toggle the digital output values.

Operator Interface Status Indicators

The Operator Interface has a number of status indicators that can be controlled by the robot controller. These include a number of LEDs and a 3 digit display.

Functions

```
void SetOILED(unsigned char led, unsigned char value);
void SetUserDisplay(unsigned char value);
```

The Operator Interface can either show the LED values or the User Display value but not both at the same time. The user display is selected on the OI but cycling through the OI display options using the button on the OI. When the User Display is being displayed the LEDs are not updated.

WPILib determines when sets of values are being displayed by the UI and sends the correct values.

The `led` argument in `SetOILED` sets the LED being controlled and is one of the following values:

```
#define PWM1_GREEN 0
#define PWM1_RED 1
#define PWM2_GREEN 2
#define PWM2_RED 3
#define RELAY1_GREEN 4
#define RELAY1_RED 5
#define RELAY2_GREEN 6
#define RELAY2_RED 7
#define SWITCH1_LED 8
#define SWITCH2_LED 9
#define SWITCH3_LED 10
```

The LED value is either a 0 or 1 to turn the chosen light on or off.

Serial Port

The robot controller has two serial ports. Port 1 is generally used for program downloading and debugging. All `printf` output is directed back through serial port 1 by default. This is the port with the DB-9 connector (on the FRC robot controllers) or the modular phone jack (on the VEX robot controllers). Port 2 runs at TTL levels and is generally used for the CMU Camera.

Both serial ports are initialized automatically at the start of the program.

Functions

```
unsigned char ReadSerialPortOne(void);
unsigned char ReadSerialPortTwo(void);
void WriteSerialPortOne(unsigned char byte);
void WriteSerialPortTwo(unsigned char byte);
void OpenSerialPortOne(unsigned baudRate);
void OpenSeialPortTwo(unsigned baudRate);
```

Serial ports can be read and written by calling the functions listed above. The Microchip C library writes to serial port 1 by default and the camera reads and writes from serial port 2.

Each serial port has two ring buffers associated with it, an input and output buffer. Characters are read and written to/from these buffers by the serial port interrupt service routines.

Opening Serial Ports

By default each serial port is opened on startup and set to 115,200 baud. You can override the port speed by opening it with another baud rate. This is done using the OpenSerialPortOne or OpenSerialPortTwo functions. The baudRate is one of:

```
#define BAUD_4800 0x0081
#define BAUD_9600 0x0040
#define BAUD_14400 0x01AD
#define BAUD_19200 0x0181
#define BAUD_28800 0x0156
#define BAUD_38400 0x0140
#define BAUD_57600 0x012A
#define BAUD_115200 0x0115
```

These values are defined in BuiltIns.h. The terminal window in easyC and the IFI Loader both run at 115,200 baud (the default setting).

Writing

Characters being sent out accumulate in the buffer it the serial port hardware is busy. If the buffer fills, programs will wait until there is space available.

Reading

Characters incoming from an attached device will be buffered in the input ring buffers until the program takes them out. Trying to read bytes when the buffer is empty immediately returns zero characters.

The PIC serial hardware does not have an internal FIFO buffer as many dedicated serial port interfaces do. This means that it is easy to miss receiving characters if there is high interrupt activity preventing the buffer from being cleared before the next character comes in. In this case an overrun condition will occur and data will be lost.

Timing Things on Your Robot

Often it is a requirement for the robot to do operations based on time. For example:

- the robot might need to drive for exactly one second,
- or precisely measure the time for an ultrasonic ping traveled from a sensor, to a target, and back in order to calculate the range
- or take samples of a gyro at precise intervals in order to develop compute a heading

WPILib has a range of features that provide support these kinds of requests and more. They are all fundamentally based on hardware timers that are built into the PIC microprocessor inside the robot controller.

There are fundamentally three types of timer functions built into WPILib:

Software timers	6 software timers are built into WPILib and are started and stopped by or read by the program at any time. These timers have 1 millisecond (1/1000 of a second) resolution.
System Clock	There are a variety of functions that let the program: • read the time since the robot has started or time since the game began (in competitions), • wait for a specified interval with millisecond resolution.
Repetitive times	Often the program will want to do an operation repetitively, such as every 10ms or every second. The repetitive time functions are mostly used when interfacing new devices to WPILib and are more complex than the first two sets of functions. They require knowledge of writing interrupt service routines and concurrent programming. These timer functions are described in the section about writing device drivers.

Timers (Software Timers)

Software timers can be started and stopped and read at any time in a running program. They have 1ms accuracy – the number of ticks read is in units of 1/1000 of a second. This is the easiest method of measuring time between events.

Functions

```
void StartTimer(unsigned char timerNumber);
void StopTimer(unsigned char timerNumber);
void PresetTimer(unsigned char timerNumber, unsigned long value);
unsigned long GetTimer(unsigned char timerNumber);
```

The timers operate in much the same way as a stopwatch. Starting a timer causes it to start counting up. Stopping a timer makes it stop counting. `GetTimer` reads the current time value from the timer. `PresetTimer` sets a new value into the timer, usually used to zero the timer before another round of counting.

All the time values in the timers are unsigned long variables and represents milliseconds (1/1000 of a second.)

There are 6 software timers, numbered 1-6. The timer number must be included with any of the above functions. Each timer keeps its own time and starting or stopping one timer doesn't affect the operation of any of the others.

Timers default to zero, so if a timer is just started, it will begin counting from zero.

Timers (System Clock)

There are two system clocks that can be read by the running program, one in seconds for coarse grain measurements like knowing when an autonomous competition is about to end. Another is in microsecond units and is useful for writing devices that require measurement of short times, like the ultrasonic sensor.

```
unsigned long GetUsClock(void);
```

This function returns the time since the robot was started in units of microseconds. It is important to note that in microseconds the time will wrap around the 32-bit long value after about 71 minutes. This is longer than the batteries will probably last in your robot and shouldn't be a problem.

```
unsigned long GetMsClock(void);
```

This function gets the time since the robot was started in units of milliseconds.

```
unsigned GetSecondClock(void);
```

This function returns the number of seconds since the robot has started. Notice that the number of seconds is a truncated version of the millisecond clock so if you read both, the number of milliseconds will not be exactly 1000 times the number of seconds.

```
unsigned GetGameTime(void);
```

The game timer represents the time in seconds since the program entered the Autonomous function in the case of a competition mode project or the time that the program entered the main function in the case of a non-competition project.

```
Void Wait(unsigned long milliseconds);
```

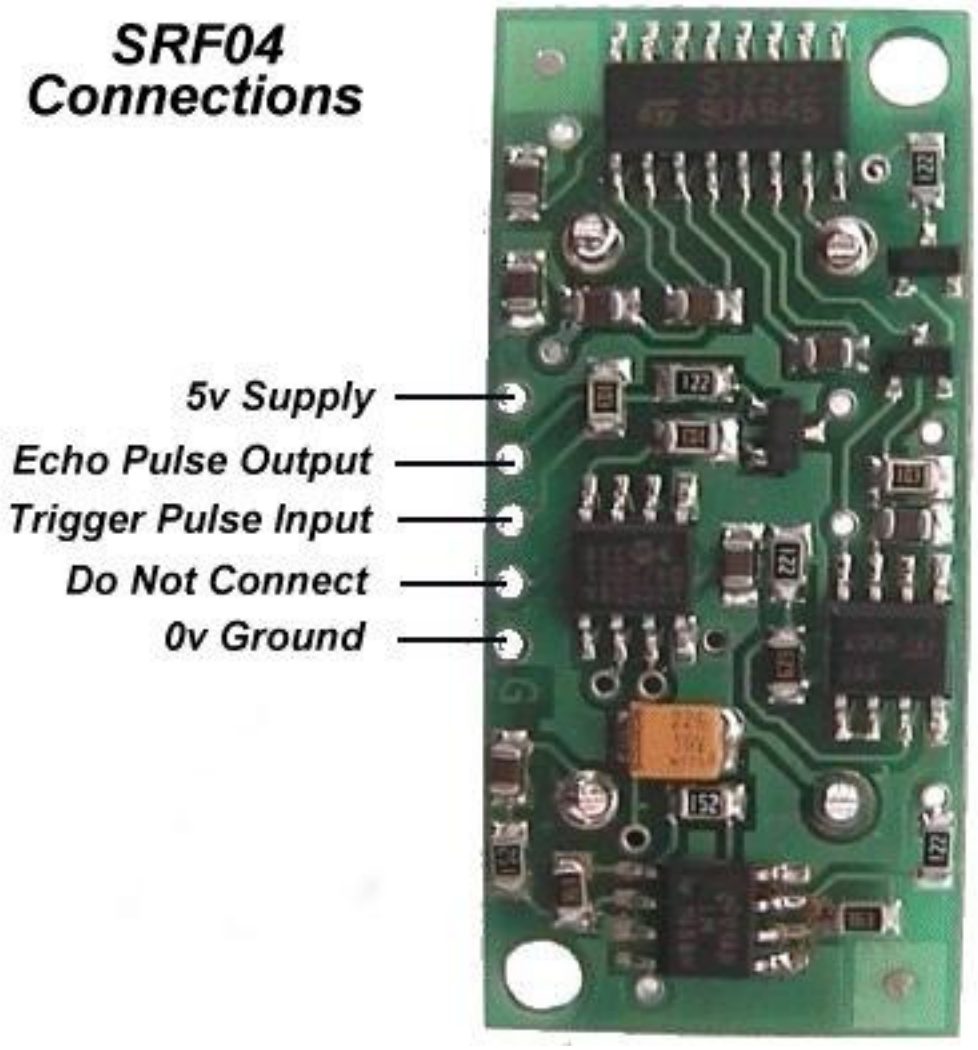
The `Wait` function will pause the execution of the program for the indicated number of milliseconds. This represents one of the major differences between WPILib and programming the controller using the default code. The program can wait without having to return to the main loop. It is perfectly acceptable to wait for an arbitrary amount of time.

Note that motors will continue running at the previously set speed while waiting, and sensors will continue to operate while the robot is waiting since those operations run in the background.

Wait will almost always wait slightly less than the number of milliseconds requested because it uses a 1ms clock, and the Wait always ends exactly as the clock ticks over to the requested millisecond, but the start time might have had some remainder.

Ultrasonic Rangefinder

WPILib has built in support the Daventech SRF04 Ultrasonic Rangefinder. This is also sold by Radio Shack as the VEX Ultrasonic Range Module. This inexpensive device consists of two small ultrasonic transducers, a speaker for sending out ultrasonic pings and microphone for receiving the echo. The software measures the **time of flight** (time from outgoing ping to the returning echo) and returns a value that can be converted to inches or other standard units. The effective range of the SRF04 is about 3” to 10’ with approximately 1” accuracy.



The Echo Pulse Output pin **must** be connected to the interrupt port. On a VEX ultrasonic rangefinder, this port is labeled Output. The trigger pulse input is connected to any other digital output port. On a VEX ultrasonic rangefinder, this port is labeled Input.

If multiple rangefinders are connected to the robot controller they will be fired sequentially to avoid interference between them.

Functions

```
void StartUltrasonic(unsigned char echo, unsigned char ping);
unsigned GetUltrasonic(unsigned char echo, unsigned char ping);
void StopUltrasonic(unsigned char echo, unsigned char ping);
```

To use the Ultrasonic Rangefinder it is first started. Once started, it will continuously send out ping signals and compute distances every 100ms. Whenever the program calls `GetUltrasonic` it will return the most recent distance measured. If it is called more often than 100ms it will return the same result each time.

Example

The following program assumes that an ultrasonic sensor is mounted on the front of the robot and that it is connected to digital I/O ports 1 (echo) and 7 (ping). On a VEX controller the echo signal must be connected to an interrupt port.

The robot will drive until it comes within 15" of an obstacle then stop.

```
#include "WPILib.h"
main()
{
	TwoWheelDrive(2,1);		// use a two wheel drive robot
	StartUltrasonic(1, 7);		// start ultrasonic sensor ranging
	Drive(70, 0);			// start robot driving forward
	while (GetUltrasonic(1, 7) > 15) // continue while distance > 15"
	{
		Wait(20);		// do nothing while driving forward
	}
	Drive(0, 0);			// stop driving
}
```

Writing Device Drivers

Writing device drivers has been greatly simplified by the framework supplied by WPILib but it is still a complex undertaking that should only be taken on by experienced C programmers. Before writing a driver for a new device it is important to fully understand the operation of the device including its I/O and interrupt requirements. Once written, a device driver can easily be used by less experienced programmers.

WPILib is designed to be modular. This means that device drivers can be added and changed without changing robot programs linked with the library. There is no source code merging required to update to a newer version of a driver or to add one all together.

In general WPILib devices have the following skeleton (but there is no hard requirement that this format be followed):

```
void InitializeDevice(unsigned char ports, other parameters);
void StartDevice(unsigned char ports);
void StopDevice(unsigned char ports);
type GetDevice(unsigned char ports);
void DeviceISR(unsigned char port, unsigned char value);
```

The Initialize function (if needed) sets up the device parameters. For example:

- The CMU Camera's Initialize function sets up the color tracking parameters that should be used by the camera.
- The gyro driver does calibration of bias and drift in the Initialize function.

`InitializeDevice` functions can be called more than once if required to change parameters or to recalibrate a device. Many devices have no need to do initialization and can just be started.

The `StartDevice` function starts the device operating – generally this is where the interrupt service routine gets connected and starts running. Some examples in common devices are:

- The CMU Camera driver starts reading tracking packets continuously from the serial port.
- The accelerometer driver begins sampling acceleration values and averaging.
- Encoders start watching the inputs and counting ticks.

The `StopDevice` function stops the recording of data from the device. This is useful for performance reasons. Two very high load devices that can't operate simultaneously can each be started and stopped.

The `GetDevice` function returns the status from the device. It may do unit conversions, calculations, or other functions before delivering the result. For example:

- The ultrasonic rangefinder will compute the most recent calculation from sound round trip times to inches.
- The gyro computes heading information in degrees based on the summed rate information collected.

Generally the `GetDevice` function will get the current value for the device even it is slightly out of date. For example, the CMU Camera constantly receives tracking packets and the `CaptureTrackingData` function will return the most recent results. If `CaptureTrackingData` is called five times between two complete received packets, it will return the same values each time.

The `DeviceISR` is the Interrupt Service Routine (ISR) for the device if one is necessary. There are generally two types of ISRs: actual interrupt handlers for the device signals and timer ISRs to handle periodic device action. Each of these is described in the next sections.

Interrupt Service Routine Considerations

Interrupt Service Routines are one of the most complex parts of real time programming. WPILib has a framework to make this easier, but don't be misled into thinking that all of the usual complexities are gone. There are two important rules about writing ISRs that must always be followed:

1. In general **it is critical that the device driver makes the ISR as short as possible**. This cannot be stressed enough! **While in the ISR for one device, no other device interrupts will be serviced**. An ISR that does extended calculations like floating point math will cause other devices to miss receiving interrupts and will give the appearance that those devices are failing.
2. Variables that are modified inside the ISR and read or changed outside the ISR must be used very carefully. The programmer must constantly be aware of how those variables are modified inside and outside the ISR. In general when accessing variables outside of an ISR that may be modified by the ISR interrupts must be disabled (or at least the interrupts for that particular device or timer).

Accessing Variables Shared with an ISR

This is the most complex part of device programming!

Variables that are modified by the ISR and read outside the ISR can change any time there is an interrupt. And they can change in very subtle ways. The programmer has to constantly be asking, "What would happen if the device interrupted here?" Here are a few examples that will hopefully make the problem clear.

Example 1

Example with multiple variables

Example 2

Example with a single variable with multiple bytes changing

Periodic Timed Events

Devices drivers often need to periodically poll their inputs especially in the case of analog inputs where there are no other device interrupts available. Some examples of devices needing periodic event notification are:

- The gyro reads the analog port every 20ms to get the current rate information so that it can integrate this value and compute the heading.

- The Ultrasonic rangefinder needs to send “pings” every 100ms to for distance measurements.

Interrupt Service Routines

Device drivers may request to be notified through the use of callback routines whenever an interrupt occurs on a particular interrupt line. On the FRC controllers these are Digital I/O ports 1-6 and on VEX controllers these are the 6 Interrupt ports. To request notification the driver should use the following functions:

```
void RegisterInterruptHandler(unsigned char port,
                              unsigned char edge,
                             void (*handler)(unsigned char port, unsigned char value));
void UnregisterInterruptHandler(unsigned char port);
void SetInterruptEdge(unsigned char port, unsigned char edge);
```

The first function, `RegisterInterruptHandler` gives a port number (1-6) to start watching for interrupts. Interrupts can occur either on the rising edge (transition from 0 to 1) or falling edge (transition from 1 to 0) of an input signal. This is specified in the edge argument with either `RISING_EDGE` or `FALLING_EDGE`. Only a single edge can be watched at a time.

The handler argument is the address of an ISR for this particular interrupt. The ISR is passed the port number that interrupted and the edge that caused the interrupt.

When it is no longer necessary to receive interrupts for the device a call to `UnregisterInterruptHandler` will take the device out of the interrupt chain. This is typically done on stopping the device with the `StopDevice` call such as `StopUltrasonic`.

Occasional for some devices it is necessary to change the edge that is watched for interrupts. For example, the Ultrasonic Sensor looks for the time between a rising edge and falling edge to compute the distance. When the ISR is first registered `RISING_EDGE` is specified. After a rising edge interrupt comes in, the edge is changed to `FALLING_EDGE` to catch the other end of the pulse. The current time (in microseconds) is recorded at the rising edge and then subtracted at the falling edge to get the difference. This difference is stored and returned to the user when they call the `GetUltrasonic` function.

Example – A Digital Compass Driver

Here is an example for a simple device, the CMPS03 digital compass. This is not the actual device driver, but a simplified version to help understand how to write device drivers. Unlike most of the other devices in WPILib, the compass driver only supports a single device connected to the robot. This makes it an ideal example to show how to write a driver.

Initialization

The compass driver registers for rising edge interrupts on its port.

```
void InitializeCompass(unsigned char port)
{
	RegisterInterruptHandler(port, RISING_EDGE, &CompassInterruptHandler);
}
```

Interrupt Service Routine

The interrupt handler times the difference between the rising edge and the falling edge of the compass output PWM port. This difference in time is the returned heading.

```
static unsigned long heading, compassRisingEdgeTime;

static void CompassInterruptHandler(unsigned char port, unsigned char state)
{
	if (state)			// check if this is a rising or falling edge
	{				// this is a rising edge
		SetInterruptEdge(port, FALLING_EDGE);	// look for other edge
		compassRisingEdgeTime = GetUSClock();	// record the time
	}
	else
	{				// this is the falling edge
		SetInterruptEdge(port, RISING_EDGE);	// record time
		heading = GetUSClock() – compassRisingEdgeTime;	// compute
	}
}
```

The interrupt service routine is first called on the rising edge from the compass. It records the current time in the `compassRisingEdgeTime` variable, then sets the interrupt handler to look for the falling edge next time. Remember it's the time between the rising edge and the falling edge that is the heading.

Next time the interrupt handler is called, it's for the falling edge. Set the handler to call next time for the rising edge, to start a new cycle. Now the heading is computed.

GetCompassHeading Function

The `GetCompassHeading` function returns the result. It looks like this:

```
Unsigned GetCompassHeading(void)
{
	int temp;
	DISABLE_INTERRUPTS;
		temp = heading;
	ENABLE_INTERRUPTS;
	return temp / 100;
}
```

This function actually computes the heading. Two things to notice here:

1. Interrupts are disabled while the heading is first copied into a temporary variable. This is because if an interrupt were to occur while the individual bytes of the heading were being transferred the heading would have some bytes from the previous sample and other bytes from the next sample. Disabling interrupts ensures that an interrupt won't occur while the `GetCompassHeading` function makes a local copy of the heading to work with.

2. Also notice that the divide by 100 is not done while the interrupts are disabled. Doing long operations while interrupts are disabled will cause devices to miss interrupts and get bad data.

Appendix A: Differences from the IFI Default Code

If you've had experience with the default code supplied by Innovation First, this section will describe the changes in using WPILib. It should be noted that the WPILib code is based on the IFI default code and the libraries supplied by IFI for each of the supported controllers.

Fast and Slow Code

In the default code there was the concept of fast and slow code. The slow code runs every 26ms (or 14ms on VEX) – basically whenever there is a data exchange between the master and user processor. The idea is that the program would read data from the master processor by calling the GetData function – retrieving all the OI inputs. Then the program would process this data, generating motor commands based on the input. The program would then write the data to the master processor by calling PutData. Since nothing externally could change (i.e. motor speeds) except when PutData is called the program would just be bracketed by these data exchanges.

Sometimes sensor data would come in faster than once every 26ms and this code (fast) would just run continuously regardless of the timing of the communications between the master and user processors.

In WPILib there is no concept of "fast" and "slow" code. All code runs as fast as the processor can go. You update motor speeds and request OI values as often as the program cares to do it. There is still the 26ms restriction imposed by the architecture, so only the last motor update just before a PutData will be transmitted, and all reads of the OI will return the same results until the next 26ms GetData update.

This simplifies the design of user programs, developers don't have to be very concerned about the whole concept of fast and slow code.

No GetData/PutData

The GetData and PutData requests described in the previous section are completely handled in the background of the running program. Every millisecond an interrupt service routine checks to see if there is data ready to be received from the master processor. If there is, it happens in the background. The next time the program requests OI data, the new updated values are returned. The same is true of PWM outputs; the program can set the PWMs as often as desired but it will only be transmitted when the master processor is ready to accept new data.

The program no longer needs to be aware of GetData and PutData functions. The whole program structure can now change – you can have local loops in the code where the robot might just drive straight for 2 seconds by starting the motors going, and waiting for the two seconds to run out. Previously it was impossible to do this because the code had to get execute the GetData/PutData function calls every 26ms otherwise the user processor would be halted.

Appendix B: Frequently Asked Questions

Is the source code for WPILib available?

Unfortunately it is not being released at this time. However, we will be available to provide help for any team needing it throughout the season and we will be responsive as possible in putting out updates to the code as bugs are reported or improvements are made. Advanced users may write device drivers for WPILib and make those devices available to others.

Are you taking suggestions for new features for the library?

We are always looking for ways to improve the library, additional sensors to support, and other features that will help users extend the library. We are also willing to incorporate new devices that are submitted. These devices will become available to all teams in the next release after we get the code.

How is WPILib related to easyC?

WPILib is the underlying runtime code that easyC for FRC is based on. easyC uses a friendly drag-and-drop programming metaphor to help users write programs. All the devices are implemented in WPILib.

WPILIb is sometimes referred to as a *framework*. What's this all about?

A **framework** is a partially completed piece of code that is intended to be extended to suit the needs of a particular user of set of users. WPILib is a framework that is may be extended with support for new devices and new functions cleanly without the need to integrate those new features into the source code of end-users programs. For example, if a device driver is written for a new Gyro device, it can be added to WPILib and no existing programs need to change, unless they take advantage of the new the device. And then it's only a matter of initializing the device and getting values from it. There is no complicated source code merging required.

Appendix C: WPILib Functions Reference

Robot Initialization

```
void IO_Initialization(void);
void Set_Number_of_Analog_Channels(unsigned char numberOfChannels);
unsigned char IsAutonomous(void);
unsigned char IsEnabled(void);
```

Vex

```
void DefineControllerIO(unsigned char numberOfAnalogChannels,
                        unsigned char p1, unsigned char p2, unsigned char p3,
                        unsigned char p4, unsigned char p5, unsigned char p6,
                        unsigned char p7, unsigned char p8, unsigned char p9,
                        unsigned char p10, unsigned char p11, unsigned char p12,
                        unsigned char p13, unsigned char p14, unsigned char p15,
                        unsigned char p16);
void SetCompetitionMode(unsigned char autonomousTime,
                            unsigned char operatorTime);
```

FRC

```
void DefineControllerIO(unsigned char p1, unsigned char p2,
        unsigned char p3, unsigned char p4, unsigned char p5,
unsigned char p6, unsigned char p7, unsigned char p8,
       unsigned char p9, unsigned char p10, unsigned char p11,
       unsigned char p12, unsigned char p13, unsigned char p14,
       unsigned char p15, unsigned char p16, unsigned char p17,
       unsigned char p18);

void SetCompetitionMode(unsigned char mode);
```

General Purpose I/O

```
// direction values for SetDirection()
#define INPUT   1
#define OUTPUT  0

void SetDigitalOutput(unsigned char port, unsigned char value);

unsigned char GetDigitalInput(unsigned char port);

// set port direction (INPUT or OUTPUT)
void SetDirection(unsigned char port, unsigned char direction);
unsigned int GetAnalogInput(unsigned char port);
unsigned int Get_Analog_Value (unsigned char ADC_channel);
```

Debugging/Printing Output

```
// Non-graphical printing
void PrintToScreen(rom const char *fmt, ...);

// Graphical printing (using easyC terminal window in graphic mode only)
void SetGDWaitTime(unsigned time);
void ResetGD(void);
void ClearGD(unsigned char ucRow1, unsigned char ucCol1, unsigned char ucRow2,
             unsigned char ucCol2, unsigned char ucFrame);
```

```
void PrintTextToGD(unsigned char ucRow, unsigned char ucCol, unsigned long
                   ulColor, rom const char *szText, ...);
void PrintFrameToGD(unsigned char ucRow1, unsigned char ucCol1, unsigned char
                    ucRow2, unsigned char ucCol2, unsigned long ulColor);
```

Encoders

```
// Single line encoders (Vex encoders or other pulse counting applications)
void StartEncoder(unsigned char channel);
void StopEncoder(unsigned char channel);
long GetEncoder(unsigned char channel);
void PresetEncoder(unsigned char channel, long presetValue);

// Quadrature Encoder functions
void StartQuadEncoder(unsigned char channelA, unsigned char channelB,
                      unsigned char invert);
void StopQuadEncoder(unsigned char channelA, unsigned char channelB);
long GetQuadEncoder(unsigned char channelA, unsigned char channelB);
void PresetQuadEncoder(unsigned char channelA, unsigned char channelB,
                       long presetValue);

// Gear tooth sensor functions
void StartGTSensor(unsigned char port, unsigned char invert);
void StopGTSensor(unsigned char port);
long GetGTSensor(unsigned char port);
void PresetGTSensor(unsigned char port, long presetValue);
```

Interrupt Handling

```
// which edge will cause the ISR to be called
// int 1 & 2 done in hardware, 3-6 in software
#define RISING_EDGE 1
#define FALLING_EDGE 0

// Enabling and disabling interrupts
#define DISABLE_INTERRUPTS INTCONbits.GIEL = 0
#define ENABLE_INTERRUPTS INTCONbits.GIEL = 1

// Interrupt watchers latch state when interrupt occurs
void StartInterruptWatcher(unsigned char port, unsigned char edgeType);
void StopInterruptWatcher(unsigned char port);
unsigned char GetInterruptWatcher(unsigned char port);

// Registering interrupt handler function (ISR)
void RegisterInterruptHandler(unsigned char port, unsigned char edgeType,
                     void (*handler)(unsigned char port, unsigned char
value));
void UnRegisterInterruptHandler(unsigned char port);
void SetInterruptEdge(unsigned char port, unsigned char edgeType);
```

Ultrasonic Rangefinders

```
unsigned int GetUltrasonic(unsigned char echo, unsigned char ping);
void StartUltrasonic(unsigned char echo, unsigned char ping);
void StopUltrasonic(unsigned char echo, unsigned char ping);
```

Gyros

```
// these are gyro types that will be used for SetGyroType
#define ADXRS300    50
#define ADXRS150    125
#define ADXRS80     125

void InitGyro(unsigned char port);
void StartGyro(unsigned char port);
void StopGyro(unsigned char port);
int GetGyroAngle(unsigned char port);
void SetGyroType(unsigned char port, unsigned type);
void SetGyroDeadband(unsigned char port, char deadband);
```

Accelerometers

```
void InitAccelerometer(unsigned char port);
void StartAccelerometer(unsigned char port);
int GetAcceleration(unsigned char port);
void StopAccelerometer(unsigned char port);
```

Timing

```
// Suspend program while motors and interrupt handlers continue to run
void Wait(unsigned long ms);

// functions to support 6 timers
void StartTimer(unsigned char timerNumber);
void StopTimer(unsigned char timerNumber);
void PresetTimer(unsigned char timerNumber, unsigned long value);
unsigned long GetTimer(unsigned char timerNumber);

// Getting the time from the clock
unsigned GetSecondClock(void);   // time in seconds
unsigned long GetMsClock(void);  // time in milliseconds
unsigned GetGameTime(void);             // time since autonomous started
unsigned GetSecondClock(void);   // time in seconds
unsigned long GetUsClock(void);  // time in microseconds

// Timer interrupt handlers – called with interrupts disabled

// Timer interrupt handler function type
typedef void (*TimerHandler)(void);

// registering timer interrupt handler
void RegisterSingleTimer(unsigned long time, void (*handler)(void));
void RegisterRepeatingTimer(unsigned long time, void (*handler)(void));
void CancelTimer(void (*handler)(void));
```

Operator Interface

Operator Interface Status Displays (FRC only)

```
// OI Status LEDs definitions
#define PWM1_GREEN 0
#define PWM1_RED 1
#define PWM2_GREEN 2
#define PWM2_RED 3
#define RELAY1_GREEN 4
#define RELAY1_RED 5
#define RELAY2_GREEN 6
#define RELAY2_RED 7
#define SWITCH1_LED 8
#define SWITCH2_LED 9
#define SWITCH3_LED 10

// Operating status LEDs
void SetOILED(unsigned char led, unsigned char value);
void SetUserDisplay(unsigned char value);
```

Operator Interface Controls

Vex

```
Vex receiver to controller port numbers
#define PORT_1 1
#define PORT_2 2

// Vex transmitter channel numbers
#define CHANNEL_1 1
#define CHANNEL_2 2
#define CHANNEL_3 3
#define CHANNEL_4 4
#define CHANNEL_5 5
#define CHANNEL_6 6
#define CHANNEL_5_TOP 1
#define CHANNEL_5_BOTTOM 2
#define CHANNEL_6_TOP 3
#define CHANNEL_6_BOTTOM 4

// Vex functions for rear buttons (returns 1/0 for CHANNEL_5_TOP, etc.)
// these extra functions are for channels 5 and 6 controller back
// You can use GetRXInput instead and get back 0/255 for top/bottom or 127
// for compatibility with FRC
unsigned char GetOIDInput(unsigned char port, unsigned char channel);
unsigned char GetOIAInput(unsigned char port, unsigned char channel);

// Get a Vex receiver channel value
unsigned char GetRxInput(unsigned char port, unsigned char channel);

// Vex simplified OI to drive functions
// These functions send the receiver to the motors ONCE (must be in a loop)
void Arcade2(unsigned char port,
                unsigned char moveChannel, unsigned char rotateChannel,
                unsigned char leftPWM, unsigned char rightPWM,
                unsigned char leftInvert, unsigned char rightInvert);
void Arcade4(unsigned char port,
            unsigned char ucMoveChannel, unsigned char ucRotateChannel,
            unsigned char ucLeftfrontPWM, unsigned char ucRightfrontPWM,
            unsigned char ucLeftrearPWM, unsigned char ucRightrearPWM,
            unsigned char ucLeftfrontInvert, unsigned char ucRightfrontInvert,
            unsigned char ucLeftrearInvert, unsigned char ucRightrearInvert);
void Tank2(unsigned char port,
            unsigned char leftChannel, unsigned char rightChannel,
            unsigned char leftPWM, unsigned char rightPWM,
            unsigned char leftInvert, unsigned char rightInvert);
void Tank4(unsigned char port,
            unsigned char ucLeftChannel, unsigned char ucRightChannel,
            unsigned char ucLeftfrontPWM, unsigned char ucRightfrontPWM,
            unsigned char ucLeftrearPWM, unsigned char ucRightrearPWM,
            unsigned char ucLeftfrontInvert, unsigned char ucRightfrontInvert,
            unsigned char ucLeftrearInvert, unsigned char ucRightrearInvert);
```

FRC

```
// FRC OI Port numbers
#define PORT_1 1
#define PORT_2 2
#define PORT_3 3
#define PORT_4 4

// FRC joystick functions
#define X_AXIS 1
#define Y_AXIS 2
#define WHEEL_AXIS 3
#define AUX_AXIS 4
#define TRIGGER_SW 1
#define TOP_SW 2
#define AUX1_SW 3
#define AUX2_SW 4
#define ALT_TRIGGER 5
#define ALT_THUMB 6
#define ALT_AUX1_SW 7
#define ALT_AUX2_SW 8

// FRC Relay functions
void SetRelay(unsigned char port, char forward, char reverse);
void OIToRelay(unsigned char port, unsigned char function,
                unsigned char relayNumber, unsigned char direction);

// Set the user display value
void SetUserByte(unsigned char index, unsigned char value);

// FRC return raw value from OI for digital or analog port
unsigned char GetOIDInput(unsigned char port, unsigned char function);
unsigned char GetOIAInput(unsigned char port, unsigned char function);

// FRC Simple OI to Drive functions
// these functions send OI values to a motor ONCE (must be in a loop)
void Arcade2(unsigned char movePort, unsigned char moveChannel,
                unsigned char rotatePort, unsigned char rotateChannel,
                unsigned char leftPWM, unsigned char rightPWM,
                unsigned char leftInvert, unsigned char rightInvert);
void Arcade4(unsigned char ucMovePort, unsigned char ucMoveChannel,
            unsigned char ucRotatePort, unsigned char ucRotateChannel,
            unsigned char ucLeftfrontPWM, unsigned char ucRightfrontPWM,
            unsigned char ucLeftrearPWM, unsigned char ucRightrearPWM,
            unsigned char ucLeftfrontInvert, unsigned char ucRightfrontInvert,
            unsigned char ucLeftrearInvert, unsigned char ucRightrearInvert);
void Tank2(unsigned char leftPort, unsigned char leftChannel,
            unsigned char rightPort, unsigned char rightChannel,
            unsigned char leftPWM, unsigned char rightPWM,
            unsigned char leftInvert, unsigned char rightInvert);
void Tank4(unsigned char ucLeftPort, unsigned char ucLeftChannel,
            unsigned char ucRightPort, unsigned char ucRightChannel,
            unsigned char ucLeftfrontPWM, unsigned char ucRightfrontPWM,
            unsigned char ucLeftrearPWM, unsigned char ucRightrearPWM,
            unsigned char ucLeftfrontInvert, unsigned char ucRightfrontInvert,
            unsigned char ucLeftrearInvert, unsigned char ucRightrearInvert);

// OI values that are send directly to motors or digital outputs
void OIToDOutput(unsigned char port, unsigned char function, unsigned char
dport);
void OIToPWM(unsigned char port, unsigned char function, unsigned char pwm,
             unsigned char invert);
```

CMU Camera

```
// returned data packet from camera
typedef struct {
    unsigned char mx;
    unsigned char my;
    unsigned char x1;
    unsigned char y1;
    unsigned char x2;
    unsigned char y2;
    unsigned char regionSize;
    unsigned char confidence;
    unsigned char pan;
    unsigned char tilt;
    unsigned char length;
    unsigned char sequence;
} TPacket;

// data structure to initialize the camera parameters
typedef rom struct {
    unsigned char redMin, redMax, greenMin, greenMax, blueMin, blueMax;
    unsigned char yCrCb;
    unsigned char noiseFilter;
    unsigned char aecEnable;
    unsigned char aecLevel;
    unsigned char agcEnable;
    unsigned char agcLevel;
    unsigned char saturation;
    unsigned char blueGain;
    unsigned char redGain;
    unsigned char brightness;
    unsigned char panRangeFar;
    unsigned char panRangeNear;
    unsigned char panStep;
    unsigned char tiltRangeFar;
    unsigned char tiltRangeNear;
    unsigned char tiltStep;
} CameraInitializationData;

// initialize the camera to one of an array of CameraInitializeData elements
// you need to call one of these two functions before starting the camera

// InitCamera – designed for easyC
void InitCamera(unsigned char cameraInitIndex);

// InitializeCamera – takes a struct of camera data (designed for C
programmers)
void InitializeCamera(CameraInitializationData *c);

// camera operation – must be started before data packets can be used
void StopCamera(void);
void StartCamera(void);

// Get the current camera data values
// Use nulls for unneeded addresses
void CaptureTrackingData(
            unsigned char *centerX,
            unsigned char *centerY,
            unsigned char *x1,
            unsigned char *y1,
            unsigned char *x2,
            unsigned char *y2,
            unsigned char *regionSize,
            unsigned char *confidence,
            unsigned char *pan,
            unsigned char *tilt);

// Get the current TPacket (camera data packet)
TPacket *CopyTrackingData(void);

// sends a reset to the camera
unsigned char ResetCameraState(void);

// sends a single command to the camera like printf
// “%” characters are replaced with the numeric byte values supplied as
//     arguments
unsigned char CmdToCamera(const rom char *format, ...);

// operates the CAMERA servos – not the servos attached to PWM ports
void SetServoTracking(unsigned char panTracking, unsigned char tiltTracking);
void SetServoPosition(unsigned char servo, unsigned char position);

// Initialize the camera before starting it
unsigned char GetCameraStatus(void);
void SetCameraDebugMode(unsigned char mode);
```

Pneumatics Pressure Switch (FRC Only)

```
void InitPressureSwitch(unsigned char pressureSwitchPort, unsigned char
relayPort);
```

Serial Port

```
#define BAUD_4800 0x0081
#define BAUD_9600 0x0040
#define BAUD_14400 0x01AD
#define BAUD_19200 0x0181
#define BAUD_28800 0x0156
#define BAUD_38400 0x0140
#define BAUD_57600 0x012A
#define BAUD_115200 0x0115

unsigned char ReadSerialPortOne(void);
void WriteSerialPortOne(unsigned char);
unsigned char ReadSerialPortTwo(void);
void WriteSerialPortTwo(unsigned char);
unsigned char GetSerialPort2ByteCount(void);
unsigned char GetSerialPort1ByteCount(void);
void OpenSerialPortOne(unsigned baudRate);
void OpenSerialPortTwo(unsigned baudRate);
```

Compass

```
void InitializeCompass(unsigned char port);
unsigned GetCompassHeading(void);
```

Miscellaneous Functions

```
unsigned char GetPacketNumber(void);
```

FRC

```
unsigned GetMainBattery(void);
unsigned GetBackupBattery(void);
```

Vex

```
unsigned char ReceivingData(unsigned char port);
```

Motor and Servo Functions

```
// Set a single PWM value
void SetPWM(unsigned char port, unsigned char speed);

// Drive functions that are based on motor and direction values ranging from:
//   -127 - +127 for full backwards to full forwards
//   -127 - +127 for full right to full left turns (see Drive function)
void SetInvertedMotor(unsigned char port);
void Motor(unsigned char pwmPort, int speed);
void Motors(int leftSpeed, int rightSpeed);
void Drive(int speed, int direction);

// must call one of these functions prior to calling the above functions
void TwoWheelDrive(unsigned char _leftMotor, unsigned char _rightMotor);
void FourWheelDrive(unsigned char _leftMotor, unsigned char _frontLeftMotor,
                    unsigned char _rightMotor,
                    unsigned char _frontRightMotor);
```

Glossary

ISR	Interrupt service routine – the code associated with a device, input port, or timer that is run each time its interrupt occurs. Interrupt service routines interrupt the regular flow of execution, do their task, and then return to the main line of code that was executing at the time. Interrupt service routines represent the most difficult part of device programming.
Latency	The time from when an interrupt is detected by the hardware until the time that the code specific to that device is run. This is the overhead of the system saving state, determining the device that interrupted, and dispatching.
PWM	Pulse width modulation – a method of controlling motors by sending varying width square waves. It is the relative width of the high vs. the low portion of the signal (or duty cycle) that determines the actual speed of the motor.
Quadrature encoder	An encoder that uses two signals usually 90 degrees out of phase to let the program detect rotation direction as well as degrees of rotation.
SRF04	An ultrasonic rangefinder manufactured by Daventech and also distributed by Radio Shack as a VEX sensor.
Time of flight	The time it takes for a signal from a sensor such as an ultrasonic rangefinder to make the round-trip from the sensor to the target, reflect off the target, and return to the sensor. The length of this time usually translates into a distance between the sensor and the target object.

Index