# WPILibC++

Version 1.0

Generated by Doxygen 1.5.7.1

Sat Dec 27 17:14:39 2008

# Contents

# Chapter 1

# Todo List

**Member AnalogModule::VoltsToValue(INT32 channel, float voltage)** This assumes raw values. Oversampling not supported as is.

**Class HiTechnicCompass** Implement a calibration method for the sensor.

**Member RobotBase::IsNewDataAvailable()** The current implementation is silly. We already know this explicitly without trying to figure it out.

**Class SimpleRobot** If this is going to last until release, it needs a better name.

# Chapter 2

# Bug List

**Member SerialPort::Printf(const char ∗writeFmt,...)** All pointer-based parameters seem to return an error.

**Member SerialPort::Scanf(const char ∗readFmt,...)** All pointer-based parameters seem to return an error.

# Chapter 3

# Class Index

## 3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

# Chapter 4

# Class Index

## 4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 5

# Class Documentation

## 5.1 Accelerometer Class Reference

`#include <Accelerometer.h>`

Inherits SensorBase.

### Public Member Functions

- Accelerometer (UINT32 channel)
- Accelerometer (UINT32 slot, UINT32 channel)
- Accelerometer (AnalogChannel *channel)
- virtual ~Accelerometer ()
- float GetAcceleration ()
- void SetSensitivity (float sensitivity)
- void SetZero (float zero)

### 5.1.1 Detailed Description

Handle operation of the accelerometer. The accelerometer reads acceleration directly through the sensor. Many sensors have multiple axis and can be treated as multiple devices. Each is calibrated by finding the center value over a period of time.

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 Accelerometer::Accelerometer (UINT32 *channel*) `[explicit]`

Create a new instance of an accelerometer.

The accelerometer is assumed to be in the first analog module in the given analog channel. The constructor allocates desired analog channel.

#### 5.1.2.2 Accelerometer::Accelerometer (UINT32 *slot*, UINT32 *channel*)

Create new instance of accelerometer.

Make a new instance of the accelerometer given a module and channel. The constructor allocates the desired analog channel from the specified module

#### 5.1.2.3 Accelerometer::Accelerometer (AnalogChannel * *channel*) `[explicit]`

Create a new instance of Accelerometer from an existing AnalogChannel. Make a new instance of accelerometer given an AnalogChannel. This is particularly useful if the port is going to be read as an analog channel as well as through the Accelerometer class.

#### 5.1.2.4 Accelerometer::∼Accelerometer () `[virtual]`

Delete the analog components used for the accelerometer.

### 5.1.3 Member Function Documentation

#### 5.1.3.1 float Accelerometer::GetAcceleration ()

Return the acceleration in Gs.

The acceleration is returned units of Gs.

**Returns:**

The current acceleration of the sensor in Gs.

#### 5.1.3.2 void Accelerometer::SetSensitivity (float *sensitivity*)

Set the accelerometer sensitivity.

This sets the sensitivity of the accelerometer used for calculating the acceleration. The sensitivity varys by accelerometer model. There are constants defined for various models.

**Parameters:**

***sensitivity*** The sensitivity of accelerometer in Volts per G.

#### 5.1.3.3 void Accelerometer::SetZero (float *zero*)

Set the voltage that corresponds to 0 G.

The zero G voltage varys by accelerometer model. There are constants defined for various models.

**Parameters:**

***zero*** The zero G voltage.

The documentation for this class was generated from the following files:

- Accelerometer.h
- Accelerometer.cpp

## 5.2 AnalogChannel Class Reference

`#include <AnalogChannel.h>`

Inherits SensorBase.

### Public Member Functions

- AnalogChannel (UINT32 slot, UINT32 channel)
- AnalogChannel (UINT32 channel)
- virtual ~AnalogChannel ()
- AnalogModule * GetModule ()
- INT16 GetValue ()
- INT32 GetAverageValue ()
- float GetVoltage ()
- float GetAverageVoltage ()
- UINT32 GetSlot ()
- UINT32 GetChannel ()
- void SetAverageBits (UINT32 bits)
- UINT32 GetAverageBits ()
- void SetOversampleBits (UINT32 bits)
- UINT32 GetOversampleBits ()
- UINT32 GetLSBWeight ()
- INT32 GetOffset ()
- bool IsAccumulatorChannel ()
- void InitAccumulator ()
- void SetAccumulatorInitialValue (INT64 value)
- void ResetAccumulator ()
- void SetAccumulatorCenter (INT32 center)
- void SetAccumulatorDeadband (INT32 deadband)
- INT64 GetAccumulatorValue ()
- UINT32 GetAccumulatorCount ()
- void GetAccumulatorOutput (INT64 *value, UINT32 *count)

### 5.2.1 Detailed Description

Analog channel class.

Each analog channel is read from hardware as a 12-bit number representing -10V to 10V.

Connected to each analog channel is an averaging and oversampling engine. This engine accumulates the specified ( by SetAverageBits() and SetOversampleBits() ) number of samples before returning a new value. This is not a sliding window average. The only difference between the oversampled samples and the averaged samples is that the oversampled samples are simply accumulated effectively increasing the resolution, while the averaged samples are divided by the number of samples to retain the resolution, but get more stable values.

## 5.2.2 Constructor & Destructor Documentation

### 5.2.2.1 AnalogChannel::AnalogChannel (UINT32 *slot*, UINT32 *channel*)

Construct an analog channel on a specified module.

**Parameters:**

***slot*** The slot that the analog module is plugged into.

***channel*** The channel number to represent.

### 5.2.2.2 AnalogChannel::AnalogChannel (UINT32 *channel*) `[explicit]`

Construct an analog channel on the default module.

**Parameters:**

***channel*** The channel number to represent.

### 5.2.2.3 AnalogChannel::∼AnalogChannel () `[virtual]`

Channel destructor.

## 5.2.3 Member Function Documentation

### 5.2.3.1 UINT32 AnalogChannel::GetAccumulatorCount ()

Read the number of accumulated values.

Read the count of the accumulated values since the accumulator was last Reset().

**Returns:**

The number of times samples from the channel were accumulated.

### 5.2.3.2 void AnalogChannel::GetAccumulatorOutput (INT64 ∗ *value*, UINT32 ∗ *count*)

Read the accumulated value and the number of accumulated values atomically.

This function reads the value and count from the FPGA atomically. This can be used for averaging.

**Parameters:**

***value*** Pointer to the 64-bit accumulated output.

***count*** Pointer to the number of accumulation cycles.

#### 5.2.3.3 INT64 AnalogChannel::GetAccumulatorValue ()

Read the accumulated value.

Read the value that has been accumulating on channel 1. The accumulator is attached after the oversample and average engine.

**Returns:**

The 64-bit value accumulated since the last Reset().

#### 5.2.3.4 UINT32 AnalogChannel::GetAverageBits ()

Get the number of averaging bits previously configured. This gets the number of averaging bits from the FPGA. The actual number of averaged samples is 2∗∗bits. The averaging is done automatically in the FPGA.

**Returns:**

Number of bits of averaging previously configured.

#### 5.2.3.5 INT32 AnalogChannel::GetAverageValue ()

Get a sample from the output of the oversample and average engine for this channel. The sample is 12-bit + the value configured in SetOversampleBits(). The value configured in SetAverageBits() will cause this value to be averaged 2∗∗bits number of samples. This is not a sliding window. The sample will not change until 2∗∗(OversamplBits + AverageBits) samples have been acquired from the module on this channel. Use GetAverageVoltage() to get the analog value in calibrated units.

**Returns:**

A sample from the oversample and average engine for this channel.

#### 5.2.3.6 float AnalogChannel::GetAverageVoltage ()

Get a scaled sample from the output of the oversample and average engine for this channel. The value is scaled to units of Volts using the calibrated scaling data from GetLSBWeight() and GetOffset(). Using oversampling will cause this value to be higher resolution, but it will update more slowly. Using averaging will cause this value to be more stable, but it will update more slowly.

**Returns:**

A scaled sample from the output of the oversample and average engine for this channel.

#### 5.2.3.7 UINT32 AnalogChannel::GetChannel ()

Get the channel number.

**Returns:**

The channel number.

#### 5.2.3.8 UINT32 AnalogChannel::GetLSBWeight ()

Get the factory scaling least significant bit weight constant. The least significant bit weight constant for the channel that was calibrated in manufacturing and stored in an eeprom in the module.

Volts = ((LSB_Weight * 1e-9) * raw) - (Offset * 1e-9)

**Returns:**

Least significant bit weight.

#### 5.2.3.9 AnalogModule * AnalogChannel::GetModule ()

Get the analog module that this channel is on.

**Returns:**

A pointer to the AnalogModule that this channel is on.

#### 5.2.3.10 INT32 AnalogChannel::GetOffset ()

Get the factory scaling offset constant. The offset constant for the channel that was calibrated in manufacturing and stored in an eeprom in the module.

Volts = ((LSB_Weight * 1e-9) * raw) - (Offset * 1e-9)

**Returns:**

Offset constant.

#### 5.2.3.11 UINT32 AnalogChannel::GetOversampleBits ()

Get the number of oversample bits previously configured. This gets the number of oversample bits from the FPGA. The actual number of oversampled values is 2**bits. The oversampling is done automatically in the FPGA.

**Returns:**

Number of bits of oversampling previously configured.

#### 5.2.3.12 UINT32 AnalogChannel::GetSlot ()

Get the slot that the analog module is plugged into.

**Returns:**

The slot that the analog module is plugged into.

#### 5.2.3.13 INT16 AnalogChannel::GetValue ()

Get a sample straight from this channel on the module. The sample is a 12-bit value representing the -10V to 10V range of the A/D converter in the module. The units are in A/D converter codes. Use GetVoltage() to get the analog value in calibrated units.

**Returns:**

A sample straight from this channel on the module.

#### 5.2.3.14 float AnalogChannel::GetVoltage ()

Get a scaled sample straight from this channel on the module. The value is scaled to units of Volts using the calibrated scaling data from GetLSBWeight() and GetOffset().

**Returns:**

A scaled sample straight from this channel on the module.

#### 5.2.3.15 void AnalogChannel::InitAccumulator ()

Initialize the accumulator.

#### 5.2.3.16 bool AnalogChannel::IsAccumulatorChannel ()

Is the channel attached to an accumulator.

**Returns:**

The analog channel is attached to an accumulator.

#### 5.2.3.17 void AnalogChannel::ResetAccumulator ()

Resets the accumulator to the initial value.

#### 5.2.3.18 void AnalogChannel::SetAccumulatorCenter (INT32 *center*)

Set the center value of the accumulator.

The center value is subtracted from each A/D value before it is added to the accumulator. This is used for the center value of devices like gyros and accelerometers to make integration work and to take the device offset into account when integrating.

This center value is based on the output of the oversampled and averaged source from channel 1. Because of this, any non-zero oversample bits will affect the size of the value for this field.

#### 5.2.3.19 void AnalogChannel::SetAccumulatorDeadband (INT32 *deadband*)

Set the accumulator's deadband.

#### 5.2.3.20 void AnalogChannel::SetAccumulatorInitialValue (INT64 *initialValue*)

Set an inital value for the accumulator.

This will be added to all values returned to the user.

**Parameters:**

***initialValue*** The value that the accumulator should start from when reset.

#### 5.2.3.21 void AnalogChannel::SetAverageBits (UINT32 *bits*)

Set the number of averaging bits. This sets the number of averaging bits. The actual number of averaged samples is 2∗∗bits. Use averaging to improve the stability of your measurement at the expense of sampling rate. The averaging is done automatically in the FPGA.

**Parameters:**

***bits*** Number of bits of averaging.

#### 5.2.3.22 void AnalogChannel::SetOversampleBits (UINT32 *bits*)

Set the number of oversample bits. This sets the number of oversample bits. The actual number of oversampled values is 2∗∗bits. Use oversampling to improve the resolution of your measurements at the expense of sampling rate. The oversampling is done automatically in the FPGA.

**Parameters:**

***bits*** Number of bits of oversampling.

The documentation for this class was generated from the following files:

- AnalogChannel.h
- AnalogChannel.cpp

## 5.3 AnalogModule Class Reference

`#include <AnalogModule.h>`

Inherits Module.

### Public Member Functions

- void SetSampleRate (float samplesPerSecond)
- float GetSampleRate ()
- void SetAverageBits (UINT32 channel, UINT32 bits)
- UINT32 GetAverageBits (UINT32 channel)
- void SetOversampleBits (UINT32 channel, UINT32 bits)
- UINT32 GetOversampleBits (UINT32 channel)
- INT16 GetValue (UINT32 channel)
- INT32 GetAverageValue (UINT32 channel)
- float GetAverageVoltage (UINT32 channel)
- float GetVoltage (UINT32 channel)
- UINT32 GetLSBWeight (UINT32 channel)
- INT32 GetOffset (UINT32 channel)
- INT32 VoltsToValue (INT32 channel, float voltage)

### Static Public Member Functions

- static UINT32 SlotToIndex (UINT32 slot)
- static AnalogModule * GetInstance (UINT32 slot)

### Static Public Attributes

- static const long kTimebase = 40000000

  *40 MHz clock*

### Protected Member Functions

- AnalogModule (UINT32 slot)
- virtual ~AnalogModule ()

### 5.3.1 Detailed Description

Analog Module class. Each module can independently sample its channels at a configurable rate. There is a 64-bit hardware accumulator associated with channel 1 on each module. The accumulator is attached to the output of the oversample and average engine so that the center value can be specified in higher resolution resulting in less error.

### 5.3.2 Constructor & Destructor Documentation

#### 5.3.2.1 AnalogModule::AnalogModule (UINT32 *slot*) `[explicit, protected]`

Create a new instance of an analog module.

Create an instance of the analog module object. Initialize all the parameters to reasonable values on start. Setting a global value on an analog module can be done only once unless subsequent values are set the previously set value. Analog modules are a singleton, so the constructor is never called outside of this class.

**Parameters:**

*slot* The slot in the chassis that the module is plugged into.

#### 5.3.2.2 AnalogModule::∼AnalogModule () `[protected, virtual]`

Destructor for AnalogModule.

### 5.3.3 Member Function Documentation

#### 5.3.3.1 UINT32 AnalogModule::GetAverageBits (UINT32 *channel*)

Get the number of averaging bits.

This gets the number of averaging bits from the FPGA. The actual number of averaged samples is 2∗∗bits. The averaging is done automatically in the FPGA.

**Parameters:**

*channel* Channel to address.

**Returns:**

Bits to average.

#### 5.3.3.2 INT32 AnalogModule::GetAverageValue (UINT32 *channel*)

Get a sample from the output of the oversample and average engine for the channel.

The sample is 12-bit + the value configured in SetOversampleBits(). The value configured in SetAverageBits() will cause this value to be averaged 2∗∗bits number of samples. This is not a sliding window. The sample will not change until 2∗∗(OversamplBits + AverageBits) samples have been acquired from the module on this channel. Use GetAverageVoltage() to get the analog value in calibrated units.

**Parameters:**

*channel* Channel number to read.

**Returns:**

A sample from the oversample and average engine for the channel.

#### 5.3.3.3 float AnalogModule::GetAverageVoltage (UINT32 *channel*)

Get a scaled sample from the output of the oversample and average engine for the channel.

The value is scaled to units of Volts using the calibrated scaling data from GetLSBWeight() and GetOffset(). Using oversampling will cause this value to be higher resolution, but it will update more slowly. Using averaging will cause this value to be more stable, but it will update more slowly.

**Parameters:**

***channel*** The channel to read.

**Returns:**

A scaled sample from the output of the oversample and average engine for the channel.

#### 5.3.3.4 AnalogModule ∗ AnalogModule::GetInstance (UINT32 *slot*) `[static]`

Get an instance of an Analog Module.

Singleton analog module creation where a module is allocated on the first use and the same module is returned on subsequent uses.

**Parameters:**

***slot*** The physical slot in the cRIO chassis where this analog module is installed.

**Returns:**

A pointer to the AnalogModule.

#### 5.3.3.5 UINT32 AnalogModule::GetLSBWeight (UINT32 *channel*)

Get the factory scaling least significant bit weight constant. The least significant bit weight constant for the channel that was calibrated in manufacturing and stored in an eeprom in the module.

Volts = ((LSB_Weight ∗ 1e-9) ∗ raw) - (Offset ∗ 1e-9)

**Parameters:**

***channel*** The channel to get calibration data for.

**Returns:**

Least significant bit weight.

#### 5.3.3.6 INT32 AnalogModule::GetOffset (UINT32 *channel*)

Get the factory scaling offset constant. The offset constant for the channel that was calibrated in manufacturing and stored in an eeprom in the module.

Volts = ((LSB_Weight ∗ 1e-9) ∗ raw) - (Offset ∗ 1e-9)

**Parameters:**

***channel*** The channel to get calibration data for.

**Returns:**

Offset constant.

#### 5.3.3.7 UINT32 AnalogModule::GetOversampleBits (UINT32 *channel*)

Get the number of oversample bits.

This gets the number of oversample bits from the FPGA. The actual number of oversampled values is 2∗∗bits. The oversampling is done automatically in the FPGA.

**Parameters:**

***channel*** Channel to address.

**Returns:**

Bits to oversample.

#### 5.3.3.8 float AnalogModule::GetSampleRate ()

Get the current sample rate on the module.

This assumes one entry in the scan list. This is a global setting for the module and effects all channels.

**Returns:**

Sample rate.

#### 5.3.3.9 INT16 AnalogModule::GetValue (UINT32 *channel*)

Get a sample straight from the channel on this module.

The sample is a 12-bit value representing the -10V to 10V range of the A/D converter in the module. The units are in A/D converter codes. Use GetVoltage() to get the analog value in calibrated units.

**Returns:**

A sample straight from the channel on this module.

#### 5.3.3.10 float AnalogModule::GetVoltage (UINT32 *channel*)

Get a scaled sample straight from the channel on this module.

The value is scaled to units of Volts using the calibrated scaling data from GetLSBWeight() and GetOffset().

**Parameters:**

***channel*** The channel to read.

**Returns:**

A scaled sample straight from the channel on this module.

#### 5.3.3.11 void AnalogModule::SetAverageBits (UINT32 *channel*, UINT32 *bits*)

Set the number of averaging bits.

This sets the number of averaging bits. The actual number of averaged samples is 2∗∗bits. Use averaging to improve the stability of your measurement at the expense of sampling rate. The averaging is done automatically in the FPGA.

**Parameters:**

***channel*** Analog channel to configure.

***bits*** Number of bits to average.

#### 5.3.3.12 void AnalogModule::SetOversampleBits (UINT32 *channel*, UINT32 *bits*)

Set the number of oversample bits.

This sets the number of oversample bits. The actual number of oversampled values is 2∗∗bits. Use oversampling to improve the resolution of your measurements at the expense of sampling rate. The oversampling is done automatically in the FPGA.

**Parameters:**

***channel*** Analog channel to configure.

***bits*** Number of bits to oversample.

#### 5.3.3.13 void AnalogModule::SetSampleRate (float *samplesPerSecond*)

Set the sample rate on the module.

This is a global setting for the module and effects all channels.

**Parameters:**

***samplesPerSecond*** The number of samples per channel per second.

#### 5.3.3.14 UINT32 AnalogModule::SlotToIndex (UINT32 *slot*) `[static]`

Convert slot number to index.

**Parameters:**

***slot*** The slot in the chassis where the module is plugged in.

**Returns:**

An index to represent the module internally.

#### 5.3.3.15 INT32 AnalogModule::VoltsToValue (INT32 *channel*, float *voltage*)

Convert a voltage to a raw value for a specified channel.

This process depends on the calibration of each channel, so the channel must be specified.

**Todo**

This assumes raw values. Oversampling not supported as is.

**Parameters:**

***channel*** The channel to convert for.

***voltage*** The voltage to convert.

**Returns:**

The raw value for the channel.

The documentation for this class was generated from the following files:

- AnalogModule.h
- AnalogModule.cpp

## 5.4 AnalogTriggerOutput Class Reference

`#include <AnalogTriggerOutput.h>`

Inherits DigitalSource.

### Public Member Functions

- bool Get ()
- virtual UINT32 GetChannelForRouting ()
- virtual UINT32 GetModuleForRouting ()
- virtual bool GetAnalogTriggerForRouting ()
- virtual void RequestInterrupts (tInterruptHandler handler)

  *Asynchronus handler version.*
- virtual void RequestInterrupts ()

  *Synchronus Wait version.*

### Protected Member Functions

- AnalogTriggerOutput (AnalogTrigger ∗trigger, Type outputType)

### 5.4.1 Detailed Description

Class to represent a specific output from an analog trigger. This class is used to get the current output value and also as a DigitalSource to provide routing of an output to digital subsystems on the FPGA such as Counter, Encoder, and Interrupt.

The TriggerState output indicates the primary output value of the trigger. If the analog signal is less than the lower limit, the output is false. If the analog value is greater than the upper limit, then the output is true. If the analog value is in between, then the trigger output state maintains its most recent value.

The InWindow output indicates whether or not the analog signal is inside the range defined by the limits.

The RisingPulse and FallingPulse outputs detect an instantaneous transition from above the upper limit to below the lower limit, and vise versa. These pulses represent a rollover condition of a sensor and can be routed to an up / down couter or to interrupts. Because the outputs generate a pulse, they cannot be read directly. To help ensure that a rollover condition is not missed, there is an average rejection filter available that operates on the upper 8 bits of a 12 bit number and selects the nearest outlyer of 3 samples. This will reject a sample that is (due to averaging or sampling) errantly between the two limits. This filter will fail if more than one sample in a row is errantly in between the two limits. You may see this problem if attempting to use this feature with a mechanical rollover sensor, such as a 360 degree no-stop potentiometer without signal conditioning, because the rollover transition is not sharp / clean enough. Using the averaging engine may help with this, but rotational speeds of the sensor will then be limited.

### 5.4.2 Constructor & Destructor Documentation

#### 5.4.2.1 AnalogTriggerOutput::AnalogTriggerOutput (AnalogTrigger ∗ *trigger*, AnalogTriggerOutput::Type *outputType*) `[protected]`

Create an object that represents one of the four outputs from an analog trigger.

Because this class derives from DigitalSource, it can be passed into routing functions for Counter, Encoder, etc.

**Parameters:**

***trigger*** A pointer to the trigger for which this is an output.

***outputType*** An enum that specifies the output on the trigger to represent.

## 5.4.3 Member Function Documentation

### 5.4.3.1 bool AnalogTriggerOutput::Get ()

Get the state of the analog trigger output.

**Returns:**

The state of the analog trigger output.

### 5.4.3.2 bool AnalogTriggerOutput::GetAnalogTriggerForRouting () `[virtual]`

**Returns:**

The value to be written to the module field of a routing mux.

Implements DigitalSource.

### 5.4.3.3 UINT32 AnalogTriggerOutput::GetChannelForRouting () `[virtual]`

**Returns:**

The value to be written to the channel field of a routing mux.

Implements DigitalSource.

### 5.4.3.4 UINT32 AnalogTriggerOutput::GetModuleForRouting () `[virtual]`

**Returns:**

The value to be written to the module field of a routing mux.

Implements DigitalSource.

### 5.4.3.5 void AnalogTriggerOutput::RequestInterrupts () `[virtual]`

Synchronus Wait version.

Request interrupts synchronously on this digital input.

Implements DigitalSource.

#### 5.4.3.6 void AnalogTriggerOutput::RequestInterrupts (tInterruptHandler *handler*) `[virtual]`

Asynchronus handler version.

Request interrupts asynchronously on this digital input.

Implements DigitalSource.

The documentation for this class was generated from the following files:

- AnalogTriggerOutput.h
- AnalogTriggerOutput.cpp

## 5.5 ColorReport_struct Struct Reference

```
#include <VisionAPI.h>
```

### 5.5.1 Detailed Description

Tracking functions return this structure

The documentation for this struct was generated from the following file:

- VisionAPI.h

## 5.6 Compressor Class Reference

`#include <Compressor.h>`

Inherits SensorBase.

### Public Member Functions

- Compressor (UINT32 pressureSwitchChannel, UINT32 compressorRelayChannel)
- Compressor (UINT32 pressureSwitchSlot, UINT32 pressureSwitchChannel, UINT32 compresssorRelaySlot, UINT32 compressorRelayChannel)
- ~Compressor ()
- void Start ()
- void Stop ()
- bool Enabled ()
- UINT32 GetPressureSwitchValue ()
- void SetRelayValue (Relay::Value relayValue)

### 5.6.1 Detailed Description

Compressor object. The Compressor object is designed to handle the operation of the compressor, pressure sensor and relay for a FIRST robot pneumatics system. The Compressor object starts a task which runs in the backround and periodically polls the pressure sensor and operates the relay that controls the compressor.

### 5.6.2 Constructor & Destructor Documentation

#### 5.6.2.1 Compressor::Compressor (UINT32 *pressureSwitchChannel*, UINT32 *compressorRelayChannel*)

Compressor constructor. Given a relay channel and pressure switch channel (both in the default digital module), initialize the Compressor object.

**Parameters:**

***pressureSwitchChannel*** The GPIO channel that the pressure switch is attached to.

***compressorRelayChannel*** The relay channel that the compressor relay is attached to.

#### 5.6.2.2 Compressor::Compressor (UINT32 *pressureSwitchSlot*, UINT32 *pressureSwitchChannel*, UINT32 *compresssorRelaySlot*, UINT32 *compressorRelayChannel*)

Compressor constructor. Given a fully specified relay channel and pressure switch channel, initialize the Compressor object.

**Parameters:**

***pressureSwitchSlot*** The module that the pressure switch is attached to.

***pressureSwitchChannel*** The GPIO channel that the pressure switch is attached to.

***compresssorRelaySlot*** The module that the compressor relay is attached to.

***compressorRelayChannel*** The relay channel that the compressor relay is attached to.

#### 5.6.2.3 Compressor::∼Compressor ()

Delete the Compressor object. Delete the allocated resources for the compressor and kill the compressor task that is polling the pressure switch.

### 5.6.3 Member Function Documentation

#### 5.6.3.1 bool Compressor::Enabled ()

Get the state of the enabled flag. Return the state of the enabled flag for the compressor and pressure switch combination.

**Returns:**

The state of the compressor thread's enable flag.

#### 5.6.3.2 UINT32 Compressor::GetPressureSwitchValue ()

Get the pressure switch value. Read the pressure switch digital input.

**Returns:**

The current state of the pressure switch.

#### 5.6.3.3 void Compressor::SetRelayValue (Relay::Value *relayValue*)

Operate the relay for the compressor. Change the value of the relay output that is connected to the compressor motor. This is only intended to be called by the internal polling thread.

#### 5.6.3.4 void Compressor::Start ()

Start the compressor. This method will allow the polling loop to actually operate the compressor.

#### 5.6.3.5 void Compressor::Stop ()

Stop the compressor. This method will stop the compressor from turning on.

The documentation for this class was generated from the following files:

- Compressor.h
- Compressor.cpp

## 5.7 Counter Class Reference

```
#include <Counter.h>
```

Inherits SensorBase, and CounterBase.

Inherited by GearTooth.

### Public Member Functions

- Counter ()
- Counter (UINT32 channel)
- Counter (UINT32 slot, UINT32 channel)
- Counter (DigitalSource ∗source)
- Counter (AnalogTrigger ∗trigger)
- virtual ∼Counter ()
- void SetUpSource (UINT32 channel)
- void SetUpSource (UINT32 slot, UINT32 channel)
- void SetUpSource (AnalogTrigger ∗analogTrigger, AnalogTriggerOutput::Type triggerType)
- void SetUpSource (AnalogTrigger &analogTrigger, AnalogTriggerOutput::Type triggerType)
- void SetUpSource (DigitalSource ∗source)
- void SetUpSource (DigitalSource &source)
- void SetUpSourceEdge (bool risingEdge, bool fallingEdge)
- void ClearUpSource ()
- void SetDownSource (UINT32 channel)
- void SetDownSource (UINT32 slot, UINT32 channel)
- void SetDownSource (AnalogTrigger ∗analogTrigger, AnalogTriggerOutput::Type triggerType)
- void SetDownSource (AnalogTrigger &analogTrigger, AnalogTriggerOutput::Type triggerType)
- void SetDownSource (DigitalSource ∗source)
- void SetDownSource (DigitalSource &source)
- void SetDownSourceEdge (bool risingEdge, bool fallingEdge)
- void ClearDownSource ()
- void SetUpDownCounterMode ()
- void SetExternalDirectionMode ()
- void SetSemiPeriodMode (bool highRisingPeriod)
- void SetPulseLengthMode (float threshold)
- void Start ()
- INT32 Get ()
- void Reset ()
- void Stop ()
- void SetMaxPeriod (double maxPeriod)
- void SetUpdateWhenEmpty (bool enabled)
- bool GetStopped ()
- bool GetDirection ()

### 5.7.1 Detailed Description

Class for counting the number of ticks on a digital input channel. This is a general purpose class for counting repetitive events. It can return the number of counts, the period of the most recent cycle, and detect when the signal being counted has stopped by supplying a maximum cycle time.

## 5.7.2 Constructor & Destructor Documentation

### 5.7.2.1 Counter::Counter ()

Create an instance of a counter where no sources are selected. Then they all must be selected by calling functions to specify the upsource and the downsource independently.

### 5.7.2.2 Counter::Counter (UINT32 *channel*) `[explicit]`

Create an instance of a Counter object. Create an up-Counter instance given a channel. The default digital module is assumed.

### 5.7.2.3 Counter::Counter (UINT32 *slot*, UINT32 *channel*)

Create an instance of a Counter object. Create an instance of an up-Counter given a digital module and a channel.

**Parameters:**

- ***slot*** The cRIO chassis slot for the digital module used
- ***channel*** The channel in the digital module

### 5.7.2.4 Counter::Counter (DigitalSource * *source*) `[explicit]`

Create an instance of a counter from a Digital Input. This is used if an existing digital input is to be shared by multiple other objects such as encoders.

### 5.7.2.5 Counter::Counter (AnalogTrigger * *trigger*) `[explicit]`

Create an instance of a Counter object. Create an instance of a simple up-Counter given an analog trigger. Use the trigger state output from the analog trigger.

### 5.7.2.6 Counter::∼Counter () `[virtual]`

Delete the Counter object.

## 5.7.3 Member Function Documentation

### 5.7.3.1 void Counter::ClearDownSource ()

Disable the down counting source to the counter.

### 5.7.3.2 void Counter::ClearUpSource ()

Disable the up counting source to the counter.

#### 5.7.3.3 INT32 Counter::Get () [virtual]

Read the current counter value. Read the value at this instant. It may still be running, so it reflects the current value. Next time it is read, it might have a different value.

Implements CounterBase.

#### 5.7.3.4 bool Counter::GetDirection () [virtual]

The last direction the counter value changed.

**Returns:**

The last direction the counter value changed.

Implements CounterBase.

#### 5.7.3.5 bool Counter::GetStopped () [virtual]

Determine if the clock is stopped. Determine if the clocked input is stopped based on the MaxPeriod value set using the SetMaxPeriod method. If the clock exceeds the MaxPeriod, then the device (and counter) are assumed to be stopped and it returns true.

**Returns:**

Returns true if the most recent counter period exceeds the MaxPeriod value set by SetMaxPeriod.

Implements CounterBase.

#### 5.7.3.6 void Counter::Reset () [virtual]

Reset the Counter to zero. Set the counter value to zero. This doesn't effect the running state of the counter, just sets the current value to zero.

Implements CounterBase.

#### 5.7.3.7 void Counter::SetDownSource (DigitalSource & *source*)

Set the source object that causes the counter to count down. Set the down counting DigitalSource.

#### 5.7.3.8 void Counter::SetDownSource (DigitalSource ∗ *source*)

Set the source object that causes the counter to count down. Set the down counting DigitalSource.

#### 5.7.3.9 void Counter::SetDownSource (AnalogTrigger & *analogTrigger*, AnalogTriggerOutput::Type *triggerType*)

Set the down counting source to be an analog trigger.

**Parameters:**

***analogTrigger*** The analog trigger object that is used for the Down Source

***triggerType*** The analog trigger output that will trigger the counter.

#### 5.7.3.10 void Counter::SetDownSource (AnalogTrigger * *analogTrigger*, AnalogTriggerOutput::Type *triggerType*)

Set the down counting source to be an analog trigger.

**Parameters:**

***analogTrigger*** The analog trigger object that is used for the Down Source

***triggerType*** The analog trigger output that will trigger the counter.

#### 5.7.3.11 void Counter::SetDownSource (UINT32 *slot*, UINT32 *channel*)

Set the down counting source to be a digital input slot and channel.

#### 5.7.3.12 void Counter::SetDownSource (UINT32 *channel*)

Set the down counting source to be a digital input channel. The slot will be set to the default digital module slot.

#### 5.7.3.13 void Counter::SetDownSourceEdge (bool *risingEdge*, bool *fallingEdge*)

Set the edge sensitivity on a down counting source. Set the down source to either detect rising edges or falling edges.

#### 5.7.3.14 void Counter::SetExternalDirectionMode ()

Set external direction mode on this counter. Counts are sourced on the Up counter input. The Down counter input represents the direction to count.

#### 5.7.3.15 void Counter::SetMaxPeriod (double *maxPeriod*) `[virtual]`

Set the maximum period where the device is still considered "moving". Sets the maximum period where the device is considered moving. This value is used to determine the "stopped" state of the counter using the GetStopped method.

**Parameters:**

***maxPeriod*** The maximum period where the counted device is considered moving in seconds.

Implements CounterBase.

#### 5.7.3.16 void Counter::SetPulseLengthMode (float *threshold*)

Configure the counter to count in up or down based on the length of the input pulse. This mode is most useful for direction sensitive gear tooth sensors.

**Parameters:**

***threshold*** The pulse length beyond which the counter counts the opposite direction. Units are seconds.

#### 5.7.3.17 void Counter::SetSemiPeriodMode (bool *highSemiPeriod*)

Set Semi-period mode on this counter. Counts up on both rising and falling edges.

#### 5.7.3.18 void Counter::SetUpdateWhenEmpty (bool *enabled*)

Select whether you want to continue updating the event timer output when there are no samples captured. The output of the event timer has a buffer of periods that are averaged and posted to a register on the FPGA. When the timer detects that the event source has stopped (based on the MaxPeriod) the buffer of samples to be averaged is emptied. If you enable the update when empty, you will be notified of the stopped source and the event time will report 0 samples. If you disable update when empty, the most recent average will remain on the output until a new sample is acquired. You will never see 0 samples output (except when there have been no events since an FPGA reset) and you will likely not see the stopped bit become true (since it is updated at the end of an average and there are no samples to average).

#### 5.7.3.19 void Counter::SetUpDownCounterMode ()

Set standard up / down counting mode on this counter. Up and down counts are sourced independently from two inputs.

#### 5.7.3.20 void Counter::SetUpSource (DigitalSource & *source*)

Set the source object that causes the counter to count up. Set the up counting DigitalSource.

#### 5.7.3.21 void Counter::SetUpSource (DigitalSource * *source*)

Set the source object that causes the counter to count up. Set the up counting DigitalSource.

#### 5.7.3.22 void Counter::SetUpSource (AnalogTrigger & *analogTrigger*, AnalogTriggerOutput::Type *triggerType*)

Set the up counting source to be an analog trigger.

**Parameters:**

***analogTrigger*** The analog trigger object that is used for the Up Source

***triggerType*** The analog trigger output that will trigger the counter.

#### 5.7.3.23 void Counter::SetUpSource (AnalogTrigger * *analogTrigger*, AnalogTriggerOutput::Type *triggerType*)

Set the up counting source to be an analog trigger.

**Parameters:**

***analogTrigger*** The analog trigger object that is used for the Up Source

***triggerType*** The analog trigger output that will trigger the counter.

#### 5.7.3.24 void Counter::SetUpSource (UINT32 *slot*, UINT32 *channel*)

Set the up source for the counter as digital input channel and slot.

#### 5.7.3.25 void Counter::SetUpSource (UINT32 *channel*)

Set the upsource for the counter as a digital input channel. The slot will be the default digital module slot.

#### 5.7.3.26 void Counter::SetUpSourceEdge (bool *risingEdge*, bool *fallingEdge*)

Set the edge sensitivity on an up counting source. Set the up source to either detect rising edges or falling edges.

#### 5.7.3.27 void Counter::Start () `[virtual]`

Start the Counter counting. This enables the counter and it starts accumulating counts from the associated input channel. The counter value is not reset on starting, and still has the previous value.

Implements CounterBase.

#### 5.7.3.28 void Counter::Stop () `[virtual]`

Stop the Counter. Stops the counting but doesn't effect the current value.

Implements CounterBase.

The documentation for this class was generated from the following files:

- Counter.h
- Counter.cpp

## 5.8 CounterBase Class Reference

```
#include <CounterBase.h>
```

Inherited by Counter, and Encoder.

### 5.8.1 Detailed Description

Interface for counting the number of ticks on a digital input channel. Encoders, Gear tooth sensors, and counters should all subclass this so it can be used to build more advanced classes for control and driving.

The documentation for this class was generated from the following file:

- CounterBase.h

## 5.9 Dashboard Class Reference

`#include <Dashboard.h>`

Inherits ErrorBase.

### Public Member Functions

- void AddI8 (INT8 value)
- void AddI16 (INT16 value)
- void AddI32 (INT32 value)
- void AddU8 (UINT8 value)
- void AddU16 (UINT16 value)
- void AddU32 (UINT32 value)
- void AddFloat (float value)
- void AddDouble (double value)
- void AddBoolean (bool value)
- void AddString (char *value)
- void AddString (char *value, INT32 length)
- void AddArray (void)
- void FinalizeArray (void)
- void AddCluster (void)
- void FinalizeCluster (void)
- void Printf (const char *writeFmt,...)
- INT32 Finalize (void)

### Friends

- class DriverStation

### 5.9.1 Detailed Description

Pack data into the "user data" field that gets sent to the dashboard laptop via the driver station.

### 5.9.2 Member Function Documentation

#### 5.9.2.1 void Dashboard::AddArray (void)

Start an array in the packed dashboard data structure.

After calling AddArray(), call the appropriate Add method for each element of the array. Make sure you call the same add each time. An array must contain elements of the same type. You can use clusters inside of arrays to make each element of the array contain a structure of values. You can also nest arrays inside of other arrays. Every call to AddArray() must have a matching call to FinalizeArray().

#### 5.9.2.2 void Dashboard::AddBoolean (bool *value*)

Pack a boolean into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.3 void Dashboard::AddCluster (void)

Start a cluster in the packed dashboard data structure.

After calling AddCluster(), call the appropriate Add method for each element of the cluster. You can use clusters inside of arrays to make each element of the array contain a structure of values. Every call to AddCluster() must have a matching call to FinalizeCluster().

#### 5.9.2.4 void Dashboard::AddDouble (double *value*)

Pack a 64-bit floating point number into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.5 void Dashboard::AddFloat (float *value*)

Pack a 32-bit floating point number into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.6 void Dashboard::AddI16 (INT16 *value*)

Pack a signed 16-bit int into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.7 void Dashboard::AddI32 (INT32 *value*)

Pack a signed 32-bit int into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.8 void Dashboard::AddI8 (INT8 *value*)

Pack a signed 8-bit int into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.9 void Dashboard::AddString (char $*$ *value*, INT32 *length*)

Pack a string of 8-bit characters of specified length into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

***length*** The number of bytes in the string to pack.

#### 5.9.2.10 void Dashboard::AddString (char $*$ *value*)

Pack a NULL-terminated string of 8-bit characters into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.11 void Dashboard::AddU16 (UINT16 *value*)

Pack an unsigned 16-bit int into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.12 void Dashboard::AddU32 (UINT32 *value*)

Pack an unsigned 32-bit int into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.13 void Dashboard::AddU8 (UINT8 *value*)

Pack an unsigned 8-bit int into the dashboard data structure.

**Parameters:**

***value*** Data to be packed into the structure.

#### 5.9.2.14 INT32 Dashboard::Finalize (void)

Indicate that the packing is complete and commit the buffer to the DriverStation.

The packing of the dashboard packet is complete. If you are not using the packed dashboard data, you can call Finalize() to commit the Printf() buffer and the error string buffer. In effect, you are packing an empty structure. Prepares a packet to go to the dashboard... Pack the sequence number, Printf() buffer, the errors messages (not implemented yet), and packed dashboard data buffer.

**Returns:**

The total size of the data packed into the userData field of the status packet.

$<$ TODO: add error reporting strings.

#### 5.9.2.15 void Dashboard::FinalizeArray (void)

Indicate the end of an array packed into the dashboard data structure.

After packing data into the array, call FinalizeArray(). Every call to AddArray() must have a matching call to FinalizeArray().

#### 5.9.2.16 void Dashboard::FinalizeCluster (void)

Indicate the end of a cluster packed into the dashboard data structure.

After packing data into the cluster, call FinalizeCluster(). Every call to AddCluster() must have a matching call to FinalizeCluster().

#### 5.9.2.17 void Dashboard::Printf (const char * *writeFmt*, ...)

Print a string to the UserData text on the Dashboard.

This will add text to the buffer to send to the dashboard. You must call Finalize() periodically to actually send the buffer to the dashboard if you are not using the packed dashboard data.

The documentation for this class was generated from the following files:

- Dashboard.h
- Dashboard.cpp

## 5.10 DigitalInput Class Reference

`#include <DigitalInput.h>`

Inherits DigitalSource.

### Public Member Functions

- DigitalInput (UINT32 channel)
- DigitalInput (UINT32 slot, UINT32 channel)
- ∼DigitalInput ()
- UINT32 GetChannel ()
- virtual UINT32 GetChannelForRouting ()
- virtual UINT32 GetModuleForRouting ()
- virtual bool GetAnalogTriggerForRouting ()
- virtual void RequestInterrupts (tInterruptHandler handler)

  *Asynchronus handler version.*

- virtual void RequestInterrupts ()

  *Synchronus Wait version.*

### 5.10.1 Detailed Description

Class to read a digital input. This class will read digital inputs and return the current value on the channel. Other devices such as encoders, gear tooth sensors, etc. that are implemented elsewhere will automatically allocate digital inputs and outputs as required. This class is only for devices like switches etc. that aren't implemented anywhere else.

### 5.10.2 Constructor & Destructor Documentation

#### 5.10.2.1 DigitalInput::DigitalInput (UINT32 *channel*) `[explicit]`

Create an instance of a Digital Input class. Creates a digital input given a channel and uses the default module.

#### 5.10.2.2 DigitalInput::DigitalInput (UINT32 *slot*, UINT32 *channel*)

Create an instance of a Digital Input class. Creates a digital input given an channel and module.

#### 5.10.2.3 DigitalInput::∼DigitalInput ()

Free resources associated with the Digital Input class.

### 5.10.3 Member Function Documentation

#### 5.10.3.1 bool DigitalInput::GetAnalogTriggerForRouting () `[virtual]`

**Returns:**

The value to be written to the analog trigger field of a routing mux.

Implements DigitalSource.

#### 5.10.3.2 UINT32 DigitalInput::GetChannel ()

**Returns:**

The GPIO channel number that this object represents.

#### 5.10.3.3 UINT32 DigitalInput::GetChannelForRouting () `[virtual]`

**Returns:**

The value to be written to the channel field of a routing mux.

Implements DigitalSource.

#### 5.10.3.4 UINT32 DigitalInput::GetModuleForRouting () `[virtual]`

**Returns:**

The value to be written to the module field of a routing mux.

Implements DigitalSource.

#### 5.10.3.5 void DigitalInput::RequestInterrupts () `[virtual]`

Synchronus Wait version.

Request interrupts synchronously on this digital input. Request interrupts in synchronus mode where the user program will have to explicitly wait for the interrupt to occur. The default is interrupt on rising edges only.

Implements DigitalSource.

#### 5.10.3.6 void DigitalInput::RequestInterrupts (tInterruptHandler *handler*) `[virtual]`

Asynchronus handler version.

Request interrupts asynchronously on this digital input.

**Parameters:**

***handler*** The address of the interrupt handler function of type tInterruptHandler that will be called whenever there is an interrupt on the digitial input port. Request interrupts in synchronus mode where the user program interrupt handler will be called when an interrupt occurs. The default is interrupt on rising edges only.

Implements DigitalSource.

The documentation for this class was generated from the following files:

- DigitalInput.h
- DigitalInput.cpp

## 5.11 DigitalOutput Class Reference

```
#include <DigitalOutput.h>
```

Inherits SensorBase.

### Public Member Functions

- DigitalOutput (UINT32 channel)
- DigitalOutput (UINT32 slot, UINT32 channel)
- ~DigitalOutput ()
- void Set (UINT32 value)
- void Pulse (float length)
- bool IsPulsing ()

### 5.11.1 Detailed Description

Class to write to digital outputs. Write values to the digital output channels. Other devices implemented elsewhere will allocate channels automatically so for those devices it shouldn't be done here.

### 5.11.2 Constructor & Destructor Documentation

#### 5.11.2.1 DigitalOutput::DigitalOutput (UINT32 *channel*) `[explicit]`

Create an instance of a digital output. Create a digital output given a channel. The default module is used.

#### 5.11.2.2 DigitalOutput::DigitalOutput (UINT32 *slot*, UINT32 *channel*)

Create an instance of a digital output. Create an instance of a digital output given a slot and channel.

#### 5.11.2.3 DigitalOutput::~DigitalOutput ()

Free the resources associated with a digital output.

### 5.11.3 Member Function Documentation

#### 5.11.3.1 bool DigitalOutput::IsPulsing ()

Determine if the pulse is still going. Determine if a previously started pulse is still going.

#### 5.11.3.2 void DigitalOutput::Pulse (float *length*)

Output a single pulse on the digital output line. Send a single pulse on the digital output line where the pulse diration is specified in seconds. Maximum pulse length is 0.0016 seconds.

**Parameters:**

***length*** The pulselength in seconds

#### 5.11.3.3 void DigitalOutput::Set (UINT32 *value*)

Set the value of a digital output. Set the value of a digital output to either one (true) or zero (false).

The documentation for this class was generated from the following files:

- DigitalOutput.h
- DigitalOutput.cpp

# 5.12 DigitalSource Class Reference

`#include <DigitalSource.h>`

Inherits InterruptableSensorBase.

Inherited by AnalogTriggerOutput, and DigitalInput.

## Public Member Functions

- virtual ~DigitalSource ()

### 5.12.1 Detailed Description

DigitalSource Interface. The DigitalSource represents all the possible inputs for a counter or a quadrature encoder. The source may be either a digital input or an analog input. If the caller just provides a channel, then a digital input will be constructed and freed when finished for the source. The source can either be a digital input or analog trigger but not both.

### 5.12.2 Constructor & Destructor Documentation

#### 5.12.2.1 DigitalSource::~DigitalSource () `[virtual]`

DigitalSource destructor.

The documentation for this class was generated from the following files:

- DigitalSource.h
- DigitalSource.cpp

## 5.13 DriverStation Class Reference

`#include <DriverStation.h>`

Inherits SensorBase.

### Public Member Functions

- float GetStickAxis (UINT32 stick, UINT32 axis)
- short GetStickButtons (UINT32 stick)
- float GetAnalogIn (UINT32 channel)
- bool GetDigitalIn (UINT32 channel)
- void SetDigitalOut (UINT32 channel, bool value)
- bool GetDigitalOut (UINT32 channel)
- UINT32 GetPacketNumber ()
- float GetBatteryVoltage ()

### Static Public Member Functions

- static DriverStation ∗ GetInstance ()

### Protected Member Functions

- DriverStation ()
- void GetData ()
- void SetData ()

### 5.13.1 Detailed Description

Provide access to the network communication data to / from the Driver Station.

### 5.13.2 Constructor & Destructor Documentation

#### 5.13.2.1 DriverStation::DriverStation () `[protected]`

DriverStation contructor.

This is only called once the first time GetInstance() is called

### 5.13.3 Member Function Documentation

#### 5.13.3.1 float DriverStation::GetAnalogIn (UINT32 *channel*)

Get an analog voltage from the Driver Station. The analog values are returned as UINT32 values for the Driver Station analog inputs. These inputs are typically used for advanced operator interfaces consisting of potentiometers or resistor networks representing values on a rotary switch.

**Parameters:**

***channel*** The analog input channel on the driver station to read from. Valid range is 1 - 4.

**Returns:**

The analog voltage on the input.

#### 5.13.3.2 float DriverStation::GetBatteryVoltage ()

Read the battery voltage from the specified AnalogChannel.

This accessor assumes that the battery voltage is being measured through the voltage divider on an analog breakout.

**Returns:**

The battery voltage.

#### 5.13.3.3 void DriverStation::GetData () `[protected]`

Copy data from the DS task for the user. If no new data exists, it will just be returned, otherwise the data will be copied from the DS polling loop.

#### 5.13.3.4 bool DriverStation::GetDigitalIn (UINT32 *channel*)

Get values from the digital inputs on the Driver Station. Return digital values from the Drivers Station. These values are typically used for buttons and switches on advanced operator interfaces.

**Parameters:**

***channel*** The digital input to get. Valid range is 1 - 8.

#### 5.13.3.5 bool DriverStation::GetDigitalOut (UINT32 *channel*)

Get a value that was set for the digital outputs on the Driver Station.

**Parameters:**

***channel*** The digital ouput to monitor. Valid range is 1 through 8.

**Returns:**

A digital value being output on the Drivers Station.

#### 5.13.3.6 DriverStation * DriverStation::GetInstance () `[static]`

Return a pointer to the singleton DriverStation.

#### 5.13.3.7 UINT32 DriverStation::GetPacketNumber ()

Return the DS packet number. The packet number is the index of this set of data returned by the driver station. Each time new data is received, the packet number (included with the sent data) is returned.

#### 5.13.3.8 float DriverStation::GetStickAxis (UINT32 *stick*, UINT32 *axis*)

Get the value of the axis on a joystick. This depends on the mapping of the joystick connected to the specified port.

**Parameters:**

***stick*** The joystick to read.

***axis*** The analog axis value to read from the joystick.

**Returns:**

The value of the axis on the joystick.

#### 5.13.3.9 short DriverStation::GetStickButtons (UINT32 *stick*)

The state of the buttons on the joystick. 12 buttons (4 msb are unused) from the joystick.

**Parameters:**

***stick*** The joystick to read.

**Returns:**

The state of the buttons on the joystick.

#### 5.13.3.10 void DriverStation::SetData () `[protected]`

Copy status data from the DS task for the user. This is used primarily to set digital outputs on the DS.

#### 5.13.3.11 void DriverStation::SetDigitalOut (UINT32 *channel*, bool *value*)

Set a value for the digital outputs on the Driver Station.

Control digital outputs on the Drivers Station. These values are typically used for giving feedback on a custom operator station such as LEDs.

**Parameters:**

***channel*** The digital output to set. Valid range is 1 - 8.

***value*** The state to set the digital output.

The documentation for this class was generated from the following files:

- DriverStation.h
- DriverStation.cpp

## 5.14 Encoder Class Reference

`#include <Encoder.h>`

Inherits SensorBase, and CounterBase.

### Public Member Functions

- Encoder (UINT32 aChannel, UINT32 bChannel, bool reverseDirection=false)
- Encoder (UINT32 aSlot, UINT32 aChannel, UINT32 bSlot, UINT32 _bChannel, bool reverseDirection=false)
- Encoder (DigitalSource *aSource, DigitalSource *bSource, bool reverseDirection=false)
- virtual ~Encoder ()
- void Start ()
- INT32 Get ()
- void Reset ()
- void Stop ()
- double GetPeriod ()
- void SetMaxPeriod (double maxPeriod)
- bool GetStopped ()
- bool GetDirection ()
- float GetDistance ()
- void SetDistancePerTick (float distancePerTick)
- void SetReverseDirection (bool reverseDirection)

### 5.14.1 Detailed Description

Class to read quad encoders. Quadrature encoders are devices that count shaft rotation and can sense direction. The output of the QuadEncoder class is an integer that can count either up or down, and can go negative for reverse direction counting. When creating QuadEncoders, a direction is supplied that changes the sense of the output to make code more readable if the encoder is mounted such that forward movement generates negative values. Quadrature encoders have two digital outputs, an A Channel and a B Channel that are out of phase with each other to allow the FPGA to do direction sensing.

### 5.14.2 Constructor & Destructor Documentation

#### 5.14.2.1 Encoder::Encoder (UINT32 *aChannel*, UINT32 *bChannel*, bool *reverseDirection* = `false`)

QuadEncoder constructor. Construct a QuadEncoder given a and b channels assuming the default module.

**Parameters:**

***aChannel*** The a channel digital input channel.

***bChannel*** The b channel digital input channel.

***reverseDirection*** represents the orientation of the encoder and inverts the output values if necessary so forward represents positive values.

#### 5.14.2.2 Encoder::Encoder (UINT32 *aSlot*, UINT32 *aChannel*, UINT32 *bSlot*, UINT32 *bChannel*, bool *reverseDirection* = `false`)

QuadEncoder constructor. Construct a QuadEncoder given a and b modules and channels fully specified.

**Parameters:**

***aSlot*** The a channel digital input module.

***aChannel*** The a channel digital input channel.

***bSlot*** The b channel digital input module.

***bChannel*** The b channel digital input channel.

***reverseDirection*** represents the orientation of the encoder and inverts the output values if necessary so forward represents positive values.

#### 5.14.2.3 Encoder::Encoder (DigitalSource ∗ *aSource*, DigitalSource ∗ *bSource*, bool *reverseDirection* = `false`)

QuadEncoder constructor. Construct a QuadEncoder given a and b channels as digital inputs. This is used in the case where the digital inputs are shared. The QuadEncoder class will not allocate the digital inputs and assume that they already are counted.

**Parameters:**

***aSource*** The source that should be used for the a channel.

***bSource*** the source that should be used for the b channel.

***reverseDirection*** represents the orientation of the encoder and inverts the output values if necessary so forward represents positive values.

#### 5.14.2.4 Encoder::∼Encoder () `[virtual]`

Free the resources for a QuadEncoder. Frees the FPGA resources associated with a Quad Encoder.

### 5.14.3 Member Function Documentation

#### 5.14.3.1 INT32 Encoder::Get () `[virtual]`

Gets the current count. Returns the current count on the QuadEncoder. Quadrature encoders return 4x the expected number of counts since the hardware counts all 4 edges for higher resolution.

**Returns:**

Current count from the QuadEncoder.

Implements CounterBase.

#### 5.14.3.2 bool Encoder::GetDirection () [virtual]

The last direction the encoder value changed.

**Returns:**

The last direction the encoder value changed.

Implements CounterBase.

#### 5.14.3.3 float Encoder::GetDistance ()

Get the distance the robot has driven since the last reset

**Returns:**

The distance driven since the last reset based on the distance per tick variable being set by SetDistancePerTick(). It is just a simple multiplication, but makes the bookkeeping a little easier since the encoder remembers the scale factor.

#### 5.14.3.4 double Encoder::GetPeriod () [virtual]

Returns the period of the most recent pulse. Returns the period of the most recent Quad Encoder pulse in seconds.

**Returns:**

Period in seconds of the most recent pulse.

Implements CounterBase.

#### 5.14.3.5 bool Encoder::GetStopped () [virtual]

Determine if the encoder is stopped. Using the MaxPeriod value, a boolean is returned that is true if the encoder is considered stopped and false if it is still moving. A stopped encoder is one where the most recent pulse width exceeds the MaxPeriod.

**Returns:**

True if the encoder is considered stopped.

Implements CounterBase.

#### 5.14.3.6 void Encoder::Reset () [virtual]

Reset the QuadEncoder to zero. Resets the current count to zero on the encoder.

Implements CounterBase.

#### 5.14.3.7 void Encoder::SetDistancePerTick (float *distancePerTick*)

Set the distance per tick for this encoder. This sets the multiplier used to determine the distance driven based on the count value from the encoder. Reseting the encoder also resets the distance since it's just a simple multiply.

#### 5.14.3.8 void Encoder::SetMaxPeriod (double *maxPeriod*) [virtual]

Sets the maximum period for stopped detection. Sets the value that represents the maximum period of the QuadEncoder before it will assume that the attached device is stopped. This timeout allows users to determine if the wheels or other shaft has stopped rotating.

**Parameters:**

***maxPeriod*** The maximum time between rising and falling edges before the FPGA will consider the device stopped. This is expressed in seconds.

Implements CounterBase.

#### 5.14.3.9 void Encoder::SetReverseDirection (bool *reverseDirection*)

Set the direction sensing for this encoder. This sets the direction sensing on the encoder so that it couldl count in the correct software direction regardless of the mounting.

**Parameters:**

***reverseDirection*** true if the encoder direction should be reversed

#### 5.14.3.10 void Encoder::Start () [virtual]

Start the QuadEncoder. Starts counting pulses on the QuadEncoder device.

Implements CounterBase.

#### 5.14.3.11 void Encoder::Stop () [virtual]

Stops counting pulses on the QuadEncoder device. The value is not changed.

Implements CounterBase.

The documentation for this class was generated from the following files:

- Encoder.h
- Encoder.cpp

## 5.15 Error Class Reference

```
#include <Error.h>
```

### 5.15.1 Detailed Description

Error object represents a library error.

The documentation for this class was generated from the following files:

- Error.h
- Error.cpp

## 5.16 ErrorBase Class Reference

`#include <ErrorBase.h>`

Inherited by Dashboard, PCVideoServer, and SensorBase.

### Public Member Functions

- virtual Error & GetError ()

  *Retrieve the current error. Get the current error information associated with this sensor.*

- virtual void SetError (Error::Code code, const char ∗filename, UINT32 lineNumber) const

  *Set the current error information associated with this sensor.*

- virtual void ClearError ()

  *Clear the current error information associated with this sensor.*

- virtual bool StatusIsFatal () const

  *Check if the current error code represents a fatal error.*

### Static Public Member Functions

- static Error & GetGlobalError ()

### Protected Member Functions

- ErrorBase ()

  *Initialize the instance status to 0 for now.*

### 5.16.1 Detailed Description

Base class for most objects. ErrorBase is the base class for most objects since it holds the generated error for that object. In addition, there is a single instance of a global error object

### 5.16.2 Member Function Documentation

#### 5.16.2.1 Error & ErrorBase::GetGlobalError () `[static]`

Retrieve the current global error.

#### 5.16.2.2 void ErrorBase::SetError (Error::Code *code*, const char ∗ *filename*, UINT32 *lineNumber*) const `[virtual]`

Set the current error information associated with this sensor.

**Parameters:**

***code*** The error code

***filename*** Filename of the error source

***lineNumber*** Line number of the error source

#### 5.16.2.3 bool ErrorBase::StatusIsFatal () const `[virtual]`

Check if the current error code represents a fatal error.

**Returns:**

true if the current error is fatal.

The documentation for this class was generated from the following files:

- ErrorBase.h
- ErrorBase.cpp

## 5.17 GearTooth Class Reference

`#include <GearTooth.h>`

Inherits Counter.

### Public Member Functions

- GearTooth (UINT32 channel, bool directionSensitive=false)
- GearTooth (UINT32 slot, UINT32 channel, bool directionSensitive=false)
- GearTooth (DigitalSource ∗source, bool directionSensitive=false)
- virtual ∼GearTooth ()
- void EnableDirectionSensing (bool directionSensitive)

### Static Public Attributes

- static const double kGearToothThreshold = 55e-6

  *55 uSec for threshold*

### 5.17.1 Detailed Description

Alias for counter class. Implement the gear tooth sensor supplied by FIRST. Currently there is no reverse sensing on the gear tooth sensor, but in future versions we might implement the necessary timing in the FPGA to sense direction.

### 5.17.2 Constructor & Destructor Documentation

#### 5.17.2.1 GearTooth::GearTooth (UINT32 *channel*, bool *directionSensitive* = `false`)

Construct a GearTooth sensor given a channel.

The default module is assumed.

**Parameters:**

***channel*** The GPIO channel on the digital module that the sensor is connected to.

***directionSensitive*** Enable the pulse length decoding in hardware to specify count direction.

#### 5.17.2.2 GearTooth::GearTooth (UINT32 *slot*, UINT32 *channel*, bool *directionSensitive* = `false`)

Construct a GearTooth sensor given a channel and module.

**Parameters:**

***slot*** The slot in the chassis that the digital module is plugged in to.

***channel*** The GPIO channel on the digital module that the sensor is connected to.

***directionSensitive*** Enable the pulse length decoding in hardware to specify count direction.

#### 5.17.2.3 GearTooth::GearTooth (DigitalSource * *source*, bool *directionSensitive* = `false`)

Construct a GearTooth sensor given a digital input. This should be used when sharing digial inputs.

**Parameters:**

***source*** An object that fully descibes the input that the sensor is connected to.

***directionSensitive*** Enable the pulse length decoding in hardware to specify count direction.

#### 5.17.2.4 GearTooth::~GearTooth () `[virtual]`

Free the resources associated with a gear tooth sensor.

### 5.17.3 Member Function Documentation

#### 5.17.3.1 void GearTooth::EnableDirectionSensing (bool *directionSensitive*)

Common code called by the constructors.

The documentation for this class was generated from the following files:

- GearTooth.h
- GearTooth.cpp

## 5.18 GenericHID Class Reference

`#include <GenericHID.h>`

Inherited by Joystick.

### 5.18.1 Detailed Description

GenericHID Interface

The documentation for this class was generated from the following file:

- GenericHID.h

# 5.19 Gyro Class Reference

`#include <Gyro.h>`

Inherits SensorBase.

## Public Member Functions

- Gyro (UINT32 slot, UINT32 channel)
- Gyro (UINT32 channel)
- Gyro (AnalogChannel ∗channel)
- virtual ∼Gyro ()
- float GetAngle ()
- void SetSensitivity (float voltsPerDegreePerSecond)
- void Reset ()

## 5.19.1 Detailed Description

Use a rate gyro to return the robots heading relative to a starting position. The Gyro class tracks the robots heading based on the starting position. As the robot rotates the new heading is computed by integrating the rate of rotation returned by the sensor. When the class is instantiated, it does a short calibration routine where it samples the gyro while at rest to determine the default offset. This is subtracted from each sample to determine the heading.

## 5.19.2 Constructor & Destructor Documentation

### 5.19.2.1 Gyro::Gyro (UINT32 *slot*, UINT32 *channel*)

Gyro constructor given a slot and a channel.

**Parameters:**

***slot*** The cRIO slot for the analog module the gyro is connected to.

***channel*** The analog channel the gyro is connected to.

### 5.19.2.2 Gyro::Gyro (UINT32 *channel*) `[explicit]`

Gyro constructor with only a channel.

Use the default analog module slot.

**Parameters:**

***channel*** The analog channel the gyro is connected to.

#### 5.19.2.3 Gyro::Gyro (AnalogChannel * *channel*) [explicit]

Gyro constructor with a precreated analog channel object. Use this constructor when the analog channel needs to be shared. There is no reference counting when an AnalogChannel is passed to the gyro.

**Parameters:**

***channel*** The AnalogChannel object that the gyro is connected to.

#### 5.19.2.4 Gyro::~Gyro () [virtual]

Delete (free) the accumulator and the analog components used for the gyro.

### 5.19.3 Member Function Documentation

#### 5.19.3.1 float Gyro::GetAngle (void)

Return the actual angle in degrees that the robot is currently facing.

The angle is based on the current accumulator value corrected by the oversampling rate, the gyro type and the A/D calibration values. The angle is continuous, that is can go beyond 360 degrees. This make algorithms that wouldn't want to see a discontinuity in the gyro output as it sweeps past 0 on the second time around.

**Returns:**

the current heading of the robot in degrees. This heading is based on integration of the returned rate from the gyro.

#### 5.19.3.2 void Gyro::Reset ()

Reset the gyro. Resets the gyro to a heading of zero. This can be used if there is significant drift in the gyro and it needs to be recalibrated after it has been running.

#### 5.19.3.3 void Gyro::SetSensitivity (float *voltsPerDegreePerSecond*)

Set the gyro type based on the sensitivity. This takes the number of volts/degree/second sensitivity of the gyro and uses it in subsequent calculations to allow the code to work with multiple gyros.

**Parameters:**

***voltsPerDegreePerSecond*** The type of gyro specified as the voltage that represents one degree/second.

The documentation for this class was generated from the following files:

- Gyro.h
- Gyro.cpp

## 5.20 HiTechnicCompass Class Reference

`#include <HiTechnicCompass.h>`

Inherits SensorBase.

### Public Member Functions

- HiTechnicCompass (UINT32 slot)
- virtual ~HiTechnicCompass ()
- float GetAngle ()

### 5.20.1 Detailed Description

HiTechnic NXT Compass.

This class alows access to a HiTechnic NXT Compass on an I2C bus. These sensors to not allow changing addresses so you cannot have more than one on a single bus.

Details on the sensor can be found here: `http://www.hitechnic.com/index.html?lang=en-us&target=d17.`

**Todo**

Implement a calibration method for the sensor.

### 5.20.2 Constructor & Destructor Documentation

#### 5.20.2.1 HiTechnicCompass::HiTechnicCompass (UINT32 *slot*) `[explicit]`

Constructor.

**Parameters:**

*slot* The slot of the digital module that the sensor is plugged into.

#### 5.20.2.2 HiTechnicCompass::~HiTechnicCompass () `[virtual]`

Destructor.

### 5.20.3 Member Function Documentation

#### 5.20.3.1 float HiTechnicCompass::GetAngle ()

Get the compass angle in degrees.

The resolution of this reading is 1 degree.

**Returns:**

Angle of the compass in degrees.

The documentation for this class was generated from the following files:

- HiTechnicCompass.h
- HiTechnicCompass.cpp

## 5.21 I2C Class Reference

```
#include <I2C.h>
```

Inherits SensorBase.

### Public Member Functions

- virtual ~I2C ()
- void Write (UINT8 registerAddress, UINT8 data)
- void Read (UINT8 registerAddress, UINT8 count, UINT8 ∗data)
- void Broadcast (UINT8 registerAddress, UINT8 data)
- bool VerifySensor (UINT8 registerAddress, UINT8 count, const UINT8 ∗expected)

### 5.21.1 Detailed Description

I2C bus interface class.

This class is intended to be used by sensor (and other I2C device) drivers. It probably should not be used directly.

It is constructed by calling DigitalModule::GetI2C() on a DigitalModule object.

### 5.21.2 Constructor & Destructor Documentation

#### 5.21.2.1 I2C::~I2C () `[virtual]`

Destructor.

### 5.21.3 Member Function Documentation

#### 5.21.3.1 void I2C::Broadcast (UINT8 *registerAddress*, UINT8 *data*)

Send a broadcast write to all devices on the I2C bus.

This is not currently implemented!

**Parameters:**

***registerAddress*** The register to write on all devices on the bus.

***data*** The value to write to the devices.

#### 5.21.3.2 void I2C::Read (UINT8 *registerAddress*, UINT8 *count*, UINT8 ∗ *buffer*)

Execute a read transaction with the device.

Read 1, 2, 3, or 4 bytes from a device. Most I2C devices will auto-increment the register pointer internally allowing you to read up to 4 consecutive registers on a device in a single transaction.

**Parameters:**

***registerAddress*** The register to read first in the transaction.

***count*** The number of bytes to read in the transaction.

***buffer*** A pointer to the array of bytes to store the data read from the device.

#### 5.21.3.3 bool I2C::VerifySensor (UINT8 *registerAddress*, UINT8 *count*, const UINT8 * *expected*)

Verify that a device's registers contain expected values.

Most devices will have a set of registers that contain a known value that can be used to identify them. This allows an I2C device driver to easily verify that the device contains the expected value.

**Precondition:**

The device must support and be configured to use register auto-increment.

**Parameters:**

***registerAddress*** The base register to start reading from the device.

***count*** The size of the field to be verified.

***expected*** A buffer containing the values expected from the device.

#### 5.21.3.4 void I2C::Write (UINT8 *registerAddress*, UINT8 *data*)

Execute a write transaction with the device.

Write a byte to a register on a device and wait until the transaction is complete.

**Parameters:**

***registerAddress*** The address of the register on the device to be written.

***data*** The byte to write to the register on the device.

The documentation for this class was generated from the following files:

- I2C.h
- I2C.cpp

# 5.22 IterativeRobot Class Reference

```
#include <IterativeRobot.h>
```

Inherits RobotBase.

## Public Member Functions

- virtual void StartCompetition ()
- virtual void RobotInit ()
- virtual void DisabledInit ()
- virtual void AutonomousInit ()
- virtual void TeleopInit ()
- virtual void DisabledPeriodic ()
- virtual void AutonomousPeriodic ()
- virtual void TeleopPeriodic ()
- virtual void DisabledContinuous ()
- virtual void AutonomousContinuous ()
- virtual void TeleopContinuous ()
- void SetPeriod (double period)
- double GetLoopsPerSec ()

## Protected Member Functions

- virtual ~IterativeRobot ()
- IterativeRobot ()

### 5.22.1 Detailed Description

IterativeRobot implements a specific type of Robot Program framework, extending the RobotBase class.

The IterativeRobot class is intended to be subclassed by a user creating a robot program.

This class is intended to implement the "old style" default code, by providing the following functions which are called by the main loop, StartCompetition(), at the appropriate times:

RobotInit() – provide for initialization at robot power-on

Init() functions – each of the following functions is called once when the appropriate mode is entered:

- DisabledInit() – called only when first disabled
- AutonomousInit() – called each and every time autonomous is entered from another mode
- TeleopInit() – called each and every time teleop is entered from another mode

Periodic() functions – each of these functions is called iteratively at the appropriate periodic rate (aka the "slow loop"). The default period of the iterative robot is 0.005 seconds, giving a periodic frequency of 200Hz (200 times per second).

- DisabledPeriodic()
- AutonomousPeriodic()

- TeleopPeriodic()

Continuous() functions – each of these functions is called repeatedly as fast as possible:

- DisabledContinuous()
- AutonomousContinuous()
- TeleopContinuous()

## 5.22.2 Constructor & Destructor Documentation

### 5.22.2.1 IterativeRobot::~IterativeRobot () `[protected, virtual]`

Free the resources for a RobotIterativeBase class.

### 5.22.2.2 IterativeRobot::IterativeRobot () `[protected]`

Constructor for RobotIterativeBase

The constructor initializes the instance variables for the robot to indicate the status of initialization for disabled, autonomous, and teleop code.

## 5.22.3 Member Function Documentation

### 5.22.3.1 void IterativeRobot::AutonomousContinuous () `[virtual]`

Continuous code for autonomous mode should go here.

Users should override this method for code which will be called repeatedly as frequently as possible while the robot is in autonomous mode.

### 5.22.3.2 void IterativeRobot::AutonomousInit () `[virtual]`

Initialization code for autonomous mode should go here.

Users should override this method for initialization code which will be called each time the robot enters autonomous mode.

### 5.22.3.3 void IterativeRobot::AutonomousPeriodic () `[virtual]`

Periodic code for autonomous mode should go here.

Users should override this method for code which will be called periodically at a regular rate while the robot is in autonomous mode.

### 5.22.3.4 void IterativeRobot::DisabledContinuous () `[virtual]`

Continuous code for disabled mode should go here.

Users should override this method for code which will be called repeatedly as frequently as possible while the robot is in disabled mode.

#### 5.22.3.5 void IterativeRobot::DisabledInit () [virtual]

Initialization code for disabled mode should go here.

Users should override this method for initialization code which will be called each time the robot enters disabled mode.

#### 5.22.3.6 void IterativeRobot::DisabledPeriodic () [virtual]

Periodic code for disabled mode should go here.

Users should override this method for code which will be called periodically at a regular rate while the robot is in disabled mode.

#### 5.22.3.7 double IterativeRobot::GetLoopsPerSec ()

Get the number of loops per second for the IterativeRobot

Get the number of loops per second for the IterativeRobot. The default period of 0.005 seconds results in 200 loops per second. (200Hz iteration loop).

#### 5.22.3.8 void IterativeRobot::RobotInit () [virtual]

Robot-wide initialization code should go here.

Users should override this method for default Robot-wide initialization which will be called when the robot is first powered on. It will be called exactly 1 time.

#### 5.22.3.9 void IterativeRobot::SetPeriod (double *period*)

Set the period for the periodic functions.

The period is set in seconds for the length of time between calls to the periodic functions. Default period is 0.005 seconds (200Hz iteration loop).

#### 5.22.3.10 void IterativeRobot::StartCompetition () [virtual]

default period for periodic functions

Provide an alternate "main loop" via StartCompetition().

This specific StartCompetition() implements "main loop" behavior like that of the FRC control system in 2008 and earlier, with a primary (slow) loop that is called periodically, and a "fast loop" (a.k.a. "spin loop") that is called as fast as possible with no delay between calls.

Implements RobotBase.

#### 5.22.3.11 void IterativeRobot::TeleopContinuous () [virtual]

Continuous code for teleop mode should go here.

Users should override this method for code which will be called repeatedly as frequently as possible while the robot is in teleop mode.

#### 5.22.3.12 void IterativeRobot::TeleopInit () [virtual]

Initialization code for teleop mode should go here.

Users should override this method for initialization code which will be called each time the robot enters teleop mode.

#### 5.22.3.13 void IterativeRobot::TeleopPeriodic () [virtual]

Periodic code for teleop mode should go here.

Users should override this method for code which will be called periodically at a regular rate while the robot is in teleop mode.

The documentation for this class was generated from the following files:

- IterativeRobot.h
- IterativeRobot.cpp

## 5.23 Jaguar Class Reference

`#include <Jaguar.h>`

Inherits PWM, and SpeedController.

### Public Member Functions

- Jaguar (UINT32 channel)
- Jaguar (UINT32 slot, UINT32 channel)
- float Get ()
- void Set (float value)

### 5.23.1 Detailed Description

Luminary Micro Jaguar Speed Control

### 5.23.2 Constructor & Destructor Documentation

#### 5.23.2.1 Jaguar::Jaguar (UINT32 *channel*) `[explicit]`

Constructor that assumes the default digital module.

**Parameters:**

***channel*** The PWM channel on the digital module that the Jaguar is attached to.

#### 5.23.2.2 Jaguar::Jaguar (UINT32 *slot*, UINT32 *channel*)

Constructor that specifies the digital module.

**Parameters:**

***slot*** The slot in the chassis that the digital module is plugged into.

***channel*** The PWM channel on the digital module that the Jaguar is attached to.

### 5.23.3 Member Function Documentation

#### 5.23.3.1 float Jaguar::Get () `[virtual]`

Get the recently set value of the PWM.

**Returns:**

The most recently set value for the PWM between -1.0 and 1.0.

Implements SpeedController.

#### 5.23.3.2 void Jaguar::Set (float *speed*) [virtual]

Set the PWM value.

The PWM value is set using a range of -1.0 to 1.0, appropriately scaling the value for the FPGA.

**Parameters:**

***speed*** The speed value between -1.0 and 1.0 to set.

Implements SpeedController.

The documentation for this class was generated from the following files:

- Jaguar.h
- Jaguar.cpp

## 5.24 Joystick Class Reference

`#include <Joystick.h>`

Inherits GenericHID.

### Public Member Functions

- Joystick (UINT32 port)
- Joystick (UINT32 port, UINT32 numAxisTypes, UINT32 numButtonTypes)
- UINT32 GetAxisChannel (AxisType axis)
- void SetAxisChannel (AxisType axis, UINT32 channel)
- virtual float GetX (JoystickHand hand=kRightHand)
- virtual float GetY (JoystickHand hand=kRightHand)
- virtual float GetZ ()
- virtual float GetTwist ()
- virtual float GetThrottle ()
- virtual float GetAxis (AxisType axis)
- float GetRawAxis (UINT32 axis)
- virtual bool GetTrigger (JoystickHand hand=kRightHand)
- virtual bool GetTop (JoystickHand hand=kRightHand)
- virtual bool GetBumper (JoystickHand hand=kRightHand)
- virtual bool GetButton (ButtonType button)
- bool GetRawButton (UINT32 button)

### 5.24.1 Detailed Description

Handle input from standard Joysticks connected to the Driver Station. This class handles standard input that comes from the Driver Station. Each time a value is requested the most recent value is returned. There is a single class instance for each joystick and the mapping of ports to hardware buttons depends on the code in the driver station.

### 5.24.2 Constructor & Destructor Documentation

#### 5.24.2.1 Joystick::Joystick (UINT32 *port*) `[explicit]`

Construct an instance of a joystick. The joystick index is the usb port on the drivers station.

**Parameters:**

***port*** The port on the driver station that the joystick is plugged into.

#### 5.24.2.2 Joystick::Joystick (UINT32 *port*, UINT32 *numAxisTypes*, UINT32 *numButtonTypes*)

Version of the constructor to be called by sub-classes.

This constructor allows the subclass to configure the number of constants for axes and buttons.

**Parameters:**

***port*** The port on the driver station that the joystick is plugged into.

***numAxisTypes*** The number of axis types in the enum.

***numButtonTypes*** The number of button types in the enum.

## 5.24.3 Member Function Documentation

### 5.24.3.1 float Joystick::GetAxis (AxisType *axis*) `[virtual]`

For the current joystick, return the axis determined by the argument.

This is for cases where the joystick axis is returned programatically, otherwise one of the previous functions would be preferable (for example GetX()).

**Parameters:**

***axis*** The axis to read.

**Returns:**

The value of the axis.

### 5.24.3.2 UINT32 Joystick::GetAxisChannel (AxisType *axis*)

Get the channel currently associated with the specified axis.

**Parameters:**

***axis*** The axis to look up the channel for.

**Returns:**

The channel fr the axis.

### 5.24.3.3 bool Joystick::GetBumper (JoystickHand *hand* = `kRightHand`) `[virtual]`

This is not supported for the Joystick. This method is only here to complete the GenericHID interface.

Implements GenericHID.

### 5.24.3.4 bool Joystick::GetButton (ButtonType *button*) `[virtual]`

Get buttons based on an enumerated type.

The button type will be looked up in the list of buttons and then read.

**Parameters:**

***button*** The type of button to read.

**Returns:**

The state of the button.

#### 5.24.3.5 float Joystick::GetRawAxis (UINT32 *axis*) `[virtual]`

Get the value of the axis.

**Parameters:**

*axis* The axis to read [1-6].

**Returns:**

The value of the axis.

Implements GenericHID.

#### 5.24.3.6 bool Joystick::GetRawButton (UINT32 *button*) `[virtual]`

Get the button value for buttons 1 through 12.

The buttons are returned in a single 16 bit value with one bit representing the state of each button. The appropriate button is returned as a boolean value.

**Parameters:**

*button* The button number to be read.

**Returns:**

The state of the button.

Implements GenericHID.

#### 5.24.3.7 float Joystick::GetThrottle () `[virtual]`

Get the throttle value of the current joystick. This depends on the mapping of the joystick connected to the current port.

Implements GenericHID.

#### 5.24.3.8 bool Joystick::GetTop (JoystickHand *hand* = `kRightHand`) `[virtual]`

Read the state of the top button on the joystick.

Look up which button has been assigned to the top and read its state.

**Parameters:**

*hand* This parameter is ignored for the Joystick class and is only here to complete the GenericHID interface.

**Returns:**

The state of the top button.

Implements GenericHID.

#### 5.24.3.9 bool Joystick::GetTrigger (JoystickHand *hand* = `kRightHand`) `[virtual]`

Read the state of the trigger on the joystick.

Look up which button has been assigned to the trigger and read its state.

**Parameters:**

***hand*** This parameter is ignored for the Joystick class and is only here to complete the GenericHID interface.

**Returns:**

The state of the trigger.

Implements GenericHID.

#### 5.24.3.10 float Joystick::GetTwist () `[virtual]`

Get the twist value of the current joystick. This depends on the mapping of the joystick connected to the current port.

Implements GenericHID.

#### 5.24.3.11 float Joystick::GetX (JoystickHand *hand* = `kRightHand`) `[virtual]`

Get the X value of the joystick. This depends on the mapping of the joystick connected to the current port.

Implements GenericHID.

#### 5.24.3.12 float Joystick::GetY (JoystickHand *hand* = `kRightHand`) `[virtual]`

Get the Y value of the joystick. This depends on the mapping of the joystick connected to the current port.

Implements GenericHID.

#### 5.24.3.13 float Joystick::GetZ () `[virtual]`

Get the Z value of the current joystick. This depends on the mapping of the joystick connected to the current port.

Implements GenericHID.

#### 5.24.3.14 void Joystick::SetAxisChannel (AxisType *axis*, UINT32 *channel*)

Set the channel associated with a specified axis.

**Parameters:**

***axis*** The axis to set the channel for.

***channel*** The channel to set the axis to.

The documentation for this class was generated from the following files:

- Joystick.h
- Joystick.cpp

## 5.25 ParticleAnalysisReport_struct Struct Reference

```
#include <VisionAPI.h>
```

### 5.25.1 Detailed Description

frcParticleAnalysis returns this structure

The documentation for this struct was generated from the following file:

- VisionAPI.h

## 5.26 PCVideoServer Class Reference

`#include <PCVideoServer.h>`

Inherits ErrorBase.

### Public Member Functions

- PCVideoServer (void)

    *Constructor.*

- ~PCVideoServer ()

    *Destructor. Stop serving images and destroy this class.*

- void Start ()

    *Start sending images to the PC.*

- void Stop ()

    *Stop sending images to the PC.*

### 5.26.1 Detailed Description

Class the serves images to the PC.

The documentation for this class was generated from the following files:

- PCVideoServer.h
- PCVideoServer.cpp

## 5.27 PIDController Class Reference

```
#include <PIDController.h>
```

### Public Member Functions

- PIDController (float p, float i, float d, float period=0.05)
- ∼PIDController ()
- float Get ()
- void SetContinuous (bool continuous=true)
- void SetInput (PIDSource ∗pidInput)
- void SetInput (PIDSource ∗pidInput, float minimumInput, float maximumInput)
- void SetOutput (PIDOutput ∗pidOutput)
- void SetOutput (PIDOutput ∗pidOutput, float mimimumOutput, float maximumOutput)
- void SetSetpoint (float setpoint)
- float GetSetpoint ()
- float GetError ()
- void Enable ()
- void Disable ()
- void Reset ()

### 5.27.1 Detailed Description

Class implements a PID Control Loop.

Creates a separate thread which reads the given PIDSource and takes care of the integral calculations, as well as writing the given PIDOutput

### 5.27.2 Constructor & Destructor Documentation

#### 5.27.2.1 PIDController::PIDController (float *Kp*, float *Ki*, float *Kd*, float *period* = `0.05`)

Allocate a PID object with the given constants for P, I, D

**Parameters:**

***Kp*** the proportional coefficient

***Ki*** the integral coefficient

***Kd*** the derivative coefficient

***period*** the loop time for doing calculations. This particularly effects calculations of the integral and differental terms. The default is 50ms.

#### 5.27.2.2 PIDController::∼PIDController ()

Free the PID object

## 5.27.3 Member Function Documentation

### 5.27.3.1 void PIDController::Disable ()

Stop running the PIDController, this sets the output to zero before stopping.

### 5.27.3.2 void PIDController::Enable ()

Begin running the PIDController

### 5.27.3.3 float PIDController::Get ()

Return the current PID result This is always centered on zero and constrained the the max and min outs

**Returns:**

the latest calculated output

### 5.27.3.4 float PIDController::GetError ()

Retruns the current difference of the input from the setpoint

**Returns:**

the current error

### 5.27.3.5 float PIDController::GetSetpoint ()

Returns the current setpoint of the PIDController

**Returns:**

the current setpoint

### 5.27.3.6 void PIDController::Reset ()

Reset the previous error,, the integral term, and disable the controller.

### 5.27.3.7 void PIDController::SetContinuous (bool *continuous* = `true`)

Set the PID controller to consider the input to be continuous, Rather then using the max and min in as constraints, it considers them to be the same point and automatically calculates the shortest route to the setpoint.

**Parameters:**

***continuous*** Set to true turns on continuous, false turns off continuous

#### 5.27.3.8 void PIDController::SetInput (PIDSource * *pidInput*, float *minimumInput*, float *maximumInput*)

Sets the PIDSource object from which the PIDController gets its feedback, as well as the maximum and minimum values expected from the input.

**Parameters:**

***pidInput*** the source of feedback for the PIDController

***minimumInput*** the minimum value expected from the input

***maximumInput*** the maximum value expected from the output

#### 5.27.3.9 void PIDController::SetInput (PIDSource * *pidInput*)

Sets the PIDSource object from which the PIDController gets its feedback.

**Parameters:**

***pidInput*** the source of feedback for the PIDController

#### 5.27.3.10 void PIDController::SetOutput (PIDOutput * *pidOutput*, float *minimumOutput*, float *maximumOutput*)

Sets the PIDOutput object which the PIDController writes to, as well as the minimum and maximum values to write.

**Parameters:**

***pidOutput*** the source of feedback for the PIDController

***minimumOutput*** the minimum value to write to the output

***maximumOutput*** the maximum value to write to the output

#### 5.27.3.11 void PIDController::SetOutput (PIDOutput * *pidOutput*)

Sets the PIDOutput object which the PIDController writes to.

**Parameters:**

***pidOutput*** the source of feedback for the PIDController

#### 5.27.3.12 void PIDController::SetSetpoint (float *setpoint*)

Set the setpoint for the PIDController

**Parameters:**

***setpoint*** the desired setpoint

The documentation for this class was generated from the following files:

- PIDController.h
- PIDController.cpp

## 5.28 PIDOutput Class Reference

```
#include <PIDOutput.h>
```

### 5.28.1 Detailed Description

PIDOutput interface is a generic output for the PID class. PWMs use this class. Users implement this interface to allow for a PIDController to read directly from the inputs

The documentation for this class was generated from the following file:

- PIDOutput.h

## 5.29 PIDSource Class Reference

`#include <PIDSource.h>`

### 5.29.1 Detailed Description

PIDSource interface is a generic sensor source for the PID class. All sensors that can be used with the PID class will implement the PIDSource that returns a standard value that will be used in the PID code.

The documentation for this class was generated from the following file:

- PIDSource.h

## 5.30 PWM Class Reference

`#include <PWM.h>`

Inherits SensorBase.

Inherited by Jaguar, Servo, and Victor.

### Public Member Functions

- PWM (UINT32 channel)
- PWM (UINT32 slot, UINT32 channel)
- virtual ~PWM ()
- void SetRaw (UINT8 value)
- UINT8 GetRaw ()
- void SetPeriodMultiplier (PeriodMultiplier mult)
- void EnableDeadbandElimination (bool eliminateDeadband)
- void SetBounds (INT32 max, INT32 deadbandMax, INT32 center, INT32 deadbandMin, INT32 min)

### Protected Member Functions

- void SetPosition (float pos)
- float GetPosition ()
- void SetSpeed (float speed)
- float GetSpeed ()

### Static Protected Attributes

- static const UINT32 kDefaultPwmPeriod = 774
- static const UINT32 kDefaultMinPwmHigh = 102

### 5.30.1 Detailed Description

Class implements the PWM generation in the FPGA.

The values supplied as arguments for PWM outputs range from -1.0 to 1.0. They are mapped to the hardware dependent values, in this case 0-255 for the FPGA. Changes are immediately sent to the FPGA, and the update occurs at the next FPGA cycle. There is no delay.

As of revision 0.1.10 of the FPGA, the FPGA interprets the 0-255 values as follows:

- 255 = full "forward"
- 254 to 129 = linear scaling from "full forward" to "center"
- 128 = center value
- 127 to 2 = linear scaling from "center" to "full reverse"
- 1 = full "reverse"
- 0 = disabled (i.e. PWM output is held low)

## 5.30.2 Constructor & Destructor Documentation

### 5.30.2.1 PWM::PWM (UINT32 *channel*) `[explicit]`

Allocate a PWM in the default module given a channel.

Using a default module allocate a PWM given the channel number. The default module is the first slot numerically in the cRIO chassis.

**Parameters:**

***channel*** The PWM channel on the digital module.

### 5.30.2.2 PWM::PWM (UINT32 *slot*, UINT32 *channel*)

Allocate a PWM given a module and channel. Allocate a PWM using a module and channel number.

**Parameters:**

***slot*** The slot the digital module is plugged into.

***channel*** The PWM channel on the digital module.

### 5.30.2.3 PWM::~PWM () `[virtual]`

Free the PWM channel.

Free the resource associated with the PWM channel and set the value to 0.

## 5.30.3 Member Function Documentation

### 5.30.3.1 void PWM::EnableDeadbandElimination (bool *eliminateDeadband*)

Optionally eliminate the deadband from a speed controller.

**Parameters:**

***eliminateDeadband*** If true, set the motor curve on the Jaguar to eliminate the deadband in the middle of the range. Otherwise, keep the full range without modifying any values.

### 5.30.3.2 float PWM::GetPosition () `[protected]`

Get the PWM value in terms of a position.

This is intended to be used by servos.

**Precondition:**

SetMaxPositivePwm() called.
SetMinNegativePwm() called.

**Returns:**

The position the servo is set to between 0.0 and 1.0.

#### 5.30.3.3 UINT8 PWM::GetRaw ()

Get the PWM value directly from the hardware.

Read a raw value from a PWM channel.

**Returns:**

Raw PWM control value. Range: 0 - 255.

#### 5.30.3.4 float PWM::GetSpeed () [protected]

Get the PWM value in terms of speed.

This is intended to be used by speed controllers.

**Precondition:**

SetMaxPositivePwm() called.
SetMinPositivePwm() called.
SetMaxNegativePwm() called.
SetMinNegativePwm() called.

**Returns:**

The most recently set speed between -1.0 and 1.0.

#### 5.30.3.5 void PWM::SetBounds (INT32 *max*, INT32 *deadbandMax*, INT32 *center*, INT32 *deadbandMin*, INT32 *min*)

Set the bounds on the PWM values. This sets the bounds on the PWM values for a particular each type of controller. The values determine the upper and lower speeds as well as the deadband bracket.

**Parameters:**

***max*** The Minimum pwm value

***deadbandMax*** The high end of the deadband range

***center*** The center speed (off)

***deadbandMin*** The low end of the deadband range

***min*** The minimum pwm value

#### 5.30.3.6 void PWM::SetPeriodMultiplier (PeriodMultiplier *mult*)

Slow down the PWM signal for old devices.

**Parameters:**

***mult*** The period multiplier to apply to this channel

#### 5.30.3.7 void PWM::SetPosition (float *pos*) [protected]

Set the PWM value based on a position.

This is intended to be used by servos.

**Precondition:**

SetMaxPositivePwm() called.
SetMinNegativePwm() called.

**Parameters:**

***pos*** The position to set the servo between 0.0 and 1.0.

#### 5.30.3.8 void PWM::SetRaw (UINT8 *value*)

Set the PWM value directly to the hardware.

Write a raw value to a PWM channel.

**Parameters:**

***value*** Raw PWM value. Range 0 - 255.

#### 5.30.3.9 void PWM::SetSpeed (float *speed*) [protected]

Set the PWM value based on a speed.

This is intended to be used by speed controllers.

**Precondition:**

SetMaxPositivePwm() called.
SetMinPositivePwm() called.
SetCenterPwm() called.
SetMaxNegativePwm() called.
SetMinNegativePwm() called.

**Parameters:**

***speed*** The speed to set the speed controller between -1.0 and 1.0.

### 5.30.4 Member Data Documentation

#### 5.30.4.1 const UINT32 PWM::kDefaultMinPwmHigh = 102 [static, protected]

kDefaultMinPwmHigh is "ticks" where each tick is 6.525us

- There are 128 pwm values less than the center, so...
- The minimum output pulse length is 1.5ms - 128 $*$ 6.525us = 0.665ms
- 0.665ms / 6.525us per tick = 102

#### 5.30.4.2 const UINT32 PWM::kDefaultPwmPeriod = 774 `[static, protected]`

kDefaultPwmPeriod is "ticks" where each tick is 6.525us

- 20ms periods (50 Hz) are the "safest" setting in that this works for all devices
- 20ms periods seem to be desirable for Vex Motors
- 20ms periods are the specified period for HS-322HD servos, but work reliably down to 10.0 ms; starting at about 8.5ms, the servo sometimes hums and get hot; by 5.0ms the hum is nearly continuous
- 10ms periods work well for Victor 884
- 5ms periods allows higher update rates for Luminary Micro Jaguar speed controllers. Due to the shipping firmware on the Jaguar, we can't run the update period less than 5.05 ms.

kDefaultPwmPeriod is the 1x period (5.05 ms). In hardware, the period scaling is implemented as an output squelch to get longer periods for old devices.

Set to 5.05 ms period / 6.525us clock = 774

The documentation for this class was generated from the following files:

- PWM.h
- PWM.cpp

## 5.31 Relay Class Reference

`#include <Relay.h>`

Inherits SensorBase.

### Public Member Functions

- Relay (UINT32 channel, Direction direction=kBothDirections)
- Relay (UINT32 slot, UINT32 channel, Direction direction=kBothDirections)
- virtual ~Relay ()
- void Set (Value value)

### 5.31.1 Detailed Description

Class for Spike style relay outputs. Relays are intended to be connected to spikes or similar relays. The relay channels controls a pair of pins that are either both off, one on, the other on, or both on. This translates into two spike outputs at 0v, one at 12v and one at 0v, one at 0v and the other at 12v, or two spike outputs at 12V. This allows off, full forward, or full reverse control of motors without variable speed. It also allows the two channels (forward and reverse) to be used independently for something that does not care about voltage polatiry (like a solenoid).

### 5.31.2 Constructor & Destructor Documentation

#### 5.31.2.1 Relay::Relay (UINT32 *channel*, Relay::Direction *direction* = `kBothDirections`)

Relay constructor given a channel only where the default digital module is used.

**Parameters:**

***channel*** The channel number within the default module for this relay.

***direction*** The direction that the Relay object will control.

#### 5.31.2.2 Relay::Relay (UINT32 *slot*, UINT32 *channel*, Relay::Direction *direction* = `kBothDirections`)

Relay constructor given the module and the channel.

**Parameters:**

***slot*** The module slot number this relay is connected to.

***channel*** The channel number within the module for this relay.

***direction*** The direction that the Relay object will control.

#### 5.31.2.3 Relay::~Relay () `[virtual]`

Free the resource associated with a relay. The relay channels are set to free and the relay output is turned off.

## 5.31.3 Member Function Documentation

### 5.31.3.1 void Relay::Set (Relay::Value *value*)

Set the relay state.

Valid values depend on which directions of the relay are controlled by the object.

When set to kBothDirections, the relay can only be one of the three reasonable values, 0v-0v, 0v-12v, or 12v-0v.

When set to kForwardOnly or kReverseOnly, you can specify the constant for the direction or you can simply specify kOff and kOn. Using only kOff and kOn is recommended.

**Parameters:**

***value*** The state to set the relay.

The documentation for this class was generated from the following files:

- Relay.h
- Relay.cpp

## 5.32 Resource Class Reference

`#include <Resource.h>`

### Public Member Functions

- virtual ~Resource ()
- UINT32 Allocate ()
- UINT32 Allocate (UINT32 index)
- void Free (UINT32 index)

### Protected Member Functions

- Resource (UINT32 size)

### 5.32.1 Detailed Description

Track resources in the program. The Resource class is a convienent way of keeping track of allocated arbitrary resources in the program. Resources are just indicies that have an lower and upper bound that are tracked by this class. In the library they are used for tracking allocation of hardware channels but this is purely arbitrary. The resource class does not do any actual allocation, but simply tracks if a given index is currently in use.

WARNING: this should only be statically allocated. When the program loads into memory all the static constructors are called. At that time a linked list of all the "Resources" is created. Then when the program actually starts - in the Robot constructor, all resources are initialized. This ensures that the program is restartable in memory without having to unload/reload.

### 5.32.2 Constructor & Destructor Documentation

#### 5.32.2.1 Resource::~Resource () `[virtual]`

Delete the allocated array or resources. This happens when the module is unloaded (provided it was statically allocated).

#### 5.32.2.2 Resource::Resource (UINT32 *elements*) `[explicit, protected]`

Allocate storage for a new instance of Resource. Allocate a bool array of values that will get initialized to indicate that no resources have been allocated yet. The indicies of the resources are 0..size-1.

### 5.32.3 Member Function Documentation

#### 5.32.3.1 UINT32 Resource::Allocate (UINT32 *index*)

Allocate a specific resource value. The user requests a specific resource value, i.e. channel number and it is verified unallocated, then returned.

#### 5.32.3.2 UINT32 Resource::Allocate ()

Allocate a resource. When a resource is requested, mark it allocated. In this case, a free resource value within the range is located and returned after it is marked allocated.

#### 5.32.3.3 void Resource::Free (UINT32 *index*)

Free an allocated resource. After a resource is no longer needed, for example a destructor is called for a channel assignment class, Free will release the resource value so it can be reused somewhere else in the program.

The documentation for this class was generated from the following files:

- Resource.h
- Resource.cpp

# 5.33 RobotBase Class Reference

`#include <RobotBase.h>`

Inherited by IterativeRobot, and SimpleRobot.

## Public Member Functions

- bool IsDisabled ()
- bool IsAutonomous ()
- bool IsOperatorControl ()
- bool IsSystemActive ()
- bool IsNewDataAvailable ()
- Watchdog & GetWatchdog ()

## Static Public Member Functions

- static void startRobotTask (FUNCPTR factory)
- static void robotTask (FUNCPTR factory, Task *task)

## Protected Member Functions

- virtual ~RobotBase ()
- RobotBase ()

## Friends

- class RobotDeleter

### 5.33.1 Detailed Description

Implement a Robot Program framework. The RobotBase class is intended to be subclassed by a user creating a robot program. Overridden Autonomous() and OperatorControl() methods are called at the appropriate time as the match proceeds. In the current implementation, the Autonomous code will run to completion before the OperatorControl code could start. In the future the Autonomous code might be spawned as a task, then killed at the end of the Autonomous period.

### 5.33.2 Constructor & Destructor Documentation

#### 5.33.2.1 RobotBase::~RobotBase () `[protected, virtual]`

Free the resources for a RobotBase class. This includes deleting all classes that might have been allocated as Singletons to they would never be deleted except here.

#### 5.33.2.2 RobotBase::RobotBase () `[protected]`

Constructor for a generic robot program. User code should be placed in the constuctor that runs before the Autonomous or Operator Control period starts. The constructor will run to completion before Autonomous is entered.

This must be used to ensure that the communications code starts. In the future it would be nice to put this code into it's own task that loads on boot so ensure that it runs.

### 5.33.3 Member Function Documentation

#### 5.33.3.1 Watchdog & RobotBase::GetWatchdog ()

Return the instance of the Watchdog timer. Get the watchdog timer so the user program can either disable it or feed it when necessary.

#### 5.33.3.2 bool RobotBase::IsAutonomous ()

Determine if the robot is currently in Autnomous mode.

**Returns:**

True if the robot is currently operating Autonomously as determined by the field controls.

#### 5.33.3.3 bool RobotBase::IsDisabled ()

Determine if the Robot is currently disabled.

**Returns:**

True if the Robot is currently disabled by the field controls.

#### 5.33.3.4 bool RobotBase::IsNewDataAvailable ()

Indicates if new data is available from the driver station.

**Todo**

The current implementation is silly. We already know this explicitly without trying to figure it out.

**Returns:**

Has new data arrived over the network since the last time this function was called?

#### 5.33.3.5 bool RobotBase::IsOperatorControl ()

Determine if the robot is currently in Operator Control mode.

**Returns:**

True if the robot is currently operating in Tele-Op mode as determined by the field controls.

#### 5.33.3.6 bool RobotBase::IsSystemActive ()

Check on the overall status of the system.

**Returns:**

Is the system active (i.e. PWM motor outputs, etc. enabled)?

#### 5.33.3.7 void RobotBase::robotTask (FUNCPTR *factory*, Task * *task*) [static]

Static interface that will start the competition in the new task.

#### 5.33.3.8 void RobotBase::startRobotTask (FUNCPTR *factory*) [static]

Start the robot code. This function starts the robot code running by spawning a task. Currently tasks seemed to be started by LVRT without setting the VX_FP_TASK flag so floating point context is not saved on interrupts. Therefore the program experiences hard to debug and unpredictable results. So the LVRT code starts this function, and it, in turn, starts the actual user program.

The documentation for this class was generated from the following files:

- RobotBase.h
- RobotBase.cpp

# 5.34 RobotDeleter Class Reference

## 5.34.1 Detailed Description

This class exists for the sole purpose of getting its destructor called when the module unloads. Before the module is done unloading, we need to delete the RobotBase derived singleton. This should delete the other remaining singletons that were registered. This should also stop all tasks that are using the Task class.

The documentation for this class was generated from the following file:

- RobotBase.cpp

## 5.35 RobotDrive Class Reference

`#include <RobotDrive.h>`

### Public Member Functions

- RobotDrive (UINT32 leftMotorChannel, UINT32 rightMotorChannel, float sensitivity=0.5)
- RobotDrive (UINT32 frontLeftMotorChannel, UINT32 rearLeftMotorChannel, UINT32 frontRightMotorChannel, UINT32 rearRightMotorChannel, float sensitivity=0.5)
- RobotDrive (SpeedController ∗leftMotor, SpeedController ∗rightMotor, float sensitivity=0.5)
- RobotDrive (SpeedController ∗frontLeftMotor, SpeedController ∗rearLeftMotor, SpeedController ∗frontRightMotor, SpeedController ∗rearRightMotor, float sensitivity=0.5)
- virtual ∼RobotDrive ()
- void Drive (float speed, float curve)
- void TankDrive (GenericHID ∗leftStick, GenericHID ∗rightStick)
- void TankDrive (GenericHID ∗leftStick, UINT32 leftAxis, GenericHID ∗rightStick, UINT32 rightAxis)
- void TankDrive (float leftValue, float rightValue)
- void ArcadeDrive (GenericHID ∗stick, bool squaredInputs=true)
- void ArcadeDrive (GenericHID &stick, bool squaredInputs=true)
- void ArcadeDrive (GenericHID ∗moveStick, UINT32 moveChannel, GenericHID ∗rotateStick, UINT32 rotateChannel, bool squaredInputs=true)
- void ArcadeDrive (GenericHID &moveStick, UINT32 moveChannel, GenericHID &rotateStick, UINT32 rotateChannel, bool squaredInputs=true)
- void ArcadeDrive (float moveValue, float rotateValue, bool squaredInputs=true)
- void HolonomicDrive (float magnitude, float direction, float rotation)
- void SetLeftRightMotorSpeeds (float leftSpeed, float rightSpeed)

### 5.35.1 Detailed Description

Utility class for handling Robot drive based on a definition of the motor configuration. The robot drive class handles basic driving for a robot. Currently, 2 and 4 motor standard drive trains are supported. In the future other drive types like swerve and meccanum might be implemented. Motor channel numbers are passed supplied on creation of the class. Those are used for either the Drive function (intended for hand created drive code, such as autonomous) or with the Tank/Arcade functions intended to be used for Operator Control driving.

### 5.35.2 Constructor & Destructor Documentation

#### 5.35.2.1 RobotDrive::RobotDrive (UINT32 *leftMotorChannel*, UINT32 *rightMotorChannel*, float *sensitivity* = `0.5`)

Constructor for RobotDrive with 2 motors specified with channel numbers. Set up parameters for a two wheel drive system where the left and right motor pwm channels are specified in the call. This call assumes Jaguars for controlling the motors.

**Parameters:**

*leftMotorChannel* The PWM channel number on the default digital module that drives the left motor.

***rightMotorChannel*** The PWM channel number on the default digital module that drives the right motor.

***sensitivity*** Effectively sets the turning sensitivity (or turn radius for a given value).

#### 5.35.2.2 RobotDrive::RobotDrive (UINT32 *frontLeftMotor*, UINT32 *rearLeftMotor*, UINT32 *frontRightMotor*, UINT32 *rearRightMotor*, float *sensitivity* = 0.5)

Constructor for RobotDrive with 4 motors specified with channel numbers. Set up parameters for a four wheel drive system where all four motor pwm channels are specified in the call. This call assumes Jaguars for controlling the motors.

**Parameters:**

***frontLeftMotor*** Front left motor channel number on the default digital module

***rearLeftMotor*** Rear Left motor channel number on the default digital module

***frontRightMotor*** Front right motor channel number on the default digital module

***rearRightMotor*** Rear Right motor channel number on the default digital module

***sensitivity*** Effectively sets the turning sensitivity (or turn radius for a given value)

#### 5.35.2.3 RobotDrive::RobotDrive (SpeedController ∗ *leftMotor*, SpeedController ∗ *rightMotor*, float *sensitivity* = 0.5)

Constructor for RobotDrive with 2 motors specified as SpeedController objects. The SpeedController version of the constructor enables programs to use the RobotDrive classes with subclasses of the SpeedController objects, for example, versions with ramping or reshaping of the curve to suit motor bias or deadband elimination.

**Parameters:**

***leftMotor*** The left SpeedController object used to drive the robot.

***rightMotor*** the right SpeedController object used to drive the robot.

***sensitivity*** Effectively sets the turning sensitivity (or turn radius for a given value)

#### 5.35.2.4 RobotDrive::RobotDrive (SpeedController ∗ *frontLeftMotor*, SpeedController ∗ *rearLeftMotor*, SpeedController ∗ *frontRightMotor*, SpeedController ∗ *rearRightMotor*, float *sensitivity* = 0.5)

Constructor for RobotDrive with 4 motors specified as SpeedController objects. Speed controller input version of RobotDrive (see previous comments).

**Parameters:**

***rearLeftMotor*** The back left SpeedController object used to drive the robot.

***frontLeftMotor*** The front left SpeedController object used to drive the robot

***rearRightMotor*** The back right SpeedController object used to drive the robot.

***frontRightMotor*** The front right SpeedController object used to drive the robot.

***sensitivity*** Effectively sets the turning sensitivity (or turn radius for a given value)

#### 5.35.2.5 RobotDrive::~RobotDrive () `[virtual]`

RobotDrive destructor. Deletes motor objects that were not passed in and created internally only.

### 5.35.3 Member Function Documentation

#### 5.35.3.1 void RobotDrive::ArcadeDrive (float *moveValue*, float *rotateValue*, bool *squaredInputs* = `true`)

Arcade drive implements single stick driving. This function lets you directly provide joystick values from any source.

**Parameters:**

***moveValue*** The value to use for fowards/backwards

***rotateValue*** The value to use for the rotate right/left

***squaredInputs*** If set, increases the sensitivity at low speeds

#### 5.35.3.2 void RobotDrive::ArcadeDrive (GenericHID & *moveStick*, UINT32 *moveAxis*, GenericHID & *rotateStick*, UINT32 *rotateAxis*, bool *squaredInputs* = `true`)

Arcade drive implements single stick driving. Given two joystick instances and two axis, compute the values to send to either two or four motors.

**Parameters:**

***moveStick*** The Joystick object that represents the forward/backward direction

***moveAxis*** The axis on the moveStick object to use for fowards/backwards (typically Y_AXIS)

***rotateStick*** The Joystick object that represents the rotation value

***rotateAxis*** The axis on the rotation object to use for the rotate right/left (typically X_AXIS)

***squaredInputs*** Setting this parameter to true increases the sensitivity at lower speeds

#### 5.35.3.3 void RobotDrive::ArcadeDrive (GenericHID * *moveStick*, UINT32 *moveAxis*, GenericHID * *rotateStick*, UINT32 *rotateAxis*, bool *squaredInputs* = `true`)

Arcade drive implements single stick driving. Given two joystick instances and two axis, compute the values to send to either two or four motors.

**Parameters:**

***moveStick*** The Joystick object that represents the forward/backward direction

***moveAxis*** The axis on the moveStick object to use for fowards/backwards (typically Y_AXIS)

***rotateStick*** The Joystick object that represents the rotation value

***rotateAxis*** The axis on the rotation object to use for the rotate right/left (typically X_AXIS)

***squaredInputs*** Setting this parameter to true increases the sensitivity at lower speeds

#### 5.35.3.4 void RobotDrive::ArcadeDrive (GenericHID & *stick*, bool *squaredInputs* = `true`)

Arcade drive implements single stick driving. Given a single Joystick, the class assumes the Y axis for the move value and the X axis for the rotate value. (Should add more information here regarding the way that arcade drive works.)

**Parameters:**

***stick*** The joystick to use for Arcade single-stick driving. The Y-axis will be selected for forwards/backwards and the X-axis will be selected for rotation rate.

***squaredInputs*** If true, the sensitivity will be increased for small values

#### 5.35.3.5 void RobotDrive::ArcadeDrive (GenericHID $*$ *stick*, bool *squaredInputs* = `true`)

Arcade drive implements single stick driving. Given a single Joystick, the class assumes the Y axis for the move value and the X axis for the rotate value. (Should add more information here regarding the way that arcade drive works.)

**Parameters:**

***stick*** The joystick to use for Arcade single-stick driving. The Y-axis will be selected for forwards/backwards and the X-axis will be selected for rotation rate.

***squaredInputs*** If true, the sensitivity will be increased for small values

#### 5.35.3.6 void RobotDrive::Drive (float *speed*, float *curve*)

Drive the motors at "speed" and "curve".

The speed and curve are -1.0 to +1.0 values where 0.0 represents stopped and not turning. The algorithm for adding in the direction attempts to provide a constant turn radius for differing speeds.

This function sill most likely be used in an autonomous routine.

**Parameters:**

***speed*** The forward component of the speed to send to the motors.

***curve*** The rate of turn, constant for different forward speeds.

#### 5.35.3.7 void RobotDrive::HolonomicDrive (float *magnitude*, float *direction*, float *rotation*)

Holonomic Drive class for Mecanum wheeled robots.

Experimental class for driving with Mecanum wheeled robots. There are 4 wheels on the robot, arranged so that the front and back wheels are toed in 45 degrees.

**Parameters:**

***magnitude*** The speed that the robot should drive in a given direction.

***direction*** The direction the robot should drive. The direction and maginitute are independent of the rotation rate.

***rotation*** The rate of rotation for the robot that is completely independent of the magnitute or direction.

#### 5.35.3.8 void RobotDrive::SetLeftRightMotorSpeeds (float *leftSpeed*, float *rightSpeed*)

Set the speed of the right and left motors. This is used once an appropriate drive setup function is called such as TwoWheelDrive(). The motors are set to "leftSpeed" and "rightSpeed" and includes flipping the direction of one side for opposing motors.

**Parameters:**

***leftSpeed*** The speed to send to the left side of the robot.

***rightSpeed*** The speed to send to the right side of the robot.

#### 5.35.3.9 void RobotDrive::TankDrive (float *leftValue*, float *rightValue*)

Provide tank steering using the stored robot configuration. This function lets you directly provide joystick values from any source.

**Parameters:**

***leftValue*** The value of the left stick.

***rightValue*** The value of the right stick.

#### 5.35.3.10 void RobotDrive::TankDrive (GenericHID ∗ *leftStick*, UINT32 *leftAxis*, GenericHID ∗ *rightStick*, UINT32 *rightAxis*)

Provide tank steering using the stored robot configuration. This function lets you pick the axis to be used on each Joystick object for the left and right sides of the robot.

**Parameters:**

***leftStick*** The Joystick object to use for the left side of the robot.

***leftAxis*** The axis to select on the left side Joystick object.

***rightStick*** The Joystick object to use for the right side of the robot.

***rightAxis*** The axis to select on the right side Joystick object.

#### 5.35.3.11 void RobotDrive::TankDrive (GenericHID ∗ *leftStick*, GenericHID ∗ *rightStick*)

Provide tank steering using the stored robot configuration. Drive the robot using two joystick inputs. The Y-axis will be selected from each Joystick object.

**Parameters:**

***leftStick*** The joystick to control the left side of the robot.

***rightStick*** The joystick to control the right side of the robot.

The documentation for this class was generated from the following files:

- RobotDrive.h
- RobotDrive.cpp

## 5.36 ScopedSocket Class Reference

Implements an object that automatically does a close on a camera socket on destruction.

### 5.36.1 Detailed Description

Implements an object that automatically does a close on a camera socket on destruction.

The documentation for this class was generated from the following file:

- PCVideoServer.cpp

## 5.37 SensorBase Class Reference

`#include <SensorBase.h>`

Inherits ErrorBase.

Inherited by Accelerometer, AnalogChannel, AnalogTrigger, Compressor, Counter, DigitalOutput `[private]`, DriverStation, Encoder, Gyro, HiTechnicCompass, I2C `[private]`, InterruptableSensorBase, Module, PWM, Relay, Solenoid, Ultrasonic, and Watchdog.

### Public Member Functions

- SensorBase ()
- virtual ~SensorBase ()

### Static Public Member Functions

- static void SetDefaultAnalogModule (UINT32 slot)
- static void SetDefaultDigitalModule (UINT32 slot)
- static void SetDefaultSolenoidModule (UINT32 slot)
- static void DeleteSingletons ()
- static bool CheckDigitalModule (UINT32 slot)
- static bool CheckRelayModule (UINT32 slot)
- static bool CheckPWMModule (UINT32 slot)
- static bool CheckSolenoidModule (UINT32 slot)
- static bool CheckAnalogModule (UINT32 slot)
- static bool CheckDigitalChannel (UINT32 channel)
- static bool CheckRelayChannel (UINT32 channel)
- static bool CheckPWMChannel (UINT32 channel)
- static bool CheckAnalogChannel (UINT32 channel)
- static bool CheckSolenoidChannel (UINT32 channel)

### Protected Member Functions

- void AddToSingletonList ()

### 5.37.1 Detailed Description

Base class for all sensors. Stores most recent status information as well as containing utility functions for checking channels and error processing.

### 5.37.2 Constructor & Destructor Documentation

#### 5.37.2.1 SensorBase::SensorBase ()

Creates an instance of the sensor base and gets an FPGA handle

#### 5.37.2.2 SensorBase::∼SensorBase () `[virtual]`

Frees the resources for a SensorBase.

### 5.37.3 Member Function Documentation

#### 5.37.3.1 void SensorBase::AddToSingletonList () `[protected]`

Add sensor to the singleton list. Add this sensor to the list of singletons that need to be deleted when the robot program exits. Each of the sensors on this list are singletons, that is they aren't allocated directly with new, but instead are allocated by the static GetInstance method. As a result, they are never deleted when the program exits. Consequently these sensors may still be holding onto resources and need to have their destructors called at the end of the program.

#### 5.37.3.2 bool SensorBase::CheckAnalogChannel (UINT32 *channel*) `[static]`

Check that the analog channel number is value. Verify that the analog channel number is one of the legal channel numbers. Channel numbers are 1-based.

#### 5.37.3.3 bool SensorBase::CheckAnalogModule (UINT32 *slot*) `[static]`

Check that the analog module number is valid. Module numbers are the slot numbers that they are inserted in.

#### 5.37.3.4 bool SensorBase::CheckDigitalChannel (UINT32 *channel*) `[static]`

Check that the digital channel number is valid. Verify that the channel number is one of the legal channel numbers. Channel numbers are 0-based.

#### 5.37.3.5 bool SensorBase::CheckDigitalModule (UINT32 *slot*) `[static]`

Check that the digital module number is valid. Module numbers are the slot number that they are inserted in.

#### 5.37.3.6 bool SensorBase::CheckPWMChannel (UINT32 *channel*) `[static]`

Check that the digital channel number is valid. Verify that the channel number is one of the legal channel numbers. Channel numbers are 0-based.

#### 5.37.3.7 bool SensorBase::CheckPWMModule (UINT32 *slot*) `[static]`

Check that the digital module number is valid. Module numbers are the slot number that they are inserted in.

#### 5.37.3.8 bool SensorBase::CheckRelayChannel (UINT32 *channel*) `[static]`

Check that the digital channel number is valid. Verify that the channel number is one of the legal channel numbers. Channel numbers are 0-based.

#### 5.37.3.9 bool SensorBase::CheckRelayModule (UINT32 *slot*) `[static]`

Check that the digital module number is valid. Module numbers are the slot number that they are inserted in.

#### 5.37.3.10 bool SensorBase::CheckSolenoidChannel (UINT32 *channel*) `[static]`

Verify that the solenoid channel number is within limits.

#### 5.37.3.11 bool SensorBase::CheckSolenoidModule (UINT32 *slot*) `[static]`

Verify that the solenoid module is correct. Verify that the solenoid module is slot 8 (for now).

#### 5.37.3.12 void SensorBase::DeleteSingletons () `[static]`

Delete all the singleton classes on the list. All the classes that were allocated as singletons need to be deleted so their resources can be freed.

#### 5.37.3.13 void SensorBase::SetDefaultAnalogModule (UINT32 *slot*) `[static]`

Sets the default Analog module. This sets the default analog module to use for objects that are created without specifying the analog module in the constructor. The default module is initialized to the first module in the chassis.

#### 5.37.3.14 void SensorBase::SetDefaultDigitalModule (UINT32 *slot*) `[static]`

Sets the default Digital Module. This sets the default digital module to use for objects that are created without specifying the digital module in the constructor. The default module is initialized to the first module in the chassis.

#### 5.37.3.15 void SensorBase::SetDefaultSolenoidModule (UINT32 *slot*) `[static]`

Set the default location for the Solenoid (9472) module. Currently the module must be in slot 8, but it might change in the future.

The documentation for this class was generated from the following files:

- SensorBase.h
- SensorBase.cpp

## 5.38 SerialPort Class Reference

```
#include <SerialPort.h>
```

### Public Member Functions

- SerialPort (UINT32 baudRate, UINT8 dataBits=8, Parity parity=kParity_None, StopBits stopBits=kStopBits_One)
- ~SerialPort ()
- void SetFlowControl (FlowControl flowControl)
- void EnableTermination (char terminator= '\n')
- void DisableTermination ()
- INT32 GetBytesReceived ()
- void Printf (const char *writeFmt,...)
- void Scanf (const char *readFmt,...)
- UINT32 Read (char *buffer, INT32 count)
- UINT32 Write (const char *buffer, INT32 count)
- void SetTimeout (float timeout)
- void SetWriteBufferMode (WriteBufferMode mode)
- void Flush ()
- void Reset ()

### 5.38.1 Detailed Description

Driver for the RS-232 serial port on the cRIO.

The current implementation uses the VISA formatted I/O mode. This means that all traffic goes through the fomatted buffers. This allows the intermingled use of Printf(), Scanf(), and the raw buffer accessors Read() and Write().

More information can be found in the NI-VISA User Manual here: `http://www.ni.com/pdf/manuals/370423a.pdf` and the NI-VISA Programmer's Reference Manual here: `http://www.ni.com/pdf/manuals/370132c.pdf`

### 5.38.2 Constructor & Destructor Documentation

#### 5.38.2.1 SerialPort::SerialPort (UINT32 *baudRate*, UINT8 *dataBits* = `8`, SerialPort::Parity *parity* = `kParity_None`, SerialPort::StopBits *stopBits* = `kStopBits_One`)

Create an instance of a Serial Port class.

**Parameters:**

***baudRate*** The baud rate to configure the serial port. The cRIO-9074 supports up to 230400 Baud.

***dataBits*** The number of data bits per transfer. Valid values are between 5 and 8 bits.

***parity*** Select the type of parity checking to use.

***stopBits*** The number of stop bits to use as defined by the enum StopBits.

#### 5.38.2.2 SerialPort::~SerialPort ()

Destructor.

### 5.38.3 Member Function Documentation

#### 5.38.3.1 void SerialPort::DisableTermination ()

Disable termination behavior.

#### 5.38.3.2 void SerialPort::EnableTermination (char *terminator* = '\n')

Enable termination and specify the termination character.

Termination is currently only implemented for receive. When the the terminator is recieved, the Read() or Scanf() will return fewer bytes than requested, stopping after the terminator.

**Parameters:**

***terminator*** The character to use for termination.

#### 5.38.3.3 void SerialPort::Flush ()

Force the output buffer to be written to the port.

This is used when SetWriteBufferMode() is set to kFlushWhenFull to force a flush before the buffer is full.

#### 5.38.3.4 INT32 SerialPort::GetBytesReceived ()

Get the number of bytes currently available to read from the serial port.

**Returns:**

The number of bytes available to read.

#### 5.38.3.5 void SerialPort::Printf (const char * *writeFmt*, ...)

Output formatted text to the serial port.

**Bug**

All pointer-based parameters seem to return an error.

**Parameters:**

***writeFmt*** A string that defines the format of the output.

#### 5.38.3.6 UINT32 SerialPort::Read (char * *buffer*, INT32 *count*)

Read raw bytes out of the buffer.

**Parameters:**

***buffer*** Pointer to the buffer to store the bytes in.

***count*** The maximum number of bytes to read.

**Returns:**

The number of bytes actually read into the buffer.

#### 5.38.3.7 void SerialPort::Reset ()

Reset the serial port driver to a known state.

Empty the transmit and receive buffers in the device and formatted I/O.

#### 5.38.3.8 void SerialPort::Scanf (const char * *readFmt*, ...)

Input formatted text from the serial port.

**Bug**

All pointer-based parameters seem to return an error.

**Parameters:**

***readFmt*** A string that defines the format of the input.

#### 5.38.3.9 void SerialPort::SetFlowControl (SerialPort::FlowControl *flowControl*)

Set the type of flow control to enable on this port.

By default, flow control is disabled.

#### 5.38.3.10 void SerialPort::SetTimeout (float *timeout*)

Configure the timeout of the serial port.

This defines the timeout for transactions with the hardware. It will affect reads and very large writes.

**Parameters:**

***timeout*** The number of seconds to to wait for I/O.

#### 5.38.3.11 void SerialPort::SetWriteBufferMode (SerialPort::WriteBufferMode *mode*)

Specify the flushing behavior of the output buffer.

When set to kFlushOnAccess, data is synchronously written to the serial port after each call to either Printf() or Write().

When set to kFlushWhenFull, data will only be written to the serial port when the buffer is full or when Flush() is called.

**Parameters:**

***mode*** The write buffer mode.

#### 5.38.3.12 UINT32 SerialPort::Write (const char ∗ *buffer*, INT32 *count*)

Write raw bytes to the buffer.

**Parameters:**

***buffer*** Pointer to the buffer to read the bytes from.

***count*** The maximum number of bytes to write.

**Returns:**

The number of bytes actually written into the port.

The documentation for this class was generated from the following files:

- SerialPort.h
- SerialPort.cpp

# 5.39 Servo Class Reference

```
#include <Servo.h>
```

Inherits PWM, and SpeedController.

## Public Member Functions

- Servo (UINT32 channel)
- Servo (UINT32 slot, UINT32 channel)
- void Set (float value)
- float Get ()
- void SetAngle (float angle)
- float GetAngle ()

## 5.39.1 Detailed Description

Standard hobby style servo.

The range parameters default to the appropriate values for the Hitec HS-322HD servo provided in the FIRST Kit of Parts in 2008.

## 5.39.2 Constructor & Destructor Documentation

### 5.39.2.1 Servo::Servo (UINT32 *channel*) `[explicit]`

Constructor that assumes the default digital module.

**Parameters:**

***channel*** The PWM channel on the digital module to which the servo is attached.

### 5.39.2.2 Servo::Servo (UINT32 *slot*, UINT32 *channel*)

Constructor that specifies the digital module.

**Parameters:**

***slot*** The slot in the chassis that the digital module is plugged into.

***channel*** The PWM channel on the digital module to which the servo is attached.

## 5.39.3 Member Function Documentation

### 5.39.3.1 float Servo::Get () `[virtual]`

Get the servo position.

Servo values range from 0.0 to 1.0 corresponding to the range of full left to full right.

**Returns:**

Position from 0.0 to 1.0.

Implements SpeedController.

#### 5.39.3.2 float Servo::GetAngle ()

Get the servo angle.

Assume that the servo angle is linear with respect to the PWM value (big assumption, need to test).

**Returns:**

The angle in degrees to which the servo is set.

#### 5.39.3.3 void Servo::Set (float *value*) `[virtual]`

Set the servo position.

Servo values range from 0.0 to 1.0 corresponding to the range of full left to full right.

**Parameters:**

***value*** Position from 0.0 to 1.0.

Implements SpeedController.

#### 5.39.3.4 void Servo::SetAngle (float *degrees*)

Set the servo angle.

Assume that the servo angle is linear with respect to the PWM value (big assumption, need to test).

Servo angles that are out of the supported range of the servo simply "saturate" in that direction In other words, if the servo has a range of (X degrees to Y degrees) than angles of less than X result in an angle of X being set and angles of more than Y degrees result in an angle of Y being set.

**Parameters:**

***degrees*** The angle in degrees to set the servo.

The documentation for this class was generated from the following files:

- Servo.h
- Servo.cpp

## 5.40 SimpleRobot Class Reference

```
#include <SimpleRobot.h>
```

Inherits RobotBase.

### Public Member Functions

- virtual void Autonomous ()
- virtual void OperatorControl ()
- virtual void RobotMain ()
- void StartCompetition ()

### 5.40.1 Detailed Description

**Todo**

If this is going to last until release, it needs a better name.

### 5.40.2 Member Function Documentation

#### 5.40.2.1 void SimpleRobot::Autonomous () `[virtual]`

Autonomous should go here. Users should add autonomous code to this method that should run while the field is in the autonomous period.

#### 5.40.2.2 void SimpleRobot::OperatorControl () `[virtual]`

Operator control (tele-operated) code should go here. Users should add Operator Control code to this method that should run while the field is in the Operator Control (tele-operated) period.

#### 5.40.2.3 void SimpleRobot::RobotMain () `[virtual]`

Robot main program for free-form programs.

This should be overridden by user subclasses if the intent is to not use the Autonomous() and Operator-Control() methods. In that case, the program is responsible for sensing when to run the autonomous and operator control functions in their program.

This method will be called immediately after the constructor is called. If it has not been overridden by a user subclass (i.e. the default version runs), then the Autonomous() and OperatorControl() methods will be called.

#### 5.40.2.4 void SimpleRobot::StartCompetition () `[virtual]`

Start a competition. This code needs to track the order of the field starting to ensure that everything happens in the right order. Repeatedly run the correct method, either Autonomous or OperatorControl when the robot is enabled. After running the correct method, wait for some state to change, either the other mode starts or the robot is disabled. Then go back and wait for the robot to be enabled again.

Implements RobotBase.

The documentation for this class was generated from the following files:

- SimpleRobot.h
- SimpleRobot.cpp

## 5.41 Solenoid Class Reference

```
#include <Solenoid.h>
```

Inherits SensorBase.

### Public Member Functions

- Solenoid (UINT32 channel)
- Solenoid (UINT32 slot, UINT32 channel)
- ~Solenoid ()
- void Set (bool on)
- bool Get ()

### Protected Member Functions

- UINT32 SlotToIndex (UINT32 slot)

### 5.41.1 Detailed Description

Solenoid class for running high voltage Digital Output (9472 module).

The Solenoid class is typically used for pneumatics solenoids, but could be used for any device within the current spec of the 9472 module.

### 5.41.2 Constructor & Destructor Documentation

#### 5.41.2.1 Solenoid::Solenoid (UINT32 *channel*) `[explicit]`

Constructor.

**Parameters:**

***channel*** The channel on the module to control.

#### 5.41.2.2 Solenoid::Solenoid (UINT32 *slot*, UINT32 *channel*)

Constructor.

**Parameters:**

***slot*** The slot that the 9472 module is plugged into.

***channel*** The channel on the module to control.

#### 5.41.2.3 Solenoid::~Solenoid ()

Destructor.

## 5.41.3 Member Function Documentation

### 5.41.3.1 bool Solenoid::Get ()

Read the current value of the solenoid.

**Returns:**

The current value of the solenoid.

### 5.41.3.2 void Solenoid::Set (bool *on*)

Set the value of a solenoid.

**Parameters:**

***on*** Turn the solenoid output off or on.

### 5.41.3.3 UINT32 Solenoid::SlotToIndex (UINT32 *slot*) `[protected]`

Convert slot number to index.

**Parameters:**

***slot*** The slot in the chassis where the module is plugged in.

**Returns:**

An index to represent the module internally.

The documentation for this class was generated from the following files:

- Solenoid.h
- Solenoid.cpp

# 5.42 SpeedController Class Reference

`#include <SpeedController.h>`

Inherited by Jaguar, Servo, and Victor.

## Public Member Functions

- virtual void Set (float speed)=0
- virtual float Get ()=0

## 5.42.1 Detailed Description

Interface for speed controlling devices.

## 5.42.2 Member Function Documentation

### 5.42.2.1 virtual float SpeedController::Get () `[pure virtual]`

Common interface for getting the current set speed of a speed controller.

**Returns:**

The current set speed. Value is between -1.0 and 1.0.

Implemented in Jaguar, Servo, and Victor.

### 5.42.2.2 virtual void SpeedController::Set (float *speed*) `[pure virtual]`

Common interface for setting the speed of a speed controller.

**Parameters:**

***speed*** The speed to set. Value should be between -1.0 and 1.0.

Implemented in Jaguar, Servo, and Victor.

The documentation for this class was generated from the following file:

- SpeedController.h

## 5.43 Synchronized Class Reference

`#include <Synchronized.h>`

### Public Member Functions

- Synchronized (SEM_ID)
- virtual ~Synchronized ()

### 5.43.1 Detailed Description

Provide easy support for critical regions. A critical region is an area of code that is always executed under mutual exclusion. Only one task can be executing this code at any time. The idea is that code that manipulates data that is shared between two or more tasks has to be prevented from executing at the same time otherwise a race condition is possible when both tasks try to update the data. Typically semaphores are used to ensure only single task access to the data. Synchronized objects are a simple wrapper around semaphores to help ensure that semaphores are always signaled (semGive) after a wait (semTake).

### 5.43.2 Constructor & Destructor Documentation

#### 5.43.2.1 Synchronized::Synchronized (SEM_ID *semaphore*) `[explicit]`

Synchronized class deals with critical regions. Declare a Synchronized object at the beginning of a block. That will take the semaphore. When the code exits from the block it will call the destructor which will give the semaphore. This ensures that no matter how the block is exited, the semaphore will always be released. Use the CRITICAL_REGION(SEM_ID) and END_REGION macros to make the code look cleaner (see header file)

**Parameters:**

*semaphore* The semaphore controlling this critical region.

#### 5.43.2.2 Synchronized::~Synchronized () `[virtual]`

Syncronized destructor. This destructor frees the semaphore ensuring that the resource is freed for the block containing the Synchronized object.

The documentation for this class was generated from the following files:

- Synchronized.h
- Synchronized.cpp

## 5.44 Task Class Reference

```
#include <Task.h>
```

### Public Member Functions

- Task (char ∗name, FUNCPTR function, INT32 priority=kDefaultPriority, UINT32 stackSize=20000)
- bool Start (UINT32 arg0=0, UINT32 arg1=0, UINT32 arg2=0, UINT32 arg3=0, UINT32 arg4=0, UINT32 arg5=0, UINT32 arg6=0, UINT32 arg7=0, UINT32 arg8=0, UINT32 arg9=0)
- bool Restart (void)
- bool Stop (void)
- bool IsReady (void)
- bool IsSuspended (void)
- bool Suspend (void)
- bool Resume (void)
- bool Verify (void)
- INT32 GetPriority (void)
- bool SetPriority (INT32 priority)
- char ∗ GetName (void)
- INT32 GetID (void)

### 5.44.1 Detailed Description

WPI task is a wrapper for the native Task object. All WPILib tasks are managed by a static task manager for simplified cleanup.

### 5.44.2 Constructor & Destructor Documentation

#### 5.44.2.1 Task::Task (char ∗ *name*, FUNCPTR *function*, INT32 *priority* = `kDefaultPriority`, UINT32 *stackSize* = `20000`)

Create but don't launch a task.

**Parameters:**

*name* The name of the task. "FRC" will be prepended to the task name.

*function* The address of the function to run as the new task.

*priority* The VxWorks priority for the task.

*stackSize* The size of the stack for the task

### 5.44.3 Member Function Documentation

#### 5.44.3.1 INT32 Task::GetID (void)

Get the ID of a task

**Returns:**

task ID of this task.

#### 5.44.3.2 char * Task::GetName (void)

Returns the name of the task.

**Returns:**

Pointer to the name of the task or NULL if not allocated

#### 5.44.3.3 INT32 Task::GetPriority (void)

Gets the priority of a task.

**Returns:**

task priority or 0 if an error occured

#### 5.44.3.4 bool Task::IsReady (void)

Returns true if the task is ready to execute (i.e. not suspended, delayed, or blocked).

**Returns:**

true if ready, false if not ready.

#### 5.44.3.5 bool Task::IsSuspended (void)

Returns true if the task was explicitly suspended by calling Suspend()

**Returns:**

true if suspended, false if not suspended.

#### 5.44.3.6 bool Task::Restart (void)

Restarts a running task. If the task isn't started, it starts it.

**Returns:**

false if the task is running and we are unable to kill the previous instance

#### 5.44.3.7 bool Task::Resume (void)

Resumes a paused task. Returns true on success, false if unable to resume or if the task isn't running/paused.

#### 5.44.3.8 bool Task::SetPriority (INT32 *priority*)

This routine changes a task's priority to a specified priority. Priorities range from 0, the highest priority, to 255, the lowest priority. Default task priority is 100.

**Parameters:**

*priority* The priority the task should run at.

**Returns:**

true on success.

#### 5.44.3.9 bool Task::Start (UINT32 *arg0* = 0, UINT32 *arg1* = 0, UINT32 *arg2* = 0, UINT32 *arg3* = 0, UINT32 *arg4* = 0, UINT32 *arg5* = 0, UINT32 *arg6* = 0, UINT32 *arg7* = 0, UINT32 *arg8* = 0, UINT32 *arg9* = 0)

Starts this task. If it is already running or unable to start, it fails and returns false.

#### 5.44.3.10 bool Task::Stop (void)

Kills the running task.

**Returns:**

true on success false if the task doesn't exist or we are unable to kill it.

#### 5.44.3.11 bool Task::Suspend (void)

Pauses a running task. Returns true on success, false if unable to pause or the task isn't running.

#### 5.44.3.12 bool Task::Verify (void)

Verifies a task still exists.

**Returns:**

true on success.

The documentation for this class was generated from the following files:

- Task.h
- Task.cpp

# 5.45 Timer Class Reference

`#include <Timer.h>`

## Public Member Functions

- Timer ()
- double Get ()
- void Reset ()
- void Start ()
- void Stop ()

### 5.45.1 Detailed Description

Timer objects measure accumulated time in seconds. The timer object functions like a stopwatch. It can be started, stopped, and cleared. When the timer is running its value counts up in seconds. When stopped, the timer holds the current value. The implemenation simply records the time when started and subtracts the current time whenever the value is requested.

### 5.45.2 Constructor & Destructor Documentation

#### 5.45.2.1 Timer::Timer ()

Create a new timer object.

Create a new timer object and reset the time to zero. The timer is initially not running and must be started.

### 5.45.3 Member Function Documentation

#### 5.45.3.1 double Timer::Get ()

Get the current time from the timer. If the clock is running it is derived from the current system clock the start time stored in the timer class. If the clock is not running, then return the time when it was last stopped.

**Returns:**

unsigned Current time value for this timer in seconds

#### 5.45.3.2 void Timer::Reset ()

Reset the timer by setting the time to 0.

Make the timer startTime the current time so new requests will be relative now

#### 5.45.3.3 void Timer::Start ()

Start the timer running. Just set the running flag to true indicating that all time requests should be relative to the system clock.

#### 5.45.3.4 void Timer::Stop ()

Stop the timer. This computes the time as of now and clears the running flag, causing all subsequent time requests to be read from the accumulated time rather than looking at the system clock.

The documentation for this class was generated from the following files:

- Timer.h
- Timer.cpp

# 5.46 Ultrasonic Class Reference

`#include <Ultrasonic.h>`

Inherits SensorBase.

## Public Member Functions

- Ultrasonic (DigitalOutput ∗pingChannel, DigitalInput ∗echoChannel)
- Ultrasonic (DigitalOutput &pingChannel, DigitalInput &echoChannel)
- Ultrasonic (UINT32 pingChannel, UINT32 echoChannel)
- Ultrasonic (UINT32 pingSlot, UINT32 pingChannel, UINT32 echoSlot, UINT32 echoChannel)
- virtual ∼Ultrasonic ()
- void Ping ()
- bool IsRangeValid ()
- double GetRangeInches ()
- double GetRangeMM ()

## Static Public Member Functions

- static void SetAutomaticMode (bool enabling)

### 5.46.1 Detailed Description

Ultrasonic rangefinder class. The Ultrasonic rangefinder measures absolute distance based on the round-trip time of a ping generated by the controller. These sensors use two transducers, a speaker and a microphone both tuned to the ultrasonic range. A common ultrasonic sensor, the Daventech SRF04 requires a short pulse to be generated on a digital channel. This causes the chirp to be emmitted. A second line becomes high as the ping is transmitted and goes low when the echo is received. The time that the line is high determines the round trip distance (time of flight).

### 5.46.2 Constructor & Destructor Documentation

#### 5.46.2.1 Ultrasonic::Ultrasonic (DigitalOutput ∗ *pingChannel*, DigitalInput ∗ *echoChannel*)

Create an instance of an Ultrasonic Sensor from a DigitalInput for the echo channel and a DigitalOutput for the ping channel.

**Parameters:**

***pingChannel*** The digital output object that starts the sensor doing a ping. Requires a 10uS pulse to start.

***echoChannel*** The digital input object that times the return pulse to determine the range.

#### 5.46.2.2 Ultrasonic::Ultrasonic (DigitalOutput & *pingChannel*, DigitalInput & *echoChannel*)

Create an instance of an Ultrasonic Sensor from a DigitalInput for the echo channel and a DigitalOutput for the ping channel.

**Parameters:**

***pingChannel*** The digital output object that starts the sensor doing a ping. Requires a 10uS pulse to start.

***echoChannel*** The digital input object that times the return pulse to determine the range.

#### 5.46.2.3 Ultrasonic::Ultrasonic (UINT32 *pingChannel*, UINT32 *echoChannel*)

Create an instance of the Ultrasonic Sensor using the default module. This is designed to supchannel the Daventech SRF04 and Vex ultrasonic sensors. This constructor assumes that both digital I/O channels are in the default digital module.

**Parameters:**

***pingChannel*** The digital output channel that sends the pulse to initiate the sensor sending the ping.

***echoChannel*** The digital input channel that receives the echo. The length of time that the echo is high represents the round trip time of the ping, and the distance.

#### 5.46.2.4 Ultrasonic::Ultrasonic (UINT32 *pingSlot*, UINT32 *pingChannel*, UINT32 *echoSlot*, UINT32 *echoChannel*)

Create an instance of the Ultrasonic sensor using specified modules. This is designed to supchannel the Daventech SRF04 and Vex ultrasonic sensors. This constructors takes the channel and module slot for each of the required digital I/O channels.

**Parameters:**

***pingSlot*** The digital module that the pingChannel is in.

***pingChannel*** The digital output channel that sends the pulse to initiate the sensor sending the ping.

***echoSlot*** The digital module that the echoChannel is in.

***echoChannel*** The digital input channel that receives the echo. The length of time that the echo is high represents the round trip time of the ping, and the distance.

#### 5.46.2.5 Ultrasonic::~Ultrasonic () `[virtual]`

Destructor for the ultrasonic sensor. Delete the instance of the ultrasonic sensor by freeing the allocated digital channels. If the system was in automatic mode (round robin), then it is stopped, then started again after this sensor is removed (provided this wasn't the last sensor).

### 5.46.3 Member Function Documentation

#### 5.46.3.1 double Ultrasonic::GetRangeInches ()

Get the range in inches from the ultrasonic sensor.

**Returns:**

double Range in inches of the target returned from the ultrasonic sensor. If there is no valid value yet, i.e. at least one measurement hasn't completed, then return 0.

#### 5.46.3.2 double Ultrasonic::GetRangeMM ()

Get the range in millimeters from the ultrasonic sensor.

**Returns:**

double Range in millimeters of the target returned by the ultrasonic sensor. If there is no valid value yet, i.e. at least one measurement hasn't complted, then return 0.

#### 5.46.3.3 bool Ultrasonic::IsRangeValid ()

Check if there is a valid range measurement. The ranges are accumulated in a counter that will increment on each edge of the echo (return) signal. If the count is not at least 2, then the range has not yet been measured, and is invalid.

#### 5.46.3.4 void Ultrasonic::Ping ()

Single ping to ultrasonic sensor. Send out a single ping to the ultrasonic sensor. This only works if automatic (round robin) mode is disabled. A single ping is sent out, and the counter should count the semi-period when it comes in. The counter is reset to make the current value invalid.

#### 5.46.3.5 void Ultrasonic::SetAutomaticMode (bool *enabling*) `[static]`

Turn Automatic mode on/off. When in Automatic mode, all sensors will fire in round robin, waiting a set time between each sensor.

**Parameters:**

***enabling*** Set to true if round robin scheduling should start for all the ultrasonic sensors. This scheduling method assures that the sensors are non-interfering because no two sensors fire at the same time. If another scheduling algorithm is preffered, it can be implemented by pinging the sensors manually and waiting for the results to come back.

The documentation for this class was generated from the following files:

- Ultrasonic.h
- Ultrasonic.cpp

## 5.47 Victor Class Reference

`#include <Victor.h>`

Inherits PWM, and SpeedController.

### Public Member Functions

- Victor (UINT32 channel)
- Victor (UINT32 slot, UINT32 channel)
- void Set (float value)
- float Get ()

### 5.47.1 Detailed Description

IFI Victor Speed Controller

### 5.47.2 Constructor & Destructor Documentation

#### 5.47.2.1 Victor::Victor (UINT32 *channel*) `[explicit]`

Constructor that assumes the default digital module.

**Parameters:**

***channel*** The PWM channel on the digital module that the Victor is attached to.

#### 5.47.2.2 Victor::Victor (UINT32 *slot*, UINT32 *channel*)

Constructor that specifies the digital module.

**Parameters:**

***slot*** The slot in the chassis that the digital module is plugged into.

***channel*** The PWM channel on the digital module that the Victor is attached to.

### 5.47.3 Member Function Documentation

#### 5.47.3.1 float Victor::Get () `[virtual]`

Get the recently set value of the PWM.

**Returns:**

The most recently set value for the PWM between -1.0 and 1.0.

Implements SpeedController.

#### 5.47.3.2 void Victor::Set (float *speed*) [virtual]

Set the PWM value.

The PWM value is set using a range of -1.0 to 1.0, appropriately scaling the value for the FPGA.

**Parameters:**

***speed*** The speed value between -1.0 and 1.0 to set.

Implements SpeedController.

The documentation for this class was generated from the following files:

- Victor.h
- Victor.cpp

## 5.48 Watchdog Class Reference

`#include <Watchdog.h>`

Inherits SensorBase.

### Public Member Functions

- Watchdog ()
- virtual ~Watchdog ()
- bool Feed ()
- void Kill ()
- float GetTimer ()
- float GetExpiration ()
- void SetExpiration (float expiration_ms)
- bool GetEnabled ()
- void SetEnabled (bool enabled)
- bool IsAlive ()
- bool IsSystemActive ()

### 5.48.1 Detailed Description

Watchdog timer class. The watchdog timer is designed to keep the robots safe. The idea is that the robot program must constantly "feed" the watchdog otherwise it will shut down all the motor outputs. That way if a program breaks, rather than having the robot continue to operate at the last known speed, the motors will be shut down.

This is serious business. Don't just disable the watchdog. You can't afford it!

http://thedailywtf.com/Articles/_0x2f__0x2f_TODO_0x3a__Uncomment_-Later.aspx

### 5.48.2 Constructor & Destructor Documentation

#### 5.48.2.1 Watchdog::Watchdog ()

The Watchdog is born.

#### 5.48.2.2 Watchdog::~Watchdog () [virtual]

Time to bury him in the back yard.

### 5.48.3 Member Function Documentation

#### 5.48.3.1 bool Watchdog::Feed ()

Throw the dog a bone.

When everything is going well, you feed your dog when you get home. Let's hope you don't drive your car off a bridge on the way home... Your dog won't get fed and he will starve to death.

By the way, it's not cool to ask the neighbor (some random task) to feed your dog for you. He's your responsibility!

**Returns:**

Returns the previous state of the watchdog before feeding it.

#### 5.48.3.2 bool Watchdog::GetEnabled ()

Find out if the watchdog is currently enabled or disabled (mortal or immortal).

**Returns:**

Enabled or disabled.

#### 5.48.3.3 float Watchdog::GetExpiration ()

Read what the current expiration is.

**Returns:**

The number of seconds before starvation following a meal (watchdog starves if it doesn't eat this often).

#### 5.48.3.4 float Watchdog::GetTimer ()

Read how long it has been since the watchdog was last fed.

**Returns:**

The number of seconds since last meal.

#### 5.48.3.5 bool Watchdog::IsAlive ()

Check in on the watchdog and make sure he's still kicking.

This indicates that your watchdog is allowing the system to operate. It is still possible that the network communications is not allowing the system to run, but you can check this to make sure it's not your fault. Check IsSystemActive() for overall system status.

If the watchdog is disabled, then your watchdog is immortal.

**Returns:**

Is the watchdog still alive?

#### 5.48.3.6 bool Watchdog::IsSystemActive ()

Check on the overall status of the system.

**Returns:**

Is the system active (i.e. PWM motor outputs, etc. enabled)?

#### 5.48.3.7 void Watchdog::Kill ()

Put the watchdog out of its misery.

Don't wait for your dying robot to starve when there is a problem. Kill it quickly, cleanly, and humanely.

#### 5.48.3.8 void Watchdog::SetEnabled (bool *enabled*)

Enable or disable the watchdog timer.

When enabled, you must keep feeding the watchdog timer to keep the watchdog active, and hence the dangerous parts (motor outputs, etc.) can keep functioning. When disabled, the watchdog is immortal and will remain active even without being fed. It will also ignore any kill commands while disabled.

**Parameters:**

***enabled*** Enable or disable the watchdog.

#### 5.48.3.9 void Watchdog::SetExpiration (float *expiration*)

Configure how many seconds your watchdog can be neglected before it starves to death.

**Parameters:**

***expiration*** The number of seconds before starvation following a meal (watchdog starves if it doesn't eat this often).

The documentation for this class was generated from the following files:

- Watchdog.h
- Watchdog.cpp

# Index