Funnel as3 library documentation

v 0.1 April 10, 2011

Funnel classes

Arduino
Fio
Osc
Pin
SignalScope

Use the following statement to import all of the UI classes: `import funnel.*;`

You can alternatively import only the classes you are using in your program, but it's easier to use the wildcard format above. There is no difference in code size.

Arduino

```
import funnel.Arduino;
```

Properties:

`FIRMATA:Configuration` [static][read-only]
gets the default Arduino configuration

`gui:IOModuleGUI`
a gui to represent the Arduino on screen

Example:
Create an instance of an ArduinoGUI object and assign it to the arduino.gui property

```
var gui:ArduinoGUI = new ArduinoGUI();
addChild(gui);
arduino.gui = gui;
```

`samplingInterval:int`
the rate at which the inputs are sampled from the arduino board (default = 33 milliseconds)

Public Methods:

```
Arduino(config:Configuration = null, host:String
        = localhost, portNum:Number = 9000,
        samplingInterval:int = 33)
```

Parameters:
`config:Configuration` [option] - the configuration settings (default = FIRMATA)
`host:String` [optional] - the IP address of Funnel Server (default = "localhost")
`portNum:Number` [optional] - the port number used by Funnel Server (default = 9000)
`samplingInterval:int` [optional] – the rate at which the inputs are samples from the Arduino board (default is 33 milliseconds).

Example:
```
var arduino:Arduino;
…
var config:Configuration = Arduino.FIRMATA;
arduino = new Arduino(config);
```

```
analogPin(pinNum:uint):Pin
```

Parameters:
`pinNum:uint` [required] – the pin number associated with the Pin object (for example, `arduino.analogPin(0)` will return a reference to the Pin object associated with pin 0.

returns:
a reference to the specified analog pin object

```
digitalPin(pinNum:uint):Pin
```

Parameters:
`pinNum:uint` [required] – the pin number associated with the Pin object (for example, `arduino.digitalPin(0)` will return a reference to the Pin object associated with pin 0.

returns:
a reference to the specified digital pin object

```
sendFirmataString(stringToSend:String):void
```

Parameters:
`stringToSend:String` [required] – the string to send to the Arduino board

Example:
Send the string “hello” to the arduino board. This is useful for applications that use a custom sketch running on the arduino rather than StandardFirmata.

```
arduino.sendFirmataString(“hello”);
```

You can also listen for a string sent from the Arduino board:

```
arduino.addEventListener(FunnelEvent.FIRMATA_STRING,
                         onStringEvent);

function onStringEvent(e:FunnelEvent):void {
   trace(e.message);
}
```

`sendSysexMessage(command:uint, message:Array):void`

Used to send a system exclusive message to the arduino. This is useful for custom implementations where StandardFirmata is not used. See firmata.org for details.

Parameters:
`command:uint` [required] – the message command
`message:Array` [required] – the message body as an Array

`setServoPulseRange(pinNumber:uint, minPulse:uint, maxPulse:uint):void`

Sets the minimum and maximum servo pulse range for the servo attached to the specified pin number. You will not normally need to call this method unless you are using a unique type of servo.

Parameters:
`pinNumber:uint` [required] – the pin number of the attached servo
`minPulse` [required] – the minimum pulse for the attached servo (see servo datasheet)
`maxPulse` [required] – the maximum pulse for the attached servo (see servo datasheet)

Events:

`FunnelEvent.FIRMATA_STRING`
dispatched when a Firmata string is received from the Arduino

`FunnelEvent.READY`
dispatched when the Arduino is ready

`FunnelEvent.I2C_POWER_PINS_READY`
if power pins were enabled, dispatched when I2C power pins of the Arduino are ready.

`FunnelErrorEvent.CONFIGURATION_ERROR`
dispatched when configuration settings are not specified

`FunnelErrorEvent.ERROR`
dispatched when an error occurs

`FunnelErrorEvent.REBOOT_ERROR`
dispatched when an I/O error results from a module failing to restart

---

Fio

```
import funnel.Fio;
```

The Fio is class is used to handle Fio modules. Unlike the Arduino class, when using the Fio class you need to reference the specific ioModule associated with the Fio board. This is because you can use more than one Fio board with a single XBee explorer connection and the reference to ioModule() is used to distinguish between multiple Fio boards.

For example, to get a reference to an analog pin on an Arduino board you would use something like:
```
var sensorPin:Pin  = arduino.analogPin(0);
```

but with an Fio, you need to reference the ioModule to which the Pin object is associated:
```
var sensorPin:Pin = fio.ioModule(1).analogPin(0);
```

if you had a second Fio you could reference it like this:
```
var sensorPin:Pin = fio.ioModule(2).analogPin(0);
```

Properties:

`FIRMATA:Configuration` [static][read-only]
gets the default Fio configuration

`gui:IOModuleGUI`
a gui to represent the Arduino on screen

Example:
Create an instance of an ArduinoGUI object and assign it to the fio.ioModule(1).gui property

```
var gui:ArduinoGUI = new ArduinoGUI();
addChild(gui);
fio.ioModule(1).gui = gui;
```

`samplingInterval:int`
the rate at which the inputs are sampled from the Fio board (default = 33 milliseconds)

Public Methods:

```
Fio(nodes:Array = null, config:Configuration = null,
      host:String = localhost, portNum:Number = 9000,
      samplingInterval:int = 33)
```

Parameters:
`nodes:Array` [option] - the configuration settings (default = [ ])

`config:Configuration` [option] - the configuration settings (default = Fio.FIRMATA)
`host:String` [optional] - the IP address of Funnel Server (default = "localhost")
`portNum:Number` [optional] - the port number used by Funnel Server (default = 9000)
`samplingInterval:int` [optional] – the rate at which the inputs are samples from the Fio board (default is 33 milliseconds).

Example:
```
var fio:Fio;
…
var config:Configuration = Fio.FIRMATA;
fio = new Fio([1], config);    // note the first parameter [1]
```

`analogPin``(pinNum:uint):Pin`

to call this method, you need to reference the ioModule (see example)

Parameters:
`pinNum:uint` [required] – the pin number associated with the Pin object (for example, `arduino.analogPin(0)` will return a reference to the Pin object associated with pin 0.

returns:
a reference to the specified analog pin object

Example:
You cannot call analogPin directly when using a wirelessly connected Fio, you must reference the ioModule in order to call the analogPin method

```
var sensorPin:Pin = fio.ioModule(1).analogPin(0);
```

digitalPin(pinNum:uint):Pin

to call this method, you need to reference the ioModule (see example)

Parameters:
`pinNum:uint` [required] – the pin number associated with the Pin object (for example, `arduino.digitalPin(0)` will return a reference to the Pin object associated with pin 0.

returns:
a reference to the specified digital pin object

Example:
You cannot call digitalPin directly when using a wirelessly connected Fio, you must reference the ioModule in order to call the digitalPin method

```
var ledPin:Pin = fio.ioModule(1).digitalPin(13);
```

sendFirmataString(stringToSend:String):void

to call this method, you need to reference the ioModule()

Parameters:
`stringToSend:String` [required] – the string to send to the Fio board

Example:
Send the string “hello” to the Fio board. This is useful for applications that use a custom sketch running on the Fio rather than StandardFirmata.

```
fio.ioModule(1).sendFirmataString(“hello”);
```

You can also listen for a string sent from the Fio board:

```
fio.ioModule(1).addEventListener(FunnelEvent.FIRMATA_STRING,
                                onStringEvent);

function onStringEvent(e:FunnelEvent):void {
   trace(e.message);
}
```

sendSysex(command:uint, message:Array):void

to call this method, you need to reference the ioModule

Used to send a system exclusive message to the Fio. This is useful for custom implementations where StandardFirmata is not used. See firmata.org for details.

Parameters:
`command:uint` [required] – the message command
`message:Array` [required] – the message body as an Array

`setServoPulseRange``(pinNumber:uint, minPulse:uint, maxPulse:uint):void`

to call this method, you need to reference the ioModule

Sets the minimum and maximum servo pulse range for the servo attached to the specified pin number. You will not normally need to call this method unless you are using a unique type of servo.

Parameters:
`pinNumber:uint` [required] – the pin number of the attached servo
`minPulse` [required] – the minimum pulse for the attached servo (see servo datasheet)
`maxPulse` [required] – the maximum pulse for the attached servo (see servo datasheet)

Public Constants:

`ALL` Constant
`public static const ALL:uint = 0xFFFF;`

represents all of the connected Fio modules (in the case that there are more than one). For example, to set the value of pin 10 for all connected Fio modules:
`fio.module(ALL).pin(10).value = x;`

Events:

`FunnelEvent.FIRMATA_STRING`
dispatched when a Firmata string is received from the Fio

`FunnelEvent.READY`
dispatched when the Fio is ready

`FunnelEvent.I2C_POWER_PINS_READY`
if power pins were enabled, dispatched when I2C power pins of the Fio are ready.

`FunnelErrorEvent.CONFIGURATION_ERROR`
dispatched when configuration settings are not specified

`FunnelErrorEvent.ERROR`
dispatched when an error occurs

`FunnelErrorEvent.REBOOT_ERROR`
dispatched when an I/O error results from a module failing to restart

---

Osc

```
import funnel.Osc;
```

Osc outputs a waveform on the associated PWM pin (or any digital pin if the wave function is Osc.SQUARE). For example, this can be used to blink or fade an LED on or off.

Note that many of the methods defined in the OSC class will only work if the Osc filter is set for a PWM pin on the Arduino or Fio. When using a PWM pin you must set the configuration of the pin as follows:
`config.setDigitalPinMode(9, PWM); // pin 9 set to pwm`

Properties:

`amplitude:Number`
the amplitude of the waveform used to drive the oscillation (default = 1)

`freq:Number`
the frequency of the waveform used to drive the oscillation (default = 1)

`offset:Number`
the offset of the waveform used to drive the oscillation (default = 0)

`phase:Number`
the phase of the waveform used to drive the oscillation (default = 0)

`serviceInterval:uint`
the service interval of the oscillation in milliseconds

`times:Number`
the repeat count from 0 to infinity (default = 0)

`value:Number` [read-only]
the generated number

`value:Number` [read-only]
the generated number

`wave:Function`
the wave function (default = Osc.SIN)

Public Methods:

```
Osc(wave:Function = null, freq:Number = 1,
                amplitude:Number = 1, offset:Number = 0,
                phase:Number = 0, times:Number = 0)
```

Parameters:
`wave:Function` [optional] - the function generating the waveform (default = `Osc.SIN`)
`freq:Number` [optional] - the frequency of the waveform (default = 1)
`amplitude:Number` [optional] - the amplitude of the wave (default = 1)
`offset:Number` [optional] - the offset of the wave (default = 0)
`phase:Number` [optional] - the phase of the wave (default = 0)
`times:Number` [optional] - the repeat count of the wave (default = 0)

Example:
use a square wave to blink an LED attached to digital pin 13

```
var osc:Osc = new Osc(Osc.SQUARE, 0.5, 1, 0, 0);
arduino.digitalPin(13).filters = [osc];
osc.start();
```

here's another way to accomplish the same task:

```
var led = new LED(arduino.digitalPin(13));
led.blink(1000, 0, Osc.SQUARE); // blink every 1 second
forever
```

```
IMPULSE(val:Number, lastVal:Number):Number
```

Parameters:
`val:Number` [required] - desc
`lastVal:Number` [required] - desc

Example:
use an impulse wave for the oscillator.

```
var osc:Osc = new Osc(Osc.IMPULSE);
```

Returns:
Something

LINEAR(val:Number, lastVal:Number):Number

Parameters:
`val:Number` [required] - desc
`lastVal:Number` [required] - desc

Example:
use an linear wave for the oscillator.

```
var osc:Osc = new Osc(Osc.LINEAR);
```

Returns:
something

reset():void

resets the oscillator

SAW(val:Number, lastVal:Number):Number

Parameters:
`val:Number` [required] - desc
`lastVal:Number` [required] - desc

Example:
use an sawtooth wave for the oscillator.

```
var osc:Osc = new Osc(Osc.SAW);
```

Returns:
something

SIN(val:Number, lastVal:Number):Number

Parameters:
`val:Number` [required] - desc
`lastVal:Number` [required] - desc

Example:
use an sine wave for the oscillator.

```
var osc:Osc = new Osc(Osc.SIN);
```

Returns:
something

`SQUARE(val:Number, lastVal:Number):Number`

Parameters:
`val:Number` [required] - desc
`lastVal:Number` [required] - desc

Example:
use an square wave for the oscillator.

```
var osc:Osc = new Osc(Osc.SQUARE);
```

Returns:
something

`start():void`

starts the oscillator

`stop():void`

stops the oscillator

`TRIANGLE(val:Number, lastVal:Number):Number`

Parameters:
`val:Number` [required] - desc
`lastVal:Number` [required] - desc

Example:
use an triangle wave for the oscillator.

```
var osc:Osc = new Osc(Osc.TRIANGLE);
```

Returns:
something

`update``(interval:int = -1):void`

the interval at which the oscillator is updated. Default is 33 milliseconds.

Parameters:
`interval:int` [required] - update interval in milliseconds (default = 33 ms)

Events:

`GeneratorEvent.UPDATE : String = update`
dispatched when the output value is updated

Pin

```
import funnel.Pin;
```

The Pin class represents the input and output pins of the I/O module (Arduino or Fio)

Properties:

`average:Number` [read-only]
average of the pin values since the creation of the Pin object

`filters:Array`
get or set filters for the pin (see Funnel Filter Class Descriptions)

Example:
```
var osc:Osc = new Osc(Osc.SQUARE, 0.5, 1, 0, 0);
arduino.digitalPin(13).filters = [osc];
```

`lastValue:Number` [read-only]
returns the last set value

`maximum:Number` [read-only]
the maximum value set for the pin since the creation of the Pin object

`minimum:Number` [read-only]
the minimum value set for the pin since the creation of the Pin object

`number:uint` [read-only]
the port number for the pin

`preFilterValue:Number` [read-only]
the value of the Pin before the filter is applied

`type:uint` [read-only]
the port type (AIN, DIN, AOUT/PWM, DOUT, SERVO, I2C)

`value:Number`
the input value from a sensor or actuator

Public Methods:

`Pin``(number:uint, type:uint)`

Parameters:
`number:uint` [required] - port number (pin number on Arduino or Fio)
`type:uint` [required] – port type (AIN, DIN, AOUT/PWM, DOUT, SERVO, I2C)

Example:
```
config.setDigitalPinMode(9, PWM);
config.setDigitalPinMode(10, SERVO);

arduino = new Arduino(config);
…
var sensorPin:Pin = arduino.analogPin(1);
var led:Pin = arduino.digitalPin(13);
var pwmPin:Pin = arduino.digitalPin(9);
var servoPin:Pin = arduino.digitalPin(10);
```

`addFilter``(newFilter:*):void`

Parameters:
`newFilter:*` [required] - add a new filter to the Pin (see Funnel Filter Class Descriptions)

Example:
add a Scaler filter and a Convolution filter to analog Pin 1

```
var sensorPin:Pin = arduino.analogPin(1);
sensorPin.addFilter(new Scaler(0, 1, 0, 100));
sensorPin.addFilter(new Convolution(
                        Convolution.HPF));
```

`clear``():void`

reset the history

`removeAllFilters``():void`

removes all filters from the pin

`setFilters``(newFilters:Array):void`

sets new filters to the pin

Parameters:
`newFilters:Array` [required] - an array of filter objects to set to the pin (see Funnel Filter Class Descriptions)

Example:
create a Scaler filter and a Convolution filter and assign them to analog pin 1

```
var sensorPin:Pin = arduino.analogPin(1);
var scaler:Scaler = new Scaler(0, 1, 0, 100);
var movingAvg:Convolution = new
              Convolution(Convolution.MOVING_AVERAGE);

sensorPin.addFilters([scaler, movingAvg]);
```

Public Constants:

`AIN` Constant
`public static const AIN:uint = 0;`
analog input

`AOUT` Constant
`public static const AOUT:uint = 2;`
analog output and pwm output

`DIN` Constant
`public static const DIN:uint = 1;`
digital input

`DOUT` Constant
`public static const DOUT:uint = 3;`
digital output

`I2C` Constant
`public static const I2C:uint = 6;`
Pins for I2C

`PWM` Constant
`public static const PWM:uint = 2;`
pwm output and analog output

`SERVO` Constant
`public static const SERVO:uint = 4;`
Pins for servo

Events:

`Event.CHANGE : String = change`
dispatched when the pin value has changed

`Event.FALLING_EDGE : String = fallingEdge`
dispatched when the pin value is decreasing

`Event.RISING_EDGE : String = risingEdge`
dispatched when the pin value is increasing

SignalScope

```
import funnel.SignalScope;
```

This SignalScope class is used to visualize input signals. You can create one or more instances and assign them to various input signals.

Public Methods:

```
SignalScope(left:Number, top:Number, points:int,
            description:String, rangeMin:Number = 0,
            rangeMax:Number = 1)
```

Parameters:
`left:Number` [required] - the x position of the signal scope
`top:Number` [required] - the y position of the signal scope
`points:Number` [required] - the width of the graph (also corresponds to the number of points to display in the signal scope view)
`description:String` [required] - a text description that appears next to the signal scope object on the stage
`rangeMin` [optional] - the minimum range of the signal scope (default = 0). If assigning a value other than 0, you need to use the Scaler class to scale the input

appropriately.
`rangeMax` [optional] - the maximum range of the signal scope (default = 1). If assigning a value other than 1, you need to use the Scaler class to scale the input appropriately.

Example:

```
var sensorPin:Pin = arduino.analogPin(0);
sensorPin.addFilter(new Scaler(0, 1, 0, 100));

var scope:SignalScope;
scope = new SignalScope(10, 10, 200, "Analog Signal", 0, 100);

addEventListener(Event.ENTER_FRAME, loop);

function loop(e:Event):void {
   scope.update(arduino.analogPin(0));
}
```

update(input:*):void

update the graph with the input value

Parameters:
`input:*` [required] - the input to be plotted by the signal scope

Note: The use of the wildcard '' datatype above indicates that this variable is explicitly weakly typed.*

Funnel filter classes

Convolution
Scaler
SetPoint

Use the following statement to import all of the filter classes: `import funnel.*;`

You can alternatively import only the classes you are using in your program. Each class description below specifies which package to import.

Convolution

This class performs a convolution operation of the inputs. A low-pass filter is used to remove fine noise and a high pass filter is used to remove drift. Use this class to smooth out noisy input signals. See examples below.

```
import funnel.Convolution;
```

Properties:

`coef:Array`
an array of coefficients to be used with product-sum operations for input buffers. If assigned a new array, the input buffer will be cleared

Example:
assign a convolution filter and later change the type

```
var filter:Convolution = new Convolution(
                    Convolution.MOVING_AVERAGE));

// change the filter
filter.coef = Convolution.HPF;
```

Public Methods:

`Convolution``(kernel:Array)`

Parameters:
`kernel:Array` [required] - An array of coefficients to be used with

product-sum operations for input buffers. Pass in one of the 3 constants: Convolution.MOVING_AVERAGE, Convolution.HPF, or Convolution.LPF. Each constant defines the array to be used to perform the convolution.

Example:
add a Convolution filter to an analog pin to smooth out a noisy input signal

```
var sensorPin:Pin = arduino.analogPin(0);
sensorPin.addFilter(new Convolution(
                        Convolution.MOVING_AVERAGE));
```

Once the filter is set for the pin, when you read the value (`sensorPin.value`) you will get the filtered output. You can also return the pre-filtered value `sensorPin.preFilterValue`. To get the difference between the filtered and pre-filter values, simply subtract as follows:

```
var diff:Number = sensorPin.preFilterValue - sensorPin.value;
```

`processSample``(val:Number):Number`

Parameters:
`val:Number` [required] – the input value

Returns:
the resulting value after applying the adaptive filter

Public Constants:

`HPF` Constant
`public static const HPF:Array;`
High-pass filter kernel. Use by passing this array to the constructor.

`LPF` Constant
`public static const LPF:Array;`
Low-pass filter kernel. Use by passing this array to the constructor.

`MOVING_AVERAGE` Constant
`public static const MOVING_AVERAGE:Array;`
Moving average filter kernel. Use by passing this array to the constructor.

Example:

```
sensorPin.addFilter(new Convolution(
                           Convolution.HPF));
```

Scaler

This class scales an input value from its minimum and maximum range to a specified minimum and maximum range. Several functions are provided to map the input to the output value. See examples below.

```
import funnel.Scaler;
```

Properties:

`inMax:Number`
the maximum input value (from the sensor) you are scaling from

`inMax:Number`
the minimum input value (from the sensor) you are scaling from

`limiter:Boolean`
sets whether or not to restrict the input value if it exceeds the specified range

`outMax:Number`
the maximum output value you are scaling to

`outMin:Number`
the minimum output value you are scaling to

`type:Function`
the function used to map the input curve (**`Scaler.LINEAR`** is the default function)

Public Methods:

```
Scaler(inMin:Number = 0, inMax:Number = 1,
          outMin:Number = 0, outMax:Number = 1,
          type:Function = null, limiter:Boolean = true)
```

Parameters:
`inMin:Number` [optional] - the minimum input value from the sensor (default = 0)
`inMax:Number` [optional] – the maximum input value from the sensor (default = 1)
`outMin:Number` [optional] – the minimum output value you are scaling to (default = 0)
`outMax:Number` [optional] – the maximum output value you are scaling to (default = 1)
`type:Function` [optional] – the function used to map the input curve (default = `Scaler.LINEAR`)

`limiter:Boolean` [optional] – whether or not to restrict the input value if it exceeds the specified range (default = true)

Example:
add a Scaler filter to an analog pin to scale the input from a range of 0.3 to 0.7 to a range of -1 to 1

```
var sensorPin:Pin = arduino.analogPin(0);
sensorPin.addFilter(new Scaler(0.3, 0.7, -1, 1));
```

CUBE`(val:Number):Number`

the input curve is mapped by: y = x^4

Parameters:
`val:Number` [required] – the input value

Returns:
y = x^4 *where y is the return value and x is the parameter, val*

Example:
```
sensorPin.addFilter(new Scaler(0.3, 0.7, -1, 1, Scaler.CUBE));
```

CUBE_ROOT`(val:Number):Number`

the input curve is mapped by: y = pow(x, ¼)

Parameters:
`val:Number` [required] – the input value

Returns:
y = pow(x, ¼) *where y is the return value and x is the parameter, val*

Example:
```
sensorPin.addFilter(new Scaler(0.3, 0.7, -1, 1,
                               Scaler.CUBE_ROOT));
```

LINEAR`(val:Number):Number`

the input curve is mapped by: y = x

Parameters:
`val:Number` [required] – the input value

Returns:
y = x *where y is the return value and x is the parameter, val*

processSample`(val:Number):Number`

Parameters:
`val:Number` [required] – the input value

Returns:
the resulting value after applying the adaptive filter

SQUARE`(val:Number):Number`

the input curve is mapped by: y = x^2

Parameters:
`val:Number` [required] – the input value

Returns:
y = x^2 *where y is the return value and x is the parameter, val*

Example:
```
sensorPin.addFilter(new Scaler(0.3, 0.7, -1, 1, Scaler.SQUARE)
                              );
```

SQUARE_ROOT`(val:Number):Number`

the input curve is mapped by: y = sqrt(x)

Parameters:
`val:Number` [required] – the input value

Returns:
y = sqrt(x) *where y is the return value and x is the parameter, val*

SetPoint

When setting a single point, this class divides an input to 0 or 1 based on the threshold and hysteresis (tolerance for the threshold value - so thresh +/- hysteresis value). You can also set multiple points by providing a nested array such as [[0.4, 0.1], [0.7, 0.05]]. When you set multiple points, the lowest value will divide the input between 0 and 1, while each higher point adds 1 when the threshold is triggered. For example if you set two points: [[0.4, 0],[0.7, 0]], as the input value increases from 0, the pin output value (pinInstance.value) will read 0 until the input value crosses 0.4 at which point the output will be set to 1, then when the input value crosses 0.7 the output will be set to 2. Use this method to trigger an action when a certain threshold is crossed. See examples below.

```
import funnel.SetPoint;
```

Public Methods:

```
SetPoint(points:Array)
```

Parameters:

`points:Array` [optional] - An array of a threshold, hysteresis pair. The threshold is the point at which the output will be divided to 0 or 1. Hysteresis is the allowance for the threshold. The default array is [0.5, 0] so any value < 0.5 will be set to 0 and any value > 0.5 will be set to 1; If the hysteresis is set to 0.1, then any value < 0.4 would be set to 0 and any value > 0.6 would be set to 1

Example:

Set the threshold to 0.4 and hysteresis to 0.05. Any value < 0.35 (0.4 – 0.05) will be set to 0 and any value > 0.45 will be set to 1.

```
var sensorPin:Pin = arduino.analogPin(0);
var threshold:Number = 0.4;
var hysteresis:Number = 0.05;
sensorPin.addFilter(new SetPoint([threshold, hysteresis]));
sensorPin.addEventListener(PinEvent.RISING_EDGE, onHigh);
sensorPin.addEventListener(PinEvent.FALLING_EDGE, onLow);
```

if the sensor input value is decreasing and becomes < 0.35 (0.4 - 0.05), then onLow will be called. if the sensor input value is increasing and becomes > 0.45, then onHigh will be called.

You can also add multiple points as follows, which is a convenient way to map actions to multiple points on an input signal:

```
// this syntax is a nested array [[a, b],[c, d],[e, f]]
sensorPin.addFilter(new SetPoint([[0.25, 0],[0.5, 0],
                                    [0.75, 0]]));

sensorPin.addEventListener(PinEvent.CHANGE, onPinChange);

function onPinChange(e:PinEvent):void {
   // will the sensor value is increasing, the value will
   // change from 0 to one when 0.25 is cross, then it will
   // change from 1 to 2 when 0.5 is crossed and from 2 to
   // 3 when 0.75 is crossed. The opposite happens when the
   // sensor value is descreasing
   trace(e.target.value);
}
```

`addPoint``(threshold:Number, hysteresis:Number = 0):void`

Parameters:
`threshold:Number` [required] – the threshold to divide an input to 0 or 1
`hysteresis:Number` [optional] – the allowance for the threshold (default = 0)

Example:
```
var filter:SetPoint = new SetPoint(); // use default values
filter.addPoint(0.7, 0.1); // set threshold and hysteresis
```

`processSample``(val:Number):Number`

Parameters:
`val:Number` [required] – the input value

Returns:
the resulting value after applying the adaptive filter

`removeAllPoints``():void`

removes all points that had been set

`removePoint``(threshold:Number):void`

removes a point set at the specified threshold

Parameters:
`threshold:Number` [required] – the threshold previously set that you now want to remove

Example:
```
var filter:SetPoint = new SetPoint(); // use default values
filter.addPoint(0.7, 0.1); // set threshold and hysteresis
```

later…
```
filter.removePoint(0.7); // removes the 0.7 threshold point
```

Funnel UI classes

Accelerometer
Button
DCMotor
LED
Potentiometer
RGBLED
Servo
SoftPot

Use the following statement to import all of the UI classes: `import funnel.ui.*;`

You can alternativly import only the UI classes you are using in your program. Each class description below specifies which package to import. Some UI classes such as Button and SoftPot require importing both the UI class and the coresponding event class.

Accelerometer

```
import funnel.ui.Accelerometer;
```

Properties:

`rotationX:Number` [read-only]
returns the rotation in degrees around the x axis

`rotationY:Number` [read-only]
returns the rotation in degrees around the y axis

`rotationZ:Number` [read-only]
returns the rotation in degrees around the z axis

`x:Number` [read-only]
the acceleration on the x axis either in the range of -1.0 to 1.0 or in units of gravity (Gs)

`y:Number` [read-only]
the acceleration on the x axis either in the range of -1.0 to 1.0 or in units of gravity (Gs)

`z:Number` [read-only]

the acceleration on the x axis either in the range of -1.0 to 1.0 or in units of gravity (Gs)

`dynamicRange:Number` [read-only]
returns the range of the sensor. If the value is 1.0, the range of the x, y, and z values will be from -1.0 to 1.0 instead of the actual range in units of gravity. If the range in units of gravity was set by the 4th parameter of the `setRangeFor` method, then `dynamicRange` returns the range in units of gravity (Gs)

Public Methods:

```
Accelerometer(xPin:Pin, yPin:Pin, zPin:Pin,
                    smoothing:Boolean = true)
```

Parameters:
`xPin:Pin` [required] - the funnel Pin object corresponding to the sensor x axis
`yPin:Pin` [required] - the funnel Pin object corresponding to the sensor y axis
`zPin:Pin` [required] - the funnel Pin object corresponding to the sensor z axis
`smoothing:Boolean` [optional] – set whether or not smoothing is enabled for the sensor values (default value is true)

Throws:
```
(newArgumentError("At - least one axis should NOT be null"))
```

Example:
```
var xPin:Pin = arduino.analogPin(0);
var yPin:Pin = arduino.analogPin(1);
var zPin:Pin = arduino.analogPin(2);
var accel:Accelerometer = new Accelerometer(xPin, yPin, zPin,
3);
```
Note: for the ADXL335 accelerometer, the 4th parameter is set to 3 because the dynamic range of the accelerometer is -3 to 3 Gs

```
setRangeFor(axis:uint, minimum:Number, maximum:Number,
                    range:Number = 1):void
```

Parameters:
`axis:uint` [required] – the axis (`X_AXIS`, `Y_AXIS` or `Z_AXIS`) to set the range for
`minimum:Number` [required] the minimum value returned by the sensor for the specified axis
`maximum:Number` [required] the maximum value returned by the sensor for the specified axis
`range:Number = 1` [optional] the dynamic range of the accelerometer (get the value from the accelerometer data sheet), the default value is 1 which will return a range from -1 to 1 instead of the true range in units of Gravity (Gs)

Public Constants:

X_AXIS Constant
Public static const X_AXIS:uint = 0;

Y_AXIS Constant
Public static const Y_AXIS:uint = 0;

Z_AXIS Constant
Public static const Z_AXIS:uint = 0;

Example:
```
accel.setRangeFor(Accelerometer.X_AXIS, 0.2, 0.8, 3);
```

Events:

```
Event.CHANGE
```

Example:
```
accel.addEventListener(Event.CHANGE, onAccelValueChange);

function onAccelValueChange(evt:Event):void {
     trace("x value = " + accel.x);
     trace("y value = " + accel.y);
     trace("z value = " + accel.z);
}
```

Button

```
import funnel.ui.Button;
import funnel.ui.ButtonEvent;
```

Properties:

debounceInterval:int
get or set the interval in milliseconds between two digital input reads. This is to ensure that false reads are not triggered during button press due to mechanical error. The default value is 20, but I've found that 100 is more reliable.

Public Methods:

```
Button(buttonPin:Pin, buttonMode:uint,
                 longPressDelay:Number = 1000)
```

Parameters:
`buttonPin:Pin` [required] - the funnel Pin object corresponding to the connected button

`buttonMode:uint` [required] - the button mode (PULL_DOWN or PULL_UP)
`longPressDelay:Number` [optional] - the time in milliseconds when a long press delay event will fire when the button is continuously depressed.

Throws:
```
(newArgumentError("buttonMode - should be PULL_DOWN or
PULL_UP"))
```

Example:
```
var longPress:uint = 2000;
var button:Button = new Button(arduino.digitalPin(2),
Button.PULL_DOWN, longPress);
```

Note: PULL_UP vs PULL_DOWN. If the other end of the resistor connected to the button is connected to ground, configuration is PULL_DOWN, if the resistor is connected to power, then the configuration is PULL_UP.

Public Constants:

`PULL_DOWN` Constant
```
Public static const PULL_DOWN:uint = 0;
```

`PULL_UP` Constant
```
Public static const PULL_UP:uint = 0;
```

Events:

```
ButtonEvent.PRESS : String = press
ButtonEvent.RELEASE : String = release
ButtonEvent.LONG_PRESS : String = longPress
ButtonEvent.SUSTAINED_PRESS : String = sustainedPress
```

Example:
```
btn.addEventListener(ButtonEvent.PRESS, onBtnPress);

function onBtnPress(evt:ButtonEvent):void {
     trace("button pressed"};
}
```

DCMotor

```
import funnel.ui.DCMotor;
```

A class for controlling an H-bridge. This class is compatible with the following H-bridge ICs:

TA7291P
SN754410
TB6612FNG
DB6211F

Properties:

`value:Number`
get or set the current value corresponding the the motor state. 0 is brake, < 0 is value of reverse motion, > 0 is value of forward motion.

Public Methods:

```
DCMotor(forwardPin:Pin, reversePin:Pin, pwmPin:Pin,
                minimumVoltage:Number = 1,
                maximumVoltage:Number = 9,
                supplyVoltage:Number = 9)
```

Some H-bridges such as the TA7291P do not require the 3rd parameter (pwmPin) so it should be set to `null`

Note that the forward, reverse, and pwm (if supported) pins of the H-bridge must be connected to PWM pins on the Arduino or Fio board. When using a PWM pin you must set the configuration of the pin as follows:

```
config.setDigitalPinMode(9, PWM); // pin 9 set to pwm
config.setDigitalPinMode(10, PWM); // pin 10 set to pwm
config.setDigitalPinMode(11, PWM); // pin 11 set to pwm
```

Parameters:
`forwardPin:Pin` [required] - the funnel Pin object corresponding to the connected forward pin of the H-bridge (use PWM pin
`reversePin:Pin` [required] - the funnel Pin object corresponding to the connected reverse pin of the H-bridge
`pwmPin:Pin` [required] - the funnel Pin object corresponding to the connected pwm pin of the H-bridge (set to `null` if the H-bridge doesn't support this feature)
`minimumVoltage:Number` [optional] - the minimum voltage (default = 1)
`maximumVoltage:Number` [optional] - the maximum voltage (default = 9)
`supplyVoltage:Number` [optional] - the supply voltage (default = 9)

Example:

```
var fwdPin:Pin = arduino.digitalPin(9);
var revPin:Pin = arduino.digitalPin(10);
var pwmPin:Pin = arduino.digitalPin(11);

var motor:DCMotor;
motor = new DCMotor(fwdPin, revPin, pwmPin);
```

despin`(useBrake:Boolean = true):void`

Parameters:
`useBrake:Boolean` [optional] - specify whether or not to use the brake (default is true)

forward`(val:Number = 1):void`

Parameters:
`val:Number` [optional] - the new voltage to set in the range of 0 to 1.0 (default is 1)

reverse`(val:Number = 1):void`

Parameters:
`val:Number` [optional] - the new voltage to set in the range of 0 to 1.0 (default is 1)

LED

```
import funnel.ui.LED;
```

Note that many of the methods defined in the LED class will only work if the LED is connected to a PWM pin on the Arduino or Fio. When using a PWM pin you must set the configuration of the pin as follows:
`config.setDigitalPinMode(9, PWM); // pin 9 set to pwm`

Properties:

`intensity:Number`
same as value

`value:Number`
get or set the brightness of the led ranging from 0 to 1.0 where 0 is all the way off and 1 is all the way on

Public Methods:

LED`(ledPin:Pin, driveMode:uint)`

Parameters:
`ledPin:Pin` [required] - the funnel Pin object corresponding to the connected LED
`driveMode:uint` [optional] - the LED drive mode (`SOURCE_DRIVE` or

`SYNC_DRIVE`). Default value is `SOURCE_DRIVE`.

Note: SOURCE_DRIVE vs SYNC_DRIVE. If the Anode (longer LED pin) is connected to the microcontroller pin, then it is SOURCE_DRIVE. If the Cathode is connected to the microcontroller pin, then it is SYNC_DRIVE.

Throws:
```
(newArgumentError("driveMode – should be SOURCE_DRIVE or
SYNC_DRIVE"))
```

Example:
```
var led:LED = new LED(arduino.digitalPin(9));
```

blink(interval:Number, times:Number = 1, wave:Function = null):void

Parameters:
`interval:Number` [required] - the time interval between blinks (or oscillations)
`times:Number` [optional] - set the number of times the LED will blink (default is 1)
`wave:Function` [optional] – set the waveform used to control the blink (default is `Osc.SQUARE`). See the `Osc` class description for more detail.

Note: In order to use wave functions other than Osc.SQUARE, the LED must be connected to a PWM pin on the Arduino or Fio board.

fadeIn(time:Number = 1000):void

Parameters:
`time:Number` [optional] - the time in milliseconds that it takes to fade in (default is 1000 ms)

Note: In order to use this method, the LED must be connected to a PWM pin on the Arduino or Fio board.

fadeOut(time:Number = 1000):void

Parameters:
`time:Number` [optional] - the time in milliseconds that it takes to fade out (default is 1000 ms)

Note: In order to use this method, the LED must be connected to a PWM pin on the Arduino or Fio board.

fadeTo(to:Number, time:Number = 1000):void

Parameters:

`to:Number` [required] - he new intensity to fade to
`time:Number` [optional] - the time in milliseconds that it takes to fade to the new intensity (default is 1000 ms)

Note: In order to use this method, the LED must be connected to a PWM pin on the Arduino or Fio board.

isOn`():Boolean`

Returns:
`true` if the LED is on, `false` if the LED is off

off`():void`

turns the LED off

on`():void`

turns the LED on

stopBlinking`():void`

If the LED is blinking (the `blink` method was called) it will stop blinking

toggle`():void`

toggles the LED value, so if it is off and you call this method it will turn the LED on and vice versa.

Public Constants:

`SOURCE_DRIVE` Constant
`Public static const SOURCE_DRIVE:uint = 0;`

`SYNC_DRIVE` Constant
`Public static const SYNC_DRIVE:uint = 0;`

Potentiometer

```
import funnel.ui.Potentiometer;
```

Properties:

`value:Number` [read-only]
returns the value of the potentiometer (ranging from 0.0 to 1.0)

Public Methods:

`Potentiometer``(potPin:Pin)`

Parameters:
`potPin:Pin` [required] - the funnel Pin object corresponding to the connected potentiometer

Example:
```
var pot:Potentiometer;
pot = new Potentiometer(arduino.analogPin(0));
```

`setRange``(minimum:Number, maximum:Number):void`

Parameters:
`minimum:Number` [required] - the minimum value returned by the potentiometer (to be used in the case that the minimum value returned by the potentiometer is not 0 but you want to scale it to 0)
`maximum:Number` [required] - the maximum value returned by the potentiometer (to be used in the case that the maximum value returned by the potentiometer is not 1.0 but you want to scale it to 1.0)

For example if you connect a potentiometer and trace it's output (`trace(pot.value)`) and turn the pot to each extreme and the lowest and highest values are something like 0.15 and 0.94, the setRange will scale the values from 0.0 to 1.0.

Events:

```
Event.CHANGE
```

Example:
```
pot.addEventListener(Event.CHANGE, onPotValueChange);

function onPotValueChange(evt:Event):void {
	trace("pot value = " + evt.target.value);
}
```

RGBLED

```
import funnel.ui.RGBLED;
```

Note that the RGB pins of the led must be connected to PWM pins on the Arduino or Fio board. When using a PWM pin you must set the configuration of the pin as follows:

```
config.setDigitalPinMode(9, PWM); // pin 9 (red) set to pwm
config.setDigitalPinMode(10, PWM); // pin 10 (green) set to pwm
config.setDigitalPinMode(11, PWM); // pin 11 (blue) set to pwm
```

Public Methods:

```
RGBLED(redLEDPin:Pin, greenLEDPin:Pin, blueLEDPin:Pin,
                driveMode:uint)
```

Parameters:
`redLEDPin:Pin` [required] - the funnel Pin object corresponding to the connected red LED pin (normally the pin closest to the flat side of the LED lens)
`greenLEDPin:Pin` [required] - the funnel Pin object corresponding to the connected green LED pin
`blueLEDPin:Pin` [required] - the funnel Pin object corresponding to the connected blue LED pin
`driveMode:uint` [optional] - the drive mode (`COMMON_ANODE` or `COMMON_CATHODE`). Default value is `COMMON_ANODE`.

Note: COMMON_ANODE vs COMMON_CATHODE. You can determine if your RGB LED is common anode or common cathode by reading the datasheet. The RGB LED in the exercises is common cathode. The cathode is connected to ground and the 3 anode pins are connected to the Arduino PWM pins via 330 ohm resistors. For a common anode LED, the anode is connected to power and the 3 cathode pins are connected to the Arduino PWM pins via resistors.

Example:

```
var red:Pin = arduino.digitalPin(9);
var green:Pin = arduino.digitalPin(10);
var blue:Pin = arduino.digitalPin(11);
var rgbLed:RGBLED;
rgbLed = new RGBLED(red, green, blue, RGBLED.COMMON_CATHODE);
```

```
fadeIn(time:Number = 1000):void
```

Parameters:
`time:Number` [optional] - the time in milliseconds that it takes to fade in (default is 1000 ms)

`fadeOut(time:Number = 1000):void`

Parameters:
`time:Number` [optional] - the time in milliseconds that it takes to fade out (default is 1000 ms)

`fadeTo(red:Number, green:Number, blue:Number, time:Number = 1):void`

fades from current color to new color specified in this method call

Parameters:
`red:Number` [required] - the red value (in range of 0 to 1)
`green:Number` [required] - the green value (in range of 0 to 1)
`blue:Number` [required] - the blue value (in range of 0 to 1)
`time:Number` [optional] - the time in milliseconds that it takes to fade to the new intensity (default is 1000 ms)

Note: There is a bug in this method in which the default time is set to 1 millisecond instead of 1000 ms, so you must always specify the time. Will be fixed in next release.

`setColor(red:Number, green:Number, blue:Number, time:Number = 1):void`

sets the color of the LED

Parameters:
`red:Number` [required] - the red value (in range of 0 to 1)
`green:Number` [required] - the green value (in range of 0 to 1)
`blue:Number` [required] - the blue value (in range of 0 to 1)

Note: to specify the color using more familiar values in the range of 0-255, divide the color value by 255. For example setColor(114/255, 80/255, 255/255), or setColor(0xCC/255, 0x33/255, 0xFF/255) .

Public Constants:

`COMMON_ANODE` Constant
`Public static const COMMON_ANODE:uint = 1;`

`COMMON_CATHODE` Constant
`Public static const COMMON_CATHODE:uint = 0;`

Servo

```
import funnel.ui.Servo;
```

Note that the servo must be connected to PWM pins on the Arduino or Fio board. When using a servo you must set the configuration of the pin as follows:
`config.setDigitalPinMode(9, SERVO); // pin 9 set to pwm`

Properties:

`angle:Number`
get or set the value the angle of the servo head. The typical range is from 0 to 180 degrees.

Public Methods:

`Servo``(servoPin:Pin, minAngle:int = 0, maxAngle:int = 180)`

Parameters:
`servoPin:Pin` [required] - the funnel Pin object corresponding to the connected servo
`minAngle` [optional] - the minimum angle of the servo head rotation (default is 0 degrees)
`maxAngle` [optional] - the maximum angle of the servo head rotation (default is 180 degrees)

Example:
```
var servo:Servo;
servo = new Servo(arduino.digitalPin(9));
```

Throws:
`ArgumentError`

SoftPot

```
import funnel.ui.SoftPot;
import funnel.ui.SoftPotEvent;
```

Properties:

`distanceFromPressed:Number` [read-only]
while dragging this property indicates the distance your finger (or other object) is from where it originally triggered the press event that began the drag event

`isPressed:Boolean` [read-only]
weather or not the softpot is currently pressed

`value:Number` [read-only]
the current value corresponding to the current touch point (where the finger or other object is on the softpot)

Public Methods:

`SoftPot``(potPin:Pin, softPotLength:Number = 100)`

Parameters:
`potPin:Pin` [required] - the funnel Pin object corresponding to the connected softpot sensor
`softPotLength:Number` [optional] - the length of the sensor in millimeters (common lengths are 50mm, 100mm, and 210mm). Specify the length to obtain more accurate results.

Example:
```
var sensorLength = 50; // 50 mm softpot
var softPot:SoftPot;
softPot = new SoftPot(arduino.analogPin(0), sensorLength);
```

`setMinFlickMovement``(num:Number):void`

use this method to fine tune the flick behavior

Parameters:
`num:Number` [required] - the minimum distance an object must travel along the softpot surface within 200 milliseconds in order to generate a flick event. The default value is set by the following equation:
minFlickMovement = 1.0/softPotLength * 5.0;

`setRange``(minimum:Number, maximum:Number):void`

Parameters:
`minimum:Number` [required] - the minimum value returned by the softpot (to be used in the case that the minimum value returned by the softpot is not 0 but you want to scale it to 0)
`maximum:Number` [required] - the maximum value returned by the softpot (to be used in the case that the maximum value returned by the softpot is not 1.0 but you want to scale it to 1.0)

For example if you connect a softpot and trace it's output (`trace(softPot.value)`) and the lowest and highest values at each end of the sensor are something like 0.15 and 0.94, the setRange will scale the values from 0.0 to 1.0.

Events:

```
SoftPotEvent.PRESS : String = press
SoftPotEvent.RELEASE : String = release
SoftPotEvent.TAP : String = tap
SoftPotEvent.DRAG : String = drag
SoftPotEvent.FLICK_DOWN : String = flickDown
SoftPotEvent.FLICK_UP : String = flickUp
```

Example:

```
softPot.addEventListener(SoftPot.FLICK_UP, onFlick);
softPot.addEventListener(SoftPot.FLICK_DOWN, onFlick);

function onFlick(evt:SoftPotEvent):void {
   switch (evt.type) {
      case "flickUp":
            trace("got flick up");
            break;
      case "flickDown":
            trace("got flick down");
            break;
   }
}
```

Funnel I2C classes

ADXL345
BlinkM
HMC6352
GyroITG3200

Use the following statement to import all of the UI classes: `import funnel.i2c.*;`

You can alternatively import only the I2C classes you are using in your program. Each class description below specifies which package to import. Some I2C classes such as GyroITG3200 and require importing both the I2C class and the corresponding event class.

ADXL345

`import funnel.i2c.ADXL345;`

This class enables the use of the ADXL345 triple axis accelerometer in i2c mode. You can select between +/- 2G, +/-4G, +/- 8G, and +/- 16G measurement ranges. Acceleration values are returned both in units of gravity and as raw acceleration values from the sensor.

Please note: this sensor is rated for 3.3v or less and should not be directly interfaced with a 5v Arduino without regulating the voltage appropriately on the digital input and output pins. See this link for details: http://www.sparkfun.com/tutorials/65

Sparkfun sells and easy to use breakout board for this sensor: http://www.sparkfun.com/products/9836

Properties:

`dynamicRange:Number` [read-only]
returns the dynamic range of the accelerometer in Gs (either +/-2G, +/-4G, +/-8G, or +/-16G)

`isRunning:Boolean` [read-only]
returns true if read continuous mode is enabled, false if it is not

`rawX:Number` [read-only]
returns the raw x acceleration value from the sensor

`rawY:Number` [read-only]
returns the raw y acceleration value from the sensor

`rawZ:Number` [read-only]
returns the raw z acceleration value from the sensor

`sensitivityX:Number`
`sensitivityY:Number`
`sensitivityZ:Number`
the sensitivity value is multiplied by the raw acceleration value to obtain the acceleration value in units of Gravity. The default value is `ADXL345.DEFAULT_SENSITIVITY` and generally should not need to be changed.

`x:Number` [read-only]
the acceleration on the x axis units of gravity (Gs)

`y:Number` [read-only]

the acceleration on the x axis units of gravity (Gs)

`z:Number` [read-only]
the acceleration on the x axis units of gravity (Gs)

`dynamicRange:Number` [read-only]
returns the range of the sensor. If the value is 1.0, the range of the x, y, and z values will be from -1.0 to 1.0 instead of the actual range in units of gravity. If the range in units of gravity was set by the 4th parameter of the **`setRangeFor`** method, then **`dynamicRange`** returns the range in units of gravity (Gs)

Public Methods:

```
ADXL345(ioModule:*, autoStart:Boolean = false,
                    address:uint, range:Number)
```

Parameters:
`ioModule:*` [required] - a reference to the Arduino or Fio I/O module
`autoStart:Boolean` [optional] - true if read continuous mode should start automatically upon instantiation (default = false)
`address:uint` [optional] - the i2c address of the sensor
(default = `ADXL345.DEVICE_ID`)
`range:Number` [optional] – the dynamic range selection in Gs
(default = `ADXL345.RANGE_2G`)

Example:
create an instance of the ADXL345 i2c accelerometer, set autoStart to false and set the dynamic range to +/- 4G

```
var accel:ADXL345;
accel = new ADXL345(fio.ioModule(1), false, ADXL345.DEVICE_ID,
ADXL345.RANGE_4G);
accel.addEventListener(Event.CHANGE, onAccelUpdate);

accel.startReading();    // enable read continuous mode

function onAccelUpdate(e:Event):void {
   trace("x = " + e.currentTarget.x);
   trace("y = " + e.currentTarget.y);
   trace("z = " + e.currentTarget.z);
}
```

`getAxisOffset``():Object`

Returns:
An object containing the x, y, and z offset values

Example:

```
var offsetVals:Object = getAxisOffset();
var offsetX:Number = offsetVals.x;
var offsetY:Number = offsetVals.y;
var offsetZ:Number = offsetVals.z;
```

`setAxisOffset(xVal:int, yVal:int, zVal:int):void`

use this method to calibrate the accelerometer by passing +/- values. The scale factor is 15.6 mg/LSB, so setAxisOffsit(-2, -2, 4) offsets the x and y axis by -.0312 Gs and offsets the z axis by .0625 Gs.

Parameters:

`xVal:int` [required] - the amount to add or subtract from the x axis
`yVal:int` [required] - the amount to add or subtract from the y axis
`zVal:int` [required] - the amount to add or subtract from the z axis

Example:

offset the x, y, and z values so that in each orientation the values add up to approximately +1 or -1

```
accel.setAxisOffset(-2, -2, 4);
```

To find the appropriate values, hold the accelerometer in each of the orientations depicted below and note the outputs for each axis per the orientation. Trace the accelerometer outputs and note the difference between the actual acceleration data and the target data in the image below. Divide the difference by 0.0156. The result is the amount to offset the axis by (either positive or negative)

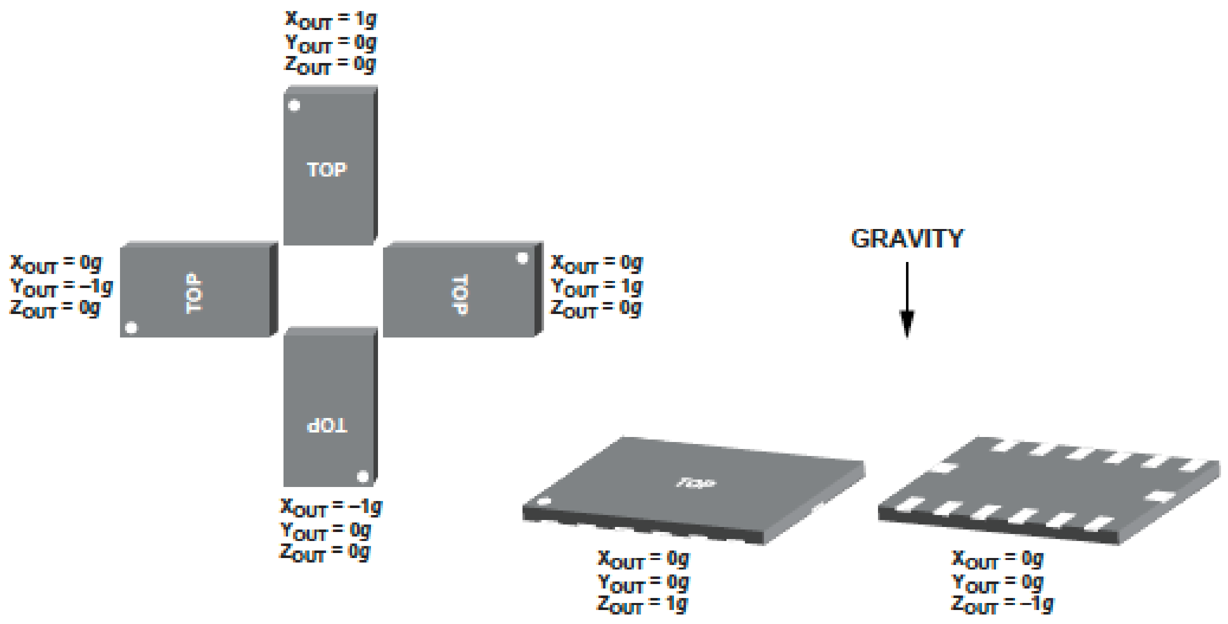

Figure 17. Output Response vs. Orientation to Gravity

`startReading():void`

start continuous reading of the accelerometer. In this mode, you should not call the update() method, because the code running on the Arduino or Fio board is automatically returning the accelerometer data every 33 milliseconds (or at the rate set by `samplingInterval` if it was changes). The CHANGE event will be dispatched each time a new reading is received.

`stopReading():void`

stops continuous reading of the accelerometer. To get data from the accelerometer after calling this method, you must either call the update() method (each call to update will only return one set of acceleration values), or call startReading() to begin read continuous mode.

`update():void`

sends a read request to the accelerometer and updates the x, y, and z acceleration values. If you call update while in read continuous mode, read continuous mode will stop (stopReading() will be called in the background) and you will need to continue to call update on a regular basis to return the latest values. You can also call startReading() to enter continuous read mode.

Public Constants:

`DEFAULT_SENSITIVITY` Constant
```
public static const DEFAULT_SENSITIVITY:Number = 0.00390625;
```

`DEVICE_ID` Constant
```
public static const DEVICE_ID:uint = 0x53;
```

`RANGE_16G` Constant
```
public static const RANGE_16G:Number = 16;
```

`RANGE_2G` Constant
```
public static const RANGE_16G:Number = 2;
```

`RANGE_4G` Constant
```
public static const RANGE_16G:Number = 4;
```

`RANGE_8G` Constant
```
public static const RANGE_16G:Number = 8;
```

Events:

`Event.CHANGE : String = change`
dispatched when a new set of acceleration values are received

Example:
```
accel.addEventListener(Event.CHANGE, onAccelUpdate);

function onAccelUpdate(e:Event):void {
   trace("x = " + e.currentTarget.x);
   trace("y = " + e.currentTarget.y);
   trace("z = " + e.currentTarget.z);
}
```

BlinkM

```
import funnel.i2c.BlinkM;
```

This class enables the use of the BlinkM rgbLED module.

Please refer to ThingM for documentation: http://thingm.com/products/blinkm

Public Methods:

`BlinkM``(ioModule:*, address:uint = 0x09)`

Parameters:
`ioModule:*` [required] - a reference to the Arduino or Fio I/O module
`address:uint` [optional] - the i2c address of the sensor
(default = `0x09`)

Example:
```
var blinkM:BlinkM;
blinkM = new BlinkM(arduino);
blinkM.stopScript(); // stop the default script

var red:uint = 255;
var green:uint = 128;
var blue:uint = 64;
// pass RGB color values as an Array
blinkM.goToRGBColorNow([red, green, blue]);
```

`fadeToHSBColor``(color:Array, speed:int = -1):void`

Fade from the current color to the specified HSB (Hue Saturation Brightness) color value.

Parameters:
`color:Array` [required] - an 3 element array containing the H, S, B color values
`speed:uint` [optional] - the time it takes to complete the fade
(default = 15 ticks which is about `500 ms`). Supply speed as number of ticks where 1 tick = 33 milliseconds. So 15 ticks = 495 ms.

Example:
```
var hue:uint = 255;
var saturation:uint = 128;
var brightness:uint = 64;
// fade in 1 second (30 * 33 is about 1000 ms)
blinkM.fadeToHSBColor([hue, saturation, brightness], 30);
```

`fadeToRandomHSBColor``(color:Array, speed:int = -1):void`

Fade from the current color to a random HSB color. Pass an array containing the range for each color. A range of 0 means no change for the color value; a range of 255 means any possible value of that color.

Parameters:
`color:Array` [required] - an 3 element the range for each color

`speed:uint` [optional] - the time it takes to complete the fade (default = 15 ticks which is about `500 ms`). Supply speed as number of ticks where 1 tick = 33 milliseconds. So 15 ticks = 495 ms.

Example:

```
var hRange:uint = 255; // any red value between 0 and 255
var sRange:uint = 0;   // do not change current green value
var bRange:uint = 255; // any blue value between 0 and 255
// fade in 1 second (30 * 33 is about 1000 ms)
blinkM.fadeToRandomHSBColor([hRange, sRange, bRange], 30);
```

fadeToRandomRGBColor(color:Array, speed:int = -1):void

Fade from the current color to a random RGB color. Pass an array containing the range for each color. A range of 0 means no change for the color value, a range of 255 means any possible value of that color.

Parameters:
`color:Array` [required] - an 3 element the range for each color
`speed:uint` [optional] - the time it takes to complete the fade (default = 15 ticks which is about `500 ms`). Supply speed as number of ticks where 1 tick = 33 milliseconds. So 15 ticks = 495 ms.

Example:

```
var rRange:uint = 255; // any red value between 0 and 255
var gRange:uint = 0;   // do not change current green value
var bRange:uint = 255; // any blue value between 0 and 255
// fade in 1 second (30 * 33 is about 1000 ms)
blinkM.fadeToRandomRGBColor([rRange, gRange, bRange], 30);
```

fadeToRGBColor(color:Array, speed:int = -1):void

Fade from the current color to the specified RGB color value.

Parameters:
`color:Array` [required] - an 3 element array containing the R, G, B color values
`speed:uint` [optional] - the time it takes to complete the fade (default = 15 ticks which is about `500 ms`). Supply speed as number of ticks where 1 tick = 33 milliseconds. So 15 ticks = 495 ms.

Example:

```
var red:uint = 255;
var green:uint = 128;
var blue:uint = 64;
// fade in 0.5 second (15 * 33 is about 500 ms)
blinkM.fadeToRGBColor([red, green, blue], 15);
```

goToRGBColorNow(color:Array):void

Immediately change to the specified RGB color.

Parameters:
`color:Array` [required] - an 3 element array containing the R, G, B color values

Example:
```
var red:uint = 255;
var green:uint = 0;
var blue:uint = 0;
// immediately turn LED red
blinkM.goToRGBColorNow([red, green, blue]);
```

playLightScript(scripted:uint, theNumberOfRepeats:uint = 1, lineNumber:uint = 0):void

The blinkM module is preloaded with 19 light scripts. New scripts can also be loaded via the blinkM sequencer application that can be downloaded from the ThingM website: http://thingm.com/products/blinkm. See the blinkM datasheet for descriptions of the light scripts.

Parameters:
`scriptId:uint` [required] - the ID of the script (0 – 18)
`theNumberOfRepeats:uint` [optional] - the number of times to play the script (default = 1)
`lineNumber:uint` [optional] - the script line number to start playing from. A value of 0 will play the script forever (default = 0)

Example:
play the virtual candle (random yellows) script 5 times

```
blinkM.playLightScript(12, 5);
```

setFadeSpeed(speed:uint):void

set the rate at which color fading happens

Parameters:
`speed:uint` [required] - the rate at which color fading happens. Values are between 1 and 255 where 1 is slowest and 255 is fastest (or instant fade).

stopScript():void

stop the script (if one is playing). You should always call this immediately after creating an instance of a new blinkM module or the default script will play upon power up**.**

HMC6352

```
import funnel.i2c.HMC6352;
```

This class enables the use of the HMC6352 digital compass module. It simply returns the current heading in degrees (0 to 360) each time the CHANGE event is dispatched. You must hold the sensor level (just like you would with an analog compass) in order to get the most accurate reading.

Sparkfun sells and easy to use breakout board for this sensor: http://www.sparkfun.com/products/7915

Properties:

`heading:Number` [read-only]
the compass heading in degrees (0 to 360)

Public Methods:

`HMC6352``(ioModule:*, address:uint)`

Parameters:
`ioModule:*` [required] - a reference to the Arduino or Fio I/O module
`address:uint` [optional] - the i2c address of the sensor
(default = `HMC6352.DEVICE_ID`)

Example:
```
var compass:HMC6352;
compass = new HMC6352(fio.ioModule(1));
compass.addEventListener(Event.CHANGE, onHeadingChange);

function onHeadingChange(e:Event):void {
   trace("heading = " + e.target.heading);
}
```

Public Constants:

`DEVICE_ID` Constant
`public static const DEVICE_ID:uint = 0x21;`

Events:

`Event.CHANGE : String = change`
dispatched when a new heading value is received

Example:
```
compass.addEventListener(Event.CHANGE, onHeadingChange);

function onHeadingChange(e:Event):void {
   trace("heading = " + e.target.heading);
}
```

GyroITG3200

```
import funnel.i2c.GyroITG3200;
import funnel.i2c.GyroEvent;
```

This class enables the use of the InvenSense ITG3200 triple axis MEMS gyro in i2c mode.

Please note: this sensor is rated for 3.3v or less and should not be directly interfaced with a 5v Arduino without regulating the voltage appropriately on the digital input and output pins. See this link for details: http://www.sparkfun.com/tutorials/65

Sparkfun sells and easy to use breakout board for this sensor: http://www.sparkfun.com/products/9801

Properties:

`isRunning:Boolean` [read-only]
returns true if read continuous mode is enabled, false if it is not

`rawX:Number` [read-only]
returns the raw x output value from the sensor

`rawY:Number` [read-only]
returns the raw y output value from the sensor

`rawZ:Number` [read-only]
returns the raw z output value from the sensor

`x:Number` [read-only]
returns the angular velocity on the x axis in degrees/second

`y:Number` [read-only]
returns the angular velocity on the y axis in degrees/second

`z:Number` [read-only]
returns the angular velocity on the z axis in degrees/second

Public Methods:

```
GyroITG3200(ioModule:*, autoStart:Boolean = true,
                        autoCalibrate:Boolean = false,
                        address:uint)
```

Parameters:
`ioModule:*` [required] - a reference to the Arduino or Fio I/O module
`autoStart:Boolean` [optional] - true if read continuous mode should start automatically upon instantiation (default = true)
`autoCalibrate:Boolean` [optional] - true if read calibration routine should start automatically upon instantiation (default = false)
`address:uint` [optional] - the i2c address of the sensor
(default = `GyroITG3200.DEVICE_ID`)

Example:
create an instance of the ITG3200 i2c gyro, pass a reference to the fio module and use default parameters (autoStart = true, autoCalibrate = false)

```
var gyro:GyroITG3200;
gyro = new GyroITG3200(fio.ioModule(1));
gyro.addEventListener(Event.CHANGE, onGyroUpdate);

gyro.setOffsets(0, 2.5, -5);  // calibrate x, y, and z to 0

function onGyroUpdate(e:Event):void {
   // print angular velocity (degrees/second) for each axis
   trace("x = " + e.currentTarget.x);
   trace("y = " + e.currentTarget.y);
   trace("z = " + e.currentTarget.z);
}
```

```
calibrate(totalSamples:uint):void
```

call this method to automatically calibrate the sensor by setting the x, y, and z values to zero when the sensor is stationary upon startup.

PLEASE NOTE: The calibration method may take several seconds to complete. Listen for `GyroEvent.CALIBRATION_PROGRESS` *to get current progress, reading the progress property of the GyroEvent.*

Parameters:
`totalSamples:uint` [optional] - total number of samples to use for calibration (default = 500)

Example:
```
var gyro:GyroITG3200;
// autoStart = false, autoCalibrate = true
gyro = new GyroITG3200(fio.ioModule(1), false, true);
gyro.addEventListener(GyroEvent.CALIBRATION_PROGRESS,
                      onProgress);
gyro.addEventListener(GyroEvent.CALIBRATION_COMPLETE,
                      onCalComplete);

function onProgress(e:GyroEvent):void {
   // print calibration progress (0 to 100%)
   trace("calibration " + evt.progress + "% complete");
}

function onCalComplete(e:GyroEvent):void {
   // now that the gyro is calibrated, start listening
   // for data
   gyro.addEventListener(Event.CHANGE, onGyroUpdate);
}
```

setGains`(xGain:Number, yGain:Number, zGain:Number):void`

set the gain value for the x, y, or z output

Parameters:
`xGain:Number` [required] - the gain value for the x axis
`yGain:Number` [required] - the gain value for the y axis
`zGain:Number` [required] - the gain value for the z axis

setOffsets`(xGain:Number, yGain:Number, zGain:Number):void`

use this method to manually zero-calibrate the gyro. On startup, when the gyro is a stationary position the x, y, and z values should be as close to 0 as possible.

Example:
offset the x, y, and z values so that they are as close to 0 as possible when the sensor is in a station position on startup

```
gyro.setOffsets(0, 2.5, -5); // values will be different for
```

```
                                    // each sensor
```

`setReversePolarity``(xPol:Boolean, yPol:Boolean,`
`zPol:Boolean):void`

set the polarity of the x, y, and z output values. True sets polarity to negative, false sets to positive value

Parameters:
`xPol:Boolean` [required] - the polarity of the x axis (default = false)
`yPol:Boolean` [required] - the polarity of the x axis (default = false)
`zPol:Boolean` [required] - the polarity of the x axis (default = false)

`startReading``():void`

start continuous reading of the accelerometer. In this mode, you should not call the update() method, because the code running on the Arduino or Fio board is automatically returning the accelerometer data every 33 milliseconds (or at the rate set by `samplingInterval` if it was changes). The CHANGE event will be dispatched each time a new reading is received.

`stopReading``():void`

stops continuous reading of the accelerometer. To get data from the accelerometer after calling this method, you must either call the update() method (each call to update will only return one set of acceleration values), or call startReading() to begin read continuous mode.

`update``():void`

sends a read request to the gyro and updates the x, y, and z values. If you call update while in read continuous mode, read continuous mode will stop (stopReading() will be called in the background) and you will need to continue to call update on a regular basis to return the latest values. You can also call startReading() to enter continuous read mode.

Public Constants:

`DEVICE_ID` Constant
`public static const DEVICE_ID:uint = 0x69;`

Events:

`Event.CHANGE : String = change`
dispatched when a new set of acceleration values are received

Example:
```
gyro.addEventListener(Event.CHANGE, onGyroUpdate);

function onGyroUpdate(e:Event):void {
   // print angular velocity (degrees/second) for each axis
   trace("x = " + e.currentTarget.x);
   trace("y = " + e.currentTarget.y);
   trace("z = " + e.currentTarget.z);
}
```

`GyroEvent.CALIBRATION_COMPLETE : String = calibrationComplete`
dispatched when the calibration routine has completed

Example:
```
gyro.addEventListener(GyroEvent.CALIBRATION_COMPLETE,
                      onCalComplete);
```

`GyroEvent.CALIBRATION_PROGRESS : String = calibrationProgress`
dispatched continuously during calibration progress. Use to show show calibration progress by tracing output or displaying progress bar.

Example:
```
gyro.addEventListener(GyroEvent.CALIBRATION_PROGRESS,
                      onProgress);
```

`GyroEvent.GYRO_READY : String = gyroReady`
dispatched when the gyro is initialized. The gyro requires approximately 70 milliseconds to settle upon startup. Use this event to be sure startup is complete before beginning to read data or send any commands to the gyro.

Example:
```
gyro.addEventListener(GyroEvent.GYRO_READY, onGyroReady);
```