**Importing the Bindings**

To use these bindings you must import them into Python. We have provided a file called ev3.py. You should move a copy of this program to the same directory as the program you are writing. We will import this file using the following command. This should go at the beginning of your Python program:

from ev3 import \*

Now we can use the ev3dev commands to control motors and sensors.

**Attaching a Motor**

Before we can use a motor or a sensor we need to create a variable we can associate with the device. In these examples we will use the Medium Motor. Using a Large Motor is similar.

For example to control the Medium Motor we could use the following:

medium = MediumMotor()

Here medium is the name we choose to use for the motor. You can use any name you wish. This command will search the ports for a Medium Motor and save its location if it finds one. You will get an error if it doesn't find one.

You can find out which port it is connected to by checking its address.

medium.address

If the Medium Motor is connected to port A then the resulting value will be 'outA'.

You will probably have situations where you have more than one of the same type of device installed. You can specify the output port to which it is attached. For example if the Medium Motor is attached to output port A we could do the following.

medium = MediumMotor('outA')

If we need to, we can check to see if the motor is connected.

medium.connected

This will have the value True if the motor is connected and False if it is not.

**Running Motors**

**Running a Motor for a Specified Time**

The following is an example of a command we could send the motor.

medium.run\_timed(time\_sp = 3000, duty\_cycle\_sp = 75)

This will run the motor for a specified time (3000 milliseconds = 3 seconds) at 75% of full power. The duty cycle can be set to any value in the range [-100, 100]. Negative values will run the motor backwards!

**Running a Motor Forever**

You can also let the motor run until you stop it.

medium.run\_forever(duty\_cycle\_sp = 75)

You stop it by either giving it a new instruction, or using the following command.

medium.stop()

**Running a Motor by Degrees**

You can tell the motor to run though an approximate number of degrees. This works because the motors are "smart" and have a sensor that can detect the number of degrees the motor has turned.

medium.run\_to\_rel\_pos(position\_sp=360)

This will produce approximately on full revolution from the motor. You can check this by checking the position of the motor before and after running the motor.

Begin = medium.position

medium.run\_to\_rel\_pos(position\_sp=360)

end = medium.position

On one example run the value of end-begin was 421. The motor overshot the target by 61 degress! This is because the power is cut to the motor when it reaches 360 degrees but momentum carried it farther. How much farther it will go beyond the set point depends on factors like the speed, load on the motor, etc. You can add a stop command to reduce this.

Begin = medium.position

medium.run\_to\_rel\_pos(position\_sp=360, stop\_command='brake')

end = medium.position

Now the value of end-begin on a sample run was 379. The motor only overshot the target set point by 19 degrees. Not perfect, but much better.

**More on Setting Parameters**

Within your program the values of the motor parameters will be remembered from one command to the next. You do not need to reset them each time. For example if you give following command:

medium.run\_timed(time\_sp=3000, duty\_cycle\_sp=-75)

If you want to make a run for the same time but change the duty cycle to 75 we can just do the following.

medium.run\_timed(duty\_cycle\_sp=75)

You can also set the parameters first and then issue the command.

medium.time\_sp=2000

medium.duty\_cycle\_sp=90

medium.run\_timed()

This list of instructions has the same effect as the following:

medium.run\_timed(time\_sp=2000, duty\_cycle\_sp=90)

**More Motors, Commands and Parameters**

There are two types of motors in the EV3 set and as you can see there are many options for controlling the motors. There are commands we can send to the motor and there are parameters we can set that control how these commands are executed.

**Motors**

* LargeMotor() this function is used to attach a Large Motor. The system will search for a Large Motor or you can specify the port ('outA', 'outB', 'outC', or 'outD'). You will get an error if the motor cannot be found or is not at the specified port.
* MediumMotor() this function is used to attach a Medium Motor. The system will search for a Medium Motor or you can specify the port ('outA', 'outB', 'outC', or 'outD'). You will get an error if the motor cannot be found or is not at the specified port.

**Commands for Motors**

* run\_forever() will cause the motor to run until another command is sent.
* run\_to\_abs\_pos() will run to an absolute position specified by position\_sp and stop. Its behavior when as it stops is controlled using the command specified in stop\_command.
* run\_to\_rel\_pos() will run to a position relative to the current position value. The new position will be current position + position\_sp. Its behavior when as it stops is controlled using the command specified in stop\_command.
* run\_timed() will run the motor for the amount of time specified in time\_sp. . Its behavior when as it stops is controlled using the command specified in stop\_command.
* stop() will stop any of the run commands before they are complete. Its behavior when as it stops is controlled using the command specified in stop\_command.
* reset() will reset all of the motor parameter attributes to their default value. This will also have the effect of stopping the motor.

**Motor Parameters**

* duty\_cycle\_sp This controls the amount of electricity sent to the motor. Units are in percent. Valid values are -100 to 100. A negative value causes the motor to rotate in reverse. This value is only used when speed\_regulation is off.
* polaritySets the polarity of the motor. With normal polarity, a positive duty cycle will cause the motor to rotate clockwise. With inversed polarity, a positive duty cycle will cause the motor to rotate counter-clockwise. Valid values are 'normal' and 'inversed'.
* positionReturns the current position of the motor in degrees. When the motor rotates clockwise, the position will increase. Likewise, rotating counter-clockwise causes the position to decrease. You can set to current position to be any value you like.
* position\_spThis is the target position for the run\_to\_abs\_pos and run\_to\_rel\_pos commands. Units are in degrees.
* ramp\_down\_spThis controls the ramp down set point. Units are in milliseconds. When set to a value > 0, the motor will ramp the power sent to the motor from 100% duty cycle down to 0 over the span of this set point when stopping the motor. If the starting duty cycle is less than 100%, the ramp time duration will be less than the full span of the set point. This appears to only work with run\_to\_rel\_pos and run\_to\_abs\_pos. This could be helpful in preventing the motor from running beyond the desired position set point.
* ramp\_up\_spThis controls the ramp up set point. Units are in milliseconds. When set to a value > 0, the motor will ramp the power sent to the motor from 0 to 100% duty cycle over the span of this set point when starting the motor. If the maximum duty cycle is limited by duty\_cycle\_sp or speed regulation, the actual ramp time duration will be less than the set point.
* speedReturns the current motor speed in degrees per second.
* speed\_regulation\_enabledTurns speed regulation on or off. If speed regulation is on, the motor controller will vary the power supplied to the motor to try to maintain the speed specified in speed\_sp. If speed regulation is off, the controller will use the power specified in duty\_cycle\_sp. Valid values are 'on' and 'off'.
* speed\_spWriting sets the target speed in degrees per second and is used when speed regulation is on.
* stop\_command The value determines the motors behavior when command is set to stop. Also, it determines the motors behavior when a run command completes. Possible values are 'coast', 'brake' and 'hold'. Coast means that power will be removed from the motor and it will freely coast to a stop. Brake means that power will be removed from the motor and a passive electrical load will be placed on the motor. This load will absorb the energy from the rotation of the motors and cause the motor to stop more quickly than coasting. Hold does not remove power from the motor. Instead it actively tries to hold the motor at the current position. If an external force tries to turn the motor, the motor will ‘push back’ to maintain its position.

**Attaching a Sensor**

Attaching a senor is similar to attaching a motor. In these examples we will use the Touch Sensor. Using other sensors is similar.

For example to control the Touch Sensor we could use the following:

touch = TouchSensor()

Here touch is the name we choose to use for the sensor. You can use any name you wish. This command will search the ports for a Touch Sensor and save its location if it finds one. You will get an error if it doesn't find one.

You can find out which port it is connected to by checking its address.

touch.address

If the Medium Motor is connected to port 1 then the resulting value will be 'in1'.

You may have situations where you have more than one of the same type of sensor installed. You can specify the input port to which it is attached. For example if the Touch Sensor is attached to input port 1 we could do the following.

touch = TouchSensor('in1')

If we need to, we can check to see if the sensor is connected.

touch.connected

This will have the value True if the sensor is connected and False if it is not.

**Reading the Touch Sensor**

You can ask the sensor to report its state by doing the following:

touch.value()

This function will return a numeric value. Since this is a touch sensor, there are only two possible values: 0 tells you the button is not currently pushed, 1 tells you the button is pushed.

You could have the Touch Sensor and Medium Motor interact using the following simple program.

from ev3 import \*

m = MediumMotor()

t = TouchSensor()

m.run\_forever(duty\_cycle\_sp=50)

while (t.value() == 0):

pass

m.stop()

This program is tm.py and is available on the Wikispaces site. It will run the motor until the sensor is pressed.

**Reading the Color Sensor**

As you might imagine there is more going on with the Color Sensor. This sensor has several modes. It can detect reflected light intensity ('COL-REFLECT'), ambient light intensity ('COL-AMBIENT'), and color ('COL-COLOR'). You set the mode by setting the mode parameter. For example to detect ambient light intensity you could do the following:

color = ColorSensor()

color.mode = 'COL-AMBIENT'

color.value()

This would return the current ambient light intensity as a number between 0 and 100 where 0 is complete darkness.

The process for measuring reflected light is the same except the mode is set to 'COL-REFLECT'.

The color mode is a little different. If you set the mode to 'COL-COLOR' then the return value will be an integer in the range 0-7. The values are as follows

* 0 none
* 1 black
* 2 blue
* 3 green
* 4 yellow
* 5 red
* 6 white
* 7 brown

The following program would run the Medium Motor until the Color Sensor sees red.

from ev3 import \*

m = MediumMotor()

c = ColorSensor()

c.mode = 'COL-COLOR'

m.run\_forever(duty\_cycle\_sp=50)

while (c.value() != 5):

pass

m.stop()

This program is cm.py and is available on the Wikispaces site.

**Reading the Ultrasonic Sensor**

The Ultrasonic Sensor measures distance and so is somewhat simpler than the Color Sensor. It sends out ultrasonic pulses and listens to see how long it takes for to hear an echo (echo location). It can listen continuously or send out a single pulse. Finally it can report its distances in inches or centimeters.

If you select continuous measurement the sensor will continuously measure the distance and when you check the value you will get the value it is currently seeing. The modes for this are 'US-DIST-CM' (measurement in centimeters) and 'US-DIST-IN' (measurement in inches). The value will actually be in 10th of an inch or 10th of a centimeter. As an example, suppose you give the following commands:

ultra = UltrasonicSensor()

ultra.mode = 'US-DIST-CM'

ultra.value()

If the return value is 324 it means that the sensor measured a distance of 32.4 cm.

If you select to make a single measurement then the measurement will take place when the mode is set. The distance measured will be remembered and reported when value is called. The modes for this are 'US-SI-CM' and 'US-SI-IN'.

If you give the command

ultra.mode = 'US-SI-CM'

move the sensor so that the distance is different and then issue the command

ultra.value()

you will get the distance from when the mode was set in the first command. To make another measurement you must set the mode again.

The following program would run the Medium Motor until the Ultrasonic Sensor measures an object at a distance of 5 cm.

from ev3 import \*

m = MediumMotor()

u = UltrasonicSensor()

u.mode = 'US-DIST-CM'

m.run\_forever(duty\_cycle\_sp=50)

while (u.value() != 50):

pass

m.stop()

**Reading the Gyro Sensor**

The Gyro Sensor measures rotations about the axis indicated by the arrows on the top of the sensor itself. It actually works using accelerometers and so is not a true gyroscope. It has two modes, one ('GYRO-ANG') measures the number of degrees rotated since the sensor was initialized. The other ('GYRO-RATE') measures the rate at which the sensor is rotating in degrees per second.

For example the following code could be used to measure the rate of rotation of the sensor.

from ev3 import \*

gyro = GyroSensor()

gyro.mode = 'GYRO-RATE'

gyro.value()

**Reading the Temperature Sensor**

The Temperature Sensor measures temperature. This is a Lego Mindstorms NXT sensor and not an EV3 sensor. Because of this, binding to control this sensor is not complete. We will finish it in of one of our projects.

It has three modes: the first ('TEMP-C') measures temperature in degrees Celsius, the second ('TEMP-F') measures temperature in degrees Fahrenheit, and the final one ('TEMP-K') measures the temperature in degrees Kelvin. In the current ev3.py these modes do not work correctly.

One additional thing to note about this sensor is that you must supply an input port ('in1', 'in2', 'in3', or 'in4'). It cannot be detected automatically like the other sensors.

For example the following code could be used to measure the temperature in degrees Celsius.

from ev3 import \*

temp = TemperatureSensor('in1')

temp.mode = 'TEMP-C'

temp.value()