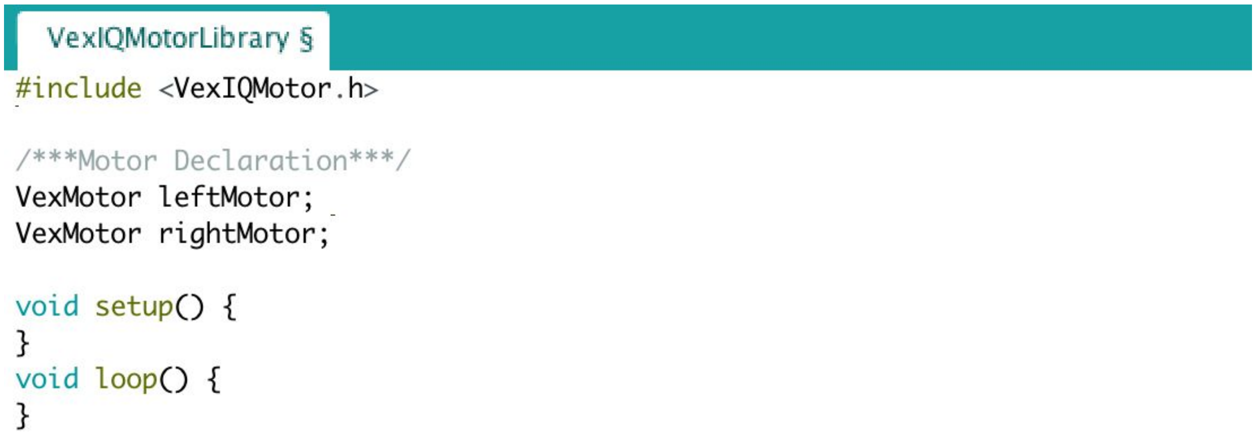
VexIQMotor Library API

Jeremy Desmond

DISCLAIMER: This API is not an official Vex Document. Simply a project of mine.

VexMotor()

No arg constructor. Creates an instance of the VexMotor class and initializes all values to zero. Later on the values will be updated by motorSetup(). *Make sure this is defined globally.*



```
#include <VexIQMotor.h>

/***Motor Declaration***/
VexMotor leftMotor;
VexMotor rightMotor;

void setup() {
}
void loop() {
}
```

initBroadcast()

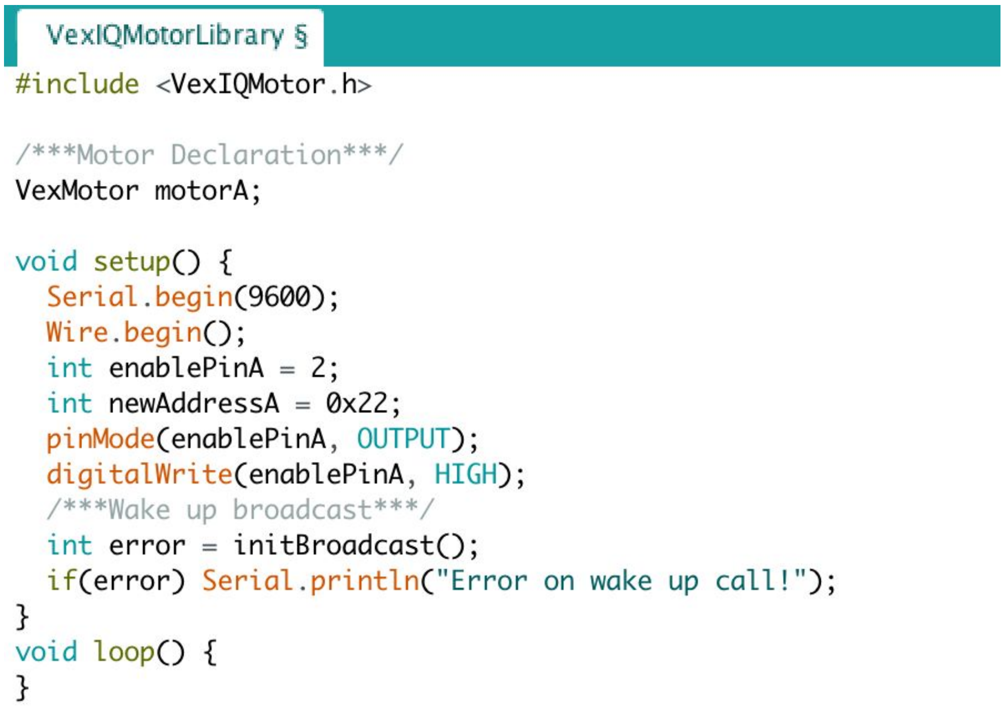
Initial broadcast that is to be sent to the motors prior to initialization. This function is a set of commands that “wakes up” the motors and prepares them for the initialization process by sending a set of commands to all devices on the I2C bus. Without calling this function you will not be able to call motorSetup or use the motors. IMPORTANT! All enable line *must* be set to OUTPUT and HIGH before issuing the initBroadcast().

Parameters:

NONE

Return:

int error - The error value from the end transmission. If the return value is anything other than 0, then something went wrong and you will need to restart the whole process.



```
#include <VexIQMotor.h>

/***Motor Declaration***/
VexMotor motorA;

void setup() {
  Serial.begin(9600);
  Wire.begin();
  int enablePinA = 2;
  int newAddressA = 0x22;
  pinMode(enablePinA, OUTPUT);
  digitalWrite(enablePinA, HIGH);
  /***Wake up broadcast***/
  int error = initBroadcast();
  if(error) Serial.println("Error on wake up call!");
}
void loop() {
}
```

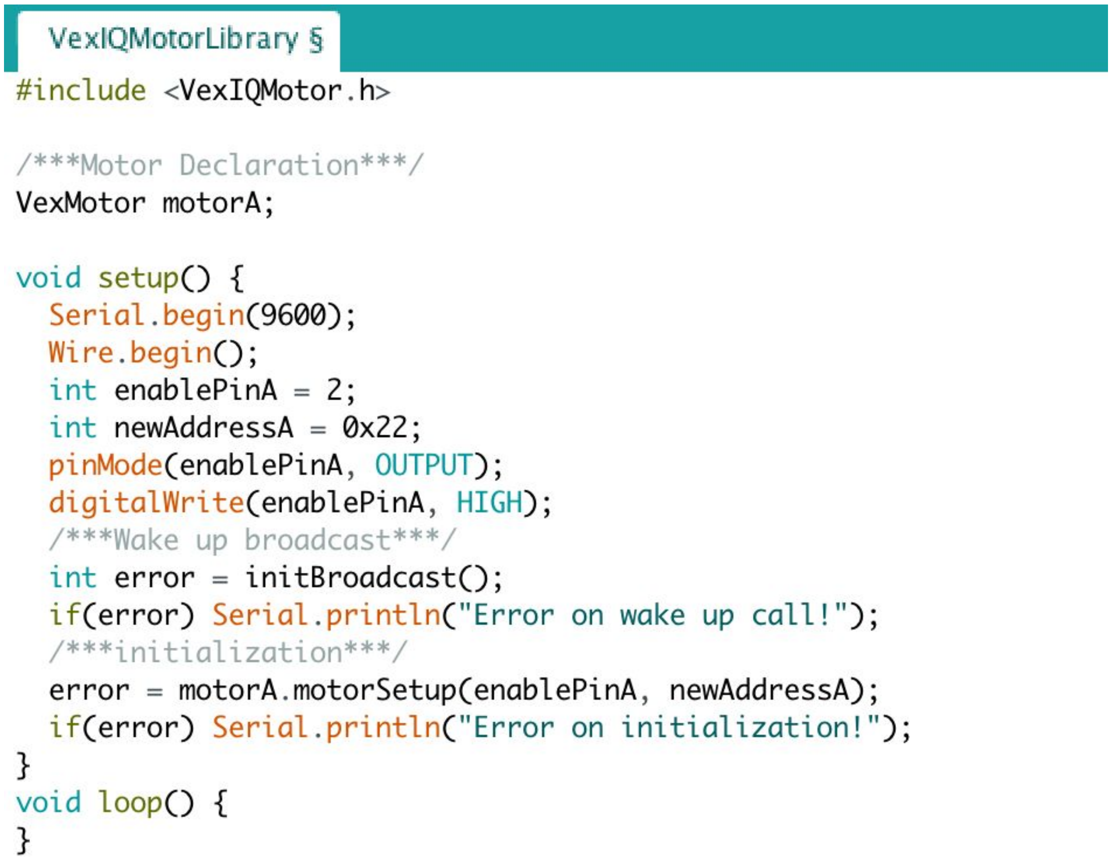
motorSetup(int enablePin, int newAddress)

motorSetup is where all of the initialization work is done. You can only call this after initBroadcast() has been successfully called.

Parameters:
int enablePin - the pin number for the I/O port wired to the enable pin of the vex motor. This pin is pinged during the setup process in order to specify which motor is being initialized. *IMPORTANT*: make sure that initBroadcast() has been successfully called before using motorSetup.

int newAddress - The new hex address that will be assigned to the motor.

Return:
int error - the error from the end transmission. If the function call return anything other than 0, then the command did not execute properly.



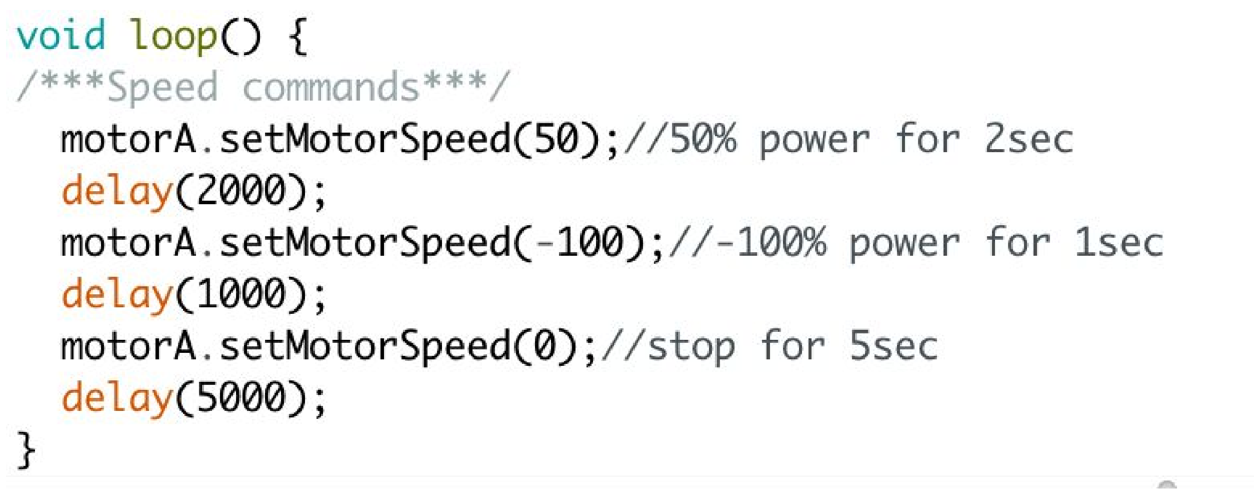
```
#include <VexIQMotor.h>

/***Motor Declaration***/
VexMotor motorA;

void setup() {
  Serial.begin(9600);
  Wire.begin();
  int enablePinA = 2;
  int newAddressA = 0x22;
  pinMode(enablePinA, OUTPUT);
  digitalWrite(enablePinA, HIGH);
  /***Wake up broadcast***/
  int error = initBroadcast();
  if(error) Serial.println("Error on wake up call!");
  /***initialization***/
  error = motorA.motorSetup(enablePinA, newAddressA);
  if(error) Serial.println("Error on initialization!");
}
void loop() {
}
```

setMotorSpeed(float spd)

Set the speed by specifying the percentage of power to be applied to the motor. The range is -100 to 100. Setting the speed to 0 will result in a coasting stop, but other brake modes are available as well. The motor will continue at the specified power level until instructed to stop, or power is no longer supplied to the motor. Any values enter that are outside the range will be mapped back to the proper values.

Parameters:
float spd - Power level of the motor. Negative values result in clockwise turning (backwards) and positive values result in counterclockwise turning (forwards).

Return:
int error - The error from the end transmission of the I2C calls. If the value returned is 0, then the function worked fine. Any other values, such as 2, means that there was a problem with the I2C communication either from a wiring problem or because the motor was not properly initialized prior to calling this function.



```
void loop() {
/***Speed commands***/
  motorA.setMotorSpeed(50);//50% power for 2sec
  delay(2000);
  motorA.setMotorSpeed(-100);//-100% power for 1sec
  delay(1000);
  motorA.setMotorSpeed(0);//stop for 5sec
  delay(5000);
}
```

holdBrake()

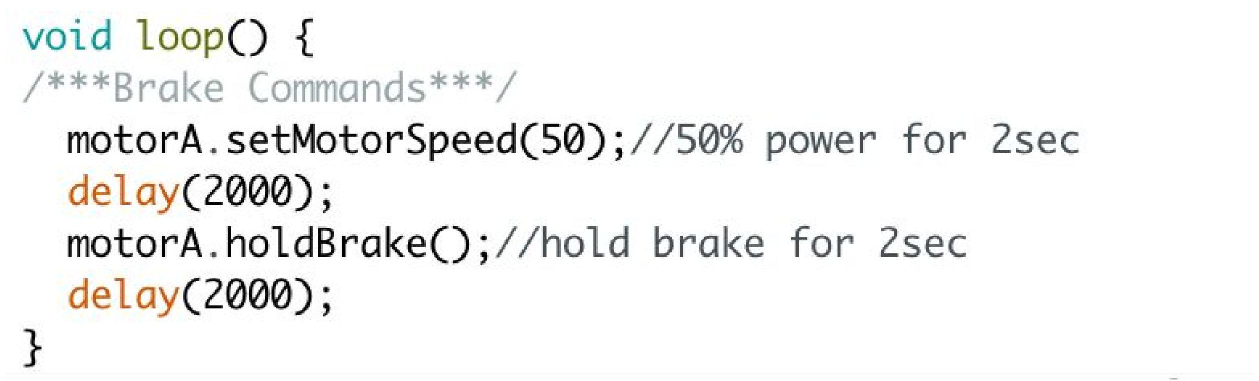
Method that applies a hold brake to the specified motor. The hold brake applies power to actively stop, and hold the current position.

Parameters:

NONE

Return:

int error - The error from the end transmission in the I2C communication. Return value of 0 means it worked fine, any other value indicates and I2C error.

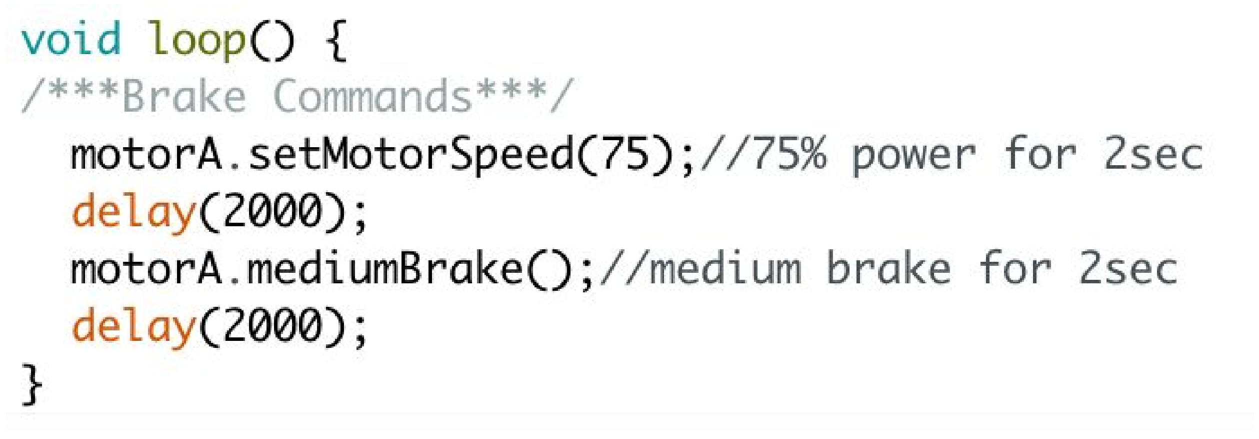


```
void loop() {
/***Brake Commands***/
  motorA.setMotorSpeed(50);//50% power for 2sec
  delay(2000);
  motorA.holdBrake();//hold brake for 2sec
  delay(2000);
}
```

mediumBrake()

Applies a medium level brake to the motor. Not as strong as the hold brake, but still applies some power to actively stopping the motion and holding the position.

Parameters:
NONE
Return:
int error - The error from the end transmission in the I2C communication. Return value of 0 means it worked fine, any other value indicates and I2C error.

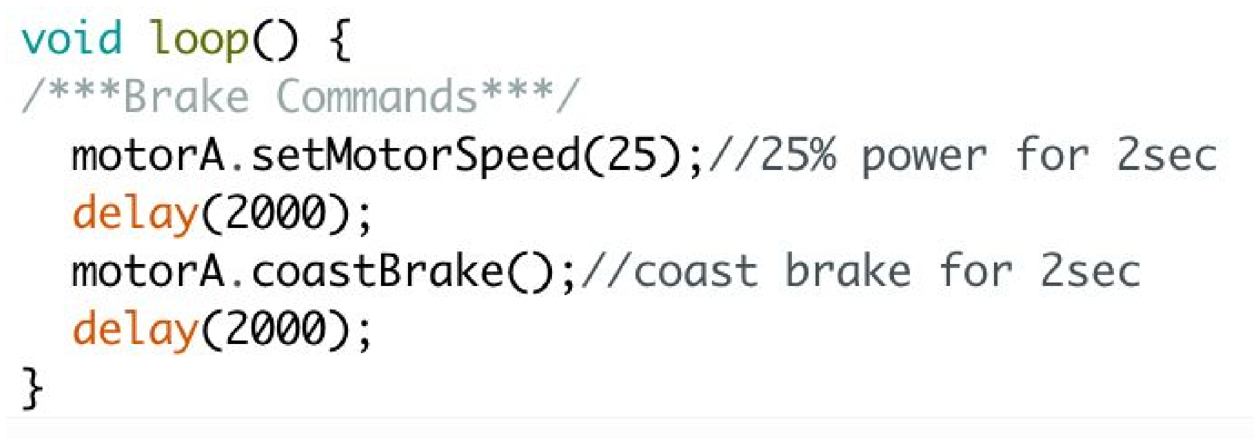


```
void loop() {
/***Brake Commands***/
  motorA.setMotorSpeed(75);//75% power for 2sec
  delay(2000);
  motorA.mediumBrake();//medium brake for 2sec
  delay(2000);
}
```

coastBrake()

Stops the motor by simply cutting the power. This is the same as setting the speed to zero. The robot will coast to a stop.

Parameters:
NONE
Return:
int error - The error from the end transmission in the I2C communication. Return value of 0 means it worked fine, any other value indicates and I2C error.



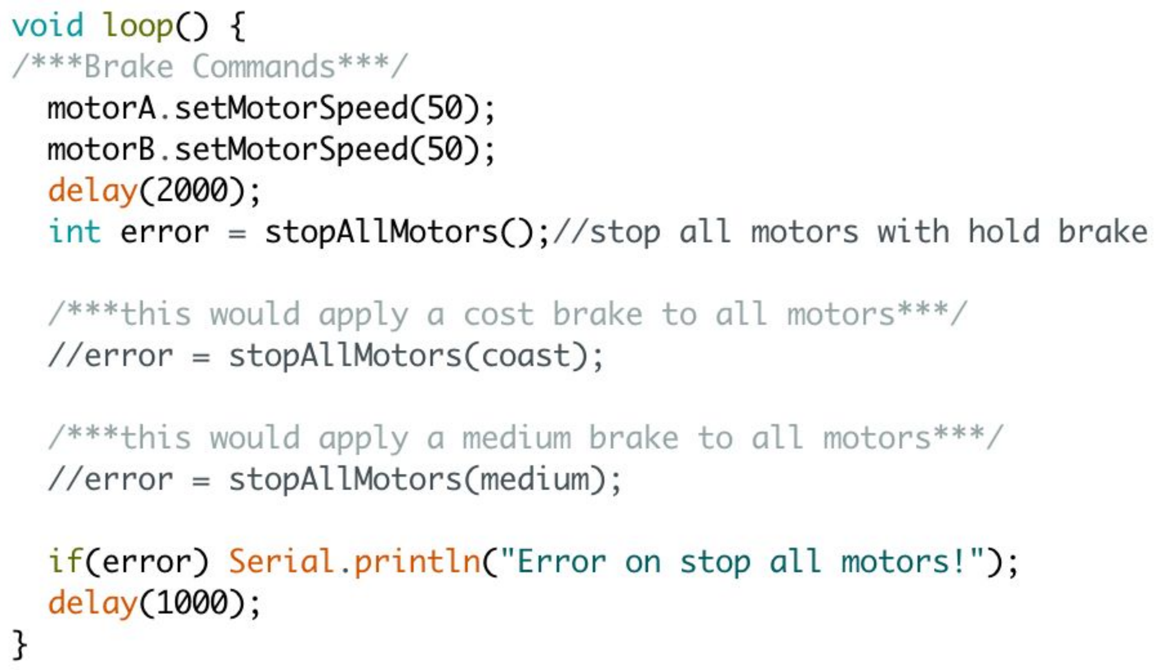
```
void loop() {
/***Brake Commands***/
  motorA.setMotorSpeed(25);//25% power for 2sec
  delay(2000);
  motorA.coastBrake();//coast brake for 2sec
  delay(2000);
}
```

stopAllMotors(enBrakeMode option = hold)

This is not a method, a member of the VexMotor class, and can be called as a regular function. By default it applies the hold brake to all of the motors that have been initialized with motorSetup. The parameter is **optional**, which means you can call this function without any parameters at all and the brake mode will be hold by default.

Parameter:
enBrakeMode option - By default this is set to hold. So you can omit the argument and it will apply a hold brake to all the motors. However, you have the option to change the brake mode to either coast or medium. enBrakeMode is an enumerated type that contains the brake modes.

Return:
int error - The error value returned from the end transmission of the I2C communication. An error of 0 means that everything worked fine, any other value indicates a wire or other problem.

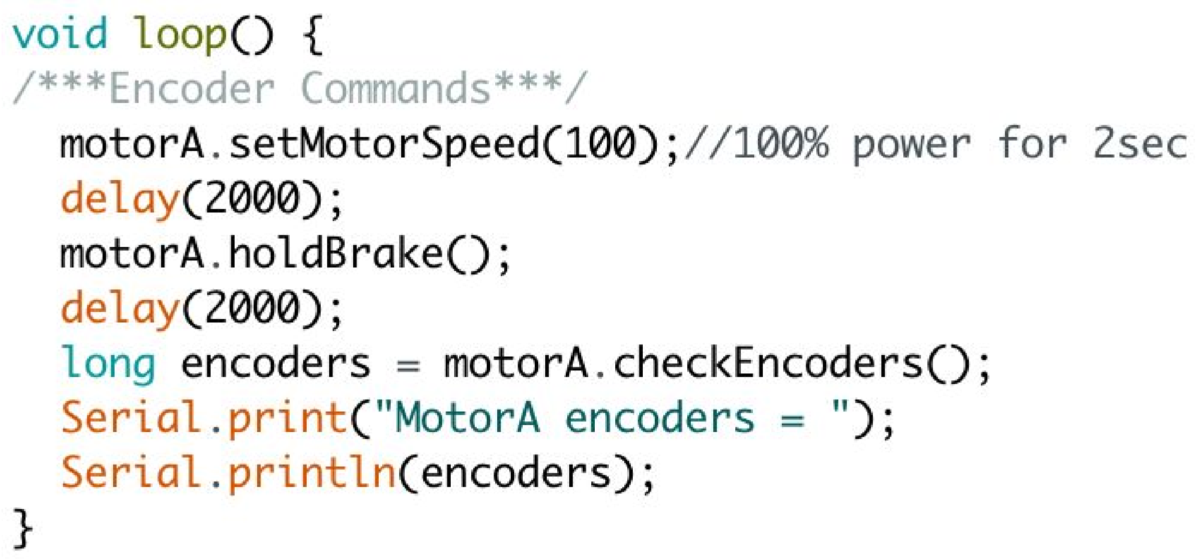


```
void loop() {
/***Brake Commands***/
  motorA.setMotorSpeed(50);
  motorB.setMotorSpeed(50);
  delay(2000);
  int error = stopAllMotors();//stop all motors with hold brake

  /***this would apply a cost brake to all motors***/
  //error = stopAllMotors(coast);

  /***this would apply a medium brake to all motors***/
  //error = stopAllMotors(medium);

  if(error) Serial.println("Error on stop all motors!");
  delay(1000);
}
```

checkEncoders()

Returns the motors current encoder count. There are 960 encoders in one full rotation. Negative speeds result in negative encoders.

Parameters:
NONE
Return:
long enc - The motors curren encoder count. If the return value is constantly 1, 2, 3, or 4 then it is actually returning an error from the end transmission, you may have a wire problem.



```
void loop() {
/***Encoder Commands***/
  motorA.setMotorSpeed(100);//100% power for 2sec
  delay(2000);
  motorA.holdBrake();
  delay(2000);
  long encoders = motorA.checkEncoders();
  Serial.print("MotorA encoders = ");
  Serial.println(encoders);
}
```

resetEncoders()

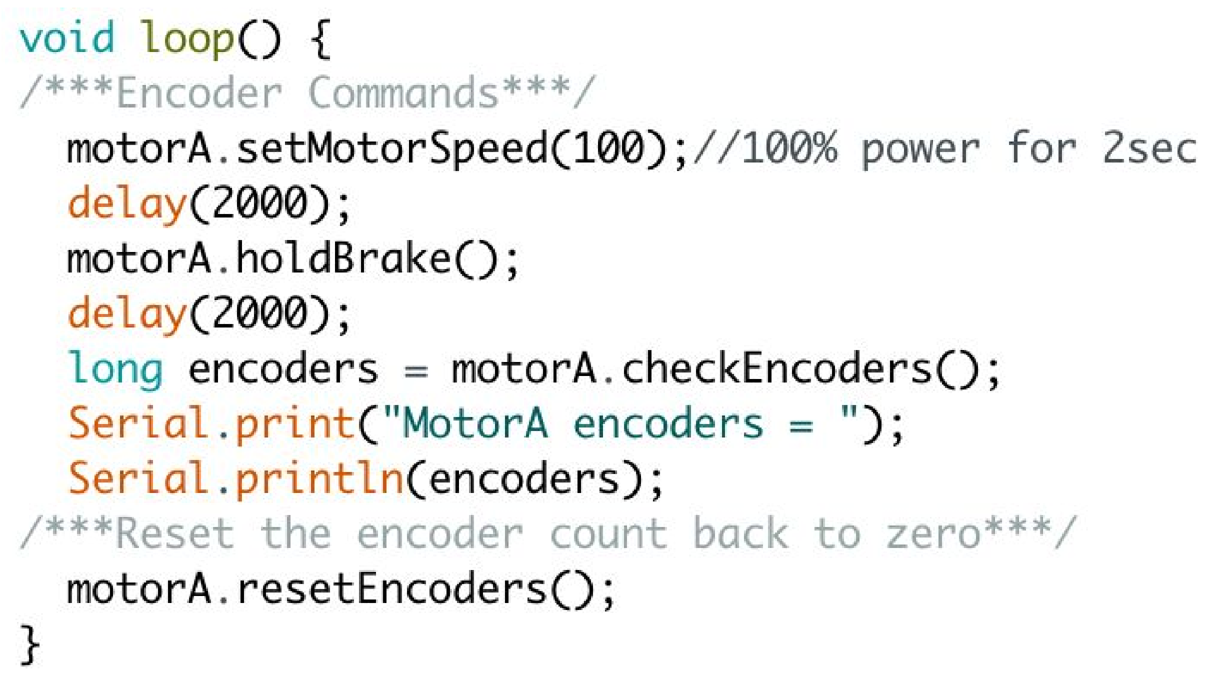
Resets the encoder count on the motor back to zero.

Parameters:

NONE

Return:

int error - The error from the end transmission of the I2C call. Any value other than 0 means the command was not executed properly and the encoders did not reset. A return value of 0, however, means everything worked fine.



```
void loop() {
/***Encoder Commands***/
  motorA.setMotorSpeed(100);//100% power for 2sec
  delay(2000);
  motorA.holdBrake();
  delay(2000);
  long encoders = motorA.checkEncoders();
  Serial.print("MotorA encoders = ");
  Serial.println(encoders);
/***Reset the encoder count back to zero***/
  motorA.resetEncoders();
}
```

encoderTarget(float spd, long encTarget, bool stateRelative = true)

This function is used to travel to a specified motor encoder target. There are two variation of the function. If you wish to go backwards you need to make one, or both, of the arguments negative (i.e. if the speed is possitive make the encTarget negative to move backwards). There are two states of this function: relative and absolute.
Relative State: The encTarget will add on to the current encoder count. So if the motor is at 2000 encoders and encTarget is 960, the motor will move 960 encoders for a final cumulative count of 2960.
Absolute State: The encTarget is the motors destination based on its absolute encoder count. So if the motor is at 2000 encoders and encTarget is 960 with the stateRelative set to false, the motor will move backwards -1040 encoders to get to 960 encoders as the absolute count.
The relative state is set by default to true, but you can select to use the absolute state by setting stateRelative to false.

Parameters:
float spd - The speed at which the motor will travel to the target. A negative value will result in backwards, or clockwise, movement.
long encTarget - The target distance that the motor will travel. A negative value will result in backwards, or clockwise, movement.
bool stateRelative - By default set to true. True results in moving the encoder target amount no matter what the current value is on the motor. False results in moving to the encoder target as an absolute value of the motors encoder count.

Return:

int error - The error on the end transmissions. Returns 0 if it was successful and any other value from 1 to 4 if the I2C commands failed.

(Check below mototActive() for an example)

motorActive()

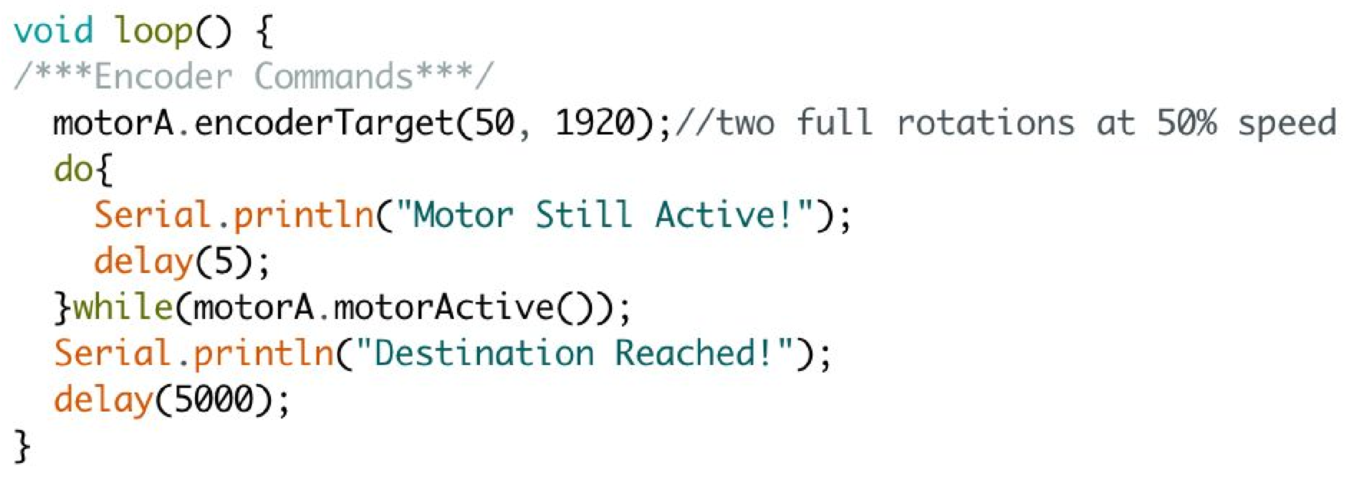
This can be used to check and see if a motor is still traveling to its encoder target. While using encoderTarget you can check to see if the motor has reached its destination by using motorActive. If the motor has not yet reached its target the return value will be true. Once the motor reaches its target and is ready for use again the return value is false. The alternative is to use timing and wait until the motor is done, but that is highly inaccurate.

Parameters:

NONE

Return:

bool - If the motor is still moving towards the target, or in motion at all, the return value is true. If the motor is stopped and waiting for commands the return value is false, for not active.



```
void loop() {
/***Encoder Commands***/
  motorA.encoderTarget(50, 1920);//two full rotations at 50% speed
  do{
    Serial.println("Motor Still Active!");
    delay(5);
  }while(motorA.motorActive());
  Serial.println("Destination Reached!");
  delay(5000);
}
```

servoMode(long deg)

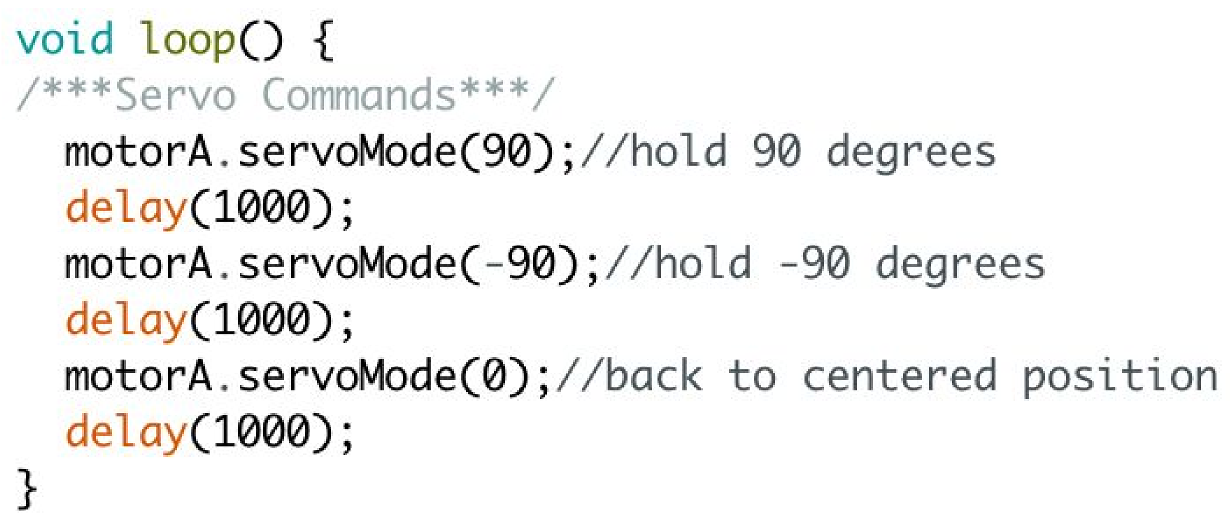
This movement function allows you to operate the motor as a servo motor. Entering a degrees value will turn the motor to that point and hold the position. The zero encoder mark is the centered position and any degrees entered will turn relative to the zero degree mark.

Parameters:

long deg - The amount of degrees the motor will turn. A hold brake will be used to stay in the position.

Return:

int error - The error from the I2C end transmission. If the error is 0 there were no problems, however any other error value indicates an I2C communication issue.



```
void loop() {
/***Servo Commands***/
  motorA.servoMode(90);//hold 90 degrees
  delay(1000);
  motorA.servoMode(-90);//hold -90 degrees
  delay(1000);
  motorA.servoMode(0);//back to centered position
  delay(1000);
}
```