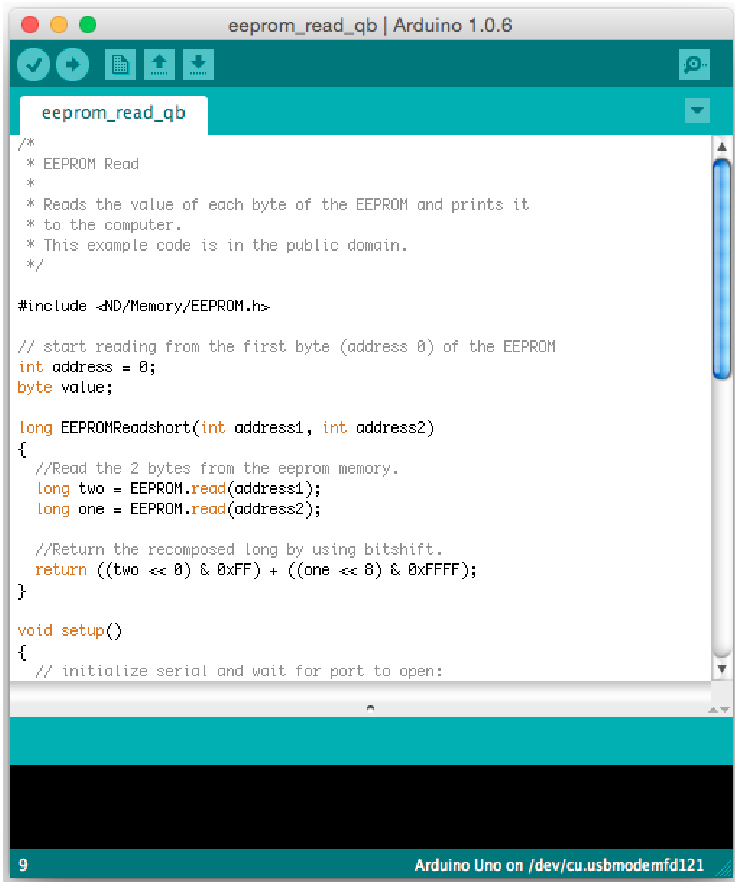
Standard Base Assembly: Code Decoded

At Notre Dame, the linemen are controlled either via XBee linked to an Arduino Uno R3 microcontroller connected to Xbox controllers, or via BlueTooth connected to a wireless PS4 controller. The drive stack as a whole consists of the Arduino, the screw shield, the XBee shield, and the XBee chip. The controller stack consists of the Arduino, the USB host shield, the XBee shield, and the XBee chip.

When using a new Arduino Uno that hasn't been programmed before (or if you don't know if the EEPROM was ever previously cleared), you must clear its EEPROM in order to use the Notre Dame code. Upload the *eeprom_setup* program to the Arduino Uno and let it run for a second, and the Arduino Uno should be ready to be programmed. The common problem you will notice with the robots if the EEPROM has not been cleared is that it is very hard to calibrate the driving offsets for the robot to drive straight.

In order to program an Arduino microcontroller, the Arduino IDE (Integrated Development Environment) must be downloaded from this site online, http://arduino.cc/en/main/software. At the time of this draft, the current version of the Arduino IDE, version 1.6.9, has bugs in it that prevent it from working properly with our codes. New additions of the IDE may resolve this issue, but until then it is necessary to use the Arduino IDE version 1.0.6 which can be found online at http://www.arduino.cc/en/Main/OldSoftwareReleases - previous.


eeprom_read_qb | Arduino 1.0.6
eeprom_read_qb
9
Arduino Uno on /dev/cu.usbmodemfd121

```
/*
 * EEPROM Read
 *
 * Reads the value of each byte of the EEPROM and prints it
 * to the computer.
 * This example code is in the public domain.
 */

#include <ND/Memory/EEPROM.h>

// start reading from the first byte (address 0) of the EEPROM
int address = 0;
byte value;

long EEPROMReadshort(int address1, int address2)
{
  //Read the 2 bytes from the eeprom memory.
  long two = EEPROM.read(address1);
  long one = EEPROM.read(address2);

  //Return the recomposed long by using bitshift.
  return ((two << 0) & 0xFF) + ((one << 8) & 0xFFFF);
}

void setup()
{
  // initialize serial and wait for port to open:
```

Library Files: ND

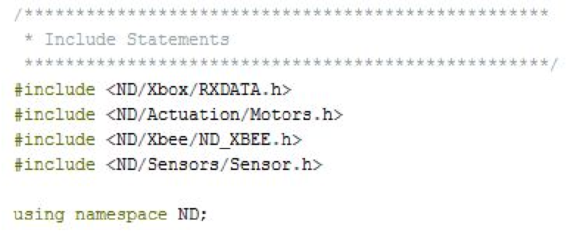
Before you add the ND library to your core directory, the EEPROM.h file is one that may already exist in your Arduino cores library. If it does, you should find and delete it. You should be able to find this using a simple search function in your windows browser. The reason you must do this is because there is another EEPROM.h file that is located in the 'ND' folder that you will add to your core directory. Your code may not compile if you have two instances of the same header file. The difference between these files is that in the EEPROM.h file that is in the 'ND' folder, there has been constants added that provide addresses for storing data in the EEPROM. These constants are for driving offsets, which are used in all of the robots. More addresses can be added to this header file if there is need to permanently store more constants, but know that the EEPROM has a limited number of write/erase cycles.

In order to set up the Notre Dame header files and classes, the ND folder must be pasted into the location, hardware → arduino → cores → arduino. During compilation, the Arduino IDE looks in three directories for any libraries or header files. These directories are the library folder of the program, the sketchbook library folder, and the core directory (hardware → arduino → cores → arduino). First, locate the hardware directory, which is most likely in the same folder as the executable file (.exe) where the IDE was installed. Then, follow the path for the core directory, found above, and paste the folder called 'ND' at this location. Now, all the main programs can be uploaded to the board because the compiler and assembler can access the proper header files and classes.

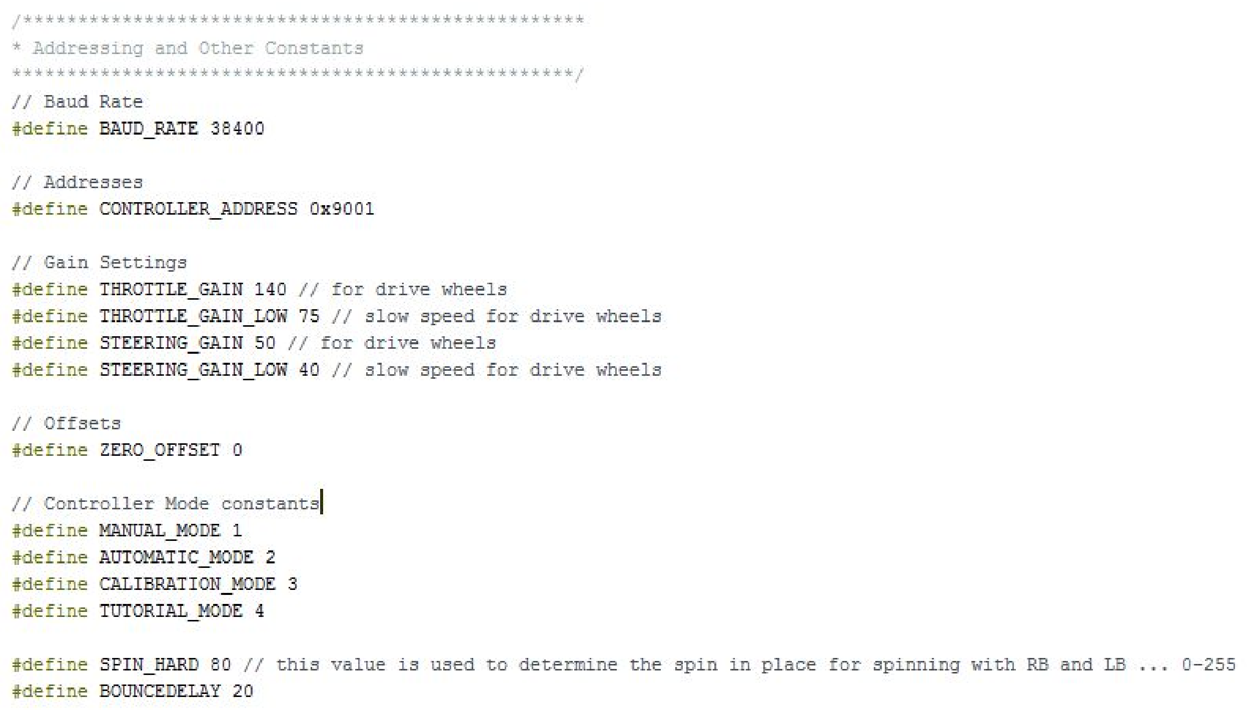
Drive Code: StdBase_2015_Drive.ino

This section of the Code guide aims to show the reader parts of the drive code and explain what each section does. This is the code that is uploaded onto the drive stack housed within the robot. This is not however a line by line guide, but the essential parts of each section of code will be discussed.

For Arduino, it is necessary to include several header files (.h files). This is accomplished by using the syntax `#include <`*`filename`*`.h>`. Define statements, which are created similarly to include statements, make specified strings equal to a value or another string.

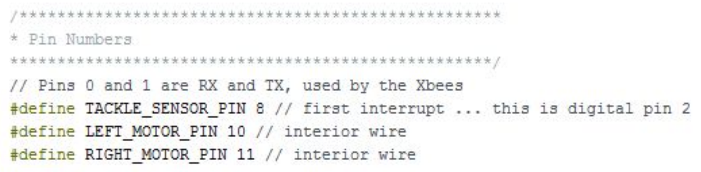


```
/***************************************************
 * Include Statements
 ***************************************************/
#include <ND/Xbox/RXDATA.h>
#include <ND/Actuation/Motors.h>
#include <ND/Xbee/ND_XBEE.h>
#include <ND/Sensors/Sensor.h>

using namespace ND;
```



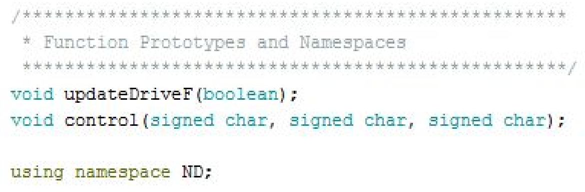
```
/***************************************************
* Addressing and Other Constants
***************************************************/
// Baud Rate
#define BAUD_RATE 38400

// Addresses
#define CONTROLLER_ADDRESS 0x9001

// Gain Settings
#define THROTTLE_GAIN 140 // for drive wheels
#define THROTTLE_GAIN_LOW 75 // slow speed for drive wheels
#define STEERING_GAIN 50 // for drive wheels
#define STEERING_GAIN_LOW 40 // slow speed for drive wheels

// Offsets
#define ZERO_OFFSET 0

// Controller Mode constants
#define MANUAL_MODE 1
#define AUTOMATIC_MODE 2
#define CALIBRATION_MODE 3
#define TUTORIAL_MODE 4

#define SPIN_HARD 80 // this value is used to determine the spin in place for spinning with RB and LB ... 0-255
#define BOUNCEDELAY 20
```

The *BAUD_RATE* is a measure of the number of bits that can flow through a communications channel per second. The *CONTROLLER_ADDRESS* allows the XBee to determine which controller it should be receiving data from. Each robot should have its own address, or else the signals from their controllers will interfere. The *Throttle* and *Steering* Gains modify the x and y inputs from the Xbox controller joysticks used to drive the robot. The left joystick controls the *THROTTLE_GAIN* value, which controls forward and backward motion, and the right joystick controls the *STEERING_GAIN* value, which controls left and right turning.



```
/***************************************************
* Pin Numbers
***************************************************/
// Pins 0 and 1 are RX and TX, used by the Xbees
#define TACKLE_SENSOR_PIN 8 // first interrupt ... this is digital pin 2
#define LEFT_MOTOR_PIN 10 // interior wire
#define RIGHT_MOTOR_PIN 11 // interior wire
```

These lines of code define the pin for the left motor to be pin 10 and the right to be pin 11. The tackle sensor pin will be pin 8.



```
/***************************************************
 * Function Prototypes and Namespaces
 ***************************************************/
void updateDriveF(boolean);
void control(signed char, signed char, signed char);

using namespace ND;
```

Here, the coder declares the names of two functions, *updateDriveF* and *control*, and defines their output and input data types. The namespace is called *ND*.

```
/***************************************************
* Debugging Variables
***************************************************/
//#define DEBUG_TIME
#ifdef DEBUG_TIME
  long timedbg;
#endif

//#define DEBUG
#ifdef DEBUG // variables used in debugging
  int * vals1, *vals2;
  motor_offsets mo;
  uint8_t val;
#endif
```

These lines of code relate to variables needed for the debugging process. As we are assuming that this drive code is functional, we will not investigate this in detail.

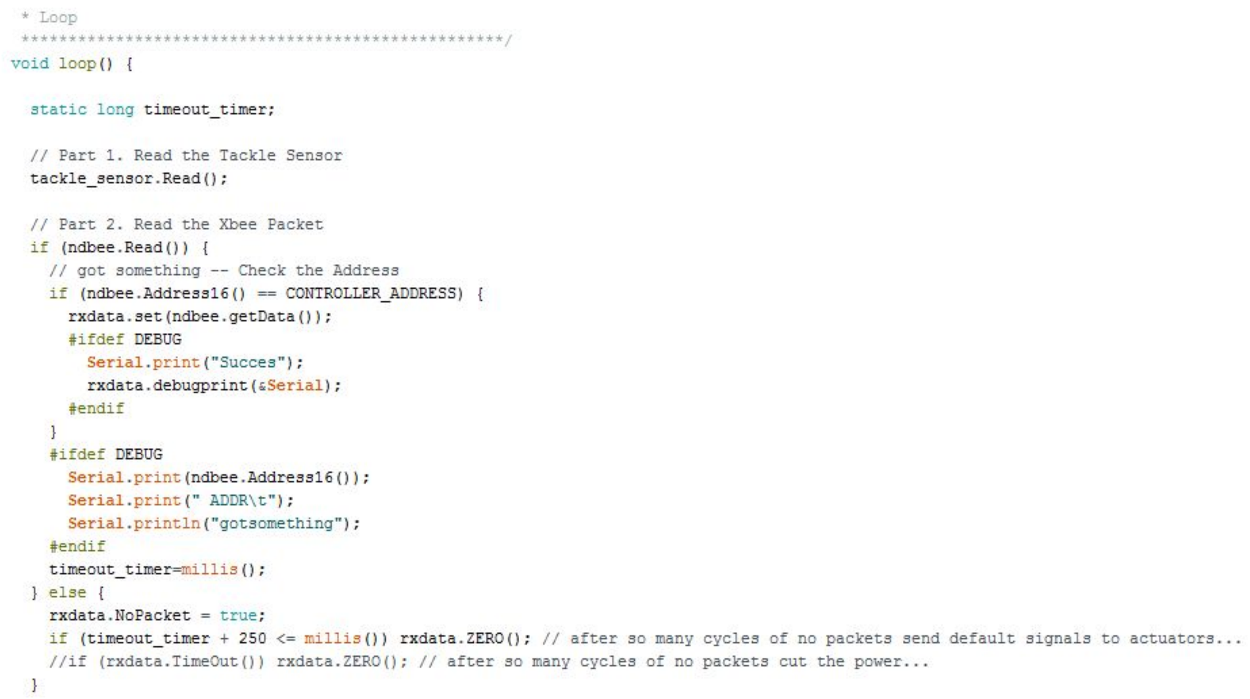
```
/***************************************************
* Global Variables
***************************************************/
// Motor Class
Servo lm; Servo rm;
Motors Wheels(&lm,&rm,THROTTLE_GAIN,STEERING_GAIN,ZERO_OFFSET); // used to control a pair of steering motors
boolean blnSlowGains = false;

// Data Classes
ND_XBEE ndbee; // used for RF communication
Rx16Response rx16 = Rx16Response();
RXDATA rxdata; // used to handle the incoming data from the xbox controller... button presses, releases, etc.

// Sensor Class
bool tackled = false; // determines if you are tackled or not
Tackle_Sensor tackle_sensor(TACKLE_SENSOR_PIN);
uint8_t message[1] = {1};

int count;
```

These lines create variable definitions and also organize the various data inputs and outputs into data classes. These variables, since they are defined outside of the main loop and any interior functions, are treated as global variables.

```
/***************************************************
 * Setup
 ***************************************************/
void setup() {
  // Setup motors;
  lm.attach(LEFT_MOTOR_PIN); rm.attach(RIGHT_MOTOR_PIN);
  Wheels.steering_gain = STEERING_GAIN;
  Wheels.throttle_gain = THROTTLE_GAIN;
  Wheels.Stop();
  Serial.begin(BAUD_RATE);
  // setup the RF communication
  ndbee = ND_XBEE();
  ndbee.xbee.setSerial(Serial);
  rxdata = RXDATA();
  Serial.println("STARTING...");
  delay(2000); // delay 2 seconds
}
```

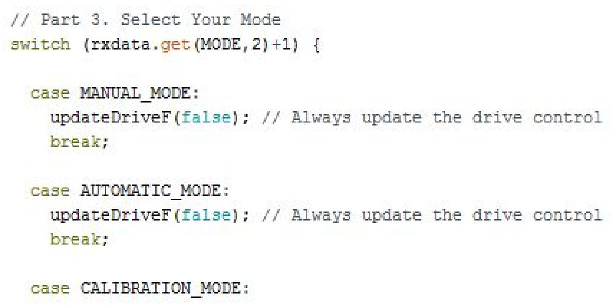
This *void setup()* function is applied right before beginning the code's main loop. In this code, the servos *lm* and *rm* are attached to their corresponding pins. The gains of both throttle and steering are also set to their corresponding values. *Setup()* also immediately sets the wheels at *Stop*, so the robot doesn't take off on activation. After that line, the code prompts the Arduino to begin looking for the serial date at the designated baud rate, though there is an initial 2 second delay to process everything, so that the robot doesn't immediately drive away.

The remaining code all takes place within the *void loop()* function.



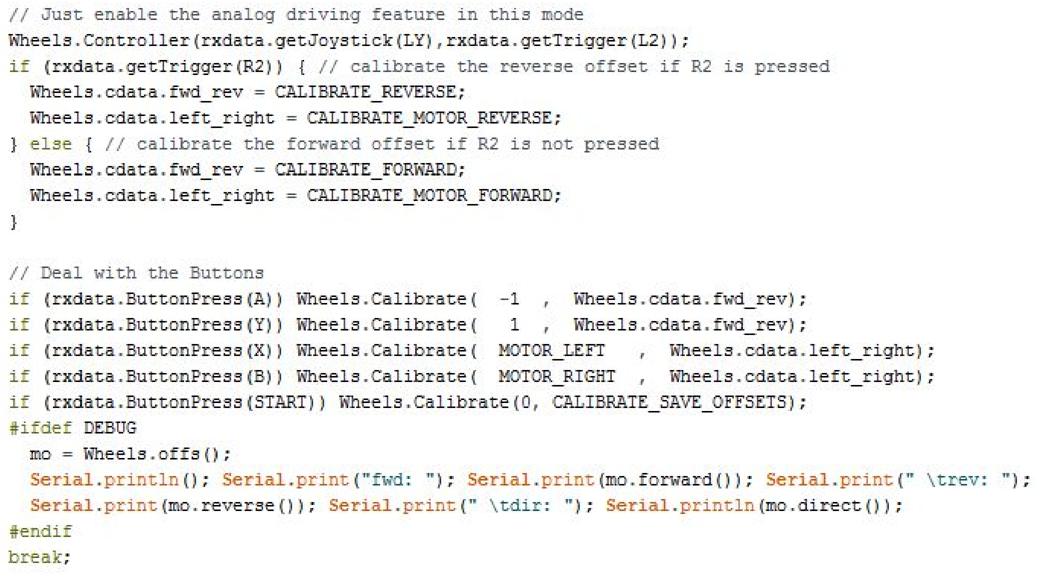
```
 * Loop
 ****************************************************/
void loop() {

  static long timeout_timer;

  // Part 1. Read the Tackle Sensor
  tackle_sensor.Read();

  // Part 2. Read the Xbee Packet
  if (ndbee.Read()) {
    // got something -- Check the Address
    if (ndbee.Address16() == CONTROLLER_ADDRESS) {
      rxdata.set(ndbee.getData());
      #ifdef DEBUG
        Serial.print("Succes");
        rxdata.debugprint(&Serial);
      #endif
    }
    #ifdef DEBUG
      Serial.print(ndbee.Address16());
      Serial.print(" ADDR\t");
      Serial.println("gotsomething");
    #endif
    timeout_timer=millis();
  } else {
    rxdata.NoPacket = true;
    if (timeout_timer + 250 <= millis()) rxdata.ZERO(); // after so many cycles of no packets send default signals to actuators...
    //if (rxdata.TimeOut()) rxdata.ZERO(); // after so many cycles of no packets cut the power...
  }
```

These lines of code are split into two parts. The first part is fairly self explanatory in that it looks for data input from *tackle_sensor* to see if it has been triggered. The second part looks at the incoming XBee packet from the controlling Xbox controller. It also includes several lines for the case of debugging mode.



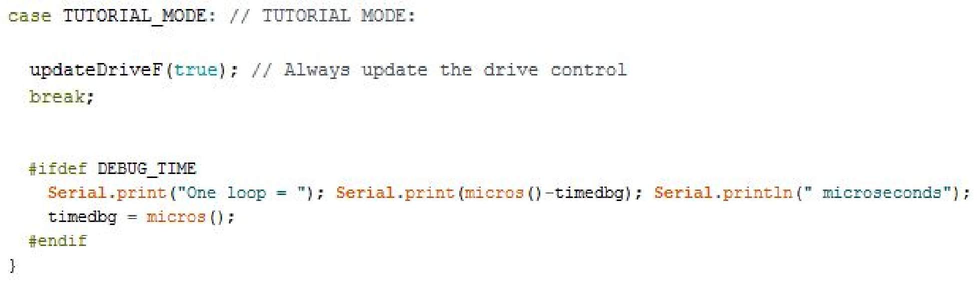
```
// Part 3. Select Your Mode
switch (rxdata.get(MODE,2)+1) {

  case MANUAL_MODE:
    updateDriveF(false); // Always update the drive control
    break;

  case AUTOMATIC_MODE:
    updateDriveF(false); // Always update the drive control
    break;

  case CALIBRATION_MODE:
```

The three cases outlined here are simply to select the operating mode that it functions in. Automatic and Manual mode are similar in that they constantly use the *updateDriveF* and update the drive control. The calibration mode has several commented lines of code associated with it for the purpose of calibrating the button inputs to certain actions of the robot. The commented lines are not shown but the functional code is.



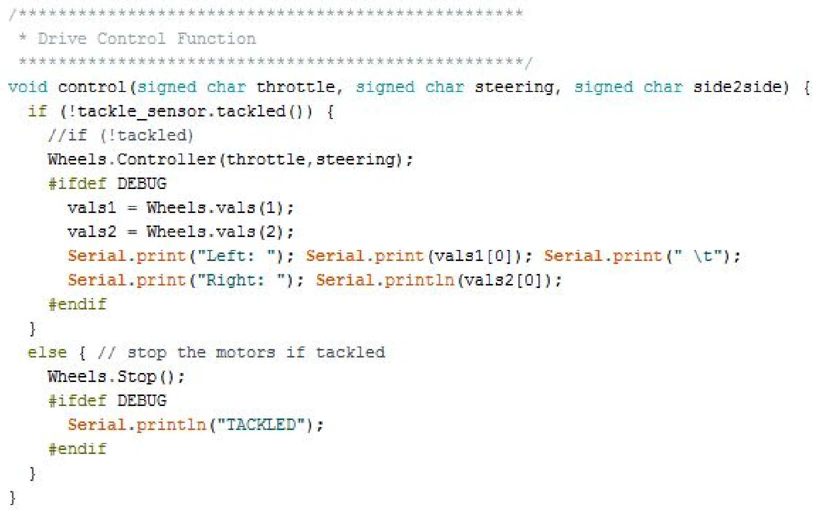
```
// Just enable the analog driving feature in this mode
Wheels.Controller(rxdata.getJoystick(LY),rxdata.getTrigger(L2));
if (rxdata.getTrigger(R2)) { // calibrate the reverse offset if R2 is pressed
  Wheels.cdata.fwd_rev = CALIBRATE_REVERSE;
  Wheels.cdata.left_right = CALIBRATE_MOTOR_REVERSE;
} else { // calibrate the forward offset if R2 is not pressed
  Wheels.cdata.fwd_rev = CALIBRATE_FORWARD;
  Wheels.cdata.left_right = CALIBRATE_MOTOR_FORWARD;
}

// Deal with the Buttons
if (rxdata.ButtonPress(A)) Wheels.Calibrate(  -1  ,  Wheels.cdata.fwd_rev);
if (rxdata.ButtonPress(Y)) Wheels.Calibrate(   1  ,  Wheels.cdata.fwd_rev);
if (rxdata.ButtonPress(X)) Wheels.Calibrate(  MOTOR_LEFT   ,  Wheels.cdata.left_right);
if (rxdata.ButtonPress(B)) Wheels.Calibrate(  MOTOR_RIGHT  ,  Wheels.cdata.left_right);
if (rxdata.ButtonPress(START)) Wheels.Calibrate(0, CALIBRATE_SAVE_OFFSETS);
#ifdef DEBUG
  mo = Wheels.offs();
  Serial.println(); Serial.print("fwd: "); Serial.print(mo.forward()); Serial.print(" \trev: ");
  Serial.print(mo.reverse()); Serial.print(" \tdir: "); Serial.println(mo.direct());
#endif
break;
```

The last mode mentioned in the code is a tutorial mode.



```
case TUTORIAL_MODE: // TUTORIAL MODE:

  updateDriveF(true); // Always update the drive control
  break;


  #ifdef DEBUG_TIME
    Serial.print("One loop = "); Serial.print(micros()-timedbg); Serial.println(" microseconds");
    timedbg = micros();
  #endif
}
```

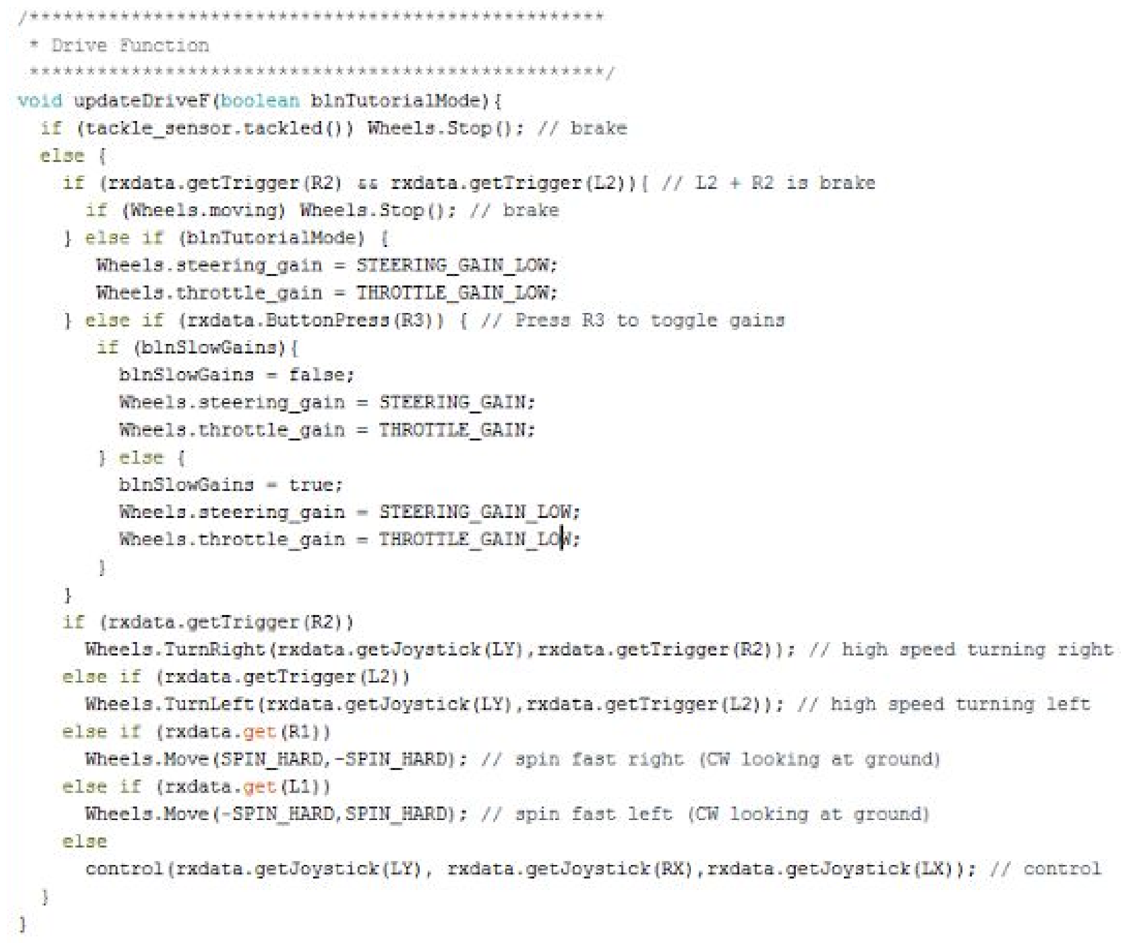
The latter half of the above section of code is for the debugger mode.



```
/***************************************************
 * Drive Control Function
 ***************************************************/
void control(signed char throttle, signed char steering, signed char side2side) {
  if (!tackle_sensor.tackled()) {
    //if (!tackled)
    Wheels.Controller(throttle,steering);
    #ifdef DEBUG
      vals1 = Wheels.vals(1);
      vals2 = Wheels.vals(2);
      Serial.print("Left: "); Serial.print(vals1[0]); Serial.print(" \t");
      Serial.print("Right: "); Serial.println(vals2[0]);
    #endif
  }
  else { // stop the motors if tackled
    Wheels.Stop();
    #ifdef DEBUG
      Serial.println("TACKLED");
    #endif
  }
}
```

The above code uses a function previously defined as *control*. Here it is fleshed out in detail. The first line of code checks to see if the tackle sensor has been activated. If not, it uses the function

Wheels.Controller() to drive the wheels of the robot. If the tackle sensor is triggered, then the function forces the wheels to come to a stop.

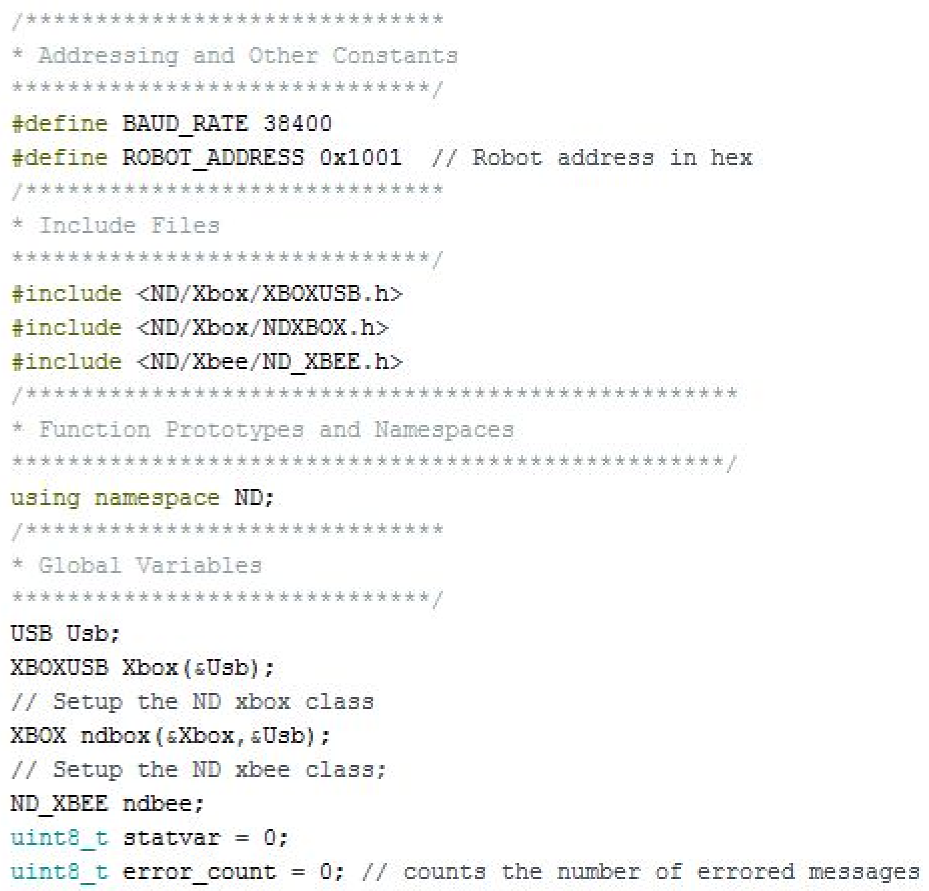


```
/**************************************************
 * Drive Function
 **************************************************/
void updateDriveF(boolean blnTutorialMode){
  if (tackle_sensor.tackled()) Wheels.Stop(); // brake
  else {
    if (rxdata.getTrigger(R2) && rxdata.getTrigger(L2)){ // L2 + R2 is brake
      if (Wheels.moving) Wheels.Stop(); // brake
    } else if (blnTutorialMode) {
       Wheels.steering_gain = STEERING_GAIN_LOW;
       Wheels.throttle_gain = THROTTLE_GAIN_LOW;
    } else if (rxdata.ButtonPress(R3)) { // Press R3 to toggle gains
       if (blnSlowGains){
         blnSlowGains = false;
         Wheels.steering_gain = STEERING_GAIN;
         Wheels.throttle_gain = THROTTLE_GAIN;
       } else {
         blnSlowGains = true;
         Wheels.steering_gain = STEERING_GAIN_LOW;
         Wheels.throttle_gain = THROTTLE_GAIN_LOW;
       }
    }
    if (rxdata.getTrigger(R2))
      Wheels.TurnRight(rxdata.getJoystick(LY),rxdata.getTrigger(R2)); // high speed turning right
    else if (rxdata.getTrigger(L2))
      Wheels.TurnLeft(rxdata.getJoystick(LY),rxdata.getTrigger(L2)); // high speed turning left
    else if (rxdata.get(R1))
      Wheels.Move(SPIN_HARD,-SPIN_HARD); // spin fast right (CW looking at ground)
    else if (rxdata.get(L1))
      Wheels.Move(-SPIN_HARD,SPIN_HARD); // spin fast left (CW looking at ground)
    else
      control(rxdata.getJoystick(LY), rxdata.getJoystick(RX),rxdata.getJoystick(LX)); // control
  }
}
```

This second function, *updateDriveF* does a few things. The first thing it checks is if *R3* (joystick button) has been pressed down. If yes, then the robot switches to a low gain mode, which gives the controller more control at the expense of speed. Pressing *R3* again will reset back to normal gain mode. The *updateDriveF* function also checks both the left and right bumper buttons for inputs, and writes actions for the wheels that are appropriate for the button input.

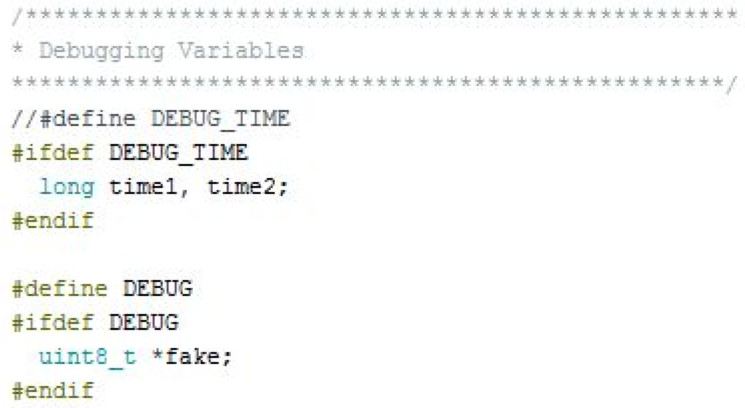
<u>**Controller Code: FinalTx_Proto.ino**</u>

This section of the Code guide aims to show the reader parts of the controller code and explain what each section does. This is the code that is uploaded onto the controller stack that is connected to the Xbox controller.



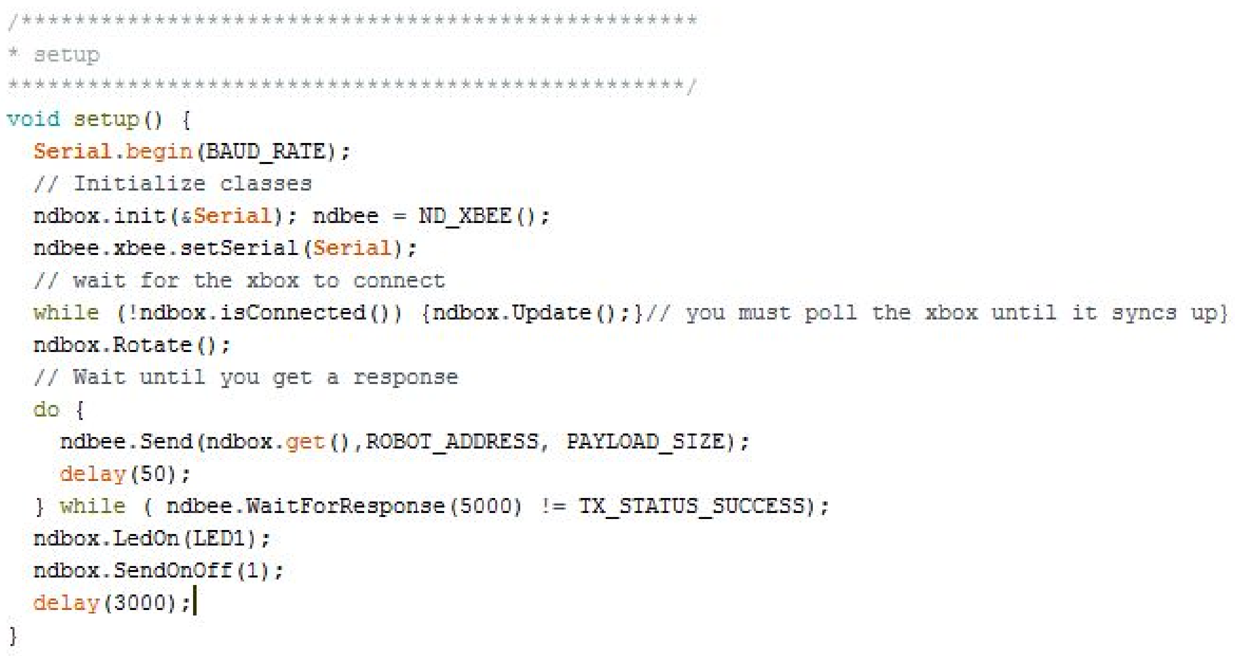
```
/******************************
* Addressing and Other Constants
******************************/
#define BAUD_RATE 38400
#define ROBOT_ADDRESS 0x1001  // Robot address in hex
/******************************
* Include Files
******************************/
#include <ND/Xbox/XBOXUSB.h>
#include <ND/Xbox/NDXBOX.h>
#include <ND/Xbee/ND_XBEE.h>
/**************************************************
* Function Prototypes and Namespaces
**************************************************/
using namespace ND;
/******************************
* Global Variables
******************************/
USB Usb;
XBOXUSB Xbox(&Usb);
// Setup the ND xbox class
XBOX ndbox(&Xbox,&Usb);
// Setup the ND xbee class;
ND_XBEE ndbee;
uint8_t statvar = 0;
uint8_t error_count = 0; // counts the number of errored messages
```

This first section of code establishes the controller's outgoing baud rate, the address of the robot it is connected to, function prototypes, and the different header file inclusions needed for this Arduino code. It also creates different global variables and classes.



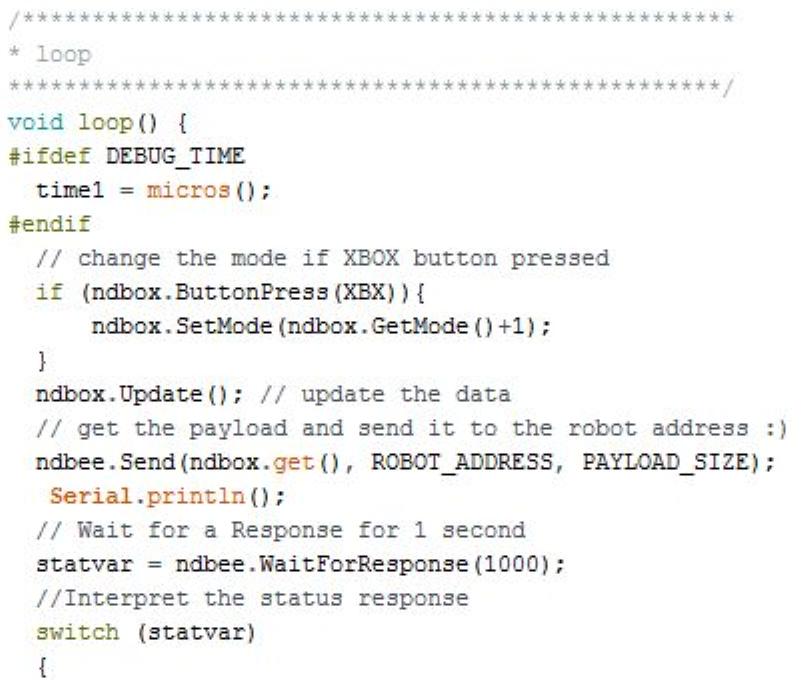
```
/**************************************************
* Debugging Variables
**************************************************/
//#define DEBUG_TIME
#ifdef DEBUG_TIME
  long time1, time2;
#endif

#define DEBUG
#ifdef DEBUG
  uint8_t *fake;
#endif
```

Once again, the relevant debugging section of the code is included but not discussed in this tutorial.



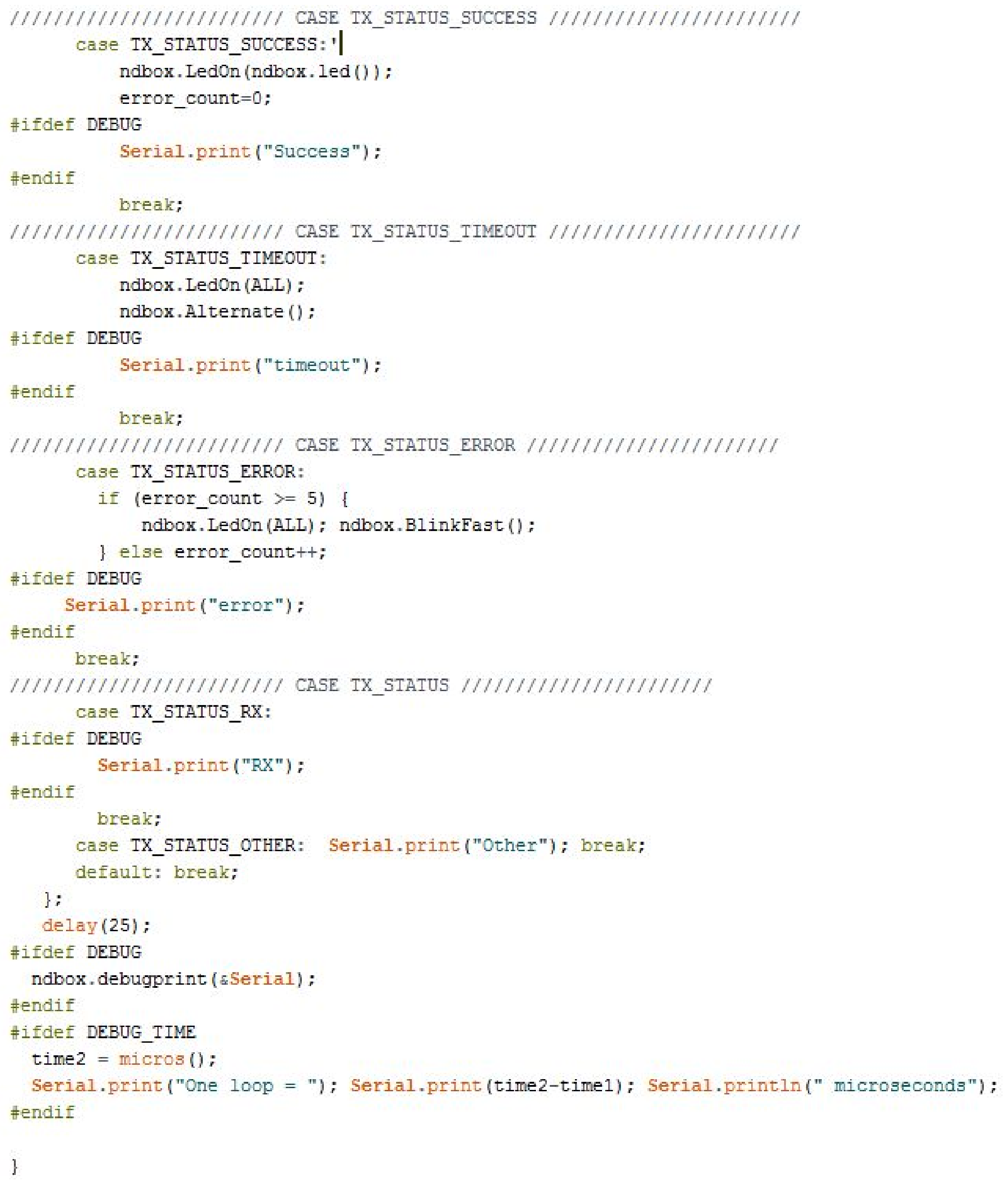
```
/**************************************************
* setup
**************************************************/
void setup() {
  Serial.begin(BAUD_RATE);
  // Initialize classes
  ndbox.init(&Serial); ndbee = ND_XBEE();
  ndbee.xbee.setSerial(Serial);
  // wait for the xbox to connect
  while (!ndbox.isConnected()) {ndbox.Update();}// you must poll the xbox until it syncs up}
  ndbox.Rotate();
  // Wait until you get a response
  do {
    ndbee.Send(ndbox.get(),ROBOT_ADDRESS, PAYLOAD_SIZE);
    delay(50);
  } while ( ndbee.WaitForResponse(5000) != TX_STATUS_SUCCESS);
  ndbox.LedOn(LED1);
  ndbox.SendOnOff(1);
  delay(3000);
}
```

This setup function first establishes, then initializes the serial connection between the controller and the lineman. The controller first searches for the standard base, and then waits until the standard base responds to the connection. When connection is successful, the controller has its “first player” LED light up (the top left light of the four lights that surround the center Xbox button), and then waits 3 seconds before going into the main loop.



```
/**************************************************
* loop
**************************************************/
void loop() {
#ifdef DEBUG_TIME
  time1 = micros();
#endif
  // change the mode if XBOX button pressed
  if (ndbox.ButtonPress(XBX)){
      ndbox.SetMode(ndbox.GetMode()+1);
  }
  ndbox.Update(); // update the data
  // get the payload and send it to the robot address :)
  ndbee.Send(ndbox.get(), ROBOT_ADDRESS, PAYLOAD_SIZE);
   Serial.println();
  // Wait for a Response for 1 second
  statvar = ndbee.WaitForResponse(1000);
  //Interpret the status response
  switch (statvar)
  {
```

This section of the main loop changes the functioning mode of the lineman. From the setup function, the serial data connection has already been established, and the *ndbox.Update* and *ndbee.Send* functions update the linemen based on controller inputs.



```
////////////////////////// CASE TX_STATUS_SUCCESS ///////////////////////
      case TX_STATUS_SUCCESS:'
          ndbox.LedOn(ndbox.led());
          error_count=0;
#ifdef DEBUG
          Serial.print("Success");
#endif
          break;
////////////////////////// CASE TX_STATUS_TIMEOUT ///////////////////////
      case TX_STATUS_TIMEOUT:
          ndbox.LedOn(ALL);
          ndbox.Alternate();
#ifdef DEBUG
          Serial.print("timeout");
#endif
          break;
////////////////////////// CASE TX_STATUS_ERROR ///////////////////////
      case TX_STATUS_ERROR:
        if (error_count >= 5) {
            ndbox.LedOn(ALL); ndbox.BlinkFast();
        } else error_count++;
#ifdef DEBUG
     Serial.print("error");
#endif
      break;
////////////////////////// CASE TX_STATUS ///////////////////////
      case TX_STATUS_RX:
#ifdef DEBUG
        Serial.print("RX");
#endif
        break;
      case TX_STATUS_OTHER:  Serial.print("Other"); break;
      default: break;
   };
   delay(25);
#ifdef DEBUG
  ndbox.debugprint(&Serial);
#endif
#ifdef DEBUG_TIME
  time2 = micros();
  Serial.print("One loop = "); Serial.print(time2-time1); Serial.println(" microseconds");
#endif

}
```

The CASE TX_STATUS part of the code involves the time the driver has to wait until the lineman and the controller are synced up. If the connection is successful the top left light on the center of the controller will light up. If the connection is not yet established then the lights will rapidly alternate, and if the connection fails or there is some kind of error, all of the LEDs on the center button will rapidly blink in unison.

Uploading Code

After the drive code or the controller code is downloaded from your browser, it must be uploaded to its respective Arduino microcontroller. The drive code is to be uploaded to the Arduino placed within the robot, and the controller code is to be uploaded to the Arduino attached to the Xbox controller. The following steps to upload code to an Arduino are provided for the Arduino IDE (Integrated Development Environment) for both Windows and in Mac. The concepts can be applied to the IDE in other operating systems if necessary.

Windows

1. Make sure the serial cord is connecting the board to the computer.
2. Select the board type (Tools→Board→Arduino Uno). Note that the computer usually configures this for you.
3. Select the COM port (Tools→Serial Port→COM #).
4. If an XBee shield is attached to the board, make sure that the switch is moved towards 'USB' or 'DLINE' and not 'XBEE' or 'UART'.
5. Upload the code by clicking the right arrow in the upper left corner next to the verify button (checkmark).
6. Wait until IDE is done uploading and the program should be running on the Arduino.

Macintosh

1. Make sure the serial cord is connecting the board to the computer.
2. Select the board type (Tools→Board→Arduino Uno). Note that the computer usually configures this for you.
3. Select the COM port (Tools→Serial Port→COM #). The COM # will either be '/dev/cu.usbmodemfa121' or the '/dev/cu.usbmodemfa131' for MacBook Pro's depending on which usb port you are using.
4. If an XBee shield is attached to the board, make sure that the switch is moved towards 'USB' or 'DLINE' and not 'XBEE' or 'UART'.
5. Upload the code by clicking the right arrow in the upper left corner next to the verify button (checkmark).
6. Wait until IDE is done uploading and the program should be running on the Arduino.

Common Arduino IDE Problems

There are many quirks when programming in the Arduino IDE. When first starting the Arduino IDE, the first compilation takes a longer time than subsequent compilations so do not be impatient during the first compilation. Here is a list of other common error messages that show up in the Arduino status window while uploading and compiling:

P: Uploading Error. *avrdude: stk500 getsync(): not in sync: resp=0x00*

S: The Xbee Shield is probably switched to 'XBEE' or 'UART'. Switch it to 'USB' or 'DLINE'.

S: This could be a number of different problems. Consult http://www.ladyada.net/learn/arduino/help.html

P: Compilation Error. *unable to rename core.a ; reason: File exists*

S: Two or more programs are trying to be compiled at once. Wait until the program finishes compiling and compile again.

P: Uploading Error. *processing.app.SerialNotFoundException: Serial port COM# not found.* This message is much longer, but this is how it begins. This error also may come in the form of the Arduino IDE asking to upload to a different COM port if the COM ports are available.

S: The IDE does not recognize the COM port. Follow the list below:

1. Select the COM port (Tools→Serial Port→COM #). If the COM port is not there,
2. 2. Turn off all wireless communication to that Arduino (turn controller off), and make sure the Xbee shield is switched to USB (if applicable).
3. Unplug the serial cord to cut off power to the board and then reattach the serial cord.
4. Select the COM port (Tools→Serial Port→COM #). If the COM port is still not recognized,
5. Close all instances of the Arduino IDE and restart the program. If the COM port is still not recognized,
6. Restart the computer and reopen the Arduino IDE. If the COM port is still not recognized, consult Google.

There are some different uploading errors that occur occasionally that seem to not make sense. Try uploading the code again to see if the error persists. Usually this will just go away. In the case of any problem, always try Googling it because there are many forums and resources available.