Kinect Chapter 6. The Tilt Motor, LED, and Accelerometer

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/kinect/; only important fragments are described here.]

The Kinect is more than just a collection of cameras, as Figure 1 shows.

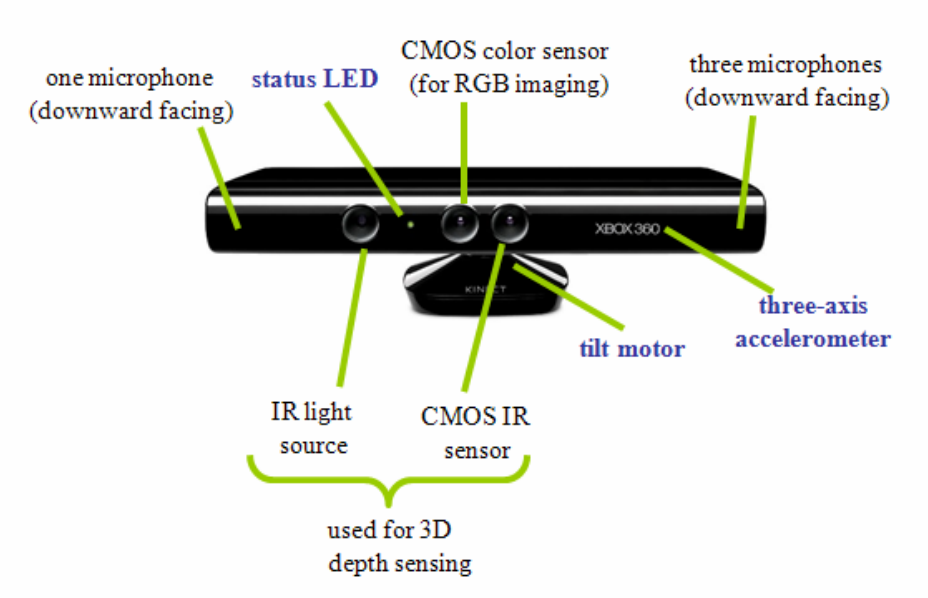

Figure 1. The Kinect Sensor.

Up to now, I've focused on the color and IR sensors, but this chapter is about the less important, but still useful, tilt motor, status LED, and accelerometer.

These controls should be managed by OpenNI's "Kinect Motor" driver, but it doesn't currently support them. A popular alternative is to install the "Xbox NUI Motor" driver from the OpenKinect libfreenect library. The steps involved in this are clearly explained in Den Delimarky's DZone article "Kinect drivers can be inter-changed for experimentation purposes" at http://dotnet.dzone.com/articles/kinect-drivers-can-be-inter. The principle drawback is the need to deal with both OpenNI and OpenKinect on one machine, *and* their Java wrappers, just to access a few additional functions. Also, some wrappers don't actually include methods for accessing the tilt motor, LED, and accelerometer.

My approach is (arguably) simpler and more flexible. I'll utilize libusb-win32 to create my own driver for the Kinect motor, and then communicate with it using a libusbjava wrapper. It's simpler because I've already used libusb-win32 and libusbjava twice before (for the missile launcher in NUI chapter 4 ?? and the robot arm in NUI chapter 6 ??). It's more flexible because I get to choose what features to include in my MotorCommunicator class.

If you haven't looked at NUI chapters 4 and 6 ??, then it's worthwhile reading at least the early sections of chapter 4 which introduce USB and its support in Java.

1. Creating a Libusb Driver for the Kinect Motor

I need the Kinect motor's vendor and product IDs, which I can find using USBDeview (free from http://www.nirsoft.net/utils/usb_devices_view.html). Figure 2 shows USBDeview's information for the motor.

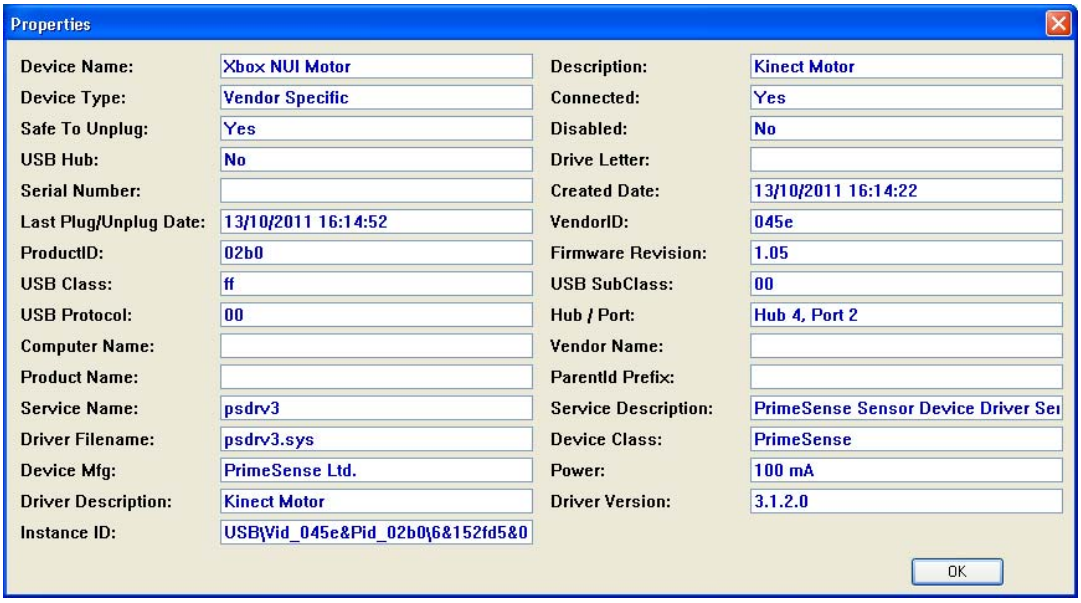

Figure 2. USBDeview's Details on the Kinect Motor.

The important entries for me are the VendorID and ProductID hexadecimals: 045e and 02b0 (about a third of the way down the two columns).

The three elements involved in my libusbjava interface to the motor are shown in Figure 3.

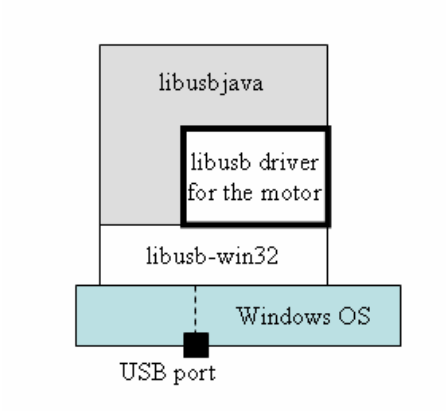

Figure 3. The libusb Driver for the Kinect Motor.

LibusbJava relies on libusb-win32, a Windows port of a widely used USB library for Linux. The current version can be downloaded from http://sourceforge.net/apps/trac/libusb-win32/wiki.

The libusb-win32 bin\ directory contains a inf-wizard.exe tool, which I can employ to create a INF file for my libusb-win32 motor driver. inf-wizard.exe starts by listing all the devices connected to the PC, in terms of their vendor ID, product ID, and device description (see Figure 4).

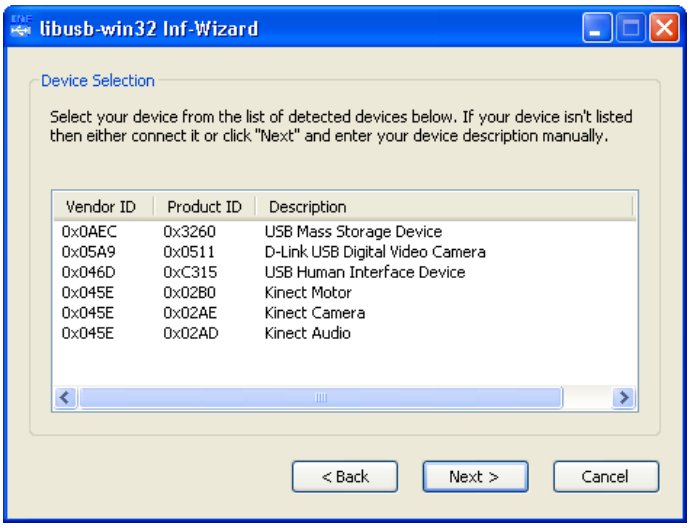

Figure 4. The inf-wizard.exe Application.

I know which entry to choose by referring to the vendor and product IDs I obtained from USBDeview, although it's quite obvious by looking at the descriptions in this case. I also need to invent a name for my driver (I chose "My Kinect Motor"), and then inf-wizard.exe generates an INF file for it (as shown in Figure 5).

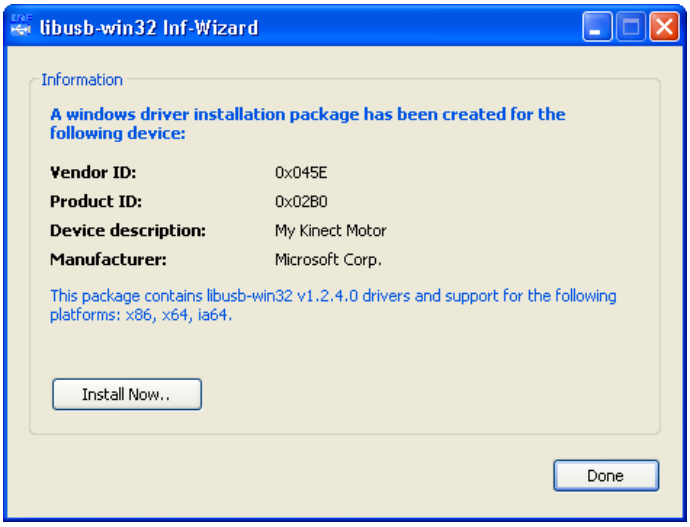

Figure 5. Installing my libusb Driver.

Once installed, the driver shows up in the "libUSB-win32 Devices" category in Window's device manager (Figure 6). Note that the Kinect must be plugged into the PC at this stage.

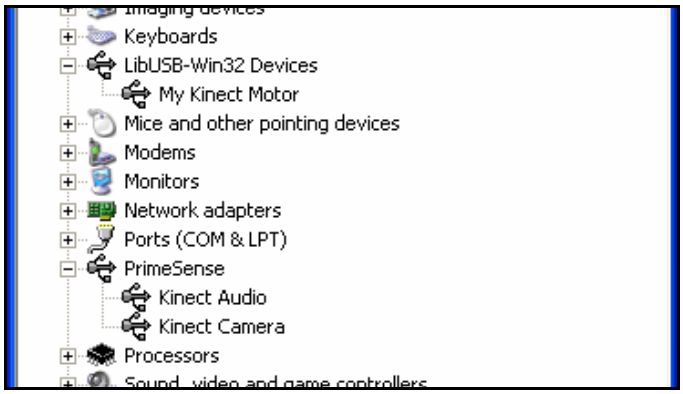

Figure 6. Device Manager Showing the Three Kinect Drivers.

Figure 6 also lists the OpenNI drivers under a "PrimeSense" heading; its Kinect Motor entry has disappeared, replaced by my libusb-win32 driver.

2. Using LibusbJava

The LibusbJava library can be downloaded from http://libusbjava.sourceforge.net/wp/. The necessary files are a JAR (ch.ntb.usb-0.5.9.jar) and a zipped DLL (LibusbJava_dll_0.2.4.0.zip). I placed the JAR and unzipped DLL in a directory on my c:\ drive (c:\libusbjava\), but anywhere is fine.

The JAR includes a number of test applications, the simplest being a viewer for all the libusb-win32 USB devices connected to the machine. Figure 7 shows its information on the Kinect motor.

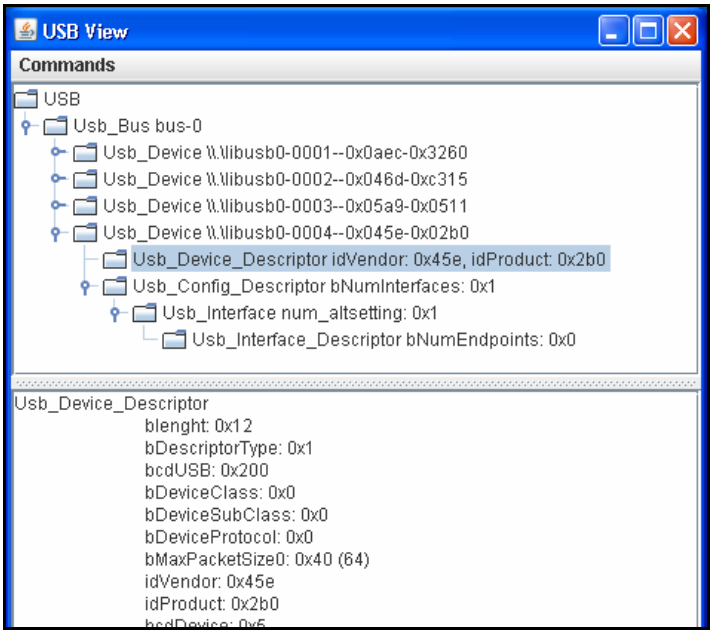

Figure 7. The USB View Application.

The viewer (which is part of ch.ntb.usb-0.5.9.jar) is started using the command line:

```
java -Djava.library.path="c:\libusbjava"
     -cp "c:\libusbjava\ch.ntb.usb-0.5.9.jar;."
                                ch.ntb.usb.usbView.UsbView
```

Of special note is the fact that the device is listed as having 0 endpoints (on the last row of the top panel in Figure 7). An endpoint is a USB communication channel. However, the viewer is being somewhat misleading since every USB device has an endpoint 0 configured for control transfers.

A great advantage of using LibusbJava is the simplicity of opening and closing a device. A short program for using the Kinect motor is shown below. It finds the device, opens it, and then immediately closes it:

```
// globals
private static final short VENDOR_ID = (short)0x045e;
private static final short PRODUCT_ID = (short)0x02b0;


public static void main(String[] args)
{
  // find the device using the vendor and product IDs
  Device dev = USB.getDevice(VENDOR_ID, PRODUCT_ID);
  if (dev == null) {
    System.out.println("Device not found");
    System.exit(1);
  }

  System.out.println("Opening device");
  try {
    dev.open(1, 0, -1);
    System.out.println("Opened device");
  }
  catch (USBException e) {
    System.out.println(e);
    System.exit(1);
  }

  System.out.println("Closing device");
  try {
    if (dev != null)
      dev.close();
  }
  catch (USBException e) {
    System.out.println(e);
  }
}  // end of main()
```

Unfortunately, there's a problem when I try to run this MotorTest.java program:

```
> java -cp "d:\LibUsbJava\ch.ntb.usb-0.5.9.jar;."
       -Djava.library.path="d:\LibUsbJava;." MotorTest
Opening device
ch.ntb.usb.USBException: No USB endpoints found. Check the device
```



```
configuration
```

The program failed to open the device because LibusbJava believes that the motor has no endpoints. The library doesn't recognize the control endpoint 0.

I'm not the first person to have this problem with LibusbJava; Sivan Toledo came across it when he was programming an AVR microcontroller (http://sivantoledotech.wordpress.com/2010/09/14/controlling-an-si570-from-java-and-matlab/). He downloaded the LibusbJava sources from its Subversion repository at http://libusbjava.sourceforge.net, and fixed things.

The problem arises because of a test for maximum packet size in updateMaxPacketSize() in LibusbJava's Device class. The test can be commented out without affecting the rest of the library. The libusbjava developer knows about this issue, and will fix the library in the future. For now, I've included a modified version of the ch.ntb.usb-0.5.9.jar in the downloadable code for this chapter. After replacing the original JAR by this new version, MotorTest.java executes as expected:

```
> java -cp "d:\LibUsbJava\ch.ntb.usb-0.5.9.jar;."
       -Djava.library.path="d:\LibUsbJava;." MotorTest
Opening device
Opened device
Closing device
```

3. Motor Protocol Discovery

The Kinect motor communicates via a single control endpoint, which means that all the messaging between Java and the device can be carried out in terms of control transfers. This only requires a single libusbJava method from the Device class:

```
public int controlMsg(int requestType,
                      int request, int value, int index,
                      byte[] data, int size,
                      int timeout,boolean reopenOnTimeout)
               throws USBException
```

I utilized Device.controlMsg() extensively in NUI chapters 4 and 6 ??, so that's another reason for reading them. In NUI chapter 4, I worked out what values had to be passed to the controlMsg() calls by listening in on control transfers between my laptop and the missile launcher with the USBTrace analyzer. In NUI chapter 6, I employed the freeware SnoopyPro (http://sourceforge.net/projects/usbsnoop/) instead.

The good news is that I don't need protocol analyzers this time, since all the relevant information have already been collected in the OpenKinect wiki, in the entries on protocol documentation (http://openkinect.org/wiki/Protocol_Documentation) and USB devices (http://openkinect.org/wiki/USB_Devices).

I'll start by looking at how to change the LED status light and operate the tilt motor. Then I'll describe how to access the motor's current state, and extract it's status, tilt angle (if it's stationary) and tilting speed (if it's moving). I'll also read the current (x, y, z) accelerometer values.

4. Setting the Tilt Motor, LED, and Accelerometer

Before I can write to (or read from) the motor, LED, or accelerometer, the motor has to be initialized. This involves similar code to MotorTest.java. The main difference is that the details are hidden inside a MotorCommunicator class, which is called like so:

```
public static void main(String[] args)
{
  MotorCommunicator motor = new MotorCommunicator();
       // find and open the motor

  // use the motor. . .

  motor.close();
}  // end of main()
```

The MotorCommunicator constructor finds and opens the device:

```
private static final short VENDOR_ID = (short)0x045e;
private static final short PRODUCT_ID = (short)0x02b0;

private Device dev = null;   // to communicate with USB device


public MotorCommunicator()
{
  System.out.println("Looking for device: (vendor: " +
                       toHexString(VENDOR_ID) +
       "; product: " + toHexString(PRODUCT_ID) + ")");
  dev = USB.getDevice(VENDOR_ID, PRODUCT_ID);
  if (dev == null)
    System.out.println("Device not found");

  try {
    System.out.println("Opening device");
    dev.open(1, 0, -1);
    System.out.println("Opened device");
  }
  catch (USBException e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of MotorCommunicator()
```

An optional extra is to test if the motor is ready for communication by sending it the control transfer arguments listed in Table 1.

request type	request	value	index	byte array	array size
0xC0	0x10	0x0	0x0	empty 1-element array	1

Table 1. Control Transfer Arguments for the isReady Test.

If the device is ready to communicate, then the byte array's single element will be assigned 0x22.

The control transfer involves a call to libusbJava's Device.controlMsg(), which is hidden away inside my MotorCommunicator.sendMessage():

```
private int sendMessage(int requestType, int request, int value,
                            byte[] data, int size)
// send a control transfer; the byte array may be modified
{
  int rval = -1;
  try {
    rval = dev.controlMsg(requestType, request, value, 0,
                                    data, size, 2000, false);
    if (rval < 0)
      System.out.println("Control Transfer Error (" +
            rval + "):\n  " +   LibusbJava.usb_strerror() );
  }
  catch (USBException e) {
    System.out.println(e);
  }
  return rval;
}  // end of sendMessage()
```

sendMessage() handles exceptions, and also fixes the control transfer's index and timeout settings.

The readiness check can be implemented as a boolean function which calls sendMessage() with arguments taken from Table 1:

```
public boolean isReady()
{
  byte[] buf = new byte[1];   // one-element empty array
  int rval = sendMessage(0xC0, 0x10, 0, buf, 1);
  if (rval == -1)
    return false;
  return (buf[0] == 0x22);    // 0x22 means the motor is ready
}  // end of isReady()
```

MotorCommunicator.isReady() should be called in main() after the constructor has opened the device:

```
public static void main(String[] args)
{
  MotorCommunicator motor = new MotorCommunicator();
                    // find and open the motor
  if (!motor.isReady()) {
    System.out.println("Kinect motor not ready");
    System.exit(-1);
  }

  // use the motor. . .

  motor.close();
```



```
}  // end of main()
```

4.1. Closing down the Motor

MotorCommunicator closes the device by accessing the global dev object:

```
public void close()
{
  System.out.println("Closing device");
  try {
    if (dev != null)
      dev.close();
  }
  catch (USBException e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of close()
```

4.2. Setting the LED Status Light

The status LED is set by sending the control transfer arguments shown in Table 2.

request type	request	value	index	byte array	array size
0x40	0x06	LED status integer	0x0	empty	0

Table 2. Control Transfer Arguments for Setting the LED.

The LED status integers represent different colored lights, and whether the light is blinking or not. My names for these lights are listed in Table 3.

LED Status Integer	Name
0	LED_OFF
1	LED_GREEN
2	LED_RED
3	LED_ORANGE
4	LED_BLINK_ORANGE
5	LED_BLINK_GREEN
6	LED_BLINK_RED_ORANGE

Table 3. LED Status Integers and their Names.

Blinking green is used by the Kinect to indicate that the device is ready for use, and so your code should finish by resetting the LED to that color.

The most direct mapping of the light values into Java is as an enumeration:

```
public enum LEDStatus
{
  LED_OFF(0),
  LED_GREEN(1),
  LED_RED(2),
  LED_ORANGE(3),
  LED_BLINK_ORANGE(4),
  LED_BLINK_GREEN(5),
  LED_BLINK_RED_ORANGE(6);

  private int code;

  private LEDStatus(int c)
  {  code = c; }

  public short getCode()
  { return (short)code;  }

}  // end of LEDStatus class
```

Setting the LED involves mapping a status name to its integer code, and then calling MotorCommunicator.sendMessage() with the arguments shown in Table 2.

```
public void setLED(LEDStatus status)
{
  System.out.println("Setting LED to " + status);
  sendMessage(0x40, 0x06, status.getCode(), new byte[1], 0);
}
```

In main(), MotorCommunicator.setLED can be called like so:

```
motor.setLED(LEDStatus.LED_BLINK_RED_ORANGE);
    :
    :  // at the end of programming
motor.setLED(LEDStatus.LED_BLINK_GREEN);
```

4.3. Tilting the Kinect

The tilt range of the Kinect motor is +31 degrees (up) and -31 degrees (down). If you try to tilt the Kinect beyond these limits, the motor will stop.

One oddity associated with setting the angle is that the control transfer must be supplied with a value that's double what you require.

Another quirk is that the Kinect treats the angle as being relative to the *horizon*, not its base, probably because it uses its accelerometers to control the rotation. For instance, if you activate the tilt motor while skiing down a mountain, 0 degrees means the horizon, not parallel to the slope.

People have measured the angle change and it's not particularly accurate. Also, the tilting speed is fairly slow, and the motor is quite tiny with fragile plastic gears. You can see pictures of them at iFixit's Kinect teardown at http://www.ifixit.com/Teardown/Microsoft-Kinect-Teardown/4066/1 The quality is similar to the motors and gears used in the robot arm in NUI chapter 6 ??.

The tilt angle is set by sending the control transfer arguments shown in Table 4.

request type	request	value	index	byte array	array size
0x40	0x31	2 * desired angle	0x0	empty	0

Table 4. Control Transfer Arguments for Tilting.

This is implemented in setAngle() by a suitable call to sendMessage():

```
// globals
// tilt range: + is up; - is down; 0 is straight forward
private static final double MAX_ANGLE = 31;
private static final double MIN_ANGLE = -31;


private void setAngle(int angle)
{
  System.out.println("Rotate to angle " + angle);
  if ((angle < MIN_ANGLE) || (angle > MAX_ANGLE))
    System.out.println("Angle outside tilt range: " + angle);
  else
    sendMessage(0x40, 0x31, (short)(2*angle), new byte[1], 0);
}  // end of setAngle()
```

5. Reading the Motor's State

The motor state can be retrieved as a 10-byte array containing its speed (if the motor is moving), its accelerometer readings in the x-, y-, and z- directions, its current title angle (if the motor is stationary), and a status code. The location of this data in the array is shown in Figure 8.

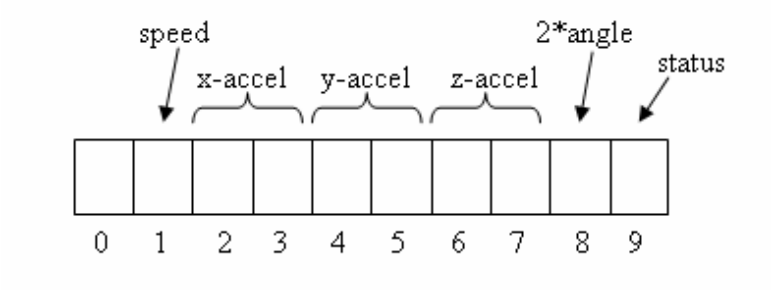

Figure 8. The Motor State Byte Array.

Each of the accelerometer values is spread over two bytes. It seems that the first element of the array isn't used for anything.

The state information is accessed by sending the Kinect the control transfer arguments shown in Table 5.

request type	request	value	index	byte array	array size
0x40	0x32	0x0	0x0	empty 10-element array	10

Table 5. Control Transfer Arguments for Accessing the Motor State.

An empty 10-element byte array is passed to the Kinect, which fills it with the state information, formatted as in Figure 8. This is implemented by getMotorInfo() using a call to sendMessage():

```
public byte[] getMotorInfo()
{
  byte[] buf = new byte[10];
  sendMessage(0xC0, 0x32, 0, buf, 10);   // don't test result
  return buf;
}  // end of getMotorInfo()
```

getMotorInfo() is called by other get methods that access parts of the state information in buf[], as explained in the following sub-sections.

5.1. What is the Motor's Current Status?

There appears to be five status codes, although the number varies depending on which documentation you read. I've encoded them as an enumeration called MotorStatus, assigning a name to each of the status codes.

```
public enum MotorStatus
{
  STOPPED(0), AT_LIMIT(1), MOVING(4), QUICK_BREAK(8), UNKNOWN(-1);

  private int code;

  private MotorStatus(int c)
  {  code = c; }

  public int getCode()
  { return code;  }


  public static MotorStatus of(int code)
  // convert status code into a MotorStatus object
  {
    switch (code) {
      case 0: return STOPPED;
```



```
      case 1: return AT_LIMIT;
      case 4: return MOVING;
      case 8: return QUICK_BREAK;
      default: return UNKNOWN;  // any other int means unknown
    }
  }  // end of of()

}  // end of MotorStatus class
```

If the Kinect is moving when the state information is assembled, then the status code will be 4. Alternatively, the motor may be stationary, and so returns 0. There are two 'error' situations which may cause the motor to stop moving: 1 means that the motor has tilted to its maximum (or minimum) extent and has to stop, while 8 indicates that the motor has encountered some kind of obstacle, so will take a short time-out before trying to move again. An unknown state is reported as -1.

The static method MotorStatus.of() converts an integer into a MotorStatus object. MotorStatus.of() is utilized by the MotorCommunicator.getStatus() method:

```
public MotorStatus getStatus()
{
  byte[] buf = getMotorInfo();
  int status = (int) buf[9];
  return MotorStatus.of(status);
}
```

5.2. Reading the Tilt Angle

The current tilt angle is stored in the 9th element of the byte array returned by getMotorInfo() (see Figure 8). To be precise, the value is 2*angle, so getAngle() has to divide the data by two.

```
public int getAngle()
{
  byte[] buf = getMotorInfo();

  if (buf[8] == -128) {
    System.out.println("Angle unavailable since Kinect is moving");
    return buf[8];
  }
  return ((int)buf[8])/2;
}  // end of getAngle()
```

The angle is only available when the tilt motor has stopped, and is set to -128 if the Kinect is still moving. An important point is that the angle is measured relative to the horizon, not the Kinect's base

5.3. Reading the Accelerometer

The iFixit teardown of the Kinect (http://www.ifixit.com/Teardown/Microsoft-Kinect-Teardown/4066/2) identifies the accelerometer as probably being a Kionix MEMS KXSD9 (http://www.kionix.com/accelerometers/accelerometer-KXSD9.html).

In the KXSD9, external accelerations move a silicon structure causing a change in capacitance. This change is converted into integer 'acceleration counts' for the x-, y, and z- axes, with 819 counts equal to 1g of acceleration.

When the Kinect is stationary, the x- and z- axis acceleration counts will be 0 (or very close to 0), while the y-axis acceleration count will be reported as 819 (or very close to 819), due to gravity of 1g.

The Kinect spreads each of its three acceleration counts value over two bytes in the byte array returned by getMotorInfo() (see Figure 8), so getAccel() does some bit manipulation to convert the byte pairs into integers.

```
public int[] getAccel()
// acceleration counts
{
  byte[] buf = getMotorInfo();
  int[] accel = new int[3];   // for x, y, z acceleration counts
  // each acceleration count is stored in a byte pair
  accel[0] = (int) (((short)buf[2] << 8) | buf[3]);  // x
  accel[1] = (int) (((short)buf[4] << 8) | buf[5]);  // y
  accel[2] = (int) (((short)buf[6] << 8) | buf[7]);  // z
  return accel;
}  // end of getAccel()
```

getAccel() returns the three integer accelerations in an array, which is a bit ugly but does the job.

The acceleration counts are converted into g-force accelerations in getAccelG() by dividing them by an ACCEL_COUNT constant (819):

```
// global
public static final double ACCEL_COUNT = 819.0;    // counts/g


public double[] getAccelG()
// accelerations as real g forces
{
  int[] accel = getAccel();
  double[] accelG = new double[3];
  accelG[0] = accel[0]/ACCEL_COUNT;  // x
  accelG[1] = accel[1]/ACCEL_COUNT;  // y
  accelG[2] = accel[2]/ACCEL_COUNT;  // z
  return accelG;
}  // end of getAccelG()
```

5.4. Reading the Kinect's Speed

The speed of the Kinect's servo motor is available in the 2nd element of the byte array (see Figure 8), which is retrieved by getSpeed():

```
public int getSpeed()
{ byte[] buf = getMotorInfo();
  return (int) buf[1];
}
```


6. Detecting Tilting Limits

If the sensor is tilted beyond its limits relative to its base, the motor will stall and stop turning. This can't be good for the hardware, and should be avoided.

Unfortunately, the current angle returned in the byte array is relative to the *horizon* because the Kinect uses its accelerometers to calculate rotation. This means that there's no sure way to decide if a base-related limit has been reached by looking only at the value returned by getAngle().

To detect if a rotation limit has been reached, we can examine the motor status via getStatus() to see if it is AT_LIMIT. This isn't sufficient either since AT_LIMIT doesn't distinguish between the positive and negative rotation limit, and often isn't detected (at least by my Kinect). One way around this might be to execute the rotation operation anyway, and check the Kinect's servo motor speed, via getSpeed(). If the requested rotation could not be carried out then the resulting speed will be 0 because of motor stalling. However, this strategy will put strain on the servo and its gears. Also, my Kinect almost always reports a 0 speed, even when it is rotating correctly.

Probably the simplest solution is to assume that the Kinect **is** positioned on a flat surface, and so its base **is** parallel to the horizon. In that case, the angle passed to setAngle() can be compared with the MAX_LIMIT and MIN_LIMIT values (31 and -31) to decide whether a rotation should be carried out. This is how my setAngle() method is implemented.

7. Testing the MotorCommunicator

The MotorCommunicator class contains a short main() function which illustrates how to use various get/set methods, and also acts as a test-rig for my code.

A connection is opened to the Kinect, which is signaled by changing its status LED to blinking red/orange. The code enters a loop, which waits for the user to input an angle for rotating the Kinect. While a rotation is in progress, various state information is reported.

The user finishes the session by typing 'q', and the Kinect light goes back to blinking green.

```
public static void main(String[] args)
{
  MotorCommunicator motor = new MotorCommunicator();
  if (!motor.isReady()) {
    System.out.println("Kinect motor not ready");
    System.exit(-1);
  }

  motor.setLED(LEDStatus.LED_BLINK_RED_ORANGE);

  System.out.println("Enter an angle (range -31 to 31):");
  String line = null;
  try {
    BufferedReader reader = new BufferedReader(
                                new InputStreamReader(System.in));
```



```
    System.out.print(">> ");
    while ((line = reader.readLine()) != null) {
      if (line.length() == 0)
        break;
      if (line.charAt(0) == 'q')
        break;
      else {
        try {
          motor.setAngle( Integer.parseInt(line) );
          motor.printMotorInfo();
          System.out.println("status: " + motor.getStatus());
          System.out.println("speed: " + motor.getSpeed());
          System.out.println("Current angle: " + motor.getAngle());
          motor.printAccel();
          motor.printAccelG();
        }
        catch (NumberFormatException e) {
          System.out.println("\"" + line + "\" not a number.");
        }
      }
      System.out.print(">> ");
    }
    reader.close();
  }
  catch (IOException e) {
    e.printStackTrace();
  }

  motor.setLED(LEDStatus.LED_BLINK_GREEN);
  motor.close();
}  // end of main()
```