NUI Chapter 6. Controlling a Robot Arm

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

In chapter 4, I developed a controller for a toy missile launcher, utilizing libusb-win32 and LibusbJava. I'll be using the same libraries again here, to control the OWI-535 robotic arm (http://www.owirobot.com/products/Robotic-Arm-Edge.html) shown in Figure 1.

Figure 1. The OWI-535 Robotic Arm with USB Interface.

To be precise, I'm using the UK version of the arm, available from Maplin Electronics (http://www.maplin.co.uk/robotic-arm-kit-with-usb-pc-interface-266257). It's the same product but the box includes a OWI USB interface, which is ***not*** part of the basic OWI-535 package.

One downside of the arm is its under-powered control software which only offers time-based movement of the arm's joints (e.g. rotate the wrist for 0.5 second). It would be much more useful to define joint rotations using absolute and relative angles, or to specify (x, y, z) locations (e.g. move the arm's grippers to (10, 15, 7)).

I'll be implementing control software that offers all these features. The code forms a loose hierarchy – the lowest level moves the arm using USB control transfers while the top-tier utilizes coordinates. Each level is implemented using capabilities supplied by the next level down, as illustrated in Figure 2.

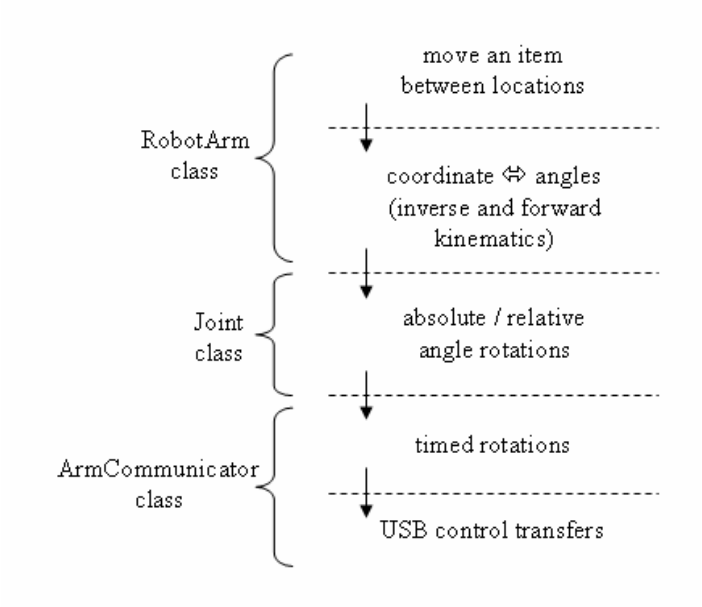

Figure 2. The Robot Arm Functionality Hierarchy.

The USB control transfer software at the lowest level is quite similar to the missile launcher code from chapter 4 (so I won't be explaining all the details again). However, for the protocol detection work, I turned to the popular freeware, SnoopyPro (http://sourceforge.net/projects/usbsnoop/), instead of the commercial USBTrace tool from chapter 4; the main reason for changing was to illustrate an alternative form of sniffing.

At the second level, sequences of control transfers are employed to implement timed rotation requests for the arm's joints. This is sufficient for programming simple user interactions with the arm, via my ArmCommunicator class.

My Joint class utilizes ArmCommunicator to define joint rotations using absolute angles, and as offsets from the current position. This requires the utilization of rotational timing information, and the storage of joint orientations.

Forward kinematics employs joint orientations and link dimensions (e.g. the length of the upper arm) to calculate the location of the robot arm's gripper. Far more useful is inverse kinematics (IK), which calculates the joint rotations necessary to move the gripper to a specified position (e.g. move to (-9, 7, 5)). IK is a formidable problem for most robot arm designs, but quite simple for the OWI.

With the availability of kinematics information, the arm's command language can specify movements in terms of where to pick up and deposit an object.

The classes that implements the hierarchy are shown in Figure 3.

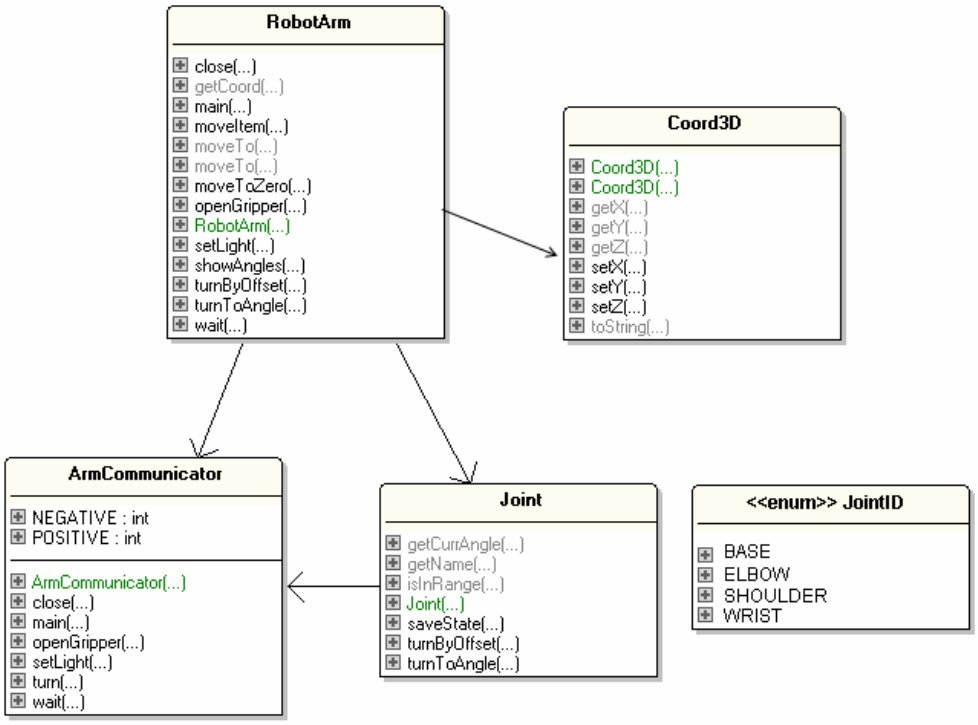

Figure 3. Class Diagrams for the Robot Arm.

The USB control messages and timed rotations are managed by the ArmCommunicator class. A Joint instance is created for each of the arm's four rotational joints. Each object maintains its joint's current orientation so that it can support rotations employing absolute and relative angles. Each joint has an ID, defined in the JointID enumeration.

The top-level RobotArm class implements forward and reverse kinematics as moveTo() methods that utilize 3D coordinates (created using the Coord3D class).

In the following sections, I'll give more technical details about the arm, explain how SnoopyPro was used to discover the arm's USB communication protocol, and explain the ArmCommunicator, Joint, and RobotArm classes in detail.

1. The OWI-535 Robot Arm

As I mentioned in the introduction, if you do buy a OWI-535 arm (http://www.owirobot.com/products/Robotic-Arm-Edge.html), make sure to also purchase its USB interface (a small circuit board, cable, and CD). I obtained mine from Maplin Electronics in the UK (http://www.maplin.co.uk/robotic-arm-kit-with-usb-pc-interface-266257) for a very reasonable 35 UK pounds. It seems that most online electronic hobbyist sites stock the bundle, such as *Planet Robotic* (http://www.planetrobotic.com/content/robotic-arm-edge-bundle-owi-535-kit-and-usb-interface). The driver software may not include a Windows 7 version, but the

OWI Robotics website has drivers free for download at
http://www.owirobot.com/pages/Downloads.html

The arm is cheap because it's sold as a kit, which worried me a bit since I'm a rather useless “maker”. In fact, its construction is no harder than a typical plastic model or Lego kit involving gears. No soldering is necessary, and the assembly guide is very clear. There's also a five-part “Robotic Arm USB Kit Assembly” video online at http://www.youtube.com/watch?v=0r4ZoNprt5Q.

My OWI arm is shown in Figure 4, and Figure 5 is a simplified schematic that labels the joints and limb dimensions.

Figure 4. My OWI Arm
(Viewed from the Right)

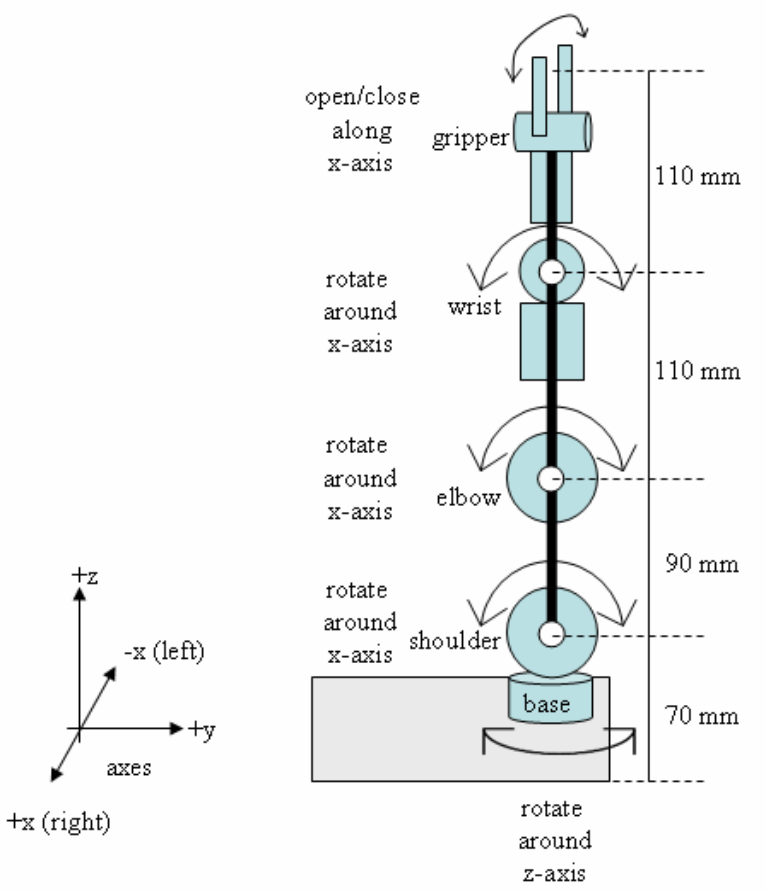

Figure 5. Schematic of the Robot Arm
(Viewed from the Right).

The arm has four rotational joints, which I'll be calling the base, shoulder, elbow and wrist. The base rotates the arm around the vertical z-axis, while the other three rotate it around the x-axis. The positive x-axis is pointing out of the page in Figure 5, and I'll also refer to it as on the "right". No joint rotates around the y-axis, which limits the arm's movements but also makes the kinematics calculations later in this chapter much easier.

Each joint has a rotation limit -- in the backwards and forwards directions for the wrist, elbow and shoulder, and to the left and right for the base, which will become important later when I implement rotations using angle values.

The arm's gripper opens and closes via rotating gearwheels, but its prongs are connected to the wheels in such a way that they stay roughly parallel to each other as they move. Since the gripper's movement is so restricted, I won't treat it as a joint. Figure 6 shows the gripper viewed from the origin, along the positive y-axis.

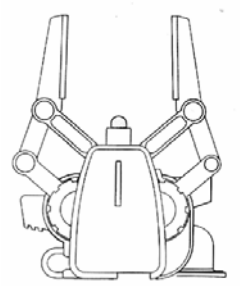

Figure 6. The Gripper
(Viewed from the Origin, down the +y Axis).

The distance between the joints is shown in millimeters in Figure 5. The distance from the wrist to the gripper ends at the midpoint of the gripper prongs when they're closed.

If you're looking for precision-tooled robotics hardware, with movements accurate to millimeters, then the OWI isn't for you. But, considering the price, OWI Robotics have produced an amazingly fun product.

One problem is that the gear wheels tend to slip due to the weight of the arm, or if the arm is rotated into a hard obstacle, or beyond a joint's rotation limit. Gravity also means that the arm's rotational velocity isn't constant -- for example, rotating the shoulder joint downwards takes less time than rotating it upwards by the same amount. Another factor is the battery power supply – as the batteries fade, so does the arm's speed.

The arm has no feedback mechanism, either for sensing when the gripper has closed around something, or to judge the joints' true positions. Fortunately, a number of makers have come to the rescue, adding a range of sensors to the basic arm, such as optocouplers (http://www.instructables.com/id/Modifications-to-Robot-Arm-for-Opto-Coupler-Feedba/) and potentiometers (http://www.youtube.com/watch?v=ArEjPsYxM4A). Incidentally, YouTube is a great resource for OWI ideas – typing "OWI robotic arm" into its search engine returns over 50 interesting results as of mid 2011.

2. Becoming a USB Detective Again

I'll assume that you've read chapter 4 about my 'sleuthing' of the Dream Cheeky Missile Launcher. I'll be going through many of the same steps again, so won't report all the gory details. I'll employ USBDeview (http://www.nirsoft.net/utils/usb_devices_view.html) to find the arm's vendor and product IDs, but use SnoopyPro (http://sourceforge.net/projects/usbsnoop/) rather than USBTrace (http://www.sysnucleus.com/) for protocol discovery. One reason for the change is that SnoopyPro is freeware, and another is that several people emailed me after I'd written chapter 4 asking me how to use SnoopyPro. Its biggest drawback is a lack of documentation but, to be fair, it's so widely used that most of its problems/confusions have been documented online somewhere (if you can find them).

A general headache-reducing strategy for USB protocol discovery and programming is to use *two* machines. The first is for detective work only, running the OWI software and USBDeview and SnoopyPro. The second machine is only for USB programming using libusb-win32 and LibusbJava. The OWI USB driver should also be installed (it's called the ELAN USB device in the documentation), and a libusb-win32 driver generated for it using the libusb inf-wizard.exe tool. The resulting INF file should be installed, and the machine restarted. This libusb-win32 ELAN driver can now be accessed via Java code using the LibusbJava library.

Before I can start coding on machine #2, I need to gather ID and protocol details from machine #1. Figure 7 shows the USBDeview list of connected devices.

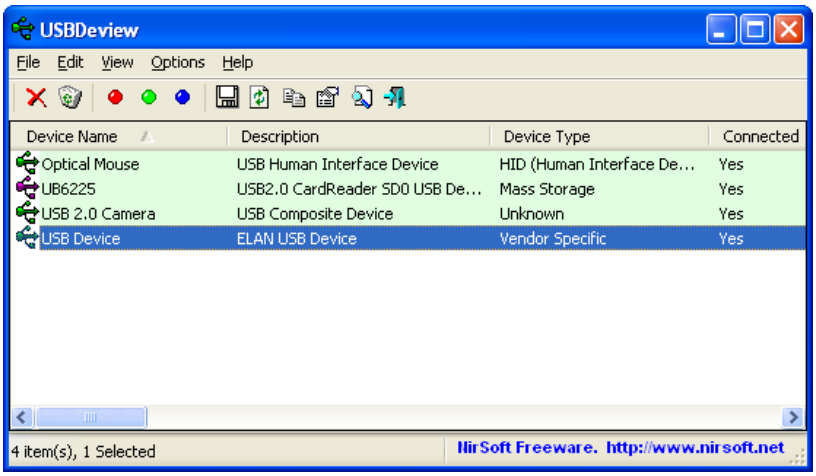

Figure 7. Connected USB Devices in USBDeview.

Double clicking on the "ELAN USB Device" row brings up a table of details shown in Figure 8.

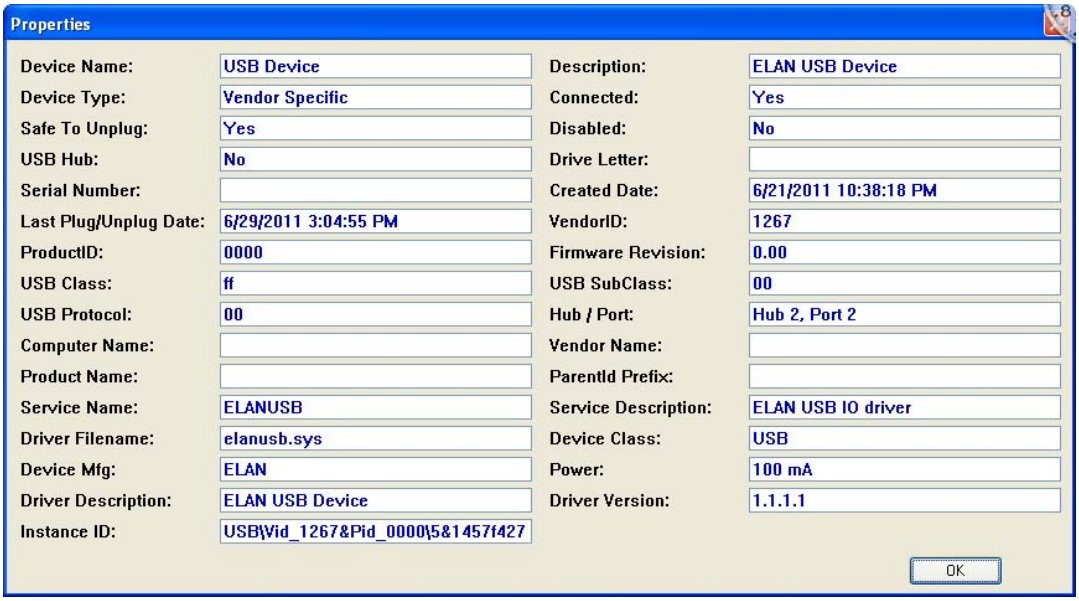

Figure 8. USBDeview's Details on the ELAN USB Device.

The important entries are the VendorID and ProductID hexadecimals: 0x1267 and 0x0000 (or 0 for short). A note should also be made of the devices description: "ELAN USB Device".

SnoopyPro is available from http://iyacht.blogspot.com/2007/04/usb-sniffer-for-windows.html. The documentation is accessed via its "Help" menu. Another useful information source is a blog post by the developer at http://iyacht.blogspot.com/2007/04/usb-sniffer-for-windows.html.

A list of USB devices (both connected and disconnected) can be displayed by typing F2 (or by selecting the "View>>USB Devices" menu item). before doing anything else, it's necessary to unpack and install the SnoopyPro drivers. Select 'Unpack Drivers' and then 'Install Service' from the 'File' menu of the devices dialog. If the installation is successful, then the text "Snpys bridge is present and accessible" will appear at the top left of the dialog window (see Figure 9), otherwise it will read "Snpys bridge is not available".

Now it's time to attach SnoopyPro to the ELAN driver; scroll down through the long list of USB devices looking in the Description column for "ELAN USB Device", right-click on the row and choose 'Install and Restart' from the pop-up menu. The result should be something like Figure 9.

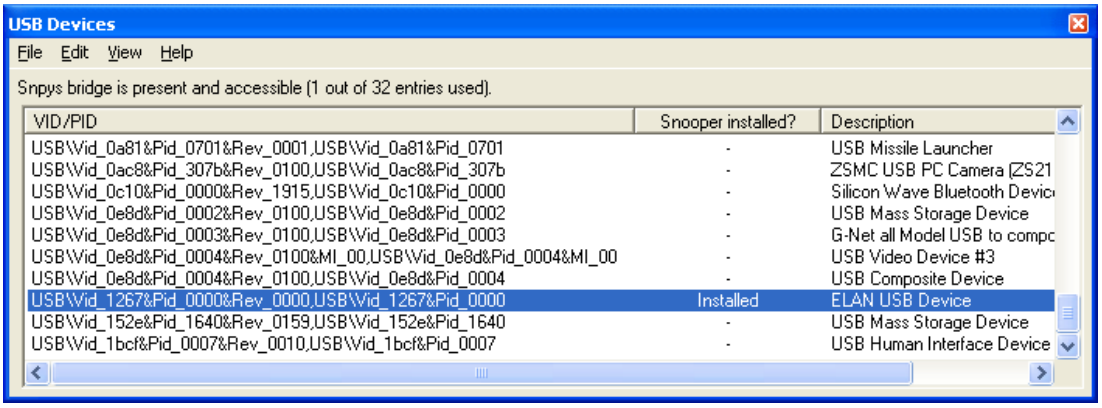

Figure 9. SnoopyPro Listening to the ELAN USB Device.

The "USB Devices" window in Figure 9 can now be closed, revealing that the SnoopyPro main window is recording a log, probably called "USBLog1". The window displays the number of packets logged so far, and the vendor and product ID details of the device. However, none of the packets are shown until the window's pause or stop button is pressed. Figure 10 shows some output after I had briefly rotated the arm's shoulder joint backwards using the OWI GUI.

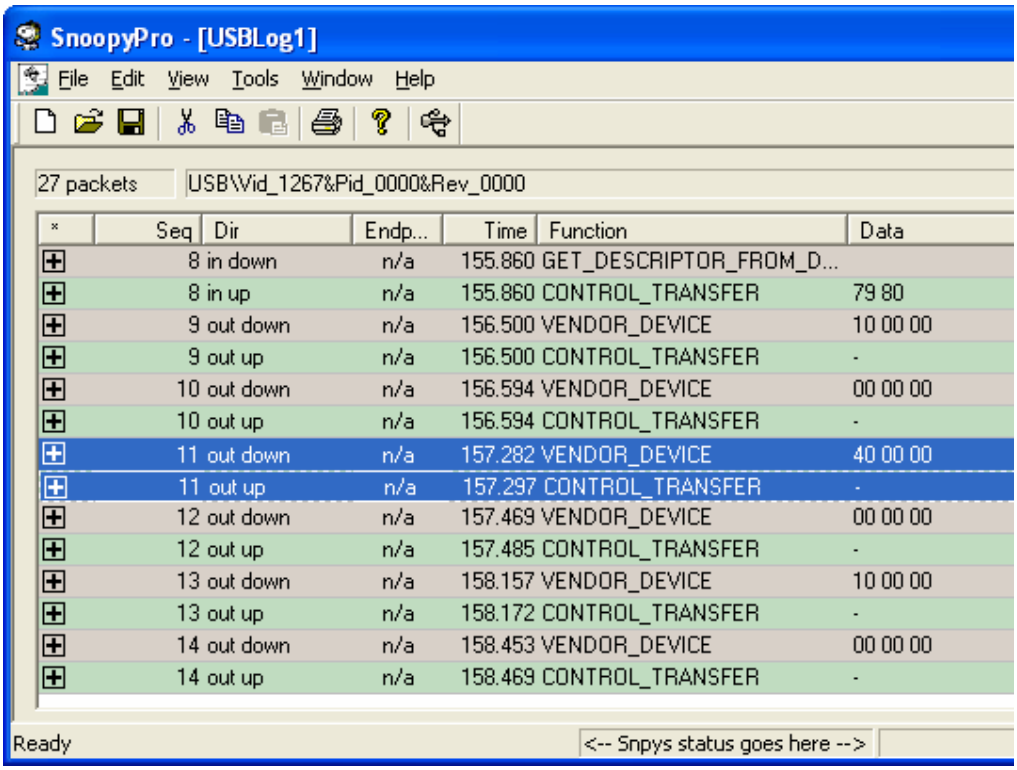

Figure 10. SnoopyPro USB Packet Information for the Arm.

Protocol discovery usually means working out the format of the "out" packets sent from the PC to the device. However, SnoopyPro displays two types of "out" packet -- "out down" and "out up" (highlighted in Figure 10). These appear in pairs, and should be considered as a *single* "out" packet. This is hinted at by the way that the sequence number of each "out down" and "out up" pair are the same (e.g. the highlighted pair in Figure 10 both have the sequence number 11).

The difference between "out down" and "out up" is illustrated by Figure 11.

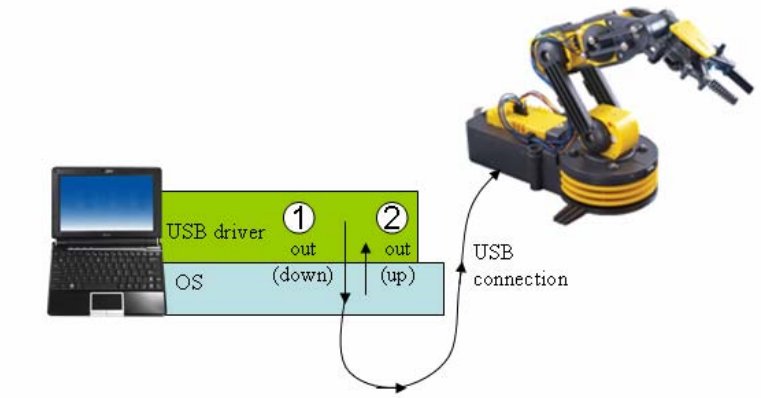

Figure 11. "out down" and "out up" USB Communication.

"out down" is a request sent from the USB driver to the OS to have data delivered to the device. "out up" is a message sent back from the OS to the driver to inform it that the data has been delivered.

An "out down" will contain information about the data sent to the arm, while an "out up" will contain details about the data's packet header. Confusingly, the "out down" will also contain packet header details, but they'll probably be changed by the OS before the packet is dispatched to the device. Any header changes will be shown in the "out up" information.

Figure 12 shows the sequence 11 "out down" packet from Figure 10.

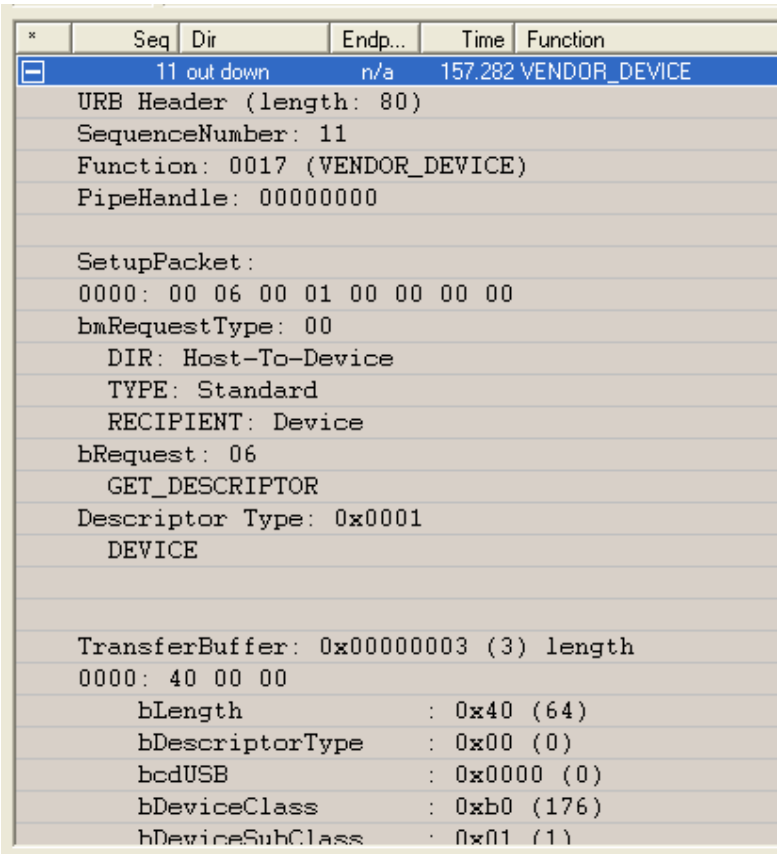

Figure 12. The Sequence 11 "out down" Packet.

The packet data consists of three bytes, "40 00 00", listed on the line after "Transfer Buffer". The header information consists of eight bytes, "00 06 00 01 00 00 00 00", listed on the line after "SetupPacket". Ignore it, because switching to the Sequence 11 "out up" packet in Figure 13 shows that the header was changed by the OS.

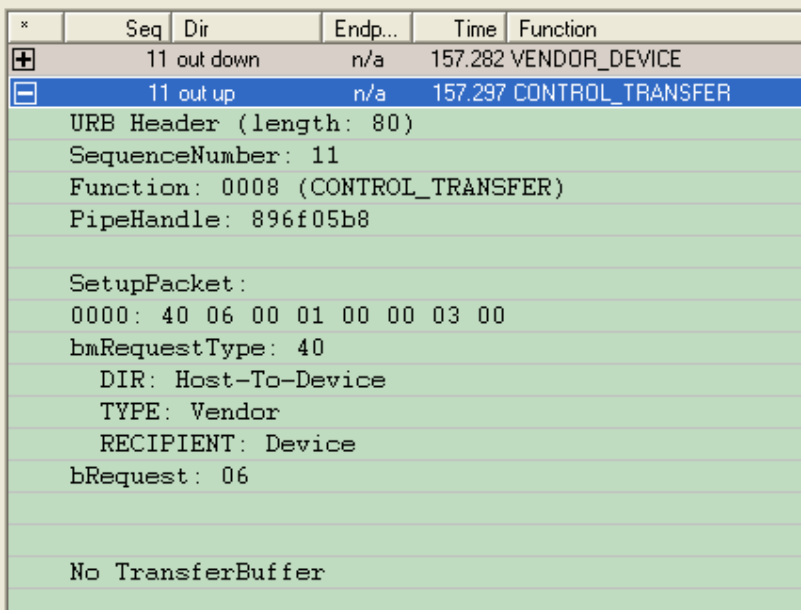

Figure 13. The Sequence 11 "out up" Packet.

The packet header is now "40 06 00 01 00 00 03 00", which is what I need for coding.

2.1. Using the USB Header

I'll be sending USB control transfers using the LibusbJava Device. controlMsg() method. Its signature is:

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

The LibusbJava API is online at http://libusbjava.sourceforge.net/wp/res/doc/, or refer back to chapter 4 for more details.

Values for the requestType, request, value, index, and size fields can be obtained from the eight byte packet header (e.g. "40 06 00 01 00 00 03 00" from Figure 13). The bytes required for the data[] array come from looking at the packet data (e.g. "40 00 00" from Figure 12). The values for Device.controlMsg()'s timeout integer and boolean are up to the programmer.

A typical packet header has a format like that shown in Figure 14.

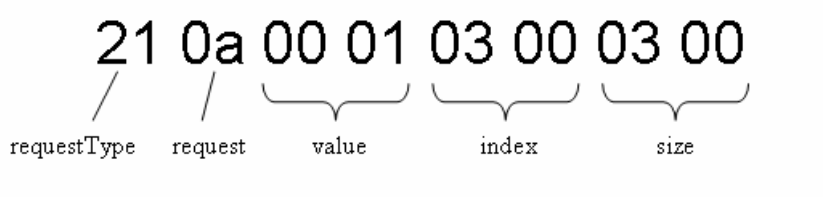

Figure 14. The Packet Header Format.

But there's a nasty catch -- USB data is sent in little endian format, so the correct way of reading the double-byte value, index, and size fields is to *reverse* their bytes. In other words, for the example in Figure 14, value is 0x0100, index is 0x0003, and size is 0x0003.

I can now return to the "out up" packet in Figure 13, where the header bytes were "40 06 00 01 00 00 03 00". Most of the controlMsg() arguments can be filled in, apart from the data[] array:

```
dev.controlMsg(0x40, 0x06, 0x0100, 0, data, 3, 2000, false);
```

I've simplified the index and size fields from 0x0000 and 0x0003 to 0 and 3.

I've set the Device.controlMsg() timeout to 2 seconds (2000 milliseconds) and disabled the reopen flag.

2.2. Using the USB Data

I know from the header field that the data consists of three bytes (i.e. size == 3), and this is confirmed by the TransferBuffer information in Figure 12: "40 00 00".

As to what these bytes actually mean, it's necessary to keep rotating different joints in the arm, noting down the corresponding data packets sent to the device. "40 00 00" is sent when the shoulder joint is rotated backwards.

It turns out that each byte in the three-byte group is used in a separate way. The first byte deals with rotating the wrist, elbow, and shoulder joints, and opening and closing the gripper, as summarized in Table 1.

Byte Value	Arm Command.
00	Stop
01	Gripper close
02	Gripper open
04	Wrist backwards
08	Wrist forwards
10	Elbow backwards
20	Elbow forwards
40	Shoulder backwards
80	Shoulder forwards

Table 1. Byte 1 Values and Commands.

A rotation command only stops when 00 is sent in byte 1. If a 00 isn't sent then eventually the joint will reach its rotation limit and keep slipping. It's possible to combine commands by bit manipulation, thereby making multiple joints move at once. For example if byte 1 is 11 than the gripper will close and the elbow rotate backwards together. However, this is of limited use since the stop command (00) stops every operation being carried out by byte 1; you can't stop the gripper without also stopping the elbow.

Byte 2 deals with the base joint, as summarized by Table 2.

Byte Value	Arm Command
00	Stop base
01	Base turns right (clockwise)
02	Base turns left (counter-clockwise)

Table 2. Byte 2 Values and Commands.

This separation of the base into its own byte means it can be rotated and stopped independently of the other joints.

Byte 3 is used to turn the gripper light on and off, as shown in Table 3.

Byte Value	Arm Command
00	Light off
01	Light on

Table 3. Byte 3 Values and Commands.

These tables define how that data[] array, passed to Device.controlMsg() should be filled in. My sendControl() method takes three joint data arguments, and builds a three-byte data array (bytes[]), as shown below:

```
// global
private Device dev;    // the robot arm


private static void sendControl(int opCode1,
                                int opCode2, int opCode3)
// send a USB control transfer
{
  byte[] bytes = { new Integer(opCode1).byteValue(),
                   new Integer(opCode2).byteValue(),
                   new Integer(opCode3).byteValue()
```



```
                };
  try {
    int rval = dev.controlMsg(
                  USB.REQ_TYPE_DIR_HOST_TO_DEVICE |
                  USB.REQ_TYPE_TYPE_VENDOR,    //0x40,
                  0x06, 0x0100, 0,
                  bytes, bytes.length, 2000, false);
    if (rval < 0) {
      System.out.println("Control Error (" + rval + "):\n  " +
                           LibusbJava.usb_strerror() );
    }
  }
  catch (USBException e) {
    System.out.println(e);
  }
}  // end of sendControl()
```

I've replaced the requestType byte in controlMsg() by LibUsbJava USB constants, and also obtained the size of the data array from the array itself rather than hardwiring "3" into the code.

In the following code fragment, sendControl() is combined with a timeout to rotate the shoulder backwards for 1.5 seconds.

```
sendControl(0x40, 0, 0);   // shoulder rotates backwards
try {
  Thread.sleep(1500);      // for 1.5 secs
}
catch(InterruptedException e) {}
sendControl(0, 0, 0);      // then arm stops
```

This paring of sendControl() calls with a timeout is so common that it's worth wrapping it up in its own sendCommand() method:

```
private void sendCommand(int opCode1, int opCode2,
                                int opCode3, int period)
// execute the operation for period millisecs;
// first version
{
  sendControl(opCode1, opCode2, opCode3);
  wait(period);
  sendControl(0, 0, 0);    // stop arm
}  // end of sendCommand()


public void wait(int ms)
// sleep for the specified no. of millisecs
{
  try {
    Thread.sleep(ms);
  }
  catch(InterruptedException e) {}
}  // end of wait()
```

Now the 1.5 second shoulder rotation can be encoded as:

```
sendCommand(0x40, 0, 0, 1500);
```


3. Implementing Timed Rotations

My ArmCommunicator class utilizes versions of the sendCommand() and sendControl() methods from the last section, and includes additional methods that hide calls to sendCommand() so the user doesn't need to remember the byte values in Tables 1-3.

The ArmCommunicator() constructor uses the arm's vendor and product IDs to open a link to the device.

```
// globals
private final static short VENDOR_ID = (short)0x1267;
private final static short PRODUCT_ID = (short)0x0;


public ArmCommunicator()
{
  dev = USB.getDevice(VENDOR_ID, PRODUCT_ID);
  try {
    System.out.println("Opening device");
    dev.open(1, 0, 0);
    // open device with configuration 1, interface 0
    // and no alt interface
  }
  catch (USBException e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of ArmCommunicator()
```

Device.open() requires configuration and interface numbers, which can be discovered by running the textlibusb-win tool (see chapter 4). The alternative interface value can be 0 or -1.

3.1. Managing the Light

When the gripper light is switched on (by setting the data array's third byte to 01) it should stay on until it's explicitly switched off. However, the first version of sendCommand() always sends a data array full of 0's at the end of an operation; this will stop the operation, but also turn off the light.

My solution is to use a global isLightOn boolean to record the state of the light (on or off) and use it to set the value of the third data array byte.

The boolean is manipulated with a pair of set and get methods:

```
// global state for the light
private boolean isLightOn = false;


private int getLightVal(boolean isLightOn)
{  return (isLightOn)? 0x01 : 0x00;  }
```



```
public void setLight(boolean turnOn)
{  isLightOn = turnOn;
   System.out.println("  Light is on: " + isLightOn);
   sendControl(0x00, 0x00, getLightVal(isLightOn));
}  // end of setLight()
```

Note that setLight() uses getLightVal() to determine the byte that's passed to sendControl(). This technique is also used in the second version of sendCommand():

```
private void sendCommand(int opCode1, int opCode2, int period)
// execute the operation for period millisecs;
// second version
{
  int opCode3 = getLightVal(isLightOn);
  if (dev != null) {
    sendControl(opCode1, opCode2, opCode3);
    wait(period);
    sendControl(0, 0, opCode3);    // stop arm
  }
}  // end of sendCommand()
```

The advantage is that the light is now independent of the joint control, and means that sendCommand() only needs to be called with two data command arguments.

3.2. Rotating a Joint

The principle rotation method in ArmCommunicator is turn(), which requires the user to supply a joint ID, a direction constant (POSITIVE or NEGATIVE), and a duration for the rotation.

```
// globals
public final static int POSITIVE = 0;
public final static int NEGATIVE = 1;
      // +ve for forwards/right turns; -ve for backwards/left turns


public void turn(JointID jid, int dir, int period)
{
  int opCode1 = 0x00;
  int opCode2 = 0x00;

  if (jid == JointID.BASE)
    opCode2 = (dir == POSITIVE)? 0x01 : 0x02;
  else if (jid == JointID.SHOULDER)
    opCode1 = (dir == POSITIVE)? 0x80 : 0x40;
  else if (jid == JointID.ELBOW)
    opCode1 = (dir == POSITIVE)? 0x20 : 0x10;
  else if (jid == JointID.WRIST)
    opCode1 = (dir == POSITIVE)? 0x08 : 0x04;
  else
    System.out.println("Unknown joint ID: " + jid);

  if (period < 0) {
    System.out.println("Turn period cannot be negative");
    period = 0;
```



```
  }

  sendCommand(opCode1, opCode2, period);
}  // end of turn()
```

There are four Joint IDs (one for each joint), defined as an enumerated type:

```
public enum JointID { BASE, SHOULDER, ELBOW, WRIST }
```

turn() hides of the data byte commands from the user, replacing them with more intuitive direction constants.

3.3. Opening and Closing the Gripper

The gripper isn't treated as a joint; instead a boolean specifies whether it is to be opened or closed.

```
// globals
private final static int GRIPPER_PERIOD = 1700;
                                 // ms time to open/close


public void openGripper(boolean isOpen)
{
  if (isOpen) {
    System.out.println("  Gripper: open");
    sendCommand(0x02, 0x00, GRIPPER_PERIOD);
  }
  else {
    System.out.println("  Gripper: close");
    sendCommand(0x01, 0x00, GRIPPER_PERIOD);
  }
}  // end of openGripper()
```

The GRIPPER_PERIOD value was arrived at by timing the gripper in action. openGripper()'s boolean behavior simplifies its interface, but also means that there's no way to partially or incrementally open or close it. For example, when the gripper closes around an object, the gears will continue to operate for 1.7 (GRIPPER_PERIOD) seconds even if the object is firmly grasped after a second. For the rest of the time, the gripper gears will grind away, doing nothing.

I chose this approach because the gripper has no touch feedback, and so it's not possible to have the program detect when an object is fully gripped and then cut short the closing rotation.

Another weakness is that there's no global gripper state to record whether the gripper is currently open or closed. That means that it's possible to call openGripper(true) twice, and have the gripper fully open and then try to open even more.

3.4. Recreating the Arm's Basic Programming Mode

The GUI control software for the OWI offers two programming interfaces. The 'basic' mode shown in Figure 15 uses the keyboard and mouse to move the joints.

Figure 15. The Arm's Basic Programming Mode.

There's also a 'program' mode where joints are assigned timed rotations.

The basic mode can be easily reimplemented using the ArmCommunicator class, which is useful when the arm needs to be positioned under user control.

ArmCommunicator's main() contains a loop that reads letters from the keyboard.

```
// in the ArmCommunicator class
public static void main(String[] args)
{
  ArmCommunicator arm = new ArmCommunicator();

  System.out.println("Enter a single letter command (and <ENTER>:");
  printHelp();
  Console console = System.console();
  String line = null;
  char ch;

  System.out.print(">> ");
  while ((line = console.readLine()) != null) {
    if (line.length() == 0)
      break;
    ch = line.charAt(0);
    if (ch == 'q')
      break;
    else if (ch == '?')
      printHelp();
    else
      doArmOp(ch, arm);
    System.out.print(">> ");
  }

  arm.close();
```



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

The letters are the same as those used in the OWI's basic mode. Also, '?' prints out a list of the letters, and 'q' or a new line makes the loop exit.

doArmOp() is an if-test that converts a letter into a call to ArmCommunicator.turn() or a method that affects the gripper or light.

```
// global
private static final int DELAY = 250;

private static void doArmOp(char ch, ArmCommunicator arm)
// use POSITIVE for forwards/right turns;
// NEGATIVE for backwards/left turns
{
  if (ch == 'w')           // close gripper
    arm.openGripper(false);
  else if (ch == 's')      // open
    arm.openGripper(true);

  else if (ch == 'e')      // wrist backwards
    arm.turn(JointID.WRIST, ArmCommunicator.NEGATIVE, DELAY);
  else if (ch == 'd')      // forwards
    arm.turn(JointID.WRIST, ArmCommunicator.POSITIVE, DELAY);

  else if (ch == 'r')      // elbow backwards
    arm.turn(JointID.ELBOW, ArmCommunicator.NEGATIVE, DELAY);
  else if (ch == 'f')      // forwards
    arm.turn(JointID.ELBOW, ArmCommunicator.POSITIVE, DELAY);

  else if (ch == 'u')      // shoulder backwards
    arm.turn(JointID.SHOULDER, ArmCommunicator.NEGATIVE, DELAY);
  else if (ch == 'j')      // forwards
    arm.turn(JointID.SHOULDER, ArmCommunicator.POSITIVE, DELAY);

  else if (ch == 'k')      // base left
    arm.turn(JointID.BASE, ArmCommunicator.NEGATIVE, DELAY);
  else if (ch == 'i')      // right
    arm.turn(JointID.BASE, ArmCommunicator.POSITIVE, DELAY);

  else if (ch == 'l')      // switch light on
    arm.setLight(true);
  else if (ch == 'p')      // off
    arm.setLight(false);

  else
    System.out.println("Unknown command: " + ch);
}  // end of doArmOp()
```

The rotation duration (the DELAY constant) is 250 milliseconds because it seems to produce a useful small rotation. If larger rotations are required then the user has to repeatedly enter the rotation's letter.

4. Rotating Using Angles

Timed rotations aren't that useful for programming a robot, since it's much more common to want to express a rotation in terms of an absolute angle (e.g. rotate the elbow to the +45 degrees position) or as an offset from the current orientation (e.g. rotate the wrist by -10 degrees).

This functionality is best supported by a Joint class, which is used to create four Joint instances at run time (for the base, shoulder, elbow, and wrist). The class has several features:

- each joint instance stores its current orientation;
- each joint converts angles into timed rotations;
- each joint knows the limits of its rotation, in both directions.

The gripper could also be represented as a joint, but I'll continue to follow my decision in ArmCommunicator and treat it as a binary device that is either open or closed.

I need a protractor and stopwatch to gather joint information for the class. The protractor is for measuring a joint's rotation limits, and the stopwatch times how long it takes the joint to rotate from one limit to the other. The information is summarized in Figure 16.

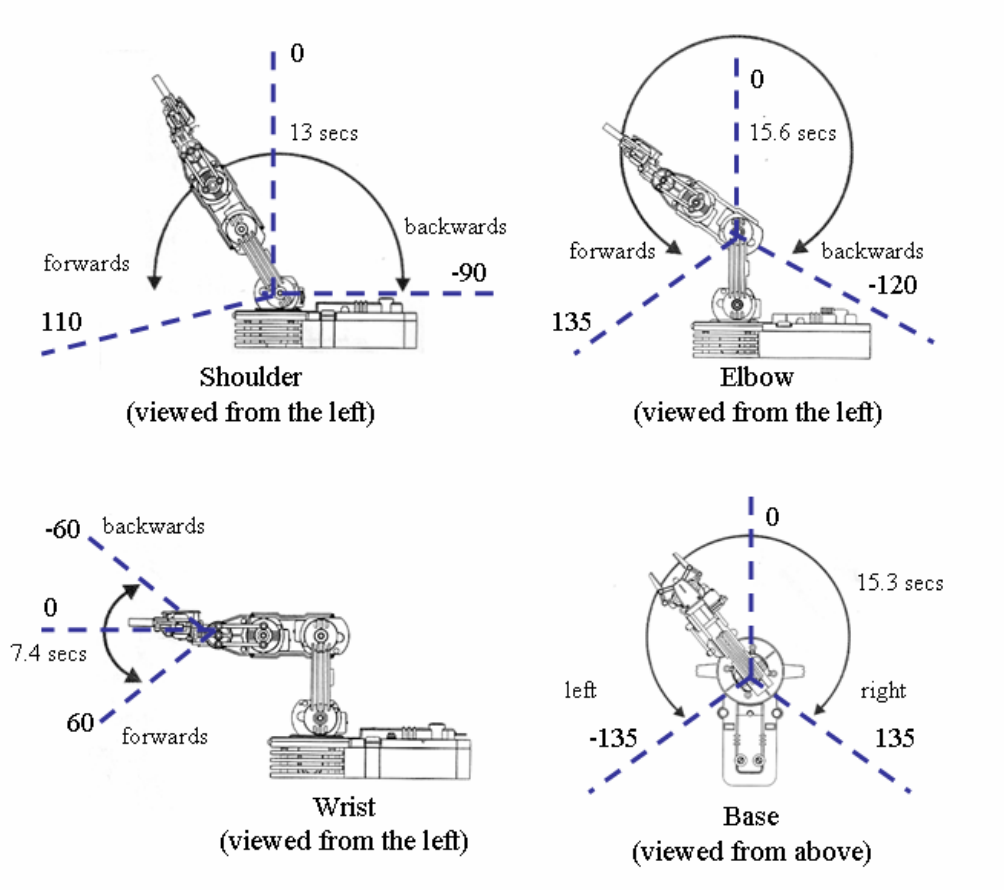

Figure 16. Joint Rotation Limits (in Degrees) and Times.

The rotation limits are specified in degrees, and the times are for a single rotation from one limit to the other. For example it takes the base 15.3 seconds to rotate from its right hand limit over to the left limit (and the same time in the other direction).

I'll use the timing and limit information to calculate a rotation *rate* for each joint:

rotation rate = ((positive angle limit) - (negative angle limit))/ rotation time

This rate is divided into a user-supplied angle offset (e.g. turn by 30 degrees) to convert it into a rotation time (30 / rotation rate).

Even while I was collecting the timing and limit values, it was clear this approach had some problems. One is that the math assumes a joint has a constant rotation rate between its two limits; this is not the case for the arm's shoulder and elbow joints. Both are affected by the arm's weight, and move noticeable slower when rotating against gravity. Fixing this is somewhat messy since the effect depends on the arm's rotation direction (forwards or backwards) and it's current position over the base.

Another problem is that the gear wheels slip when they start rotating, effectively causing the arm to move faster for the first few milliseconds. The gears also take a few milliseconds to stop moving when a joint is told to stop. These millisecond inaccuracies build up if the joints are rotated using many small angle changes as opposed to a single large adjustment.

The best solution requires hardware-based feedback, which can independently calculate the robot's rotation and/or position without relying on the gears. As I mentioned at the start of this chapter, adding sensors to the joints (e.g. potentiometers) is one approach. However, I'll be using a webcam as a feedback mechanism in chapter 18, combined with tangible drawing objects.

4.1. Storing the Joint State

Since each joint needs to maintain several pieces of state information, I decided to store it separately from the Joint class in text files for each instance. The files store the joint's rotation limits and traversal times, which are fixed, and also the joint's current orientation. This angle is written out to the file before the Joint instance terminates, and reloaded when a new instance is created later. In this way, a joint 'remembers' its position between invocations.

The data file for the shoulder, shoulderJI.txt:

```
rotate: 13000
limits: 110 -90
angle: 45
```

The first two lines are the shoulder joint information shown in Figure 16, while the angle indicates that the shoulder was turned 45 degrees forward when the joint details were last saved.

The Joint class uses readJointInfo() to load the four integers into globals:

```
// globals in the Joint class
private int rotTime;
        // time (in ms) for the joint to move between its limits
```



```
private int posLimit, negLimit;   // in degrees (negLimit is -ve)

private int currAngle;  //may range between posLimit and negLimit
```

checkInfo() performs some validity tests on the input data (e.g. is the rotation time positive?), and also calculates the rotation rate using the equation from above:

```
// global in the Joint class
private float rotRate;      // joint rotation rate (between limits)

// in Joint.checkInfo()
rotRate = ((float)(posLimit-negLimit))/rotTime;
```

4.2. Turning by an Offset Angle

Joint.turnByOffset() turns the joint by a specified offset, which may be positive or negative. A positive value is a forwards rotation for the wrist, elbow and shoulder, and to the right for the base.

```
// global
private int currAngle;


public void turnByOffset(int offsetAngle)
{
  int newAngle = withinLimits(currAngle + offsetAngle);
  timedAngleTurn(newAngle);
}
```

The offset is converted into an absolute angle, constrained between the positive and negative rotation limits. timedAngleTurn() converts the angle into a rotation time, and carries out the operation using ArmCommunicator.

```
// globals
private static final int MIN_TIME = 200;   // ms
           // don't move if time is less than MIN_TIME

private JointID jointID;
private ArmCommunicator armComms;
private int currAngle;

private float rotRate;    // joint rotation rate (between limits)


private void timedAngleTurn(int angle)
// move to angle by executing a timed rotation
{
  int offsetAngle = angle - currAngle;  //offset may be +ve or -ve

  if (offsetAngle < 0) {
    int time = Math.round( -offsetAngle/rotRate );
    if (time < MIN_TIME)
      System.out.println("  " + jointID + ": -ve turn time short (" +
                                 time + "); ignoring");
    else {
```



```
      armComms.turn(jointID, ArmCommunicator.NEGATIVE, time);
      currAngle = angle;
    }
  }
  else {    // offset is +ve
    int time = Math.round( offsetAngle/rotRate );
    if (time < MIN_TIME)
      System.out.println("  " + jointID + ": +ve turn time short (" +
                                time + "); ignoring");
    else {
      armComms.turn(jointID, ArmCommunicator.POSITIVE, time);
      currAngle = angle;
    }
  }
}  // end of timedAngleTurn()
```

timedAngleTurn() recalculates the offset of the angle relative to the current position and determines if it positive or negative so that ArmCommunicator can be called with the correct direction constant (ArmCommunicator.POSITIVE or ArmCommunicator.NEGATIVE). When the offset is converted into a rotation time, it is checked to see if it exceeds a minimum value (MIN_TIME). Small rotations are discarded because they're too brief to be carried out accurately by the arm.

4.3. Turning to an Absolute Angle

The Joint class has another rotation method, Joint.turnToAngle(), which turns a joint to the specified angle position. This is also implemented by calling timedAngleTurn().

```
public void turnToAngle(int angle)
{  timedAngleTurn( withinLimits(angle) );  }
```

5. Building a Robot Arm from Joints

My RobotArm class represents the entire robot arm, as depicted in Figure 5. It utilizes four Joint instances for the base, shoulder, elbow, and wrist, and employs ArmCommunicator to manage the gripper and light. The RobotArm() constructor shows how these elements are initialized and stored.

```
// globals
private Joint base, wrist, shoulder, elbow;
private Joint[] joints; // so can iterate through the joints
private ArmCommunicator armComms;


public RobotArm()
{
   armComms = new ArmCommunicator();

   base = new Joint(JointID.BASE, armComms);
   wrist = new Joint(JointID.WRIST, armComms);
   shoulder = new Joint(JointID.SHOULDER, armComms);
   elbow = new Joint(JointID.ELBOW, armComms);
```



```
   joints = new Joint[4];   // order is important; used by moveTo()
   joints[0] = base;
   joints[1] = wrist;
   joints[2] = shoulder;
   joints[3] = elbow;
}  // end of RobotArm()
```

There's no real need for a joints[] array, except that it makes iterating through all the joints easier. For example, RobotArm.moveToZero() moves all of them back to their 0 degree position:

```
public void moveToZero()
{
  System.out.println("Moving to zero...");
  for(int i=joints.length-1; i >= 0; i--)
    joints[i].turnToAngle(0);     // process joints in reverse
}
```

This rotates the joints so that the arm looks like Figure 4.

A tricky aspect of RobotArm is ensuring that the joints are rotated in an order that avoids knocking things over on the tabletop. For instance, in moveToZero() the elbow joint is rotated first since it's most likely to be nearest the table surface.

5.1. Turning the Joints

RobotArm offers its own versions of the joint turning functions, turnByOffset() and turnToAngle(). These check the joint ID and call the relevant Joint instance.

```
// globals
private Joint base, wrist, shoulder, elbow;


public void turnByOffset(JointID jid, int offsetAngle)
{
  if (jid ==  JointID.WRIST)
    wrist.turnByOffset(offsetAngle);
  else if (jid == JointID.ELBOW)
    elbow.turnByOffset(offsetAngle);
  else if (jid == JointID.SHOULDER)
    shoulder.turnByOffset(offsetAngle);
  else if (jid == JointID.BASE)
    base.turnByOffset(offsetAngle);
  else
    System.out.println("Unknown joint ID: " + jid);
} // end of turnByOffset()


public void turnToAngle(JointID jid, int angle)
{
  if (jid ==  JointID.WRIST)
    wrist.turnToAngle(angle);
  else if (jid == JointID.ELBOW)
    elbow.turnToAngle(angle);
  else if (jid == JointID.SHOULDER)
    shoulder.turnToAngle(angle);
```



```
  else if (jid == JointID.BASE)
    base.turnToAngle(angle);
  else
    System.out.println("Unknown joint ID: " + jid);
} // end of turnToAngle()
```

5.2. Inverse Kinematics

The RobotArm class supports both inverse and forward kinematics. Inverse kinematics (IK) is more useful, and so I'll describe it first.

IK calculates the joint rotations that will move the arm's endpoint (i.e. the gripper) to a desired (x, y, z) position. IK allows the user to interact with the arm in terms of gripper positions (i.e. where to pick something up, where to drop it), rather than using joint rotations settings.

The bad news is that IK calculations are difficult for real-world robot gear containing multiple rotational and positional (prismatic) joints. Iterative numerical solutions are the norm, involving heavy-duty matrix manipulation of Jacobean matrices.

If you have to use an iterative approach, the easiest to understand is probably Cyclic Coordinate Descent (CCD). CCD iterates through each joint turning it so the gripper moves closer to the intended position. It's simple to implement, and efficient, although can lead to the arm moving rather strangely.

A good, not too technical, introduction to IK and CCD can be found in two articles by Jeff Lander: “Oh My God, I Inverted Kine!” (http://graphics.cs.cmu.edu/nsp/course/15-464/Spring07/assignments/jlander_gamedev_sept98.pdf) and "Making Kine More Flexible" (http://graphics.cs.cmu.edu/nsp/course/15-464/Spring07/assignments/jlander_gamedev_nov98.pdf).The second article describes a simple CCD algorithm, and has some corrections to the first article. Another CCD introduction is " Cyclic Coordinate Descent in 2D" by Ryan Juckett which includes good diagrams of how CCD works (http://www.ryanjuckett.com/programming/animation/21-cyclic-coordinate-descent-in-2d).

Now the good news: the OWI is so simple that it's possible to develop an algebraic solution to the IK problem (i.e. one using simple trigonometry) without the need for Jacobean matrices or CCD.

I'll explain the algebra by referring to Figures 17 and 18.

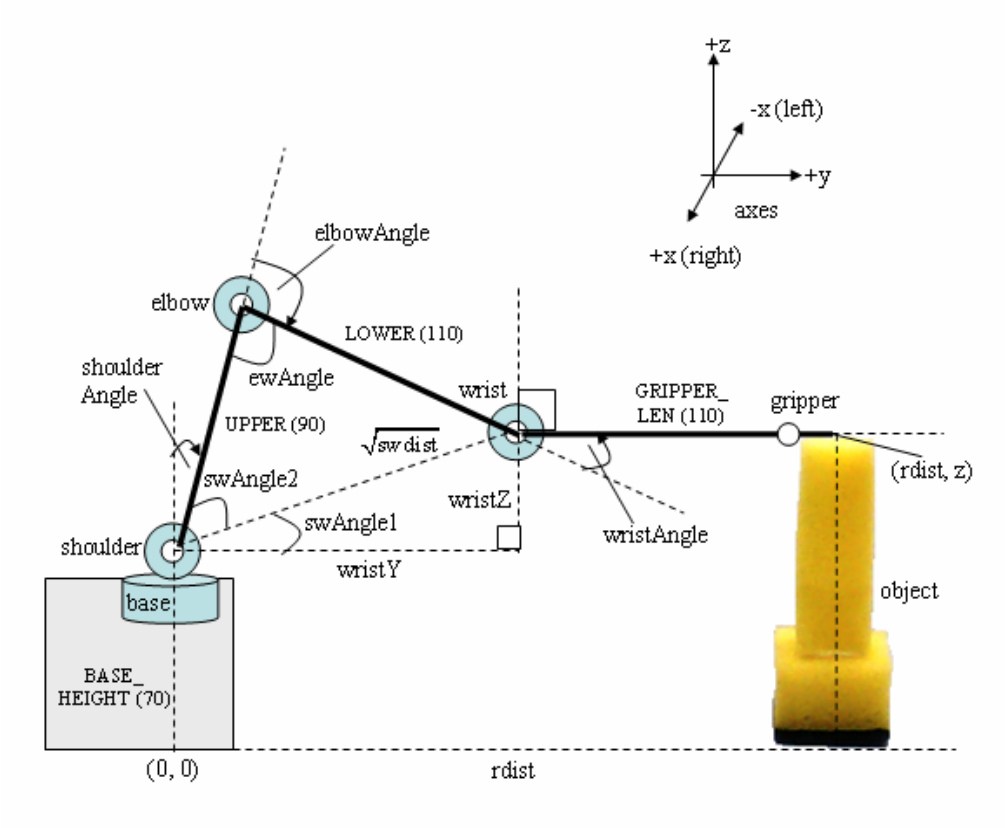

Figure 17. Inverse Kinematics for the Arm (Viewed from the Right).

The top of the object is at (x, y, z), but the (x, y) part is 'flattened' to rdist, the direct line from the arm's base to the object, as shown in Figure 18.

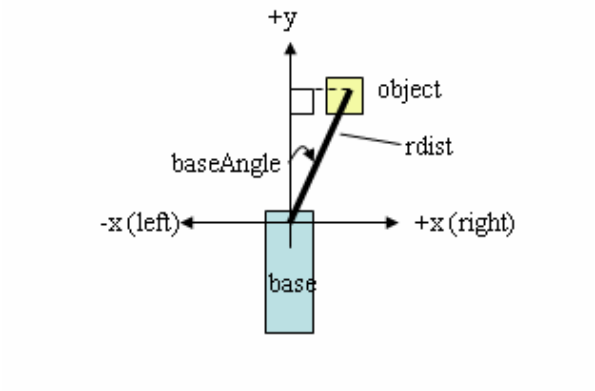

Figure 18. Inverse Kinematics for the Arm (Viewed from Above).

This simplification means that the base joint must be rotated first so the arm points directly at the object. This allows me to do manipulate algebra in the 2D plane defined by (rdist, z).

To make the problem even easier, I constrain the wrist joint's rotation (wristAngle in Figure 17) to make the wrist-gripper link always stay at the horizontal.

The math requires the law of cosines, which relates the sides of a triangle to the cosine of one of its angles (see Figure 19).

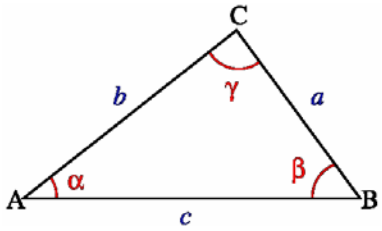

Figure 19. The Law of Cosines.

Using the notation from Figure 19, the cosines rule involving the γ angle is:

$$c^2 = a^2 + b^2 - 2ab\cos\gamma$$

Similar expressions can be written for the α and β angles.

I utilize the cosine rule in two places in Figure 17 – to calculate the angles labeled as swAngle2 and ewAngle.

The calcIK() method uses the notation of Figures 17 and 18, to calculate the baseAngle, shoulderAngle, elbowAngle, and finally the wristAngle, given a supplied (x, y, z) coordinate for an object's top.

```
// global lengths of arm elements (in mm)
private static final int GRIPPER_LEN = 110;
private static final int LOWER_ARM = 110;
private static final int UPPER_ARM = 90;
private static final int BASE_HEIGHT = 70;


private int[] calcIK(int x, int y, int z)
{
  int extent2 = (x*x) + (y*y);
  int maxExtent = GRIPPER_LEN + LOWER_ARM + UPPER_ARM;
  if (extent2 > (maxExtent*maxExtent)) {
    System.out.println("Coordinate (" + x + ", " + y + ", " + z +
                       ") is too far away on the XY plane");
    return null;
  }

  // base angle and radial distance from x,y coordinates
  double baseAngle = Math.toDegrees( Math.atan2(x,y) );
  double rdist = Math.sqrt((x*x) + (y*y));
    // radial distance now treated as the y coord for the arm

  // wrist position
  double wristZ = z - BASE_HEIGHT;
  double wristY = rdist - GRIPPER_LEN;
```



```
  // shoulder-wrist squared distance (swDist2)
  double swDist2 = (wristZ * wristZ) + (wristY * wristY);

  // shoulder-wrist angle to ground
  double swAngle1 = Math.atan2(wristZ, wristY);

  double triVal = (UPPER_ARM*UPPER_ARM + swDist2 -
       LOWER_ARM*LOWER_ARM) / (2 * UPPER_ARM * Math.sqrt(swDist2));
  if (triVal > 1.0) {
    System.out.println("Arm not long enough to reach coordinate");
    return null;
  }
  double swAngle2 = Math.acos(triVal);
  double shoulderAngle = 90.0 -
                   Math.toDegrees(swAngle1 + swAngle2);

  // elbow angle
  double ewAngle = Math.acos(
            (UPPER_ARM*UPPER_ARM + LOWER_ARM*LOWER_ARM - swDist2) /
            (2 * UPPER_ARM * LOWER_ARM) );
  double elbowAngle = 180.0 - Math.toDegrees(ewAngle);

  double wristAngle = 90.0 - (shoulderAngle + elbowAngle);

  // round angles to integers
  int baseAng = (int)Math.round(baseAngle);
  int shoulderAng = (int)Math.round(shoulderAngle);
  int elbowAng = (int)Math.round(elbowAngle);
  int wristAng = (int)Math.round(wristAngle);

  int[] angles = new int[] {baseAng, wristAng,
                            shoulderAng, elbowAng};
  return angles;
}  // end of calcIK()
```

It's possible that the (x, y, z) coordinate isn't within reach of the arm, which will become obvious when cosine(swAngle2) is calculated. Its value, stored in triVal, will be greater than 1.

Another easy out-of-bounds test is performed at the start of calcIK() when the maximum extent of the arm (when it's stretched out along the horizontal) is less than the shortest distance to the object coordinate on the XY plane.

The wrist joint angle calculation is simple, but perhaps a little mysterious:

```
double wristAngle = 90.0 - (shoulderAngle + elbowAngle);
```


Figure 20 illustrates the reasoning behind the math.

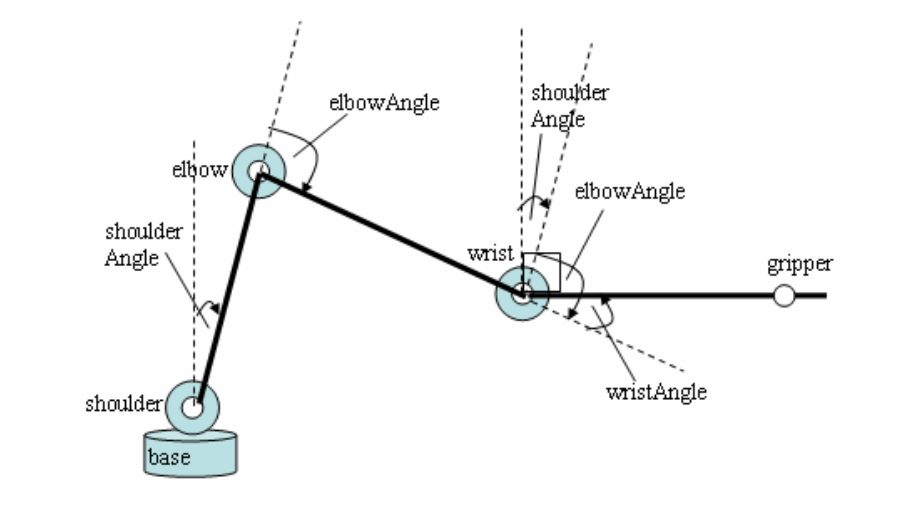

Figure 20. Calculating the Wrist Angle.

Since the wrist-gripper link always stays horizontal, it remains perpendicular to the original orientation of the shoulder-elbow link.

The four rotation angle are returned from calcIK() in a single array, after being converted and rounded to integer degrees.

5.3. Using Inverse Kinematics

The great advantage of IK is that it allows a user to manipulate the robot arm in terms of the gripper's (x, y, z) position rather than with joint rotations. The RobotArm class offers two moveTo() methods, which take a coordinate as input and attempt to move the gripper to that position.

```
// globals
private Joint[] joints;    // so can iterate through the joints


public boolean moveTo(Coord3D pt)
{  return moveTo(pt.getX(), pt.getY(), pt.getZ());  }


public boolean moveTo(int x, int y, int z)
{
  int[] angles = calcIK(x, y, z);
  if (angles == null)
    return false;

  if (!withinRanges(angles)) {
    System.out.println("Move Cancelled");
    return false;
  }
  else {   // turn all the joints
    for (int i=0; i < angles.length; i++)
      joints[i].turnToAngle(angles[i]);
```



```
    return true;
  }
}  // end of moveTo()
```

The Coord3D class used by the first moveTo() is a wrapper around three integers representing the x-, y-, and z- arguments of the coordinate.

The array of rotation angles returned by calcIK() need to be checked to see if they all fall within their joints' rotation limits.

The availability of moveTo() means that it's straightforward to write a method that moves an object between two locations:

```
public void moveItem(Coord3D fromPt, Coord3D toPt)
// move an item from one place to another
{
  boolean hasMoved = moveTo(fromPt);
  showAngles();
  System.out.println("From Coord: " + getCoord() );

  if (hasMoved) {
    openGripper(false);    // grasp object

    turnByOffset(JointID.ELBOW, -30);  // rotate away from floor
    System.out.println("Off-floor Coord: " + getCoord() );

    moveTo(toPt);   // don't check the result of this move
    showAngles();
    System.out.println("To Coord: " + getCoord() );
    openGripper(true);    // release object
  }
}  // end of moveItem()
```

openGripper() is employed to grab and later release the object. turnByOffset() moves the arm up off the floor, so it doesn't drag the object to its destination.

showAngles() prints out all the current joint angles:

```
// global
private Joint[] joints;

public void showAngles()
{
  System.out.println("Current Angles:");
  for(Joint j : joints)
    System.out.print( "  " + j.getName() + ": " + j.getCurrAngle());
  System.out.println();
}  // end of showAngles()
```

5.4. Forward Kinematics

Forward kinematics calculates a endpoint position (i.e. the location of the gripper) using the arm's joint rotations. This is less useful than IK, but can still be utilized to check the angles calculated by IK. The math is illustrated by Figure 21.

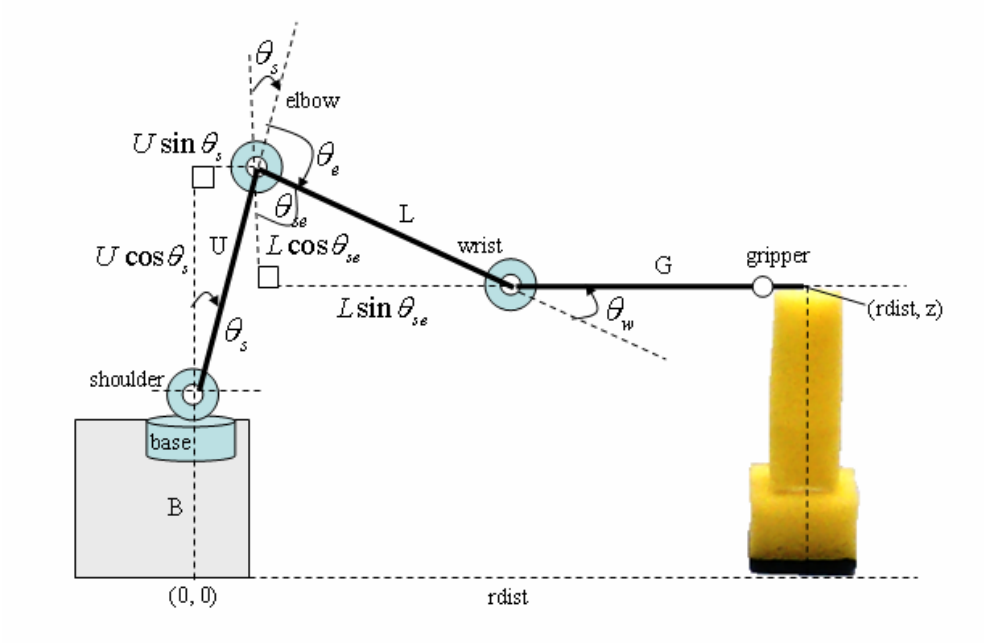

Figure 21. Forward Kinematics Calculations for the Arm
(Viewed from the Right).

The joint rotations are known for the base (θ_b), shoulder (θ_s), elbow (θ_e), and wrist (θ_w). I want to obtain the (x, y, z) coordinate at the top of the object.

I start by calculating (rdist, z), where rdist is the shortest distance from the arm to the object. From Figure 21, the coordinate values are:

$$\text{rdist} = U \sin\theta_s + L \sin\theta_{se} + G$$

$$\text{z} = B + U \cos\theta_s - L \cos\theta_{se}$$

where $\theta_{se} = 180 - (\theta_s + \theta_e)$ and U, L, and G are the lengths of the upper arm, lower arm, and gripper links respectively. Due to my fixing of the wrist-gripper link to always be horizontal, there's no need to employ the wrist angle (θ_w).

rdist is separated into x- and y- axis components, according to Figure 22.

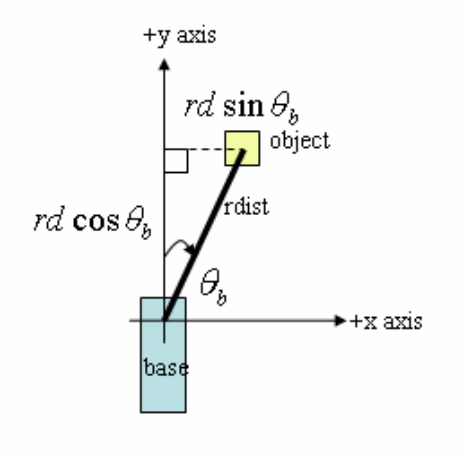

Figure 22. Forward Kinematics Calculations for the Arm (Viewed from Above).

The distances are:

$x = rdist \sin\theta_b$ and $y = rdist \cos\theta_b$

These calculations are implemented in getCoord():

```
// globals
private Joint base, wrist, shoulder, elbow;


public Coord3D getCoord()
{
  int baseAngle = base.getCurrAngle();
  int shoulderAngle = shoulder.getCurrAngle();
  int elbowAngle = elbow.getCurrAngle();
  // wristAngle not needed

  if (!base.isInRange(baseAngle))
    System.out.println("  base angle ("+baseAngle+") out of range");

  if (!shoulder.isInRange(shoulderAngle))
    System.out.println("shoulder ("+shoulderAngle+")out of range");

  if (!elbow.isInRange(elbowAngle))
    System.out.println("  elbow angle ("+elbowAngle+")out of range");

  double baseAng = Math.toRadians(baseAngle);
  double shoulderAng =  Math.toRadians(shoulderAngle);
  double elbowAng = Math.toRadians(elbowAngle);

  int seAngle = 180 - (shoulderAngle + elbowAngle);
  double seAng =  Math.toRadians(seAngle);

  double radialDist = UPPER_ARM*Math.sin(shoulderAng) +
                      LOWER_ARM*Math.sin(seAng) + GRIPPER_LEN;
  int x = (int)Math.round( radialDist*Math.sin(baseAng) );
  int y = (int)Math.round( radialDist*Math.cos(baseAng) );
```



```
  int z = (int)Math.round( BASE_HEIGHT +
                           UPPER_ARM*Math.cos(shoulderAng) -
                           LOWER_ARM*Math.cos(seAng) );
  return new Coord3D(x, y, z);
}  // end of getCoord()
```

getCoord()'s result doesn't 'prove' that the arm is in a certain position. In fact, the inaccuracies of the OWI's gears mean that after only two moves, the actual position of the gripper will be noticeable different from the calculated position. The only way to fix this is to add a feedback mechanism which independently observes the true position of the gripper, and corrects the error by rotating it back to that spot. I'll use a webcam to implement feedback of this type in chapter 18.

