Introduction

Welcome to instructional robotics laboratory. This course is unlike any of the other programming courses that you have taken. Instead of just “generating output”, your programs will be interact directly with a mechanical system. They will read sensors, cause wheels to turn, and generate audio. To be successful, you must understand not just how your program is working but also how the mechanical components are put together and how they interact. This manual will be your guide to the robot hardware and to the interface library that you will use to communicate with the hardware.

Robot Hardware Description

Your robot for this class is a Pioneer 2 DX, manufactured by Activmedia Robotics Like everyone else these days, it has a home page, http://www.activrobots.com, which you may wish to check out for additional background information.

Using the picture shown on this page as guide, let’s talk about the basic features of this robot. Starting from the bottom, the robot has three points of contact with the ground, two large wheels and a small caster wheel. Separate motors drive the large wheels independently and the caster is passive. This configuration of motors and wheels is an example of a differential direct drive, non-holonomic system with zero gyro-radius. (A holonomic system can move in any direction without turning first. A zero gyro-radius means it can turn on a dime.) If your program sets both drive motors to the same speed, the robot goes straight forward (or backward). Setting the wheels to equal speeds of opposite polarity (e.g., -200 and +200) causes the robot to spin in place. Anything in between causes a smooth turn. The base has a maximum translational speed of approximately 30 inches per second and a maximum rotational speed of 50 degrees per second.

---

1 Adapted from an original version written by Illah Nourbakhsh for CMU Computer Science 16x62
Moving to the top of the robot, notice a grey panel on the side of the robot that frame the power switch and power receptacle. The remaining electrical controls are on the top of the robot. When you turn the scout on, the red LED on the top will light and the electrical system will start up with a pleasant two-tone beep.

To charge the robot, simply plug it into the wall. There is one huge catch: **THE ROBOT MUST BE ON FOR THE CHARGING CIRCUIT TO WORK.** This means that no matter how long you leave it plugged in, if the scout is off, charging will not happen. So, at night, when you tuck the scouts in, leave them plugged in and on. As a confirmation that you have done it right, listen for two short beeps when the power cord is connected. This is the indicator that charging has begun.

The two push buttons on the top are reset buttons. The red reset button resets the controller chip in the robot electronics. The white switch enables and disables power to the drive motors. Enabling and disabling the drive motors can also be controlled by software. In fact it happens automatically each time you invoke the “openSerial” method in RobotController. Either of these buttons will stop the robot. However, if you use the white button to disable to motors, your software should be able to detect the state change and shut down more gracefully.

---

**Which way is forward?**

To orient yourself, the side of the robot where the caster wheel is located is considered to be the “back” of the robot. Thus a positive motor speed setting moves the robot forward such that it drags the caster behind. Negative motor settings move the robot such that the caster is pushed while the robot moves backward. This also is the convention used for numbering the sonar sensors and pressure switches. Number zero is always directly (180°) opposite the caster position. Numbering increases in the counterclockwise direction.

---

**What to do when the robot runs into something and won’t stop**

Of course, this will never happen to you. But just in case, if the robot’s emitting a high-pitch squeal as it sits rammed into a wall, the motors are being overdriven. Hit the white motor enable button. If your program has been written to properly monitor the status of the robot and shut down properly, it should detect the change in state. If not you may have to kill your program manually.

You should never try to move the robot manually when your program is sending motion commands to it (i.e. when the wheels are moving). Doing so will cause damage to the robot and possibly to you. You can push the robot around when it is stopped. Terminate your program and stop sending commands to the robot. Wait a few seconds to ensure that the current command has timed out. In this state, pushing the robot around is perfectly fine, but if you want to show off your muscle, you can also lift the robot.
Above the pressure switch ring is a ring of round gold ‘coins’ that are the robots’ only range sensors: the sonars. These Polaroid sonar transducers provide surprisingly accurate range information from 5 inches to 255 cm. Each has a beam width of about 25 degrees, although there may be variation in this beam width, or half cone, from sonar to sonar. A single sonar unit works as follows: The transducer emits a series of 16 pulses at a frequency of 49.4 KHZ. The echo receiver circuitry is disabled, or "blanked," for about 1 millisecond during and slightly beyond this emission phase so that the initial pulses are not detected as echoes. This blanking has a serious effect at close ranges, known as double echo. If the robot is closer than about 5 cm to an object, then the return echo from that object could occur while the echo receiver circuitry is still blanking. Therefore, the object might appear to be farther away than it actually is. Also, a sonar signal will bounce away instead of returning if it hits a smooth object at a shallow angle of attack— in effect, it can glance off coherently. This will also result in the strike distance being larger than it should be.

The sonar sensors communicate with your program through the state array. Sonar 1 is at the front of the robot. The state array is an array of integers and the sonar data is returned in the 1st through 16th element of the array, numbered as shown in the diagram to the right. Sonar return values return an integer, which is the distance in cm to a contact. However, values of five or below should not be considered to be accurate.

There is one other pair of sensors, not visible from the outside, that are of importance to you in this course: the wheel position encoders. These sensors allow you to track the position of the robot by monitoring to the rotation of the wheels. The sensors are up/down counters which count up or down 756 times per cm of forward or backward translation on each wheel. Based on the relative motion of the two wheels, the sensor controller can also track changes in the angular orientation of the robot. All three of these values are available to your program in the state array in elements 17, 18, and 19. Their indices have also been assigned symbolic constants, XPOS, YPOS, and THETA to make them easy to remember. All readings for these encoders are in units of tenths of inches for translations and tenths of degrees for orientations. For example, a 22-degree turn will be correspond to a change of 220 in the THETA encoder.
**Slip-Sliding Away: Accuracy and the Position Encoders**

If the world were covered with Velcro, we probably would not have a problem, but as it is, your robot will spend most of the time on a carpeted surface. Plastic wheels on carpeted surfaces tend to slip and this means that your encoders will include errors introduced during acceleration, deceleration and during turns. Depending on how much of a speed demon you are, this may be as little as a few tenths of an inch to an inch or more each time that you start and stop the robot. However fast you go, this can be a serious problem in two situations. First, if you rely solely on the encoders to track the robot position through a turn followed by translation, even a small error in the turn angle will introduce a large position error as the robot moves away. Second, transnational position errors will compound as you navigate a complex path involving several waypoints and turns. We will talk more about this later in the course, but for now use the position encoders only to measure relative intervals such so that position errors will not accumulate and always use both the sonars and the position encoders to solve your navigation problems.

On the top of the robot is the laptop that will run your control programs. This laptop is connected to the wireless LAN to the computers in the lab, and through the serial (COM) port to the microcontroller chip in the robot. Since we are running the LINUX operating system on all systems. It will not be necessary for you to work directly on the laptop console in order to develop and run programs on the laptop system. Using X-windows you can work remotely which should go a long way making you more comfortable as you write your program. The laptops are attached to the robots w/Velcro. PLEASE DO NOT REMOVE THE LAPTOPS FROM THE ROBOTS.

**A Peek Inside**

Before we move on to the software interface, it is important that you get a feel for how the processing and communication works inside the robot and within the lab. We will begin with a look inside the robot as shown in the figure on the next page. Inside of the robots, each of the sensors and the motor controllers communicate with an MC68111 single chip controller. The operation of the embedded controllers is not of interest in this class. However, the communication mechanism and control protocol between the single board PC and the controller system is of critical importance.
At the risk of shouting “the emperor has no clothes”, the primary communication mechanism between the PC and the controller is actually fairly low-tech. It is a 19.2 KB serial link between the COM1 port of the PC and the on-chip serial port of the 6811 controller. Robot operations are encoded into a textual command language that supports relatively high-level control operations. To get a sense of the command language, peek ahead to the software section and look at the methods in the robotController class. Each of these methods roughly translates into one of these commands. For example, invoking the `setVel(50,50)` method will send a `SET VELOCITY 50 50` command across the serial interface. As a result, the robot will begin to move forward at 5 inches per second and will continue moving forward until another command is received. The PC must initiate all communication on this interface, even if the function of the command is to send data between the sensors and the PC. The robot sensors have no mechanism to “interrupt” the PC or to communicate anything about its state unless it is polled by a command from an application program on the PC. Nor is there a mechanism to inform the 6811 controller if an unanticipated event causes the host program to unexpectedly terminate.
Losing control…

What does all this talk of communication links and control protocols mean to you and your programs? Consider the following example. Suppose that your program sends a `setVel(50,50)` command to the robot. After the robot begins moving your program hits a bug and crashes. Even though your program is dead, the robot is still moving, waiting for another command that will never come. You try to hit the bumper switch to stop it, but with no program running to poll the state array, this is useless. YIKES!!! What do you do? It just won’t stop. You can’t pick up or push the robot while it is moving. You would rather not turn the robot off since it is dangerous to shut down Windows/NT that way. RELAX, all is not lost. For exactly this reason `setVel` commands automatically time out after 2 seconds. (This can be changed if necessary using the `setTimeout` method).

Alternatively, you may wish to write an emergency shutdown program that you can start quickly to and send stop commands.

Note however, that there is not a corresponding timeout set up for the `turnSonarsOn()` command. So be prepared for the robots to continue clicking-away indefinitely if your program terminates and leaves them on.
Software Environment

The JAVA software environment for programming the robot consists of multiple layers as shown below. The primary mechanism that your applications will use to access the hardware is via the methods defined in two objects: the robotController object and the speechController object.

To access the robotController interface, each JAVA project that you create will include a source class called “robotController.” The methods in the robotController class each in turn invoke “native” routines from a C language shared library (libp2.so). These library routines are where the controller commands are assembled and sent to the COM1 port. If you would like to see the C side source code, ask one of the instructors or TA’s and they

As you know, JAVA is an object-oriented environment. So, when this manual suggests that you “invoke a method” from the class called RobotController, there is an assumption that you have declared and instantiated a RobotController object in your program using a statement of the form:

```java
RobotController myDumbRobot;
```

followed somewhere else by

```java
myDumbRobot = new RobotController;
```

At this point, the actual invocation of a method like serialOpen takes the form.

```java
myDumbRobot.serialOpen();
```
can direct you to a web site where the source code is available. A complete description of the methods for the robotController class can be found in Appendix A.

Audio output for the speech interface does not go through the COM1 interface to the sensor controller, but instead uses the audio interface in the single board PC. To access the text-to-speech routines (as well as a rudimentary voice input unit), you should include a source class called jspeech in your project. The methods for the text to speech interface. These methods are described in Appendix B.

Although there are about a dozen or so methods in the robotController class, almost everything that you will do in the class can be completed using only about five or six. For the remainder of this section we will focus on a detailed description of these most commonly used methods.

Before you can do anything with the robot, you must first open the COM1 port on the laptop. Thus, in the initialization code you will need to insert a call to the **serialOpen()** method. Similarly, before you program terminates, you should invoke **serialClose();**

Once you are up and running on the serial port your most important command to the robot is the function: **setVel(leftwheel, rightwheel)**. This function (like the several other robot commands) sends a stream of characters via the serial port to the robot. This stream of characters encodes your intended velocities for the two wheels of the robot, in tenths of inches per second. The range of velocity values is an integer between -300 and 300 (a negative velocity indicates backward movement). The indoor spaces are very confined; thus, you should not specify velocities anywhere near 300 or -300 in any case.

The behavior of the robot when it receives a command of the form **setVel()** is precisely as follows (in this order):

1) Begins executing, simultaneously, the specified wheel speeds.
2) Continues executing this motion until the robot receives a new command of any kind or until the command times out. This timeout defaults to the value 2 second. However, you have the power to change this default (to accomplish such feats, refer to Appendix A).

Note that turns are accomplished by setting the right and left motors to different speeds (or directions if you want to spin in place). There are two ways to stop moving. You can use **selVel(0,0)** or you can use a smoother and more motor-friendly method called **killMotors()**. Unless there is good reason not to, you should use the latter.

Now that you are moving, you may want to see where you are going. To use the sonars, you will need to first turn them on. Do so by invoking the **turnSonarsOn()** method. To turn them off when their clicking get on your nerves, use the **turnSonarsOff()** method. Turning off the sonars is a good idea in general because it also saves battery power. When your program exits, you need to routinely send a **turnSonarsOff()** command to the robot as part of your exit code.
The current sonar readings as well as the other sensor data for the robot are stored in an integer array called the “state array”. Your program will have a copy of this state array as part of the robot controller object. However, keep in mind that the embedded controller (on the other side of the COM1 link) maintains the real time sensor data. Before you can use the local copy of the state array, you must first update it by polling the embedded controller. The \texttt{gs()} (for “get state”) method accomplishes this task.

State array is a C-style integer array. Using the myDumbRobot object above as an example, sonar 1 can be read as follows.

```c
int sonar1;
.
.
myDumbRobot.GS();
sonar1 = array.getint(myDumbRobot.State_Array,1);
```
The following table summarizes the contents of the State_Array.

<table>
<thead>
<tr>
<th></th>
<th>Return Value</th>
<th>Sonar Values</th>
<th>Integrated Position Encoders</th>
<th>Wheel Velocities</th>
<th>Bumper Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Return Value</td>
<td>0</td>
<td>The first element of the stateArray [0] is the C-side return value of the last executed command.</td>
<td>1-16</td>
<td>17-19</td>
</tr>
<tr>
<td>1-16</td>
<td>Sonar Values</td>
<td></td>
<td>These are the 16 sonar sensor readings. Each element is an integer between 5 and 255, where the integer is precisely the number of inches from the sonar transducer to the object that caused an echo. The sonars are arranged in a counterclockwise direction with the sensor in the twelve o'clock position (the “front”) being index 1 (element 1 of the array) and the sensor at the one o'clock direction of the robot being element 16 of the array.</td>
<td>17-19</td>
<td>20-21</td>
</tr>
<tr>
<td>17-19</td>
<td>Integrated Position Encoders</td>
<td></td>
<td>These provide the robot's idea of its position based upon its encoder values. 17 provides position along the x-axis, 18 provides position along the y-axis, 19 provides the angle of the robot with respect to their orientation when the robot was turned on. When you turn the robot on, the encoders are all set to zero. That means that the robot resets its Cartesian grid to be lined up with its current orientation, whatever that may be—with the robot pointing straight up the X axis, and Theta = 0. As the robot moves, the encoders show the robot's position and orientation on that grid using tenths of inches and tenths of degrees. The range of values for the robot’s encoded angle is from 0 to 3599. X position and Y position can have positive or negative values, depending on the quadrant that the robot is in within its make-believe Cartesian grid. The x-axis is aligned with the 0 degree orientation -- if you think about this a bit, it makes perfect sense.</td>
<td>20-21</td>
<td>22</td>
</tr>
<tr>
<td>20-21</td>
<td>Wheel Velocities</td>
<td></td>
<td>These are the current translational velocities of the robot’s wheels. You probably will never look at these, although it might be fun to look at them after issuing a command to see how well the built-in PID motor controller matches the actual velocities to the ones you requested.</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Bumper Status</td>
<td></td>
<td>This is another really important sensor. The bumper status will be zero if none of the bumpers are depressed. If any bumper is depressed, this will be nonzero. By the way, there are really 6 separate bumpers on the robot, and each one can be on or off. And from this one number you can figure out which are depressed and which aren’t. But don’t you bother with that just yet.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Robot Program Example

To finish this manual, we are included a heavily commented segment of robot code. It is the button1 handler for a Java application GUI. It doesn’t do much. The robot simply moves to a point where its front and back sonars readings are nearly equal.

```java
void button1_MouseClick(java.awt.event.MouseEvent event)
{
    // to do: code goes here.
    int frontsonar, backsonar;
    boolean flag = true;

    // Code not shown here
    // RC was declared to be a RobotController Object
    // RC was instantiated
    // Serial port was opened

    // Start by turning on the Sonars
    RC.turnSonarsOn();

    // get the current state array values
    RC.GS();

    System.out.println("Hit robot bumper to quit...");

    // this loop is the main body, we are here until
    // someone hits a bumper switch
    while (flag) {
        // need to get updated state array on each loop
        // otherwise this program is an infinite loop
        RC.GS();

        // pick off the front and back sonar values from the
        // state array and debug-print them
        frontsonar = Array.getInt(RC.State_Array, 1);
        backsonar = Array.getInt(RC.State_Array, 9);
        System.out.println("Front sonar: " + String.valueOf(frontsonar) +
            "  Back sonar: " + String.valueOf(backsonar));

        // now look at bumper state and exit program if any bumper is
        // is hit (index = 22)
        if (Array.getInt(RC.State_Array, RobotController.BUMPER) != 0) {
            RC.turnSonarsOff();
            RC.killMotors();
            flag = false;
        }

        // rock-a-bye baby
        else if (frontsonar < backsonar) {
            RC.setVel(-20,-20);
        } else {  // end while (true)
            RC.setVel(20,20);
        }
    }
}
```
### Appendix A: robotController Method Summary

The following is a complete description of all robot interface functions available in the Robotcontroller Class to send commands to the robot.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int serialOpen()</code></td>
<td>Open the COM1 port of the single board PC so it can communicate with the robot.</td>
</tr>
<tr>
<td><code>int serialClose()</code></td>
<td>Close the serial port of the single board PC.</td>
</tr>
<tr>
<td><code>int turnSonarsOn()</code></td>
<td>Turns all 16 sonars on in some nice firing pattern and firing speed.</td>
</tr>
<tr>
<td><code>int turnSonarsOff()</code></td>
<td>Turn all the sonars off when they get annoying.</td>
</tr>
<tr>
<td><code>int GS()</code></td>
<td>Get the stateArray all updated. You do NOT need to call this function if you call any other robot command function, since stateArray is updated when any robot communication is performed.</td>
</tr>
<tr>
<td><code>int setVel(int leftVel, int rightVel)</code></td>
<td>Set the velocities of the left and right wheels, in tenths of inches per second.</td>
</tr>
<tr>
<td><code>int killMotors()</code></td>
<td>This command is a much more graceful way of causing the robot to stop than <code>setVel(0, 0)</code> because the latter command causes the motors to freeze while <code>killMotors</code> simply cuts off power and allows the robot to coast to a halt.</td>
</tr>
<tr>
<td><code>int setAccel(int leftAccel, int rightAccel)</code></td>
<td>Configure the acceleration of the left and right wheels of the robot (make it the same please!), in tenths of inches per second per second. Default is: about 100.</td>
</tr>
<tr>
<td><code>int configureSonars(int firingRate, int[] sonarOrder)</code></td>
<td>You can use this command to specify how long the robot should wait between firing sonars and which sonars to fire and in which order. The firing rate is a number specifying how many 4 millisecond intervals the robot should wait between firings. Set it to a number between 2 and 254. <strong>Hint:</strong> use 4. The sonarOrder is an array containing exactly sixteen integers. The order of the numbers indicates the firing order. If you want to fire only the front three sonars, sonarOrder would be <code>{1,2,16,255,0,0,0,0,0,0,0,0,0,0,0,0,0}</code>. As you can now guess, after you put in all the sonars you care about, put in a 255. Make sure the whole array is 16 long (buffer it with zeroes). If you want just the back sonar to fire, here's what you do:</td>
</tr>
</tbody>
</table>

```java
int[] sonars = new int[16];
sonars[0] = 9;
sonars[1] = 255;
sonars[2] = 0;
sonars[3] = 0;
...
sonars[15] = 0;
configureSonars(4, sonars);
```
<table>
<thead>
<tr>
<th>int setTimeout(int timeout)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This command allows you to reset the amount of time after which a velocity command is thrown away and the robot &quot;limps&quot; it if received no further communication. The default is twenty seconds. Remember, the best way to override a velocity command is to simply send another one! The parameter is an integer number of seconds. We believe if you set timeout to be 0 then a command never times out. In terms of safety, that's a bad idea!</td>
</tr>
</tbody>
</table>
**Appendix B: Speech synthesis methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>openSpeech()</code></td>
<td>Use this after you create your <code>jspeech</code> object to initialize the speech. DOES NOT WORK IN A CONSTRUCTOR</td>
</tr>
<tr>
<td><code>closeSpeech()</code></td>
<td>The opposite of <code>openSpeech</code>, this closes the speech object, may work in a constructor, but why bother?</td>
</tr>
<tr>
<td><code>voiceSpeech(int voice)</code></td>
<td>Each number from 1 to 19 represents a voice. This function MUST be called before you try to give it a string. No voice, no speech. Many of the voices are useless and unintelligible, my personal favorites are 1, 5, and 18.</td>
</tr>
<tr>
<td><code>textSpeech(String thetext)</code></td>
<td><code>textSpeech</code> is given a string of characters (words really) to say, and placed between quotes. Sometimes it takes some acrobatics to make it sound intelligible. TIP: add comma’s liberally into the text to introduce pauses, you don’t get inflection but this is the next best thing.</td>
</tr>
</tbody>
</table>