Communications

Communications is a key element in robotics. What good is our sending a billion dollar robot to Mars if we never find out what it learned? What good is our little mapping robot if we never find out where it found out where it went? The answer is not much good. Sending data from the RCX to your PC and sending data from RCX to RCX is key. You are already communicating between your PC and RCX every time you download a program. Wouldn’t it be cool to write your own programs to do the same thing? It’s amazingly easy to do.

Java programs use *streams* to communicate. Stream is a conceptual word and defining it is difficult. One definition is a stream is a connection between two independent entities used to transfer data or information. In my mind, that does not create a mental image that helps me understand. So, think about sitting with another person and having a conversation. Your spoken words are a *word stream*. For most people, the mouth is an output device and the ear is an input device. Streams are based upon connections and have only one direction; mouths can’t hear and ears can’t speak. In a conversation between you and me there are two streams: 1) I talk to you and you listen to me and 2) you talk to me and I listen to you.

Java has an extensive set of packages that deal with communications. The designers of Java generalized the idea of streams to include things that you would not expect. Virtually all communications, with the major exception of graphic user interfaces, GUI, between a program and the real world is based upon streams. Examples include surfing the web, saving a file on a hard drive, typing in the console window, and using a modem. The most basic streams are InputStream and OutputStream.

While people use word streams to communicate, computers use *byte streams*, a.k.a. *data streams*. Remember that a byte is the smallest primitive data type in Java. As it happens, ASCII characters, letters and numerals, are stored in single bytes. So each ASCII character is sent as one byte in a byte stream. Integers take four bytes so they get sent through the stream as four consecutive bytes. So if your program is on the receiving end of a data stream how can it tell the difference between an int, a byte, a float or whatever? Simple answer is that it can’t. A byte is a byte. What you have to do is create a *protocol*. A protocol is the rules for a conversation that the sender and receiver agree to follow. While you might not be familiar with the word, if you use the Internet you use protocols all the time (it’s the P in TCP, IP and HTTP). Protocols are designed to make communication easier. Think about the protocol for using a telephone. The rule of the telephone protocol is that you pick up the phone when it rings. Next, you say “hello” or a similar greeting. The caller responds and gets to the purpose of the call. When the caller is done, you have the opportunity to respond. Finally, the caller ends the call by saying “goodbye” or something similar. That is the protocol for using a telephone. The introduction of caller id illustrates how disruptive not following an established protocol is. Answering “hello Bob” when Bob is calling and not used to the new greeting, disrupts the flow of the conversation. If you haven’t done this already, try it.
**PC to RCX Communications**

Robot communication is very similar to a conversation. To have an RCX communicate, talk, to another RCX or PC, you need to do three things: add an output stream to the sender’s program, add an input stream to the receiver’s program and establish a protocol for the communication. The next two sections present the code for the PC and RCX. The protocol is very simple. The RCX will listen until it hears something. It will then read and interpret the bytes as ints until the PC closes the stream. That is it for the protocol.

**PCSend.java**

```
import java.io.*;
import josx.rcxcomm.RCXBean;

public class PCSend {
    static final int[] dist = {30, 40, 30, 40, 20};
    static final int[] angl = {0, 90, 90, 90, -225};

    public static void main(String[] args)
    {
        send();
    }

    public static void send()
    {
        RCXBean rcxb = new RCXBean();
        try {
            rcxb.setComPort("USB");
            for (int idx = 0; idx < dist.length; idx++) {
                rcxb.sendInt(dist[idx]);
                rcxb.sendInt(angl[idx]);
            }
        } catch (IOException e) {
            System.err.println(e);
        }
        rcxb.close();
    }
}
```

The program at left is the PC part of communication example. The program uses RCXBean to manage the data stream with the RCX. It starts by creating a new RCXBean. Then it specifies the port the IR tower is on. This is all the setup the RCXBean needs to be ready to communicate with the RCX. To download the data, it loops through the two arrays of ints.

The code that uses the RCXBean is inside a try-catch block. This makes it possible to gracefully deal with the variety of unexpected that could occur, e.g. tower not plugged into PC, RCX is turned off during session, sun shines on IR port blinding it, etc. If an error occurs, the program prints an error message (something we can do on the PC but not on the RCX) and exits. A graceful exit beats the alternative, which, since this is hardware that we are dealing with here, could include hanging the system and requiring a re-boot.

You can cut and paste this into the IDE. Note that you compile this with the standard JDK compiler, not the LeJOS compiler. You will need to add the LeJOS class files, classes.jar, comm.jar, and percxcomm.jar to your project classpath. In JCreator, you click on Project>Project Setting and click on the required libraries tab. Add a new set libraries, naming it LeJOS, and add the three archive, jar, files. You also need to have tower.dll available when PCSend runs. The easiest place to put it is in the same directory as your class files.
RCXRecv.java

RCXRecv is the final piece of our first attempt at PC RCX communications. On the PC, RCXBean did several things for our communication program, including hiding the underlying data stream\(^1\). On the RCX side, there is no RCXBean, so we work with the streams directly. RCXPort is the physical infrared port on the RCX. It has an InputStream that we make into a DataInputStream (if you check the docs, InputStream only has methods for reading bytes while DataInputStream provides methods for reading all of the primitive types). Then we simply wait for ints to appear on the IR link and display them on the LCD screen. The read is inside a while true loop that never exits normally. Instead we use try-catch and have the PC close the data stream. Closing the data stream throws an IOEndOfFileException, which is a derived class of IOException, and the code in the catch block is executed, the number 8888 is displayed, and then the program pauses for 2 seconds and stops running by calling System.exit.

That is all there is to this program. Note that a lot of the messy details, and communications is extremely messy, is hidden from you by the RCXPort class and the hardware-based routines it uses. The examples of the details are beyond the scope of this book but you can get an idea of how many things can go wrong by looking the source for the communications classes that we have used.

\[^1\] Check out the code for RCXBean in the LeJOS source.

```java
import josx.platform.rcx.*;
import josx.rcxcomm.*;
import java.io.*;

class RCXRecv {
    public static void main(String[] args) {
        recv();
    }

    private static void recv() {
        try {
            RCXPort port = new RCXPort();
            DataInputStream in = new DataInputStream(port.getInputStream());
            while (true) {
                int val = in.readInt();
                LCD.showNumber(val);
            }
        } catch (java.io.IOException e) {
            LCD.showNumber(8888);
        }
        try {
            Thread.sleep(2000);
        } catch (Exception e) {
        }
        System.exit(0);
    }
}
```
RCX to PC Communications

The code for going the other way, from RCX to PC, is very similar to the previous example. At right is the method for the PC to receive data from the RCX. Note that it uses a while true loop that won’t exit normally and relies on the RCX to close the data stream to break out of the loop. To make PCSend into a receiver, add this routine to your PCSend class and replace send() with recv() in the main method (note the name of the class is now somewhat misleading but won’t make the program not work).

The code at left is the RCX side for sending data. Copy this method into your RCXRecv class and replace recv() with send(sa) in the main method. Note that the send method is passed an array of ints as a parameter. This is the first step in making send into method that can be used in other programs.

After you complete the cut and pastes into PCSend and RCXRecv, compile and test the two programs. Remember that you need to compile PCSend with the J2SDK compiler and RCXRecv with the LeJOS compiler.

At this point, the question arises of what additional code is needed to get two RCXs talking to each other. The answer is none. If you have two RCXs, try running RCXSend on one and RCXRecv on the other.
**Exercises**

1. Create a PCBBean that mirrors the functionality of RCXBean. What methods in RCXBean are not necessary in PCBBean? Describe PCBBean in a UML diagram.