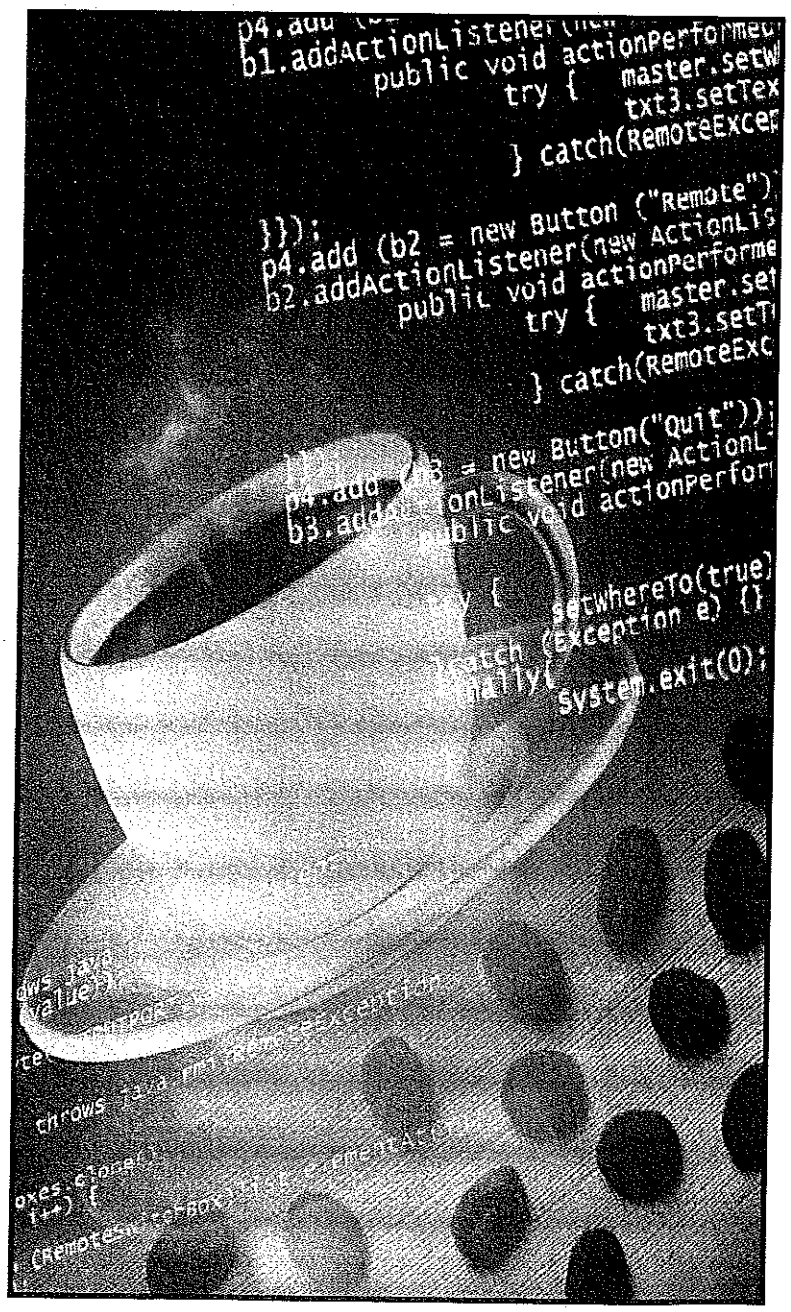
Serial and parallel ports require no additional operating system support and offer flexibility of use when it comes to interfacing. Les Hughes examines Java's support for this type of connectivity.

Ins and outs of Java



In previous articles, I have peeked at the nature of the Java technology and investigated custom interfacing at a low level. This time, I examine how Sun's Java Communications API – also known as `javax.comm` after the package in which the code is located – provides for a simple interface to RS232 and IEEE-1248 ports on Windows PCs and Solaris workstations.

Platforms

Both Win32 and Solaris platforms support `javax.comm`. Version 2 of the package can be downloaded from `java.sun.com` free of charge. Its documentation is somewhat basic although the code examples provide help in getting started. Included with the download is a number of applications; a simple reader and writer, a black box tester, a serial port chat program, etc.

Installation is relatively simple. Some files need to be copied manually, but the instructions are clear and helpful.

Linux users are not yet supported by Sun but a third-party product in the form of the Java Communications Library is available from `http://www.interstice.com/kev-inh/linuxcomm.html`. This package functions best with a newer 2.2 kernel, although certain features can be disabled during compilation to allow use with a 2.0.xx system. You will also need to obtain and install the Solaris version of `javax.comm`.

Architecture

The Java Communications API is extensible, meaning that developers are able provide support for other platforms (Linux, MAC, etc.) and interfaces (USB, ISDN, etc.) without waiting for Sun.

Within `javax.comm` is a basic communications framework around which extra classes can be built to provide support for specific hardware. As previously mentioned, this currently

extends to RS232 and IEEE-1248 ports.

Central to the whole package is the `CommPortIdentifier` class. `CommPortIdentifier` is a manager class used to determine available ports, negotiate access and to open ports. Actual hardware ports are represented by classes that extend `CommPort`. `CommPort` provides high level port methods, leaving specific things such as reading and writing to a subclass; examples of such subclasses are `SerialComPort` and `ParallelComPort`.

These classes, and their associated `CommDriver`, form the actual read/write interface to the hardware.

An example

Code examples are worth a thousand words and in the case of `javax.comm`, an example shows how simple the framework is to use.

The program below is a simple application to list all of the serial ports available to you on your particular platform. This program doesn't actually check that you have the correct hardware or that it is configured. It merely provides a list of ports that *could* be managed by Java. In list 1, the first three lines import various library classes. Note the way in which `javax.comm` is imported.

Two variables, known as fields in Java speak, are declared; a `CommPortIdentifier` and an `Enumeration` – an object that allows you to traverse through a collection or list of objects. The line:

```
ports =
CommPortIdentifier.getPortIdentifiers();
```

invokes a static method in the `CommPortIdentifier` class to obtain a list – or `Enumeration` – of manageable ports. Next, spin through this enumeration, checking each entry to see if it is a serial port. If the entry is of type `PORT_SERIAL` then print out its name.

At this point, you could also check for `PORT_PARALLEL` if we were trying to identify manageable parallel ports.

Reading and writing

Reading and writing to files or devices in Java is achieved using the 'Streams' input/output model. `InputStreams` act as a source of data to your program and `OutputStreams` are a sink for data to flow into from your program. In order to read and write to `javax.comm` ports, you need to obtain an `InputStream` and an `OutputStream`.

The abstract `CommPort` class provides methods called `getInputStream()` and `getOutputStream()` that enable us to obtain the required i/o objects. Once you have these objects, you can simply call `read()` and `write()` on them to send data to your hardware.

Figure 1 shows an interaction diagram of the steps involved in obtaining input and output objects. A Java code example shows this in action, **List 2**.

The main method in this case first creates a `ReadWrite` object and then tells that object to '`doSomething()`'. This method writes three bytes (0x01,0x02,0x03) and then reads a byte from COM1 in this case.

Note Java's error catching mechanisms at work in this example with the `try{...}catch(){}` blocks.

The streams-based i/o model allows us to add extra functionality to the i/o pipe by wrapping the basic `InputStream` or `OutputStream` with other stream objects.

For example, you could connect a stream capable of handling strings, decimal numbers or even serialised objects to our i/o channel. This would allow you to send different types of data to our hardware almost transparently. The full use of streams in this manner is beyond the scope of this article but extra information can be found in the references below.

List 1. A class that displays the ports that could be managed by Java.

```
import java.io.*;
import java.util.*;
import javax.comm.*;
  public class portlister {
    public static void main(String[] args) {
    CommPortIdentifier ID;
    Enumeration ports;
ports = CommPortIdentifier.getPortIdentifiers();
  while (ports.hasMoreElements()) {
    ID = (CommPortIdentifier) ports.nextElement();
    if
(ID.getPortType()==CommPortIdentifier.PORT_SERIAL){
      System.out.println(ID.getName());
    }
  }
}
}
```

List 2. Java code example for the interaction diagram, Fig. 1, which shows the steps involved in obtaining input and output objects. It writes three bytes, then reads a byte from COM1.

```
import javax.comm.*;
import java.io.*;
public class ReadWrite {
  SerialPort port;
  CommPortIdentifier ID;
  InputStream inp;
  OutputStream out;
  public ReadWrite() {
  try{
    ID = CommPortIdentifier.getPortIdentifier("COM1");
    port = (SerialPort) ID.open("ReadWrite",1000);
    out = port.getOutputStream();
    inp = port.getInputStream();
  }catch(Exception e) {
    System.out.println("Oops..something went wrong:");
    e.printStackTrace();
    System.exit(-1);
  }
  }
  public void doSomething() {
  try{
    out.write(1);out.write(2);out.write(3);
    System.out.println("Read: "+inp.read());
  }catch(IOException io) {
    System.out.println("Oops....");
    io.printStackTrace();
  }
  }
public static void main(String args[]){
  ReadWrite theApp = new ReadWrite();
  theApp.doSomething();
}
}
```

Port specifics

Serial and parallel ports both have specific behaviours and control signals. Each specific class – `SerialCommPort` and `ParallelCommPort` provides extra methods that enable the programmer to have full control over the device.

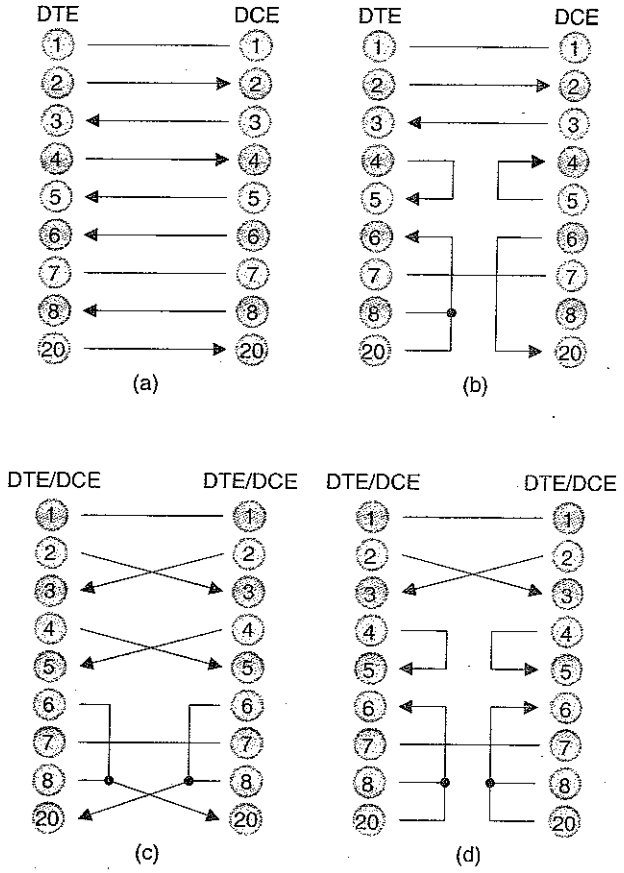
For example, the `SerialCommPort` class lets you set or check such lines as RTS/CTS, DCD, DSR, DTR, etc., while the `ParallelCommPort` class has methods for checking 'out of paper conditions', setting SPP/EPP/ECP mode, etc.

For a complete list of all available methods, refer to the API documentation included with the software download.

Following on from the serial port example above, in order to use a serial line correctly, certain parameters need to be set. To change the line speed, parity, data and stop bits and flow control discipline, use the `setSerialPortParams()` and `setFlowControlMode()` methods.

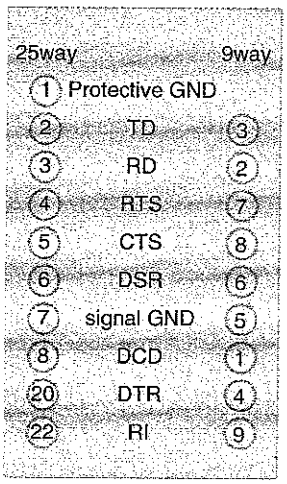
These two methods allow you to set line speed, data, stop and parity bits and to enable hardware (RTS/CTS) or software (XON/XOFF) flow control if required.

Some serial devices do not actually use the handshaking lines as the standard intends. For example, some packet radio modems use these lines to provide power. To set or to check to status of these lines, you can use such methods as setRTS(), isCTS() etc. Again, a full list is provided in the API documentation.



Zen and the Art of RS232

Serial lines can often cause problems. Is the device a DTE or a DCE ? Neither? Does your program expect and provide the correct flow control? The list is endless.



A break out box comes in handy but failing that, these diagrams show the most common connections, from a straight-through line to a full null-modem using 25 way connectors. Also 9 to 25way translations are shown.

Configuration (a) is a straight-through cable while (c) is a null modem. In (b) and (d), each side provides its own handshaking. Pin 8 is omitted for DCE/DCE. This diagram is based on the one in Horowitz and Hill's 'The Art of Electronics,' page 725 in my edition.

Listen here

The package `javax.comm` also supports Java 1.1 lightweight event subscription model, as used by the AWT, Swing, Beans, etc. Although it may sound somewhat complex, event subscription is really a simple concept to understand.

When you buy *Electronics World*, you could visit the newsagent every day until the latest copy arrives, or you could take out a subscription and let the postman deliver the magazine when it becomes ready.

Subscribers to events act in this latter manner. Your program expresses an interest in certain events and then gets on with more interesting things. When an event occurs, for example data arrives on a serial line, your program is notified and can deal with the event. If, at a later time your program is no longer interested in certain events, it can cancel its subscription.

The object that registers its interests in events is called a listener and you will find many `addXXXListener` type methods all through the core API. For serial ports, you would subscribe using `addSerialEventListener()`.

Unfortunately, `javax.comm` only supports one listener per serial port at the present. However, when modelling the actual device on the end of the serial line, you could include a method that receives serial events and then 'multicasts' these events onwards.

Events can be triggered by a number of changes on the port. Data arriving is probably the most common, but the API allows you to receive serial events for changes to CTS, CD DSR, RI, etc.

The parallel port driver also offers this type of functionality but is somewhat restricted to error and buffer empty conditions.

Putting it all together

One of my current pet projects is an MP3 player for my car. For those of you not familiar with MP3 compression, it is sufficient to say that one CDR of MP3 compressed audio can hold about 15 'normal' audio CDs! For more information on this technology, see www.mp3.com.

The system is based on a single board PC, which runs RedHat Linux, in the boot of the car. This PC has no keyboard or monitor – but it does have an ethernet adpator – so an interface was required for mounting on the dash.

Matrix Orbital manufactures a range of LCD and VFD modules, providing an RS232 and I^2C interface, keypad driver, programmable output lines, simple graphics capabilities and much more. Details of these modules can be found at `www.matrixorbital.com` or `www.linuxcentral.com`. For the MP3 player, I chose a 20 character by 4 lines LCD module with integrated 5V regulator. Its model number is *LKD204V*.

Of course, the system software was written in Java – what else? That is where `javax.comm` comes in. The *LKD204V* module has an RS232 interface and `javax.comm` is used to provide low level communications. Higher level processing of data is achieved through the use of various classes and abstractions.

Programming in the abstract

The code mentioned in this section can be obtained from my website, given later.

One of the many tenets of object-oriented programming is to isolate things that can change from things that stay the same. Isolating areas of code allows you to unplug old code and plug in new code without breaking the system.

The serial interface is one area that could change in future. Perhaps a USB or TCP/IP aware version of the module may become available. In fact, you may well want to replace the entire user interface module, so you should allow for some

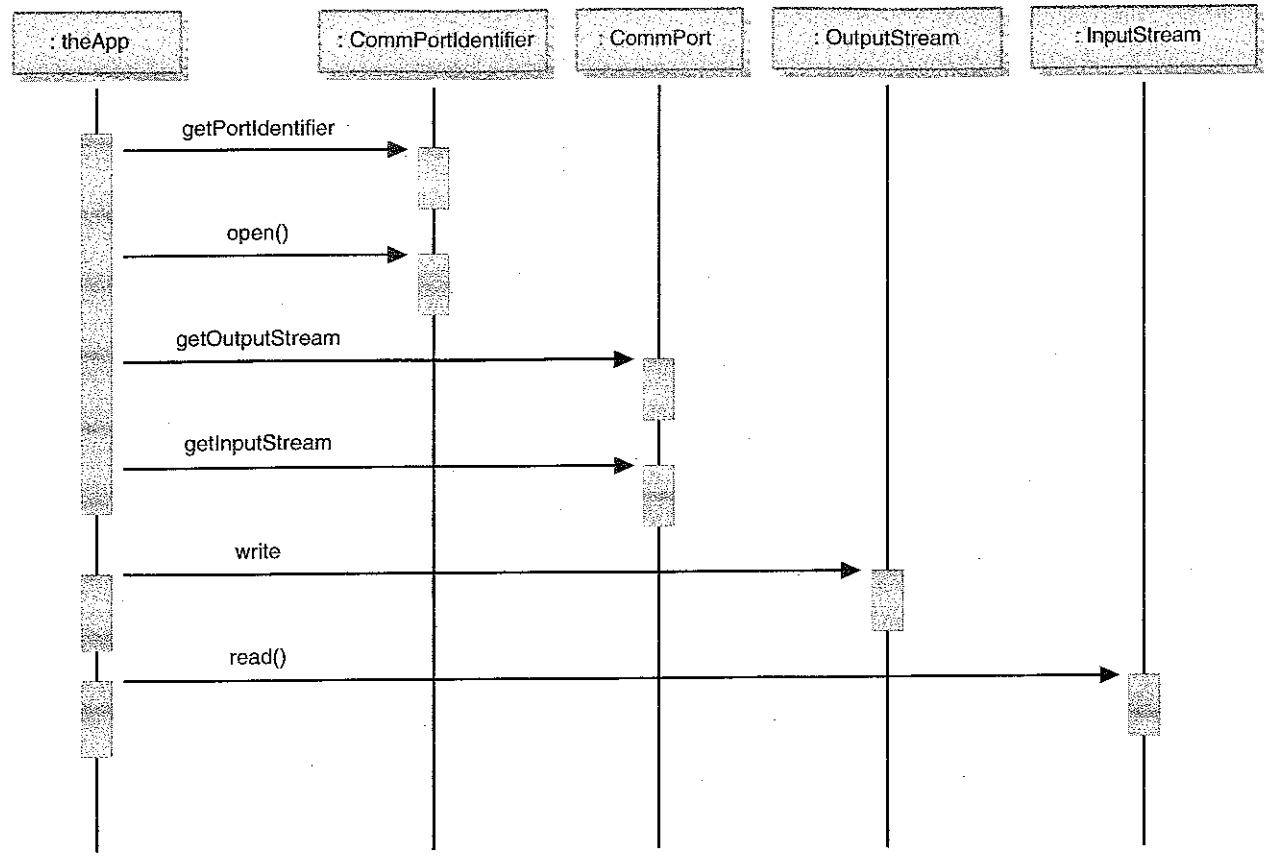


Fig. 1. Diagram illustrating interaction of the steps involved in obtaining input and output objects.

'future proofing' during your design. However, you also have to be careful not to be caught in the 'analysis paralysis' trap.

Firstly, an abstraction of the actual module was required. The class `LKD204.java` holds methods that duplicate the functionality of the module; writing text, creating custom characters, bargraphs, etc. *LKD204* is also responsible for loading a properties file from disk and creating the correct flavour of communications driver.

Each of the *LKD204* methods generates a command sequence and sends these commands to the module via an `OutputStream` obtained from the communications driver. Keypad input arrives either via an `InputStream` or through events – again obtained from the communications driver.

The `LKDCommDriver` is an abstract class, used as a 'placeholder'. In this system, the actual driver is using serial lines. By plugging in a different driver though, for I^2C for example, other communications channels become available and no other code would need changing. See the April 1999 issue of *Electronics World* for a discussion of abstract classes.

So far, I haven't mentioned the `javax.comm` serial communications side of things. In fact, all of the serial communications are factored out and encapsulated inside a concrete subclass of `LKDCommDriver`, `LKDSerialCommDriver`.

The concrete driver class for this application is called `LKDSerialCommDriver`. This class is responsible for managing serial communications with the *LKD204* module.

No other classes are aware of the fact that the module is hanging off of a serial line – it could be using the I^2C bus or even be connected to a remote serial port out on the network somewhere. `LKDSerialCommDriver` hides all of these gory details.

The complete listing for `LKDSerialCommDriver` is shown in **List 3**. The key operations are

- Obtaining a `CommPortIdentifier`
- Opening the port and obtaining a `SerialCommPort` reference
- Setting the serial port parameters
- Obtaining input and output streams
- Registering as an event listener for the port.

Again, isolating things that change, this class has no references to actual COM ports (`/dev/ttyS1` on my Linux

More information

The Java Communications API pages:
java.sun.com/products/XXXX
JCL for Linux 2.0+
www.interstice.com/kevinh/linuxcomm.html
Matrix Orbital
www.matrix-orbital.com
MP3 resources
www.mp3.com
MP3Mobile, inspiration for the author's MP3 player – written in C not Java :-(
utter.chaos.org.uk/~altman/mp3mobile
Full code examples are available from the author's web site:
www.parallax.co.uk/~leslieh

Formerly a Senior Lecturer in Software Engineering at the University of Greenwich, Les is now with Parallax, the Keane emerging technology practice (www.parallax.co.uk). He is a Sun Certified Java Programmer.

List 3. Complete listing for LKDSerialCommDriver. Key operations are, obtaining a CommPortIdentifier, opening the port and obtaining a SerialCommPort reference, setting the serial port parameters, obtaining input and output streams, and registering as an event listener for the port.

```
public class LKDSerialCommDriver extends LKDCommDriver implements SerialPortEventListener{
  SerialPort serialPort;
  CommPortIdentifier portID;
  InputStream inp;
  OutputStream out;
  public LKDSerialCommDriver(){
   try{
     portID = CommPortIdentifier.getPortIdentifier(System.getProperty("LKD204.CommPort"));
     serialPort = (SerialPort) portID.open("LKD204",1000); //1 secs timeout on open request
     int baudrate = 19200;
     try { baudrate = Integer.parseInt(System.getProperty("LKD204.BaudRate"));
     }catch(NumberFormatException n){}
     serialPort.setSerialPortParams(baudrate,
                  SerialPort.DATABITS_8,
                  SerialPort.STOPBITS_1,
                  SerialPort.PARITY_NONE);
     serialPort.setFlowControlMode(SerialPort.FLOWCONTROL_NONE);
     out = serialPort.getOutputStream();
     inp = serialPort.getInputStream();
     serialPort.addEventListener(this);
     serialPort.notifyOnDataAvailable(true);
   }catch(TooManyListenersException a){
     a.printStackTrace();
   }catch (PortInUseException piue) {
     piue.printStackTrace();
     System.exit(-1);
   }catch(NoSuchPortException nspe) {
     nspe.printStackTrace();
     System.exit(-1);
   }catch(UnsupportedCommOperationException ucoe) {
     ucoe.printStackTrace();
     System.exit(-1);
   }catch(IOException e){
     e.printStackTrace();
     System.exit(-1);
   }
  }
  public InputStream getInputStream() { return inp;}
  public OutputStream getOutputStream() {return out;}

  public void serialEvent(SerialPortEvent e){
   int data;
   Vector local = (Vector)listeners.clone();
   switch(e.getEventType()){
   case SerialPortEvent.DATA_AVAILABLE:
     try{
      data = inp.read();
     }catch(IOException ioe){
      return;
     }
     break;
   default:
     return;
   }
   for(Enumeration en = local.elements();en.hasMoreElements(); )
     {
      LKDEventListener lis = (LKDEventListener)en.nextElement();
      LKDEvent ev = new LKDEvent(LKDEvent.KEY_EVENT,data-64, this);
      lis.LKDEvent(ev);
     }

  }
}
```

List 4

```
import LKD204.*;
public class HelloLKD implements LKDListener {
  LKD204 lkd;

  public HelloLKD(){
  lkd = new LKD204();
  lkd.addListener(this);
     lkd.clear();
     lkd.setText("Hello World");
  }
  public void LKDEvent(LKDEvent ev) {
     System.out.println("Got an LKDEvent!");
     System.out.println("You pressed key" + ev.getData());
  }
public static void main(String args[]) {
     HelloLKD theApp = new HelloLKD();
}
}
```

PC or COM2 on Win32). Instead, it relies upon entries in a properties file. By using this configuration file approach, code can be moved between platforms without the need for re-compilation: the installation program need only provide a correct properties file.

The final thing that the `LKDSerialCommDriver` does is register as a listener for events on the serial port. On receiving an event from the actual RS232 driver, it reads the port and the fires another event to any objects interested in `LKDEvents`. In this way, the single listener limitation imposed by the serial port driver is circumvented. This all happens in the `serialEvent()` method.

Using the package

As ever, a 'Hello World' program demonstrates the use of the five LKD classes:

- `LKD204`,
- `LKDCommDriver`
- `LKDSerialCommDriver`
- `LKDEvent`
- `LKDEventListener`.

The program is shown in **List 4**. You can see that by encapsulating all of the gory details for each sub-section, gives the module a clean interface. You could now extend this idea further to encapsulate the LKD204 module into a user-interface framework. Again, this would allow you to unplug our present UI and to drop in for example, a touch screen LCD flat panel.

In summary

Communications provided by the Java Communications Framework is still in its infancy. Support is limited to Solaris and 32-bit Windows systems, although third party developers have ported to other platforms. In order to protect you programs from future changes and enhancements it is still necessary to provide an isolation layer by wrapping the `javax.comm` classes in your own code as demonstrated.

However, the `javax.comm` package provides a simple interface to two of the most popular types of port and affords the developer some of the platform independence and other benefits of the Java technology. ■