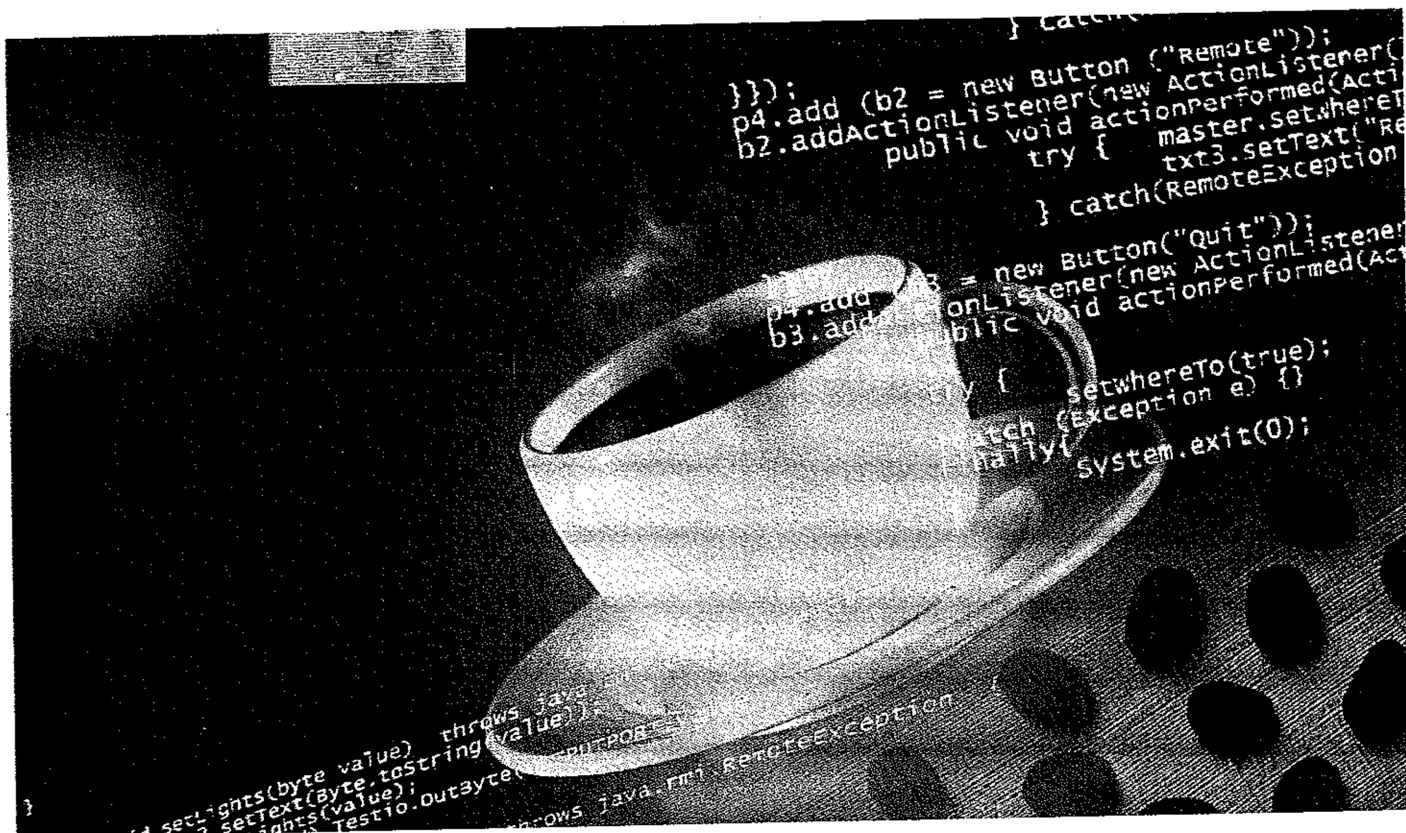

Interfacing with Java

Java prides itself on platform neutrality and portability. But what about computer interfacing with non-standard hardware? Les Hughes investigates.

Java technology was originally conceived as a simple means to power embedded devices, primarily set-top boxes – television decoders and such. However, the rise of the Internet saw Java drift off course somewhat.

Recent developments such as Embedded Java and Personal Java witness a shift back towards Java's roots. These products differ from the mainstream Java development kit in that they often require support from a real-time operating system (RTOS) of some kind. For example, WindRiver Systems was one of the first to offer Embedded Java on a true embedded platform – PowerPC, MIPS or Intel – by porting a JVM to their VxWorks RTOS.

This is fine if you have several thousand pounds to spend on development systems and a real-time operating system, but what about the original free JDK and a common PC? Can't we somehow bend Java to our will and accomplish some standard interfacing tasks? Of course we can.

Before I look at the role Java can play in interfacing, I'll briefly discuss the roles operating systems and device drivers have in allowing a program access to hardware devices.

Living with an operating system

Modern operating systems prohibit direct hardware access by user programs. Your program makes service

requests of the operating system – also known as the Kernel – which then proceeds to talk to the hardware through a device driver.

Instead of accessing hardware directly with a call such as inportb(), control is transferred to the operating system so that it can service your requirements, **Fig. 1**. This approach protects the computer from badly behaved programs and adds to the overall robustness of the operating system. The downside is that every custom device needs a custom device driver and for many operating systems, writing device drivers is no simple task, even for an advanced programmer.

However, all is not lost. Some operating systems developers – notably those working on Linux – actively encourage wide involvement in creating drivers for yet-to-be supported hardware. All of the required tools and several dozen production-quality drivers, which can be used as examples, are shipped with the OS as standard.

Also, it's a safe bet that if you're looking at doing something special with a piece of hardware on a Linux box, someone somewhere has already tried something similar and written a driver. The Linux Lab Project brings together a large quantity of these drivers and is worth a visit before reaching for your compiler.

Beyond Linux, 'generic' drivers are available for some operating systems. These drivers are not targeted at a particular expansion card; they simply provide indiscriminate access to the computer's hardware. While nowhere near robust enough for production quality systems, the use of a simple generic driver with a custom prototype board can give more than satisfactory results in an experimental environment.

A word of warning though: a generic driver is just that. Remember that they allow unrestricted access to memory and hardware, sacrificing the protection provided by the OS. It is incredibly easy to crash a system when experimenting with a generic driver. It is also theoretically possible to corrupt disks, damage monitors and inflict no end of suffering upon your poor PC if you write data into the wrong locations.

With reasonable care though, a generic driver provides a simple and safe solution to the 'unsupported hardware' problem.

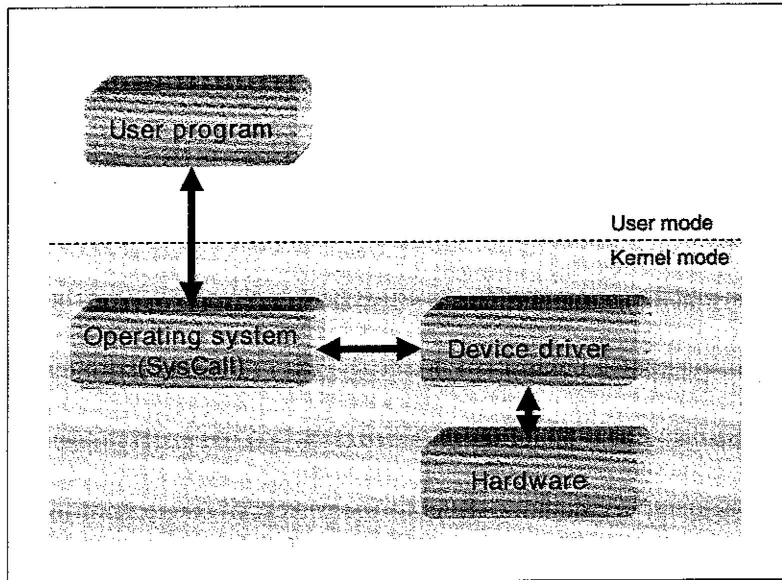

Fig. 1. Instead of accessing hardware directly with a call such as inportb(), control is transferred to the operating system so that it can service your requirements.

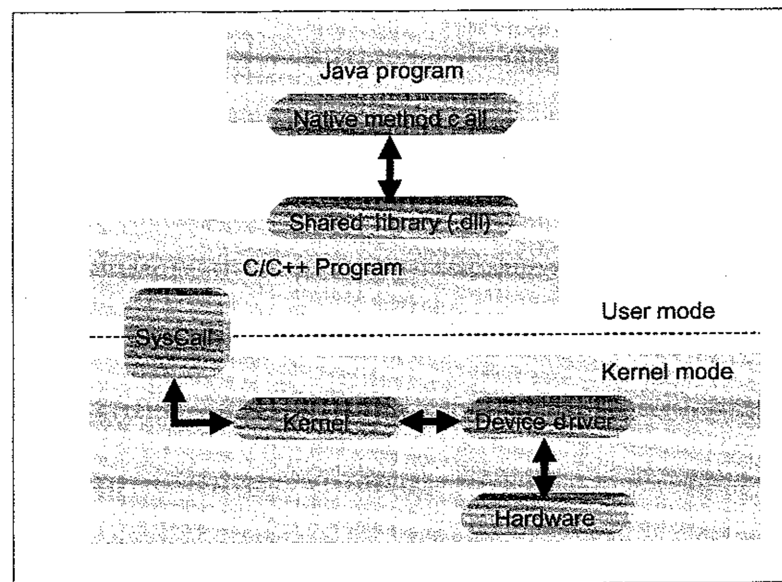

Fig. 2. The library acts as a 'wrapper' that converts Java 'method' calls into the corresponding device driver calls.

Talking to the natives

Having touched on some of the aspects of working with a modern multi-tasking, multi-user operating system, you could be forgiven for thinking that things can only get easier. Unfortunately, once you try to break free from the confines of the Java virtual machine, you encounter a number of vendor imposed difficulties.

In order to 'hook' into a device driver you need to run native code – code compiled specifically for the target platform. This code is written in C (or C++) and compiled into a shared library (.dll or .so) with a tool such as Visual Studio or GNU gcc, depending upon the platform.

EEPort.java notes

You may be wondering, "Why have an open and close method that doesn't really do anything?" A perfectly valid point. However, if you were to move your Java code to Linux for example, then you would have to actually open the device to gain access – which is not the case with the generic win32 driver. Of course, you would have to re-write your native code as well as modifying EEPort's implementation of open and close at that time. But by defining the open/close interface now, you minimise the effects of moving to another platform later.

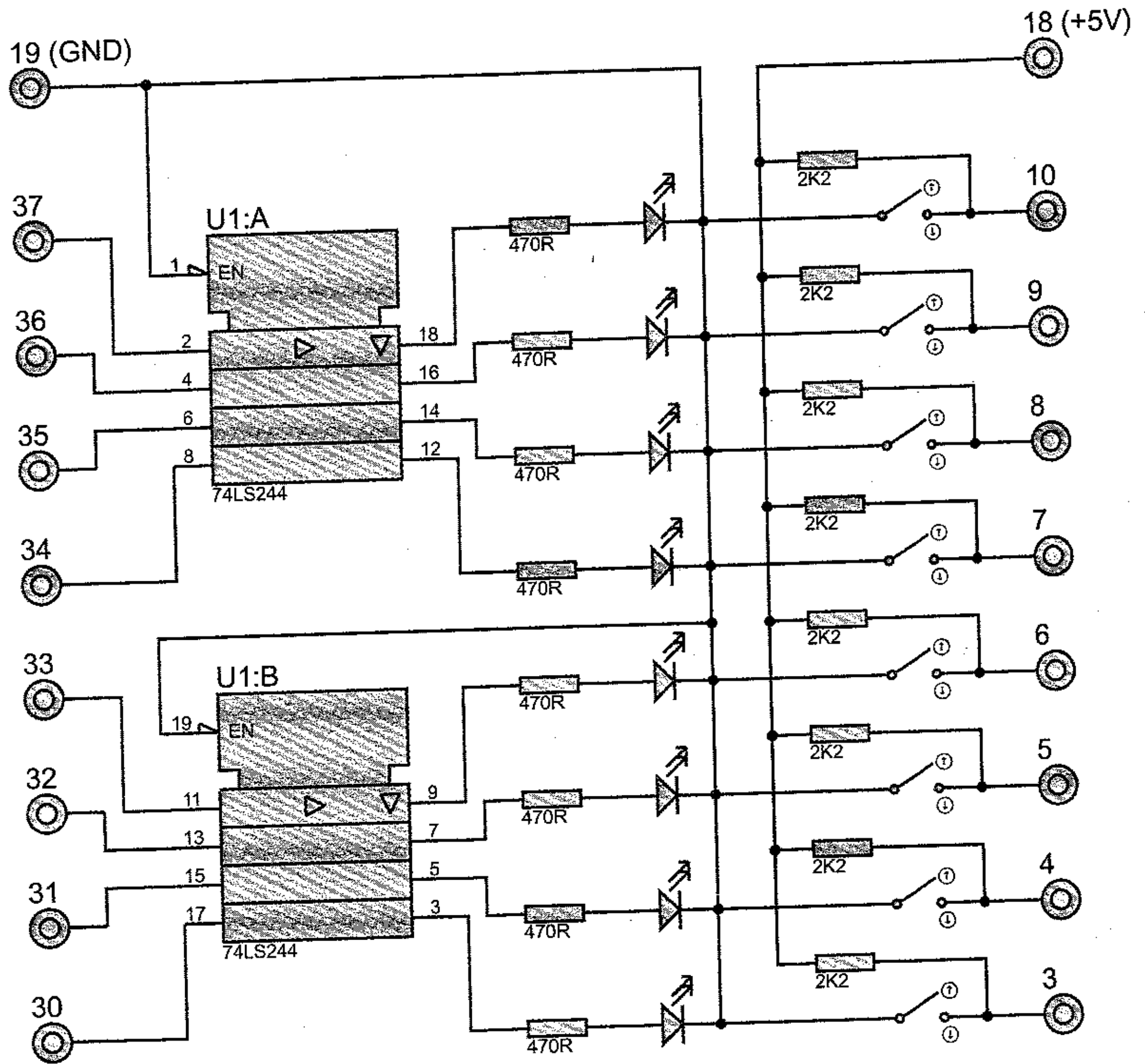

Fig. 3. Simple lights and switches circuit for interfacing to two ports of the 8255 card.

This library acts as a 'wrapper' that converts Java 'method' calls into the corresponding device driver calls, **Fig. 2**. Different platforms require different device driver calls and a different library.

Unfortunately, these 'hooks' from the Java virtual machine into native C/C++ code come in different, incompatible flavours depending upon the Java product used.

As happens all too often in computing we find that the world is divided into two camps; Microsoft's JDirect/RNI system and everyone else's Java Native Interface API. JDirect is only available on Win32 platforms giving simple access to the Win32 API. JNI on the other hand should be available wherever you find a Java virtual machine.

Originally, Microsoft decided that its way was best and declined to implement JNI. A court ruling late last year ordered Microsoft to include, amongst other things, JNI and an updated JVM is available from their website.

JNI is 'binary compatible' across platforms and Java virtual machines. Your Java code for interfacing on a Sun *shouldn't* need recompiling on a PC: you just need to supply the correct library file. Contrasting this is the fact that code written to take advantage of JDirect's simplicity will never run on anything other than a Win32 box. For this reason, I only consider JNI from here.

As with most aspects of the Java Technology, JNI could warrant a whole series of books and articles itself. Since this article mainly concerns wrapping Java methods into native API calls, I won't delve too deeply into the JNI. A further source of JNI information is listed in the references.

How EEPortTest.java works

Most of the code is spent in setting up the graphical user interface. Lines of particular interest are 27-9 which create three ports, A, B and a configuration port and lines 73-78 that actually read and write data. Notice on line 73 how we 'message' inputportA and ask it to read some data for us. Line 78 shows a similar write action, this time on outputportB.

The development system

Obviously, interfacing is hardware specific and you need to choose some hardware. Omega Engineering produces an 8255 based i/o card that, unlike some others, includes support for interrupt driven i/o.

A Linux driver is available to take advantage of this feature. However, I will concentrate on using this hard-

ware without interrupt support, on Win32 with a generic port driver.

In order to generate some input and to show some output, a test circuit will be required. Perhaps the simplest – and most widespread – is the 'lights and switches' style box; an array of eight LEDs and eight switches, **Fig. 3.** The 8255 card mentioned above provides three 8-bit ports but you only need two for what's described here; PortA for input and PortB for output.

A 74LS245 octal buffer is used to protect the outputs of the i/o card and pull-up resistors ensure correct logic levels on the inputs.

For Win32 development you will need a copy of JDK > 1.1.5, which is free from java.sun.com. You will also need a means of producing a Windows DLL, a suitable interface card and the DriverLINX Port i/o Driver for the test circuit. There's more on the driver and DLL below.

Some of these tools and utilities are available from my website, whose address is given later. Once you have obtained the hardware and software, you should follow the supplied instructions regarding installation and configuration. In particular, you should read the README file that comes with the DriverLINX Port i/o Driver.

A simple example

Enough theory. Now I will show how to create a simple application to demonstrate some of the principles involved in interfacing with Java.

Assuming you have installed all of the required hardware and software – drivers, compilers, etc. – you first need to plan your design. Java is an object-oriented language and you should be creating software 'things' if you want to exploit the power of the language fully.

For our model, an obvious 'thing' is a representation of the Port itself. Your software Port will be responsible for interacting with the OS via JNI, and will provide you with the necessary port 'behaviours'; read, write, open and close with perhaps some kind of control behaviour.

Other client objects – those that require the services of a port – aren't interested in the gory details, they just want to read and write some data. Most probably, they also won't care about others who are using the port. Your port thing should therefore marshal port accesses, locking the resource so that only one part of the program may use the resource – again, an important task, **Fig. 4.**

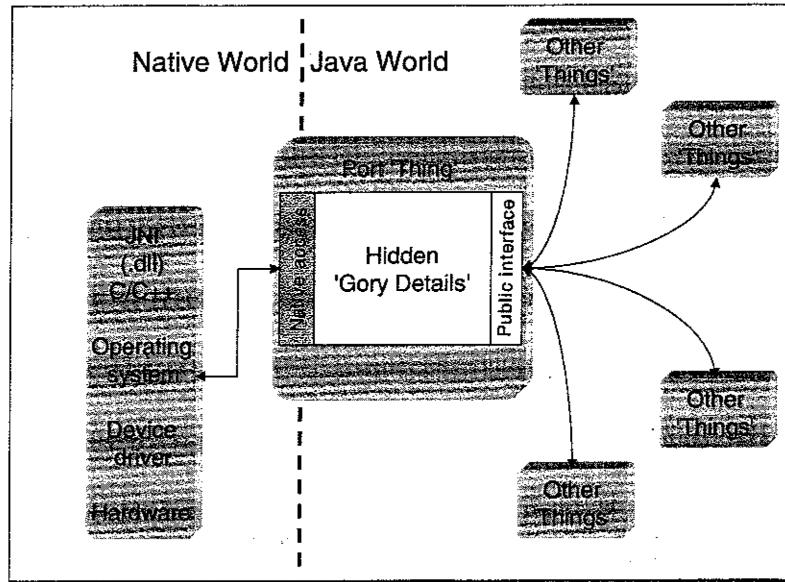

Fig. 4. Your port 'thing' should marshal port accesses, locking the resource so that only one part of the program may use the resource.

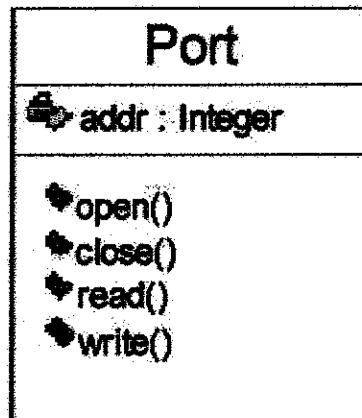

Fig. 5. Class diagram for a port, with four methods – open, close, read, write – and a privately stored address field.

Taking these ideas further, perhaps it would be a good idea to create a 'PortManager' of some kind? This isn't really necessary for your first attempts at interfacing, but for a larger system a Manager would be made responsible for allocating ports to client objects, etc. For the moment though, it is best to put these management and access ideas to one side and remember to manage the ports yourself.

The Java side of the port doesn't really have to do much for the time being. You need to define the 'services' it provides and to simply map these into our native calls. Later, you could expand each of these services, keeping the names the same but enhancing what each one does. This iterative approach is often used in object-oriented development – you can change the implementation as long as you leave the interfaces alone.

List 1. Defining an i/o port in Java.

```
public class EEPort{

static {
    System.loadLibrary("javaio");
}
    int addr;
    boolean isOpen;

    public EEPort(int addr) {
        isOpen = false;
        this.addr = addr;
    }

    public void open() {
        isOpen = true;
    }
    public void close() {
        isOpen = false;
    }
    public byte read() {
        if(isOpen) {
            return _read(addr);
        } else {
            return 0;
        }
    }
    public void write(byte data) {
        if(isOpen) {
            _write(addr, data);
        }
    }
    public void setDebug(boolean flag) {
        _setDebug(flag);
    }
    private native void _write(int addr, byte data);
    private native byte _read(int addr);
    private native void _setDebug(boolean flag);

}
```

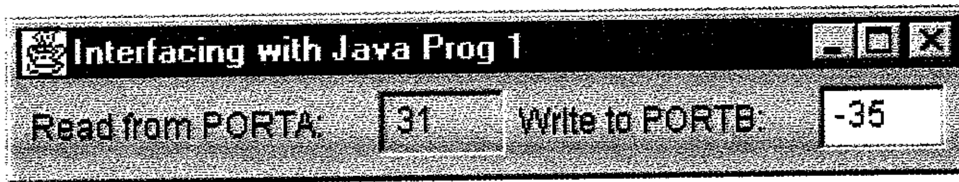

Fig. 6. List 2 in operation. This application reads port A every 100ms, displaying the result in a box.

List 2. Simple application that reads the value on PortA every 100ms, displaying results in a text box.

```
import java.awt.*;
import java.awt.event.*;
public class EEPortTest extends Frame implements Runnable {
        TextField   portAText;
        TextField   portBText;

        Label       portALabel;
        Label       portBLabel;
        EEPort      inputportA;
        EEPort      outputportB;
        EEPort      configport;
        Thread      runner;
        static final byte controlword = (byte)0x99;
        static final int PORTA           = 0x300;
        static final int PORTB           = 0x301;
        static final int PORTC           = 0x302;
        static final int CONFIG          = 0x303;
    public EEPortTest() {
        super("Interfacing with Java Prog 1");
        inputportA  =   new EEPort(PORTA);
        outputportB =   new EEPort(PORTB);
        configport  =   new EEPort(CONFIG);
        configport.open();
        configport.write(controlword);
        configport.close();
        inputportA.open();
        outputportB.open();
        setLayout(new FlowLayout());
        portAText = new TextField("0",3);
        portAText.setEditable(false);
        portBText = new TextField("0",3);
        portALabel = new Label("Read from PORTA:");
        portBLabel = new Label("Write to PORTB:");
        add(portALabel);
        add(portAText);
        add(portBLabel);
        add(portBText);
        (runner = new Thread(this)).start();
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent w) {
                System.exit(0);
            }});
        this.pack();
        this.setVisible(true);
    }
    public void run() {
        while(true) {
        try {
            byte data = inputportA.read();
            String sometext = Byte.toString(data);
            portAText.setText(sometext);
            data = Byte.parseByte(portBText.getText());
            outputportB.write(data);
            Thread.currentThread().sleep(100);
        } catch(InterruptedException expt) {
        } catch(NumberFormatException nfexpt) {}
        }
    }
    public static void main(String args[]) {
        EEPortTest porttest = new EEPortTest();
    }
}
```

Figure 5 shows a class diagram for your port, with four methods – open, close, read, write – and a privately stored address field. The class operates thus; an object creates a port to write to, telling it the address it's interested in. The port stores this for future reference. The client object then opens the port, reads and writes data from the previously announced address and finally closes the port.

Implementing a port

Let's implement the port in Java, **List 1**. First, define the class name (EEPort). Your EEPort class needs to load your compiled C/C++ library before it can do anything. This is achieved using the loadLibrary method.

You then define the constructor for your class. This piece of code is executed whenever another object creates a port to work with. The constructor accepts an integer representing an address for you to work with. The constructor also sets the isOpen flag to false, indicating that the port is closed.

Next, define your methods. Open and close simply change the isOpen flag. Read and write check the isOpen flag. If the port is currently open, perform the required action via a native call (_read and _write) shown at the end of the listing.

These native method calls are the actual hooks out into the real world. Notice how they have no body. The actual implementation for these methods takes place in our C/C++ library. To facilitate debugging, a setDebug method is included that causes the native code to generate some helpful messages.

Working through a typical scenario, imagine a client object creates and opens a port. It then calls read which in turn calls _read. This causes some native C/C++ code to execute that finally accesses the hardware. The data read from the hardware bubbles up through the various called methods and ends up back at our application.

Compiling

Assuming you are using the Sun JDK, you would compile EEPort.java using the command:

C:\> javac EEPort.java

This should produce EEPort.class. Now we have our port, let's create a simple application showing how we use it before we look at the platform specific C/C++ code.

List 2 is a simple application that reads the value on PortA of the 8255 every 100ms, displaying this data in a text box. The application also reads the value entered in a second text box and sends it to PortB, to appear on the LEDs. A screen shot of the application in action appears in **Fig. 6**. Again, use javac to compile:

C:\> **javac EEPortTest.java**

which should result in EEPortTest.class.

Going native

The last piece in the jigsaw is the native code; i.e. code written in C acting as a simple wrapper. You can now complete this jigsaw by creating your library .dll.

The links that connect Java to C are the JNI native methods. These are defined in our Java file but our C compiler knows nothing of Java declarations. Somehow we have to translate our native method declarations into C function prototypes. Fortunately, Sun has kindly provided a tool to do this for us, as part of the Java Development Kit.

The javah program scans class files extracting the required information concerning JNI calls. It then assembles this data into a C header file that defines the necessary function prototypes ready for you to implement. To generate a header file from your EEPort.class file, run the following:

C:\> **javah –jni EEPort**

This should produce EEPort.h that defines three functions corresponding to read, write and debug methods. These prototypes are somewhat confusing in nature due to the amount of information that needs to be passed outside of the JVM into native land. You won't use most of this data and you can ignore the various pointers, etc. passed to you.

A more advanced program could call back into the Java virtual machine, invoke a Java method from C or even create another JVM. All that is needed here is to send and receive some data. The completed library program to accomplish this is shown in **List 3**.

The first few lines import standard library headers as well as the header generated above and the header supplied with the port driver. The generic driver exports the functions DlPortReadUchar and DlPortWriteUchar for reading and

List 3. This program calls back into the Java virtual machine and invokes a Java method from C.

```
#include <jni.h>
#include <windows.h>
#include <conio.h>
#include "Dlportio.h"
#include "EEPort.h"
#define INP DlPortReadPortUchar
#define OUTP DlPortWritePortUchar
BOOL debug;
JNIEXPORT void JNICALL Java_EEPort__1setDebug
  (JNIEnv * env, jobject obj , jboolean flag) {
        debug = flag;
}
JNIEXPORT void JNICALL Java_EEPort__1write(JNIEnv * env, jobject obj, jint
portaddr, jbyte outval)
{
     if (debug) {
          printf("values are port=%d; value=%d\n", portaddr, outval);
          printf("About to try to write to port\n");
     }
          OUTP(portaddr, outval);
          if (debug)
          printf("writing to port done\n");
}
JNIEXPORT jbyte JNICALL Java_EEPort__1read(JNIEnv * env, jobject obj, jint
portaddr)
{
          jbyte inval;
          if (debug) printf("About to read byte from port\n");
          inval = (jbyte)INP(portaddr);
          if (debug) printf("byte read, value is %d\n", inval);
          return(inval);
}
```

List 4. A datagram transmitter. This class fires out a series of Internet protocol datagrams; packets of data based on the Unreliable Datagram Protocol.

```
import java.net.*;
import java.io.*;
public class DgramTx implements Runnable {
    private App ip;
    private InetAddress inet;
    DatagramSocket sock;
    private Thread runner;
    public DgramTx(App ip) throws Exception {
        this.ip = ip;
        inet = null;
        sock = new DatagramSocket();
        (runner = new Thread(this)).start();
    }
public void run() {
    while(true) {
        if(inet != null) {
                byte[] buf = new byte[32];
        buf[0] = ip.read();
                DatagramPacket dg = new DatagramPacket(buf, buf.length, inet,
12345);
                try {
                 sock.send(dg);
                }catch(IOException i) {}
        }
        try{
        Thread.currentThread().sleep(100);
        }catch(InterruptedException ex) {}
    }
}
public void setWhereTo(InetAddress inet) {
    this.inet = inet;
}
}
```

List 6. This, the main application, creates the GUI, the two i/o ports and the networking objects and joins them all together. It also intercepts read and write operations from DgramTX & DgramRX so as to be able to display information on the GUI.

```
import java.net.*;
import java.awt.*;
import java.awt.event.*;
public class App extends Frame {
    private TextField theHost;
    private TextField theOutput;
    private TextField theInput;
    private Button setHost;
    EEPort inp;
    EEPort oup;
    DgramTx tx;
    DgramRx rx;
    public App() throws Exception {
        super("Datagram Application");
        //COnfigure the 8255 card
        EEPort ctl = new EEPort(0x303);
        ctl.open();
        ctl.write((byte)0x99);
        ctl.close();
        ctl = null;
        //Create our in and out ports
        inp = new EEPort(0x300);
        oup = new EEPort(0x301);
        inp.open();
        oup.open();
        //Create our Network TX
        tx = new DgramTx(this);
        //Create our Network RX
        rx = new DgramRx(this);
        //Build our GUI
        setHost = new Button("Set Host");
        theHost = new TextField(20);
        theOutput = new TextField(5);
        theInput = new TextField(5);
        setLayout(new FlowLayout());
        //Add Components to form GUI
        add(new Label("RxD from Network"));
        add(theOutput);
        add(new Label("TxD to the Network"));
        add(theInput);
        add(new Label("Enter Remote Hostname"));
        add(theHost);
        add(setHost);
        //Add an actionlistener monitoring
        //the setHost button
        setHost.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent a) {
                try{
                    tx.setWhereTo(InetAddress.getByName(theHost.getText()));
                }catch(UnknownHostException ex) {
                    theHost.selectAll();
                    Toolkit.getDefaultToolkit().beep();
                }
            }});

        //Add a listener to close the App
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent w) {
                System.exit(0);
            }
        });
        pack();
        setVisible(true);
    }
    //A method to write Data to the port
    //and to the textfield
    public void write(byte data) {
        theOutput.setText(Byte.toString(data));
        oup.write(data);
    }
    //A method to read Data from the port
    //and show in the textfield
    public byte read() {
        byte data;
        data = inp.read();
        theInput.setText(Byte.toString(data));
        return data;
    }
    //The main method
    public static void main(String args[]) throws Exception{
        App theApp = new App();
    }
}
```

writing. You alias these to INP and OUTP for short.

The setDebug function appears next, setting the BOOL defined on the previous line to true or false. Notice that you only use the last parameter passed into the function, jboolean flag.

You will also notice how the names for each function (read, write and debug) contain information as to where the Java method was defined. For example, the _read method appears as Java_EEPort__1read in the native code.

One final thing is the way in which Java data types are prefixed with 'j'; jbyte, jint, etc. This helps you keep Java types and C types separate.

The read and write functions follow a similar format to the setDebug function. They do not need to access back into the JVM, or reference the object that called them and so they ignore the first two parameters. In fact, the read function only uses the jint portaddr value returning a jbyte and the write function uses a jint for the port address and a jbyte for data.

This C code is compiled into a Windows .dll – in this case javaio.dll - using Visual Studio or similar. If you are building your own .dll instead of downloading the precompiled version, you must remember to include your Java distribution's 'include' directory with your project (for jni.h) and to link against dlport.lib, which is part of the port driver package.

Exact instructions vary from compiler to compiler, so you should refer to your manuals. Once you have produced your .dll, you should copy it into the same directory as EEPort.class This way, the JVM will be able to find your native library when it needs to.

Running the test application

By now you should have EEPort.class, EEPortTest.class and javaio.dll in one directory. Start the test application:

C:\> **java EEPortTest**

You should see a window similar to that in **Fig. 6**. Enter some data into the right-hand text box and watch the lights change. Now change the switches and observe the left-hand text box reflect these changes.

Moving on

One of the nice things about Java is the simplicity with which otherwise complicated tasks can be accomplished. As a slightly more advanced

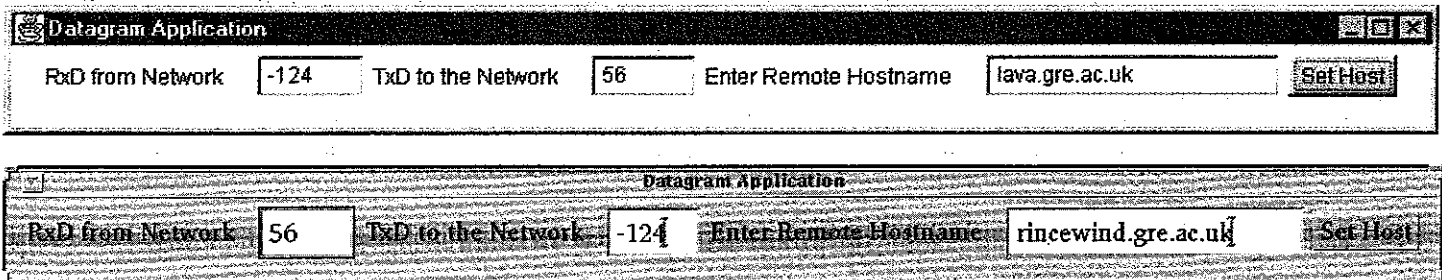

Fig. 7. Screen shots taken while list 6 – the networking demonstration application – is running.

example, a networked set of lights'n'switches can be easily constructed now that you have your i/o infrastructure.

The code shown in **List 4** represents a datagram transmitter. This class fires out a series of Internet protocol datagrams; packets of data based on the Unreliable Datagram Protocol (UDP). UDP provides for connectionless transmission where delivery is not guaranteed. However, since you are just experimenting with networking, this protocol is more than adequate.

The DgramTX class reads data from our input port every 100ms. This data is sent across the network in a UDP datagram, to be received by a DgramRX, **List 5**.

The DgramRX listens out for incoming datagrams, unpacking data from those it receives and sends this data to the output port. These two classes form the heart of our transport system.

Joining everything up is a simple class named App – the main application, **List 6**. This class creates the GUI, the two i/o ports and the networking objects and joins them all together. It also intercepts read and write operations from DgramTX & DgramRX so as to be able to display information on the GUI.

A screenshot of the running application is shown in **Fig 7**, performing reads and writes from an NT machine to a Sun workstation – all with the same basic Java code.

In order to make use of these classes, you will need a friend with an interface card, a lights'n'switches box and Internet access. Simply get on line, enter the host name or IP address of the other machine in the GUI and start sending data.

Your switch settings appear on their lights and *vice versa*. You can also enter your machine's name or IP address that will cause your LEDs to reflect the setting on your switches.

List 5. DgramRX listens out for incoming datagrams, unpacking data from those it receives and sends this data to the output port.

```
import java.net.*;
public class DgramRx implements Runnable {
    private App op;
    private DatagramSocket sock;
    private Thread runner;
    public DgramRx(App op) throws Exception {
        this.op = op;
        sock = new DatagramSocket(12345);

        (runner = new Thread(this)).start();
    }
    public void run() {
        while(true) {
            try {
                byte buf[] = new byte[32];
                DatagramPacket dg = new DatagramPacket(buf,
buf.length);
                sock.receive(dg);
                buf = dg.getData();
                op.write(buf[0]);
            }catch(Exception e) {
                    e.printStackTrace();
            }
        }
    }
}
```

In summary

To summarise, interfacing in Java is achieved in a similar manner to any other language, except for the complication of having to join platform independent Java with native C code. Once this task is complete, the generated .dll and .class files allow 'invisible' hardware access permitting you to exploit most of Java's advanced features; networking, multiple threads, object serialisation, etc. ■

Biography

Formerly a Senior Lecturer in Software Engineering with the University of Greenwich, Les is now with Parallax Solutions (www.parallax.co.uk) working in the area of object technology and, of course, Java. He is a Sun Certified Java programmer and has worked on a number of JNI related projects; Internet controlled hardware, object storage FPGAs and a rather nice networked coffee machine.

Need more information?

The Linux Lab Project http://www.llp.fu-berlin.de/
Microsoft's Java pages http://www.microsoft.com/java/
Omega Engineering (8255 card) http://www.omega.co.uk/uk/index.html
DLPort Driver from Scientific Software Tools http://www.sstnet.com/
JNI information http://java.sun.com/products/jdk/1.1/docs.html
Software mentioned in the article (javaio.dll etc) can be found at http://www.parallax.co.uk/~leslieh/index.html