Native Instrumentation Board Interface For Java-based Programs

Richard E. Pfile and William Lin

Purdue School of Engineering & Technology
Indiana University-Purdue University at Indianapolis, Indiana

Abstract

Java is becoming a popular programming language for PC-based applications programs for many reasons. Java's language rules force a natural structured approach to writing code, its strong data typing eliminates some of the subtitle errors encountered in C/C++ language, it's thoroughly object oriented, it's platform independent, and it is relatively easy to write programs with a Windows GUI using the language.

Java has a significant downside when used in data acquisition and control applications. Because Java programs run on a virtual machine and the language is strictly platform independent, there are no provisions for I/O such as that encountered when using a PC based instrumentation board. To access this type of I/O, the Java program must leave the virtual machine it runs on and run native code. A software interface can be written that will allow all of the board functionality needed to be accessed using Java calls. This paper presents the techniques to interface a Java program to native hardware using a National Instruments card as the example native hardware. The technique can be readily adapted to a variety of boards. When using the interface, student's programs can make Java function calls to access an instrumentation card.

Introduction

Java, which has long been a popular language for web applets, is becoming more popular for stand-alone applications. Programs with graphical user interfaces (GUI) are easy to write in Java, the language is fully object-oriented and programs written in Java can run, without modifications, on different platform types such as Sun workstations, Apple computers and PCs. In addition the software is free. There is reason to believe that Java's popularity will grow.

To facilitate running on different computer platforms, Java programs run on an interpreter called the Java Virtual Machine (JVM). The only system-specific I/O Java programs can perform is that allowed by the JVM such as file, keyboard, and computer monitor access. Java programs do not have a direct way of getting past the Java Virtual Machine interpreter to interface to non-standard I/O devices such as an instrumentation board. The designers of the Java language realized there would be times when Java programs need to leave the JVM and go into the nether

world of native code and they provided a way out of the virtual machine environment called the Java Native Interface (JNI).

This paper describes a simple Java program with a Windows GIU and outlines the process to go off the Java Virtual Machine to access native hardware in a computer. The hardware application is a digital I/O port on the National Instruments 6025E instrumentation card. A very simple PC-hosted program with two user control buttons is developed. One button sets a bit on the National Instruments card high and the other button clears the bit to a low. To fully comprehend this paper, familiarity with an object oriented programming language such as C++ or Java is necessary. Readers not familiar with object-oriented can get a sense of the effort required and can perhaps decide if they want to perhaps switch from a language such as Visual Basic to Java. The paper starts with the description of the Java program and then discusses the hardware interface.

One point on terminology is that subroutines (in Basic) are called methods in Java and functions in C/C++. When discussing Java code, subroutines will be referred to as methods and when discussing C code, subroutines will be referred to as functions.

Java Graphical User Interface

A Java program with a GUI, requires a Window frame. Other components such as buttons that are used in the program will be positioned inside the window frame. The frame is the familiar window frame with a title bar at the top containing the minimize and close buttons. The program GUI with the window frame and two buttons labeled *Conveyor On* and *Conveyor Off* is shown in figure 1.

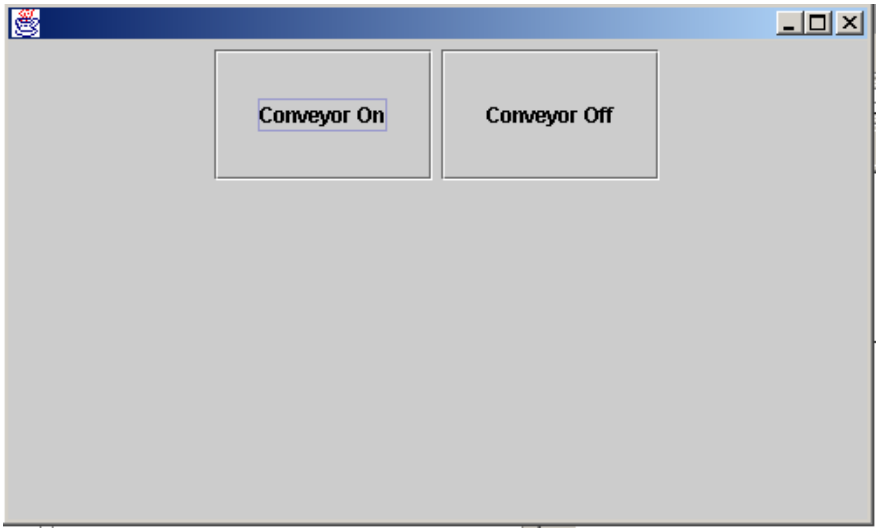

Figure 1 Window Frame with Buttons

The following code fragment shows the code to start the Java program and generate a window frame. Line numbers have been added to select lines of the code to help with the description. Only an overview of the program is given. Details about the various Java classes and rules are well documented by Sun Microsystems.[1]

```
1.      public class MainFrame
        {
2.       public static void main(String[] args)
         {
3.         MainFrame mainFrame = new MainFrame();
4.         mainFrame.App();
         }
5.       void App()
         {
6.         JFrame f = new JFrame();
7.         f.setSize(500,300);
8.         f.show();
         }
        }
```

Java programs start at function *main*. The preceding code starts at the line 2. Most Java programs then create an instance of an application class and call methods in that class. In this case an instance of the application class *MainFrame* is created (line 3) and then the *App()* (line 4) method in that class is called. Once in *App()* the window frame is created. Java has a class called *JFrame* that generates a window frame. A *JFrame* object named *f* is created (line 6) and method *setSize* (line 7) is called to make the window 500 pixels wide and 300 pixels high. The *show* method (line 8) is then called to make the window visible. At this point an empty window frame is generated. The empty frame is shown in figure 2.

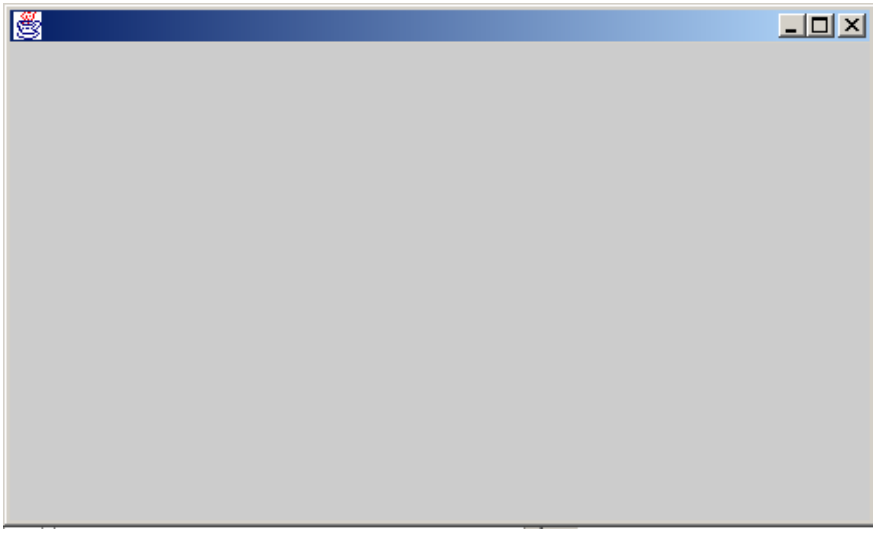

Figure 2 Empty Window Frame

Code was written to generate two user-interface buttons to control the I/O board. One button sets a bit on the National Instruments parallel I/O port when clicked. The other button clears a bit on the card. The buttons were labeled *Start Conveyor* and *Stop Conveyor* implying this particular I/O bit is connected to a conveyor.

The code to create the buttons is shown below. The Java class for a button is *JButton*. Two *JButton* objects are declared named *conveyorOnButton* and *conveyorOffButton* (lines 1 and 2). The button objects are set to be 125 pixels wide and 75 pixels high using the *setPreferredSize* (lines 3 and 4) function. After the button objects are created they are added to the window by getting the content pane for the window frame and then adding the buttons to the content pane (lines 5, 6 and 7).

```
        public class MainFrame
        {
1.       JButton conveyorOnButton = new JButton("Conveyor On");
2.       JButton conveyorOffButton = new JButton("Conveyor Off");

         void App()
         {
3.         conveyorOnButton.setPreferredSize(new Dimension(125,75));
4.         conveyorOffButton.setPreferredSize(new Dimension(125,75));

5.         Container contentPane = f.getContentPane();
6.         contentPane.add(conveyorOnButton);
7.         contentPane.add(conveyorOffButton);
         }
        }
```

The code that connects a button press event by a user to a program method that will eventually set or clear a bit on the National Instruments card is shown below. Methods *ConveyorOnButton* and *ConveorOffButton* are called whenever the user clicks their respective buttons. The *addActionListener* method (lines 1-4 and 4-8) instructs the operating system to call these methods whenever there is a button click event on the respective button. Details concerning *addActionListener* and programming button events can be found in reference 1. The *ConveyorOnButton* (lines 9-10) and *ConveyorOffButton* (lines 11-12) methods call methods in a class *JavaNative* where the native interface calls to the National Instruments card are made.

```
        public class MainFrame
        {
         void App()
         {
1.         conveyorOnButton.addActionListener(new ActionListener()
           {
2.           public void actionPerformed(ActionEvent e)
             {
```

```
3.              ConveyorOnButton();
                }
4.          });

5.          conveyorOffButton.addActionListener(new ActionListener()
            {
6.            public void actionPerformed(ActionEvent e)
              {
7.              ConveyorOffButton();
              }
8.          });
          }

          void ConveyorOnButton()
          {
9.          JavaNative javaNative = JavaNative.GetInstance();
10.         javaNative.SetDevice1Port0Line1Low();
          }

          void ConveyorOffButton()
          {
11.         JavaNative javaNative = JavaNative.GetInstance();
12.         javaNative.SetDevice1Port0Line1High();
          }
        }
```

Java Native Interface

Interfacing to the National Instruments card requires going off the Java interpreter to native C code to talk to hardware in the PC. The interface software is referred to as the Java Native Interface (JNI). Java methods that call native C code functions must be declared as native. Shown below is a Java native method that calls a C language function named *MakeDevice1Port0Line0Output()*. The C code is shown later.

```
public native void MakeDevice1Port0Line0Output();
```

It is useful to setup Java native methods for all of the I/O that may be needed in a single class. The following Java calls are native and invoke C code in the DLL to setup an I/O bit as either an input or output. The first method calls C code that sets up bit 0 of device 1 to be an output. The next method calls C code that sets up bit 0 to be an input.

```
public native void MakeDevice1Port0Line0Output();
public native void MakeDevice1Port0Line0Input();
```

To be complete methods should be written for all of the I/O bits. To keep this example short only one bit is shown. All of the Java code to access the C language DLL could be placed in a single class such as *JavaNative* that students can use (along with the corresponding C language DLL) to access the National Instruments card. If the *JavaNative* class contains all of the I/O functions on the National Instruments card that are needed, the code in this class will never need to be modified.

The next two methods are native as well and again invoke C code in a DLL. The first method calls C code that sets bit 0 high. The next method calls C code that clears bit 0 to a low.

```
public native void SetDevice1Port0Line0High();
public native void SetDevice1Port0Line0Low();
```

The following method calls C code that reads the state of bit 0 of device 1. It returns a true if the line is high and false if the line is low. Bit 0 must be an input for this method to work properly.

```
public native boolean GetDevice1Port0Line0();
```

To load the native C code for execution, the method *System.loadLibrary* must be called by the Java code. The following code shows the call to the *loadLibrary* method that specifies the DLL (in this case NatInstrInf) to load.[2] If the library file cannot be loaded, the Java program exits.

```
public class JavaNative
{
 private void jbInit() throws Exception
 {
  try
  {
   System.loadLibrary("NatInstrInf");
  }
  catch (Exception e)
  {
   System.out.println("Cannot load NatInstrInf.dll...aborting");
   System.exit(1);
  }
 }
}
```

The *loadLibrary* function is normally placed in the Java native code class. Setting up the class for a singleton access allows placing the *loadLibrary* method in the constructor that will only be invoked once.[3]

The actual native code that interfaces the National Instruments card is written in C and is built as a DLL by specifying a DLL project.[4] The C code for the Java Native Interface must follow specific rules. For each native method declared in the Java program, a C function needs to be written. A utility program named *javah.exe* comes with the Java Development Kit (JDK) that

generates a C language header file with the function prototypes for the C program functions. One C function prototype is generated with the proper names and decoration for each Java native method. The JDK available as a free download from Sun Microsystems.[5]

The Java native call followed by the C language function prototypes generated by the *javah* utility program is shown below. The prefix to the function name (Java_javainf_JavaNative) is the file path name for the Java method. It is useful to group the Java native methods in a single file so all of the C function prototypes are generated at once.

```
//Java native method
public native void SetDevice1Port0Line0High();

//C function prototype
JNIEXPORT void JNICALL Java_javainf_JavaNative_SetDevice1Port0Line0High
(JNIEnv *, jobject);
```

The function prototypes can then be cut and pasted to generate the C language functions. Definitions of names such as JNIEXPORT and JNICALL are defined in the header file JNI.H that is included with the Java Development Kit. The C language functions make calls to the appropriate library functions in the National Instruments Applications Program Interface (API). National Instruments refers to their API as the NI-DAQ library.[6]

National Instruments C Language Function Calls

The C language function below shows a call to a National Instruments function called *DIG_Out_Line*. The function writes either a 0 or 1 to the line specified and requires four arguments, the device number, port number, line number and the value (either 0 or 1) to be written to the line. The function below writes a 1 to line 0 of port 0. For this function to work properly, line 0 would need to have been made an output.

```
//Method:    SetDevice1Port0Line0High
JNIEXPORT void JNICALL Java_javainf_JavaNative_SetDevice1Port0Line0High
  (JNIEnv *, jobject)
{
        //Device=1, Port=0, Line=0, Value=1
        DIG_Out_Line(1, 0, 0, 1);
}
```

The following function call sets line 0 low.

```
//Method:    SetDevice1Port0Line0Low
JNIEXPORT void JNICALL Java_javainf_JavaNative_SetDevice1Port0Line0Low
  (JNIEnv *, jobject)
{
        //Device=1, Port=0, Line=0, Value=0
```

```
        DIG_Out_Line(1, 0, 0, 0);
}
```

The following block of code programs line 0 of port 0 to be an output. *DIG_Line_Config* is the function in the National Instruments library called to configure the bit. The function requires four arguments, the device number, port number, line number and the line direction (1 for output and 0 for input).

```
//Method:    MakeDevice1Port0Line0Output
JNIEXPORT void JNICALL Java_javainf_JavaNative_MakeDevice1Port0Line0Output
  (JNIEnv *, jobject)
{
        //Device=1, Port=0, Line=0, Output=1
        DIG_Line_Config( 1, 0, 0, 1);
}
```

The following method reads port 0, line 0 and returns a true if the bit is high and a false if the bit is low. National Instruments function *DIG_In_Line* is passed the device number, port number, line number and the address of a 16-bit variable which is set to a 1 or 0 depending on the value read on the line. The line should be programmed as an input to use this function.

```
//Method:    GetDevice1Port0Line0
JNIEXPORT jboolean JNICALL Java_javainf_JavaNative_GetDevice1Port0Line0
  (JNIEnv *, jobject)
{
        i16 state = 99;
        //device, port number, line number, state
        DIG_In_Line( 1, 0, 0, &state);          //state set to either 0 or 1
        return(state);   //0 or 1
}
```

The NI-DAQ functions are located in a DLL named *nidaq32.dll*. Library *nidaq.lib* must be linked with your C language project. Header file *nidaq.h* must be included in the C code.

Conclusion

Java has many laudable features and we believe it is a great language to consider for an instrumentation application that requires a Windows GUI. This paper has presented the steps, via a simple example, required to interface a Java program to native code. If embarking on your own program we suggest reference 1 as an excellent starting place if you need to learn Java. Reference 2 is a thorough guide to the Java Native Interface. The first two chapters provide a good JNI overview. The National Instruments documentation is quite good and provides both high-level and low-level functions to access their data acquisition cards.

[1] Cay Horstmann and Gary Cornell, *Core Java Volume 1 – Fundamentals*, Sun Microsystems Press/Prentice Hall, Upper Saddle River, New Jersey, 1999.

[2] Rob Gordon, *Essential JNI Java Native Interface*, Sun Microsystems Press/Prentice Hall, Upper Saddle River, New Jersey, 1998.

[3] Cay Horstmann and Gary Cornell, *Core Java Volume II – Advanced Features*, Sun Microsystems Press/Prentice Hall, Upper Saddle River, New Jersey, 2002.

[4] Jeffrey Richter, *Advanced Windows Third Edition*, Microsoft Press, Redmond Washington, 1997

[5] Sun Microsystems http://java.sun.com/downloads

[6] NI-DAQ User Manual for PC Compatibles Version 6.7, National Instruments, Austin, Texas, January 2000.

Richard E. Pfile is a professor of Electrical Engineering Technology at IUPUI. He received his B.S. in Electrical Engineering from the University of Louisville and his M. Eng. in Computer Information and Control Engineering from the University of Michigan. He has received several teaching awards from the School of Engineering and Technology at IUPUI. He teaches courses in microprocessor systems, computer networks, and digital signal processing and develops embedded and PC-based software for industry. He has twenty years of teaching experience and eight years of industrial experience, including three years as a systems engineer.

William Lin is currently a faculty member in the Purdue School of Engineering and Technology at IUPUI. Prior to coming to IUPUI, Bill has served as a faculty member at The Pennsylvania State University, Wayne State University, and DeVry Institute of Technology. Before joining IUPUI, he was a Full Professor in EET at the DeVry Institute in North Brunswick, New Jersey. During his tenure at DeVry, in addition to teaching and developing courses he has been involved in the development and teaching of the Team problem-solving curriculum since its inception. Dr. Lin received his Ph.D. degree in Electrical Engineering from The Pennsylvania State University, a M.S. degree in Physics from the University of Southern Mississippi, and a B.Ed. in Science Education from Taiwan.