3rd International Conference on Electrical & Computer Engineering
ICECE 2004, 28-30 December 2004, Dhaka, Bangladesh

JHI (JAVA HARDWARE INTERFACE): DEVELOPMENT OF A NEW HARDWARE INTERFACING TOOL FOR JAVA

Boshir Ahmed[1] and A.K.M. Khaled Ahsan Talukder[2]

Department of Computer Science and Engineering
Rajshahi University of Engineering and Technology, Rajshahi
Email: boshir_bd@yahoo.com[1] and akmkat_bd@yahoo.com[2]

ABSTRACT

This paper mainly represents an introduction to the new tool for interfacing hardware using Java. C/C++ has many versatile options for interfacing devices through parallel or serial ports. But network compatibility and flexibility is not completely offered by C/C++. On the other hand, Java is developed for network applications. It supports many privileges for network-based applications. But direct interface with electronic devices through parallel or serial ports are still out of the scope of Java environment. For this purpose, a new tool for port based hardware interfacing is developed. It is named as **"Java Hardware Interface (JHI)"**. This paper describes the design and implementation of JHI.

1. INTRODUCTION

The aim of the development of the Java language was to build a general platform for network accessible appliances. However, the fact is that, it has no standard method for accessing ports (serial, parallel, USB etc.) through programs. By adding some function mappers for accessing PC ports, these limitations of Java necessitate the development of JHI [5]. Access to the ports is accomplished by writing codes using other languages (such as C/C++ etc.) and it is implemented as Native Interface for the Java [1][2][4]. Then those native methods are called from Java environment. This linkage between JHI, Java and C/C++ can be illustrated as a Venn diagram (refer to fig.1).

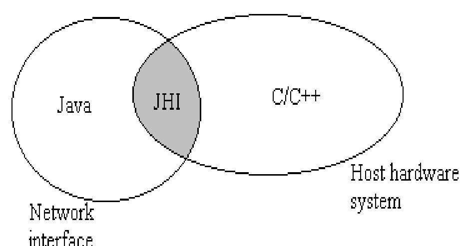

Fig. 1 Java, C/C++ and JHI

2. JHI ARCHITECTURE

In JHI, the functions for accessing hardware ports are written in pure C and are not directly supported by Java. That is why; there are still some demands for function mappers. JHI fills this gap by adding native C code that is accessible by the JVM. In fact, JHI provides communication between the port interfacing C/C++ functions and Java programs by building a layer on top of the C/C++ functions. The architecture of JHI can be depicted as fig.2 (refer to fig.2).

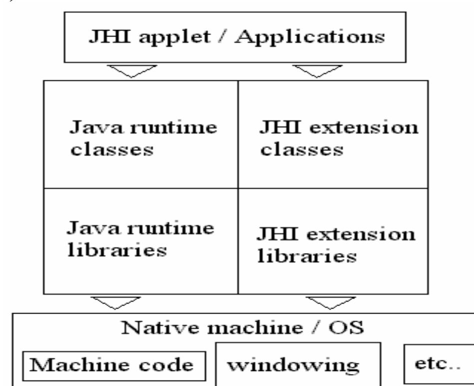

Fig. 2 JHI Architecture

ISBN 984-32-1804-4

JHI uses the Java Native Interface (JNI) [1-5] to pass all C/C++ function calls from Java to the machines native C/C++ library. So, JHI is consisted of two layers –

- Java classes
- Native library

The figure 3 shows how the JHI layer connects the Java runtime environment with hardware level machine code.

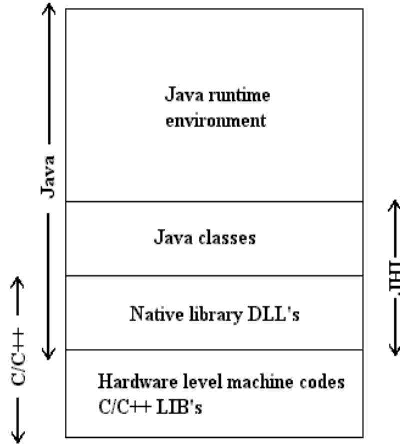

Fig.3: Layer view of JHI, Java and C/C++

3. DESIGN OF JHI

So, it becomes clear from the preceding section, that, JHI has two counter parts, the Java classes and Native library.

a. Native Libraries: Native libraries are created to support the hardware port accessing. All of them are dynamic link libraries (DLL files) written in pure C. The main codes to access the PC ports are implemented in those DLL files. So, they work for the low-level direct hardware access schemes. [3][4]

b. Java classes: next layer is composed of java classes. These classes load the c/c++ code from stub defined in the byte code. The stubs redirect the instruction pointers to native dll codes that would have to be loaded in main memory during run time [3]. For example, the method **outbyte()** (used to send data to specified port) is declared in these class files. When they are called, the c replica of **outbyte()** , **_outp()** [5] is loaded from the native library and executed. This loading phase is accomplished during the execution time of class modules. [4]

4. IMPLEMENTATION OF JHI

The native library for JHI is **"JHI.dll"** and the Java class file to access the DLL is the **"JHI.class"** In **"JHI.class"** the methods such as, **outByte()** or **inByte()** are just declared and they are implemented in C/C++ in **"JHI.dll"**. During runtime, when **outByte()** or **inByte()** methods are called, they are loaded using the procedure **System.loadLibrary()**.

For example, to interface LPT port of a PC, the typical code for implementation may look like –

```
        class LPT{
/* new JHI object named 'lpt' is created */

        JHI lpt = new JHI();
/* now use the methods defined in 'lpt' */

        public void sendData(int data){
/* Data in the argument 'data' is sent to LPT port */

        lpt.outByte(0x378, data);
                }
        public void initialize()
        {
/* point to the mode set address 0x378+2 */

/* 0x00 defines the LPT port as output port */

         lpt.outByte(0x378+2, 0x00);
                }
        }
```

In the same way, we can implement **inByte()** method to read a byte from a specified port address. Now, the class LPT can be used create some objects to implement the LPT port interfacing using Java. Some typical methods that are embedded in JHI are –

Method name	Return type	Return value	Arguments
outByte()	void	Void	Port address, data
inByte()	int	Data to be read	Port address
outWord()	void	Void	Port address, data
inWord()	int	Data to be read	Port address

The execution sequence of JHI becomes clearer from fig.4 –

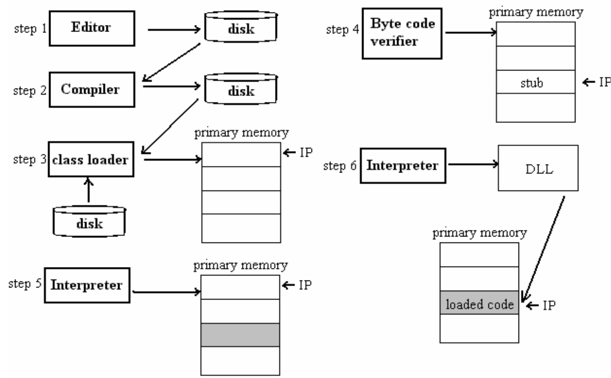

Fig. 4 JHI in action.

The overall procedure can be described as –

Step 1: Program is created in editor and stored on disk.
Step 2: Compiler creates bytecodes and stores them on disk.
Step 3: Class loader puts bytecodes in memory
Step 4: Byte code verifier confirms that all byte codes are valid and do not violate Java's security restrictions.

Step 5: Interpreters reads bytecodes and translates them in to a language that the computer can understand, possibly storing data values as the program executes.

Step 6: Interpreter finds that a stub is defined in code and then calls loader again to load the specified stub from DLL

5. EXPERIMENTAL RESULTS AND SIMULATION

The performance of JHI is tested on a circuit designed to rotate a DC motor in clockwise and counter clockwise direction and the system is interfaced with LPT port.[5] The graphics user interface (GUI) is designed with Java swing and it has some push buttons to control the motion of the motor.(refer to fig.5)

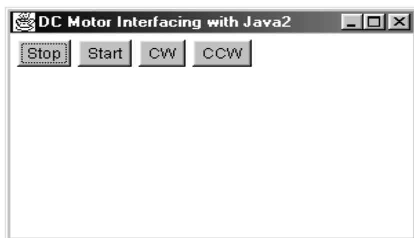

Fig. 5 GUI of commander

The circuit used to control the motor is designed as (refer to fig.6) –

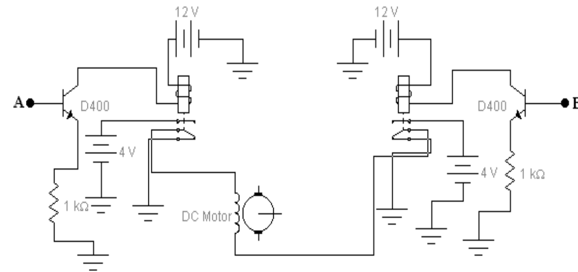

Fig. 6 Circuit used in simulation

The program uses the first two pins of the 8bit data line of the parallel port [5] and the signal from these two pins decides the behavior of the rotation. The table below illustrates the response of the motor with signal emanating from two pins A, B of the LPT port.

A	B	DIRECTION OF ROTATION
0	0	Stop
0	1	Clock wise
1	0	Counter clockwise
1	1	Invalid input

6. NETWORK IMPLEMENTATION OF THE SYSTEM

Since, the port interfacing becomes possible with the aid of JHI, the next step is to implement the overall system in a network. Java supports full flexibility in developing network applications. In the experiment a daemon (server part) and a commander (client part) program is developed to control the device from remotely located host PC. [3][4]

a. Daemon: The daemon was developed to continuously listen to the specified network port; here this port number is 5000. When it receives a hit (response) from other host running the commander (client PC), it establishes a connection through Berkeley Socket Interface (BSI) and gets ready to receive any command from the client side program. This daemon is capable of accessing LPT port through JHI interface. So, the host PC with motor controller device should run this daemon (Server) program [3][4].

b. Commander: The commander works as the client program for the daemon. It has a GUI and according

to the user command, it generates some specified messages that have to be sent to daemon through the network [3][4]. The daemon then translates these messages to motor controlling command that is compatible to the proper JHI interface.

The overall system is illustrated in fig.7 –

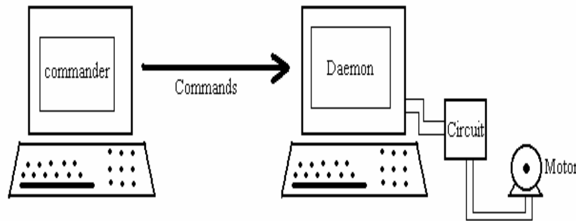

Fig. 7 Daemon and Commander

Algorithm for the Client-Server implementation –

Algorithm:

1. start daemon program.
2. wait until any connection found.
3. if a connection is found, then establish it.
4. receive commands from the client.
5. then control the LPT interfaced circuit with proper JHI methods according to received command.
6. if the process ends, then go to step 2.

7. LIMITATIONS OF JHI

The motivation behind the development of JHI is the issue of remotely located hardware accessibility. However, it also holds two important limitations –

a. Potential Security Restrictions: In order to avoid some security risks, applets cannot use native methods. Also the loading of DLL's can be restricted and sometimes their loading is subject to the approval of the security manager [2][3][4].

b. Portability loss: Since, the native code is contained in a DLL; it must be present on the machine that is executing the Java program. Further, because each native method is CPU and OS dependent, each DLL is inherently non-portable. Thus, application that uses JHI will be able to run only on a machine for which a compatible DLL is installed [2][3][4].

8. FUTURE OF JHI

a. Distributed automation system: High potential for distributed computing can be introduced as a new platform for distributed automation system. Different devices located at different host computer in a network can accomplish smaller task and their results can be integrated in a server system. This approach may become fruitful for industrial purposes.

b. Accessing hardware through WWW: Even now, JHI is not still compatible to applet, but developing a smaller daemon-commander system in a host PC, hardware access may become possible through applets as well as through WWW. So, the future of JHI shows the path to the development of a web-based technology to control port-interfaced hardware.

9.CONCLUSION

In this paper, the development of a new platform for Java hardware interfacing tool is described. Obviously, JHI meets the potential for the network based applications (applet) as well as low-level hardware based applications. That is, it will work as the superset connector for two environments. Although, some limitations still exists, the future of JHI leads to the introduction of distributed automation system. However, the methods developed here, are external port oriented, in future, other system based task will be possible through JHI.

REFERENCES

[1] Sheng Liang, *The Java Native Interface: Programmers' Guide and Specification*, 1st ed., Addison-Wesley, June 1999.

[2] Rob Gordon, *Essential JNI*, 1st ed., Prentice Hall Pty. Ltd, March 1998.

[3] JAVA 2 SDK, Standard Edition Documentation, Version 1.3.1, Sun Microsystems Inc. (http://java.sun.com)

[4] Patric Naughton & Herbert Schildt, *JAVA 2: The Complete Reference*, 2nd ed., Tata-McGraw Hill.

[5] Craig Peacock*, Interfacing the Standard Parallel Port*, 1998, http://www.senet.com.au/~cpeacock.