Working with C++ Data in Java

Presented by:
Jessica Winblad

Outline

- Working with C++ DLLs
  - Java Native Interface (JNI)
- I/O with Files & Streams
  - Byte/Bit Order
  - Data Type translation

Real World Example

- Powering test-equipment for 12.5 seconds

Project Requirements

- Send voltage out a parallel port for a specified time in seconds, to a half-second accuracy
- Run on a Windows 2000 or XP machine
- Integrate with a test-suite already written in Java

Engineering Challenge

- With Windows 98 this would have been easy
- Windows 2000 provides less transparent access to the machine's hardware
  - Only hardware device drivers can directly access the hardware
  - You can't write device drivers in Java

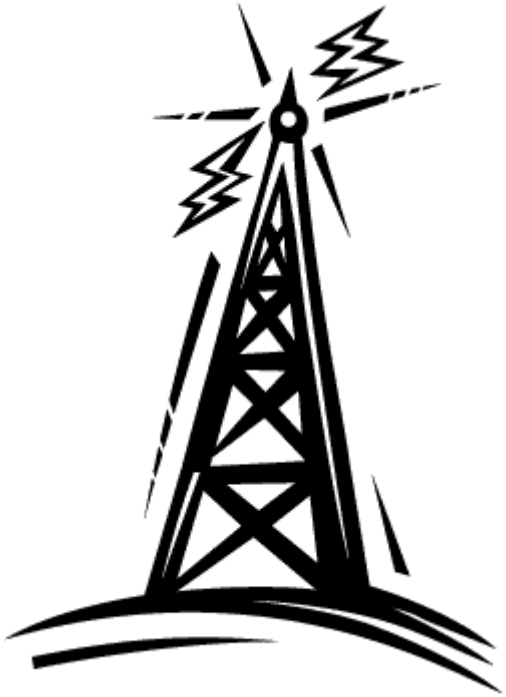
Solution

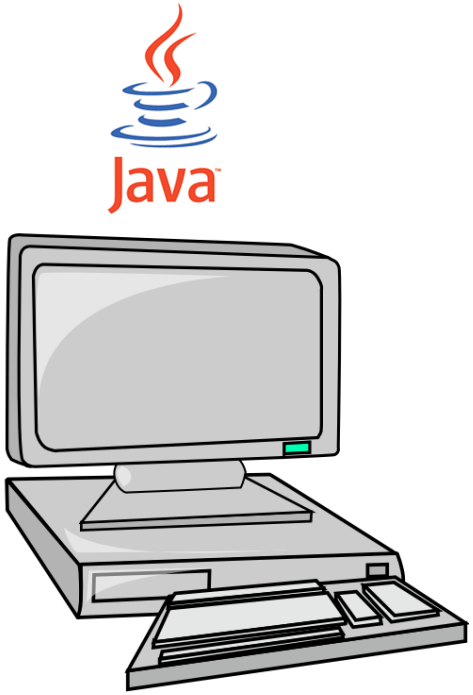
- Use a device driver DLL that allows direct programming of the parallel port
- Use JNI to access the device driver

JNI (Java Native Interface)

```
class ioPort {
    public native void Out32(
        short PortAddress, short data);
    static { System.loadLibrary("jnpout32pkg");}
}
…
ioPort pp = new ioPort();
pp.Out32(0x378, currentVal);
```

Another Real World Example





Byte and Bit Order Matter

- Big Endian ("Network Byte Order")
  - Eg: **Motorolla 68k processor**
- Little Endian
  - Eg: **x86 PC**
- May need to test for "endianness"
- Java has classes to help with correcting byte order (eg: **java.nio.ByteOrder**)

A source value of 0xFFFE = 1111 1111 1111 1110
Could be read as:

- 1111 1110 1111 1111
- 1111 1111 0111 1111
- etc.

Converting Data Types

- Applies to both streams and files
- If you have a double in C++ should you use a readDouble() method of your java stream/file reader to read it? (No)

- Java and C++ do not always have the same names for equivalent primitive types.
- Some types don't map nicely.

Internal Sizing of Data Types

Size	1 byte	2 bytes	4 bytes	8 bytes	16 bytes
	8 bits	16 bits	32 bits	64 bits	128 bits
C++	byte	short	int/long	long long	__int128
Java	byte	short	int	long	BigInteger

Size	1 byte	2 bytes	4 bytes	8 bytes	16 bytes
	8 bits	16 bits	32 bits	64 bits	128 bits
C++	char/bool		float	(long) double	
Java	boolean	char		float	double

* C++ sizes are OS/compiler dependent (Win32 shown)

Signed/Unsigned Types

- Java ensures consistency by always using signed types
- C++ supports both signed & unsigned types

	Unsigned Byte	Signed Byte	Unsigned Short	Signed Short
Size	1 byte	1 byte	2 bytes	2 bytes
Value Range	0 to 255	-128 to 127	0 to 65,535	-32,768 to 32,767

Principle of Conversion

- To read in **unsigned values** from **C++** the resulting type in **Java** needs to be **larger**
- Also, some extra conversion needs to be done to fix incorrect sign extension.

Naïve Unsigned Conversion

- short value = (short) in.readByte();

- **Question:** If a short can hold from 0 to 65,535 why doesn't this work for values 128-255?
- **Answer:** Sign Extension applied when casting

How Does Sign Extension Work?

- unsigned byte: 129 = 0x81 = $1000\ 0001_2$
- The sign bit is extended: $1111\ 1111\ 1000\ 0001_2$
- In twos compliment, if the sign bit = 1, the number is presumed negative.

Twos Compliment

Raw Bits	Signed	Unsigned
01111111 =	127	127
00000010 =	2	2
00000001 =	1	1
00000000 =	0	0
11111111 =	-1	255
11111110 =	-2	254
10000001 =	-127	129
10000000 =	-128	128

Solution: Bit Masking

byte b = in.readByte(); // reads as signed

short bitmask = (short) 0xff;

short value = (short)(b & bitmask);

```
  1111 1111 1xxx xxxx    negative
& 0000 0000 1111 1111    0xFF
-------------------------------
  0000 0000 1xxx xxxx    positive
```

Be Careful with Unsigned Ints

- long a = (long)(in.readInt() & 0xffffffff);
  - doesn't work!
- **Reason:** 0xffffffff is a **negative** value.

- **Solution:**
- long a = (long)(in.readInt() & 0xffffffffL);

Dealing with Decimals

- Going from a C++ double to a Java float is easy because both are 8-byte IEEE 754 values.
- Going from a C++ float to Java is harder because Java does not have a 4-byte float type
- But Java gives tools to make the conversion easy
- int a = in.readInt();
  float b = Float.intBitsToFloat(a);
- out.writeInt(Float.floatBitsToInt(floatValue));

Other Pitfalls & Issues

- If your C++ code used bit-fields, you will have to do bit masking and shifting to read out the individual fields
  - Or use java.util.BitSet
- Reading in text (Strings) – encoding matters
  - With "plain English" it may not, but if you have international characters in your text, it will matter
  - InputStreamReader(InputStream in, String enc);

Questions?