

# Object oriented analysis and design of IO bit and byte classes

Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek  
Software Engineering

November 17, 2008

# Hardware control from a virtual machine

Java is touted for its **compile one, run everywhere** feature. However, it cannot hold this promise once data exchange with the underlying hardware or even OS is involved. In that case you will have to speak the (native) tongue of the underlying services. In such cases the Java Native Interface approach must be used. This involves an interface (in Patterns you would call it an Adapter<sup>1</sup>). See also [http://en.wikipedia.org/wiki/Java\\_Native\\_Interface](http://en.wikipedia.org/wiki/Java_Native_Interface) and the links found there.

JNI involves calling a C or C++ function from Java. Using this technology it is possible to create methods that read or write bytes or words on the hardware level.

---

<sup>1</sup>Even a very peculiar one: a Language Adapter

# Native methods

Native methods are methods adorned with the `native`. If the compiler finds this keyword, it will try to find a matching function in non JVM libraries, typically a DLL in Windows or a shared object (.so) under Unix.

The JDK brings along a few utilities in helping to write the stub code to interface to these C and C++ libraries.

One such tool is `javah`, which generates the stub code from your Java classes which use the native keyword.

Quit often the native methods are also defined as static.

# Methods convention in this course

In this course we will use methods that return primitive types and accept primitive types, matching as much as possible the types of the underlying layers.

Examples:

`byte` 8 bit value

`short` 16 bit value

`int` 32 bit signed value

`long` 64 bit signed value

# Operations per bit

- Bit has the following operations.
  - `set()`
  - `clr()` and
  - `isSet()`

First we will do the output part, which is simplest.

# Bit operations on a byte

In real hardware, most of the time the bits are grouped in larger aggregates like bytes or words. In this example we assume the aggregate is a byte.

We also assume that we can `read()` and `write()` this byte as a whole. This assumption has been packed into the interface in the class diagram.

## Bit math

Also applies to boolean math with  $1 == \text{true}$  and  $0 == \text{false}$ .

math sym		$\neg a$	$a \wedge b$	$a \vee b$	$a \oplus b$	$a = b$	$\neg(a \wedge b)$	$\neg(a \vee b)$
techn. not.		$\bar{a}$	$a \cdot b$	$a + b$	$a \oplus b$	$a = b$	$\overline{a \cdot b}$	$\overline{a + b}$
$a$	$b$	not(a)	and	or	xor	equals	nand	nor
0	0	1	0	0	0	1	1	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	1	0
1	1	0	1	1	0	1	0	0

# Bits 'share' the byte

Bits in a byte have a *natural* aggregation: these bits 'live' in the byte. But the bits defer the operation on the real hardware to the byte, because that 'has' the hardware address and real io operation.

For many purposes you not only want to control single bits, but also want to read and write bits in groups like 2 bits for a motor control.

This is where we use **masking** operations.

# Bit ops in bytes (words etc)

To **set** a bit you use the **or** operator and 'one' at the position you want to set.

$$\begin{array}{r} a = 01000101 \\ b = 00011111 \\ \hline a \vee b = 01011111 \end{array}$$

To **clear** a bit you use the **and** operator and 'zero' at the position you want to clear.

$$\begin{array}{r} a = 01000101 \\ b = 00011111 \\ \hline a \wedge b = 00000101 \end{array}$$

# Detecting difference with xor

To detect the bit difference between two value (say old and new value) you can use the **xor** operation.

$$\begin{array}{r} a = 01000101 \\ b = 00011111 \\ \hline a \oplus b = 01011010 \end{array}$$

# Testing a bit with and and a mask

To test whether a certain bit is set you can use the **and** and **shift** operations to create a bitmask.

$$\begin{array}{r} a = 01000101 \\ \text{mask} = 01000000 \\ \hline a \wedge b = 01000000 \end{array}$$

# IOWarrior particularities

In mode 0 (simple io pin wise bit io) all input and out bits are in 4 bytes or one word. You always have to read and write all 4 bytes at the same time. We consider the handling easier if you aggregate these bits in one `int` value, the unit of reading and writing of the library in the repository.

The electronic design of the IOWarrior chip requires that a pin that you want to use as input must be set high and kept high.

Also when you change a bit by writing, it is not very useful to get signal that this bit changed. To handle this we use a `inputMask`.

# Setting a bit to a new value with masking

If each bit has an associated mask **m** (see previous slide), you can use the following three step calculation to copy the bit (boolean) into the word.

```
public void writeMasked(int mask, int value) {  
    shadow = ((shadow & ~mask) | (value & mask)) | inputMask;  
    write(shadow);  
}
```

which can then be used to set the bit in a word to its boolean like this:

```
public void set(boolean b){  
    if (b != current) {  
        current=b;  
        io.writeMasked( mask, b?0xFFFFFFFF:0);  
        updateListeners ();  
    }  
}
```

Have a look in the code for more details