

Bit Manipulation with Java

Adrian P. Robson
adrian.robson@nepsweb.co.uk

22th April 2015

Abstract

The direct manipulation of bits, or *bit twiddling*, is often necessary in network communications software. This report explains how to read and modify integer values at the bit level with the Java programming language. Data endianness, bitwise operators and bit fields are discussed.

Contents

1	Endianness and Significance	2
2	The Bitwise Operators	3
3	Bit Logic	4
4	Masks	4
5	Setting Bits	4
6	Clearing Bits	5
7	Toggling Bits	5
8	Checking a Bit	6
9	Field Insertion	6
10	Field Extraction	6
11	Building Fields	7

1 Endianness and Significance

When engaged in bit manipulation, understanding the way that data is stored in computer hardware is important.

Significance is the order of symbols in the place-value notation that we use to represent numbers. By convention we write numbers with the most significant symbols to the left. The left most symbol is often referred to as *high order*, and the right hand symbol as *low order*.

Endianness is the byte and bit ordering used in the storage and use of data. Typical examples of where endianness is important are the storage of integer values and network transmission. There are two common possibilities:

Big-endian ‘big end first’: The highest order byte is stored in the lowest address; or the highest order bit and byte are the first to be transmitted.

Little-endian ‘little end first’: The lowest order byte is stored in the lowest address; or the lowest order bit and byte are the first to be transmitted.

There is a third less common form of endianness: middle-endian or mixed-endian, where the bytes of a 32 bit word are stored as 2nd, 1st, 4th then 3rd.

Byte values are unchanged by endianness but bits are conventionally addressed in the same order as their byte’s endianness, as the following tables show:

Big-endian

byte addr	0	1	2	3
bit offset	01234567	01234567	01234567	01234567
binary	00001010	00001011	00001100	00001101
hex	0A	0B	0C	0D

Little-endian

byte addr	3	2	1	0
bit offset	76543210	76543210	76543210	76543210
binary	00001010	00001011	00001100	00001101
hex	0A	0B	0C	0D

Figure 1 shows storage endianness for a 4 byte integer. Figure 2 shows the big-endian IP header, the bits of which are processed left to right and top to bottom for transmission.

Endianness is normally prescribed by application requirements or target hardware. However, endianness can be ignored if bits are only being considered

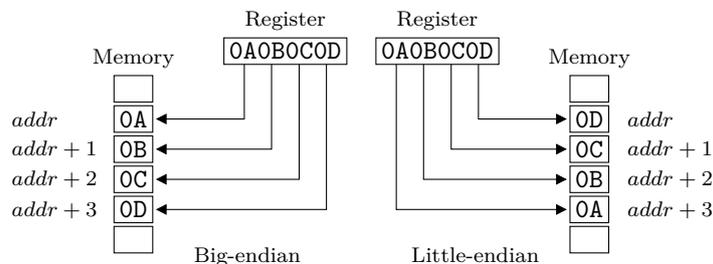


Figure 1: Big and Little Endianness

0	3 4	7 8	15 16	18 19	31
Version	IHL	ToS	Total length		
Identification			Flags	Fragment offset	
TTL		Protocol	Header checksum		
Source IP address					
Destination IP address					
Options					

Figure 2: Big-endian IP Header

in terms of their significance, rather than their addresses. If the choice is arbitrary, little-endian is conventionally used because it has the simplicity that increasing address matches increasing significance.

Endianness effects CPU, memory, buses, expansion cards and files. Most computing system are not homogeneous and will normally have mixed endianness.

- Examples of little-endian platforms include Intel 80x86 and DEC. Big-endian CPUs include Motorola 680x0, Sun Sparc, *Java Virtual Machine*, and IBM (e.g., PowerPC). MIPs and ARMs can be configured either way.
- Ethernet cards are big-endian, but the PCI bus that it might be connected to is little-endian. TCP/IP is big-endian.
- GIF files are little-endian but JPEGs are big-endian.
- The DEC PDP-11 and the sometimes the ARM are middle-endian.

2 The Bitwise Operators

Individual and blocks of bits in integer and byte variables can be modified with bitwise operators, which are:

&	bitwise AND
	bitwise inclusive OR
^	bitwise exclusive OR
>>	right shift arithmetic
>>>	right shift logical
<<	left shift
~	ones complement (unary)

The bitwise operators &, | and ^ are binary, and apply their logical operation (see §3) to pairs of bits with the same significance in their operands.

The left shift operator << moves each bit in its left-hand operand to the left by the number of positions given in its right-hand operand. Any empty bits created at the right by the shift are filled with zeroes.

The right shift operators move each bit in their left-hand operand to the right by the number of positions given in their right-hand operand. Shifting integers with >>> in Java left fills with zeroes. However, shifting with the >> operator ‘propagates the sign bit’ by filling blank bits with copies of the left most bit

value. *This arithmetic shifting normally causes problems for bit manipulation, and should be avoided in this context.*

The ones complement operator \sim performs bitwise logical negation (see §3) of its single operand. All ones are changes to zeroes, and all zeroes to ones.

3 Bit Logic

Bitwise operators are applied to bits and pairs of bits in their operands using conventional boolean logic, with 0 as false and 1 as true. There are four logical functions:

AND	\wedge	conjunction
OR	\vee	disjunction
XOR	\oplus	exclusive disjunction (exclusive OR)
NOT	\neg	negation (ones complement)

These are defined in the following truth table:

a	b	$a \wedge b$	$a \vee b$	$a \oplus b$	$\neg a$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

4 Masks

When working with bitwise operators, *masks* are used to select which bits in a variable are to be modified. The mask is the same size as the variable. It has a one in each bit location that is to be modified, and zeroes for those bits that are to remain unchanged.

For example, 00101101 would select bits 0, 2, 3 and 5, using the little-endian convention. This binary number is 44 in decimal or 2D in hexadecimal. The latter form is the most convenient for expressing masks, and would be written as 0x2D in Java.

5 Setting Bits

To set bits to one, we use an OR operation. For each bit position b in the mask M , input value V and result R , we have the following logical relation:

$$R_b = M_b \vee V_b = \begin{cases} 1 & \text{when } M_b = 1 \\ V_b & \text{when } M_b = 0 \end{cases}$$

Applying this logic to the bits of a byte would give the following:

mask	M	00000110	0x06
value	V	10101010	0xAA
result	R	10101110	0xAE

So in a program we could use the bitwise inclusive OR operator to set bits 1 and 2 of a byte value to 1 without affecting its other bits with this:

```
result = 0x06 | value;
```

or if the input value and result are the same variable, we can write:

```
value |= 0x06;
```

6 Clearing Bits

To clear bits to zero, we use the AND and NOT operations. For each bit position b in the mask M , input value V and result R , we have the following logical relation:

$$R_b = \neg M_b \wedge V_b = \begin{cases} 0 & \text{when } M_b = 1 \\ V_b & \text{when } M_b = 0 \end{cases}$$

Applying this logic to a byte would give the following:

mask	M	00000110	0x06
NOT mask	$\neg M$	11111001	0xF9
value	V	10101010	0xAA
result	R	10101000	0xA8

So in a program we could use the bitwise AND and the 1's complement operators to set bits 1 and 2 of a byte value to 0 without affecting its other bits with this:

```
result = ~0x06 & value;
```

or we can write the following if the input value and result are the same variable:

```
value &= ~0x06;
```

7 Toggling Bits

To toggle bits, we use an XOR operation. For each bit position b in the mask M , input value V and result R , we have the following logical relation:

$$R_b = M_b \oplus V_b = \begin{cases} \neg V_b & \text{when } M_b = 1 \\ V_b & \text{when } M_b = 0 \end{cases}$$

Applying this logic to a byte would give the following:

mask	M	00000110	0x06
value	V	10101010	0xAA
result	R	10101100	0xAC

When there is a zero in the mask the value bit is unchanged, but when there is a one in the mask the value bit is inverted, or toggled.

So in a program we could use the bitwise exclusive OR operator to toggle bits 1 and 2 of the byte value to 1 without affecting its other bits with this:

```
result = 0x06 ^ value;
```

or if the input value and result are the same variable, we can write:

```
value ^= 0x06;
```

8 Checking a Bit

To look at a single bit, define a mask with just that bit set and use a bitwise AND operation. This gives a zero if and only if the target bit is clear. So we can write:

```
if ( (target & 0x20) != 0 )
    System.out.printf("bit 5 is on\n");
```

More than one bit can be checked with an appropriate mask, but interpreting the result is not so simple. See section §10 for a discussion on looking at more than one bit.

9 Field Insertion

It is often necessary to insert a variable as a field, or block of bits, into an existing target. The principle of operation is to align the variable with the target field and use a mask to define the field width and location. In the mask, a 1 indicates that the equivalent target bit should be replaced; and a 0 indicates that the target's bit should be unchanged. So for each bit position b in the target T , mask M , aligned value V and result R , we have the following logical relation:

$$R_b = (\neg M_b \wedge T_b) \vee (M_b \wedge V_b) = \begin{cases} V_b & \text{when } M_b = 1 \\ T_b & \text{when } M_b = 0 \end{cases}$$

This logic can be adapted to work with program variables using the bitwise AND, bitwise OR, ones complement and left shift operators. So to insert a value into `target` we can use the statement:

```
result = ( ~(widthmask << offset) & target ) |
          ( value << offset );
```

where `offset` is the location from the right of the field in `target`, and `widthmask` defines the size of the field as a sequence of ones.

For example, consider `target` with three fields A, B and C arranged as ABBBCCCC, where we want to replace field B with the contents of `value`. This can be expressed as:

```
result = ( ~(0x07 << 3) & target ) | ( value << 3 );
```

This, of course, will only work if the value can fit in the field. So here the contents of `value` must be less than or equal to 7.

10 Field Extraction

To get a field or block of bits from a variable, it must be extracted with a mask and then shifted right. The following Java statement gets NNN from aNNNaaaa:

```
field = (target & 0x70) >>> 4;
```

For example, `field` is 2 if `target` is 0xAA (10101010₂). It is important that a logical right shift is applied, so `>>` should not be used.

11 Building Fields

The left shift operator can be used to build multiple bit fields from integer values. Consider a 8 bit variable with three fields each 2 bits wide. These represent values for action, rate and target, and there are 2 bits of padding between action and rate. This can be visualised as `AA__RRTT`, and a function to make the complete variable from three integer values is:

```
static int makeHeader( int a, int r, int t )
{
    return (a << 6) | (r << 2) | t;
}
```

However, the values used as arguments must not exceed the field sizes, which in this case are all a maximum integer value of three.