Bit Manipulation in C, C++ or Java

Adrian P. Robson

7th March 2010

Abstract
The manipulation of bits is often necessary in embedded software, hardware drivers and networking software. This report explains how to manipulate integer variables at the bit level in the C, C++ and Java programming languages. Data endianness, bitwise operators, bit fields and some common methods for addressing hardware are discussed.

Contents

1 Endianness and Significance 2

2 Bitwise Operators 3
  2.1 Masks ................................. 3
  2.2 Setting a bit ........................... 4
  2.3 Clearing a Bit ......................... 4
  2.4 Toggling a Bit ......................... 4
  2.5 Checking a Bit ........................ 5
  2.6 Named Bits ............................ 5
  2.7 Shifting in Bits ....................... 5
  2.8 Field Insertion ....................... 5
  2.9 Field Extraction ...................... 6
  2.10 Parameterised ....................... 6

3 Bit Fields 6
  3.1 Bit Field Example ..................... 7
  3.2 Addressing, Alignment and Packing 7

4 Controlling Hardware 8
  4.1 Pointers .............................. 8
  4.2 Bit Fields ............................ 9
  4.3 Compiler Extensions .................. 9
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 digits in the place-value notation that we use to represent numbers. By convention we write numbers with the most significant digits to the left. The left most digit or value is often referred to as high order, and the right hand digit 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.

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.

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.

Bits are conventionally addressed in the same order as their byte’s endianness as the following tables show:

<table>
<thead>
<tr>
<th>Big-endian</th>
<th>Little-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte addr</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>bit offset</td>
<td>01234567</td>
</tr>
<tr>
<td>binary</td>
<td>00001010</td>
</tr>
<tr>
<td></td>
<td>00001011</td>
</tr>
<tr>
<td></td>
<td>00001100</td>
</tr>
<tr>
<td></td>
<td>00001101</td>
</tr>
<tr>
<td>hex</td>
<td>0A</td>
</tr>
<tr>
<td></td>
<td>0B</td>
</tr>
<tr>
<td></td>
<td>0C</td>
</tr>
<tr>
<td></td>
<td>0D</td>
</tr>
</tbody>
</table>

![Figure 1: Big and Little Endianness](image-url)
Bit endianness is normally prescribed by the application or target device. However, endianness can be ignored if bits are only being considered in terms of their significance, rather than their addresses. If the choice is arbitrary, little-endian has the simplicity that increasing address matches increasing significance.

Endianness affects CPU, memory, buses, devices, 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 Bitwise Operators

Individual bits can be modified in an integer variable with the bitwise operators **AND**, **OR** and **XOR**. Bits can be set, cleared or toggled.

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
</tr>
</tbody>
</table>

#### 2.1 Masks

A useful way of identify which bits are to be modified is a mask, where the targeted bits are marked with a one. So 00101101 would mark bits 0, 2, 3 and 6, using the little-endian convention. This binary number is 44 in decimal or 2D in hexadecimal. The latter form is the most convenient, and would be written as 0x2D in C, C++ and Java.
2.2 Setting a bit

To set a bit to one, we use the bitwise OR operation:

\[
\begin{align*}
\text{mask} & \quad 00000110 \quad 0x06 \\
\text{target} & \quad 10101010 \quad 0xAA \\
\text{result} & \quad 10101110 \quad 0xAE = \text{target OR mask}
\end{align*}
\]

Note that where there is a 1 in the mask the bit is 1 in the result, and where there was a 0 in the mask the bit is unchanged.

In a program we would use the following statement to set bits 1 and 2:

\[\text{result} = \text{target} \mid 0x06;\]

or if the target and result were the same variable we could write:

\[\text{target} \mid= 0x06;\]

2.3 Clearing a Bit

To clear a bit to zero, we use the bitwise AND operation and the one’s complement of the mask. To make the one’s complement of the mask we simply invert the bits:

\[
\begin{align*}
\text{mask} & \quad 00000110 \quad 0x06 \\
\text{1’s complement} & \quad 11111001 \quad 0xF9 \\
\text{target} & \quad 10101010 \quad 0xAA \\
\text{result} & \quad 10101000 \quad 0xA8 = \text{target AND 1’s complement}
\end{align*}
\]

Where there is a 1 in the mask the bit is 0 in the result, and where there was a 0 in the mask the bit is unchanged.

In a program we would use the following statement to clear bits 1 and 2:

\[\text{result} = \text{target} \& \lnot 0x06;\]

or if the target and result were the same variable we could write:

\[\text{target} \&= \lnot 0x06;\]

2.4 Toggling a Bit

To toggle a bit we use the bitwise XOR operation:

\[
\begin{align*}
\text{mask} & \quad 00000110 \quad 0x06 \\
\text{target} & \quad 10101010 \quad 0xAA \\
\text{result} & \quad 10101100 \quad 0xAC = \text{target XOR mask}
\end{align*}
\]

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

In a program we would use the following statement to invert bits 1 and 2:

\[\text{result} = \text{target} \^\ 0x06;\]

or if the target and result were the same variable we could write:

\[\text{target} \^= 0x06;\]
2.5 Checking a Bit

To look at a single bit, define a mask with just that bit set. When this mask is used with a bitwise AND operator it gives a zero if and only if the target bit is clear. So in C we can write

```c
if ( target & 0x20 )
    printf("bit 6 is on\n");
```

This works because 0 is interpreted as false and no-zero as true. However, in Java this is not so, and the following is needed

```java
if ( (target & 0x20) != 0 )
    System.out.printf("bit 6 is on\n");
```

More than one bit can be checked with an appropriate mask, but interpreting the result is not so simple. See section §2.9 for a discussion on looking at more than one bit.

2.6 Named Bits

This can be made a little easier to follow by giving the bits names:

```c
#define LED1 0x02
#define LED2 0x04
:
PTD |= LED1 | LED2; // LEDs on
:
PTD &= ~(LED1 | LED2); // LEDs off
```

2.7 Shifting in Bits

The shift operator can also be used to build bit fields from integer values. Consider an 8 bit variable with three fields, action, rate and target, which are 2 bits wide, with 2 bits of padding between action and rate. This can be visualised as

```
AA__RRTT
```

A function to make it from three integer values is:

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

Java has a distinct byte type that is appropriate for bit oriented values. The approach in this case is similar:

```java
return new Integer( (a << 6) | (r << 2) | t ).byteValue();
```

2.8 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. A 1 bit in the mask indicates that the equivalent target bit should be replaced; and a 0
indicates that the target’s bit should be unchanged. For each bit position in the
target, variable and mask we have the following logical relation

\[ T'_\text{bit} = \neg M_{\text{bit}} \land T_{\text{bit}} \lor M_{\text{bit}} \land V_{\text{bit}} \]

This logic can be adapted to work with program variables. To insert \texttt{value}
into \texttt{target} we can use:

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

where \texttt{offset} is the bit location from the left of the field in \texttt{target}, and
\texttt{widthmask} defines its size.

For example consider a target with three fields \texttt{A}, \texttt{B} and \texttt{C} arranged as
\texttt{AABBBCCC}, where we want to replace field \texttt{B} with the contents of \texttt{value}. This
would be expressed as:

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

### 2.9 Field Extraction

To get a field from a variable, it must be extracted with a mask and then shifted
left. The following gets \texttt{NNN} from \texttt{xNNNxxx}:

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

For example if \texttt{target} is \texttt{0xAA} then \texttt{field} is 2.

### 2.10 Parameterised

Sometimes it is useful to be able to dynamically specify which bit is to be
modified. The following function demonstrates this how this can be done for a
little-endian byte:

```c
void setBit( unsigned char *byte, int bitAddr )
{
    *byte |= (1 << bit);
}
```

The shift left operator \(<<\) is used to create a mask, which is then applied to the
target. For big-endian bits the right shift operator can be used in \((8 >> \text{bit})\)
to produce the mask.

### 3 Bit Fields

In C and C++ there is a possible alternative for managing bits. We can declare
a structure with bit fields. This notation is provided to support compact data
storage but with care, it can be used as an alternative to bitwise logic.
3.1 Bit Field Example

Using the previous AA__RRT example (see §2.7):

typedef struct {
  unsigned char target :2;   // TT
  unsigned char rate :2;     // RR
  unsigned char pad :2;      // __
  unsigned char action :2;   // AA
} HeaderBits;

We can declare a variable of this type and assign values to the fields like this:

HeaderBits header;
:
header.rate = 3;

However, this will not allow us to modify the bits of an integer variable. To
do this we declare a union:

typedef union {
  struct {
    unsigned char target :2;
    unsigned char rate :2;
    unsigned char pad :2;
    unsigned char action :2;
  } bits;
  unsigned char byte;
} Header;

and use it like this:

unsigned char makeHeader( int a, int r, int t )
{
  Header header;
  header.bits.action = a;
  header.bits.pad = 0;
  header.bits.rate = r;
  header.bits.target = t;
  return header.byte;
}

3.2 Addressing, Alignment and Packing

The order of field declaration and addressing order are the same. If field f0 is
declared before field f1, then field f0 has the lower address. However, a field’s
significance is dictated by the platform’s endianness.

The type of a bit field affects its alignment and packing. Figure 3 shows the
bit layout of the following structure with unsigned int, unsigned short, and
unsigned char bit fields:
Figure 3: Bit Field Packing Examples

```
struct {
    type f0 :7;
    type f1 :3;
    type f2 :2;
    type f3 :2;
    type f4 :2;
} bits;
```

The short version is the most compact and concise packing. It is probably what would be used for bit manipulation. The char version aligns some fields so that they do not cross byte boundaries. However for bit manipulation it is best to use explicit fields if pad bits are needed, rather than to rely on this sort of alignment behaviour. The int version packs the same as the short version but is padded to 32 bits.

The packing and alignment of bit fields can be confusing, and a useful development check is to use the sizeof operator on the struct to confirm its expected size.

A disadvantage of bit fields is that the executable code generated by the compiler to handle them can be slow. The programming might look simpler than with bitwise logic, but the same or more processing is needed.

4 Controlling Hardware

In some cases bit manipulation is used to control hardware that is memory mapped. Typically the devices are controlled by reading and writing to ‘registers’ that are located at absolute memory addresses.

4.1 Pointers

Pointers can used to access these registers. So given that the Port D Data Register (PTD) is at memory location 0x03, a pointer could be defined like this:

```
volatile char *PTD = (char*)0x03;
```
It is declared as volatile because its value might change at any time. The cast is needed to avoid a compile error.

A mask can then be used with the dereferenced pointer to set the register thus:

*PTD |= 0x06; // switch LED 1 and 2 on

4.2 Bit Fields

The register could also be managed using bit fields as follows:

```c
typedef struct {
    unsigned char :1; // low order
    unsigned char led1 :1;
    unsigned char led2 :1;
    unsigned char :5;
} PTDbits;
```

```c
volatile PTDbits* PTD = (PTDbits*)0x03;
```

```c
PTD->led1 = 1;
PTD->led2 = 0;
```

4.3 Compiler Extensions

Often a compiler provides a notation for mapping variables onto these absolute locations. For example the Cosmic C compiler for Freescale HC08 allows the following definition:

```c
volatile char PTD @0x03; /* port D */
```

Using this notation the register can be treated as a simple integer variable like this:

```c
PTD |= 0x06; // switch LED 1 and 2 on
```