Bit Manipulation in C and C++

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

22nd April 2015

Abstract

The direct manipulation of bits, or *bit twiddling*, is often necessary in embedded software, hardware drivers and networking software. This report explains how to read and modify integer variables at the bit level in the C and C++ 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 The Operators 3

3 Logic **4**
3.1 Masks 4
3.2 Setting Bits 4
3.3 Clearing Bits 5
3.4 Toggling Bits 5
3.5 Checking a Bit 6
3.6 Named Bits 6
3.7 Field Insertion 6
3.8 Field Extraction 7
3.9 Building Fields 7
3.10 Parameterised Bits 8

4 Bit Fields **8**
4.1 Bit Field Example 8
4.2 Addressing, Alignment and Packing 9

5 Controlling Hardware **10**
5.1 Pointers 10
5.2 Bit Fields 10
5.3 Compiler Extensions 10

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.

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.

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`

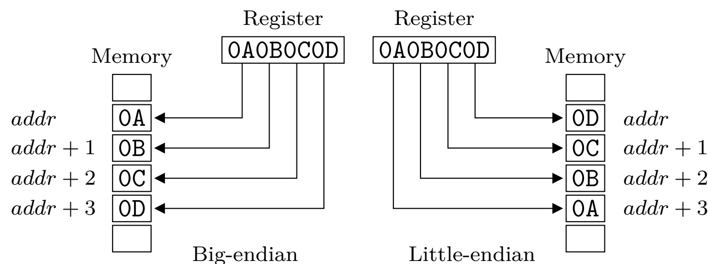

Figure 1: Big and Little Endianness

<table>
<tr><td>0 3</td><td>4 7</td><td>8 15</td><td>16 18</td><td>19 31</td></tr>
<tr><td>Version</td><td>IHL</td><td>ToS</td><td colspan="2">Total length</td></tr>
<tr><td colspan="3">Identification</td><td>Flags</td><td>Fragment offset</td></tr>
<tr><td colspan="2">TTL</td><td>Protocol</td><td colspan="2">Header checksum</td></tr>
<tr><td colspan="5">Source IP address</td></tr>
<tr><td colspan="5">Destination IP address</td></tr>
<tr><td colspan="5">Options</td></tr>
</table>

Figure 2: Big-endian IP Header

Endianness is normally prescribed by application requirements or target hardware. 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 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 Bitwise Operators

2.1 The Operators

Individual or blocks of bits can be modified in an integer variable with the bitwise and some other operators, which are:

`&`	bitwise AND
`|`	bitwise inclusive OR
`^`	bitwise exclusive OR
`>>`	right shift
`<<`	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 operator moves each bit in its left-hand operand to the right by the number of positions given in its right-hand operand. Shifting unsigned integers with `>>` left fills with zeroes. However, shifting signed integers with the `>>` operator, performs an *arithmetic shift*. This 'propagates the sign bit' by filling blank bits with copies of the previous left most bit value. *Arithmetic shifting normally causes problems for bit manipulation.* So signed integers should normally be avoided for bit manipulation work.

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

3 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

3.1 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 C or C++.

3.2 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	00000**11**0	0x06
value	V	10101**01**0	0xAA
result	R	10101**11**0	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;
```

3.3 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	00000**11**0	0x06
NOT mask	$\neg M$	11111**00**1	0xF9
value	V	10101**01**0	0xAA
result	R	10101**00**0	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;
```

3.4 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	00000**11**0	0x06
value	V	10101**01**0	0xAA
result	R	10101**10**0	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;
```

3.5 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 )
   printf("bit 5 is on\n");
```

This works because 0 is interpreted as false and non-zero as true in C and C++. However, a longer but probably clearer expression is:

```
if ( (target & 0x20) != 0 )
   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 §3.8 for a discussion on looking at more than one bit.

3.6 Named Bits

Performing bit oriented activities can be made a little easier to follow and maintain by giving the bits names. So we could have:

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

Here, the the bit names `LED1` and `LED2` are actually masks, and these are combined as required with bitwise inclusive OR operators to create working masks to actually select the bits.

3.7 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:

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

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 `AABBBCCC`, 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.

3.8 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 statement gets `NNN` from `xNNNxxxx`:

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

So in this case, if `target` is `0xAA` ($1\mathbf{010}1010_2$), the result `field` will be 2.

In general, it is best if the right shift operation is not arithmetic. So the variables used should be unsigned integers.

3.9 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:

```
unsigned char 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.

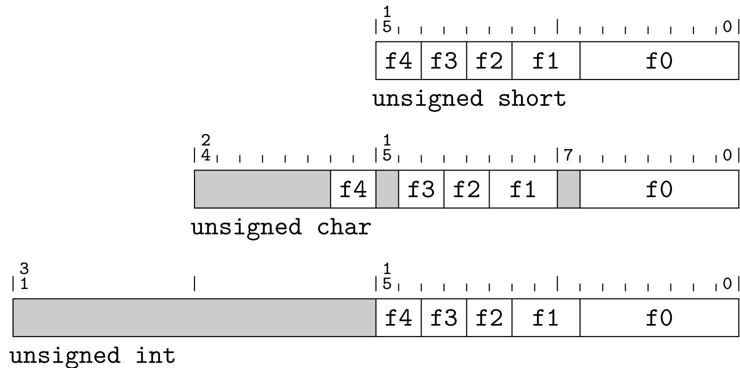

Figure 3: Bit Field Packing Examples

3.10 Parameterised Bits

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:

```
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, like this `(8 >> bit)`, to produce the mask.

4 Bit Fields

In C and C++ we can declare a structure with *bit fields*. This notation is provided to support compact data storage, but it can be used as an alternative to explicit bitwise logic.

Bit fields have a concise notation, and can be very easy to use (§4.1). However, they have no performance advantage over bit logic, and ensuring correct field alignment and padding needs some care (§4.2).

4.1 Bit Field Example

Using the previous `AA__RRTT` example (see §3.9), we can define a structure:

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

We can then 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 must 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;
```

The following function shows how the union can be used to assign field values:

```
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;
}
```

4.2 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:

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

The `short` version has the most compact packing; and 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, it is best to use explicit fields if pad bits

are needed, rather than rely on implicit alignments. 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.

5 Controlling Hardware

Operating systems and embedded control applications are often written in C because of the language's ability to access system memory using pointers. Bit manipulation is used to control hardware that is memory mapped. Devices are typically controlled by reading and writing 'registers' that are located at absolute memory addresses.

5.1 Pointers

Pointers can used to access device control registers. So given that a system's Port D Data Register (PORTD) is at memory location `0x03`, a pointer could be defined like this:

```
volatile char *PORTD = (char*)0x03;
```

It is declared as volatile because its value might be changed at any time by the port hardware. The cast is needed to avoid a compile error.

Bitwise operators can then be used with the dereferenced pointer to read or modify selected bits in the register thus:

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

5.2 Bit Fields

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

```
typedef struct {
   unsigned char      :1;  // low order
   unsigned char led1 :1;
   unsigned char led2 :1;
   unsigned char      :5;
} PORTDbits;
  :
volatile PORTDbits* PORTD = (PORTDbits*)0x03;
  :
PORTD->led1 = 1;
PORTD->led2 = 0;
```

5.3 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:

```
volatile char PORTD @0x03; /* port D */
```

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

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