Compound Bitwise Operators (&=, |=, ^=)

The compound bitwise operators perform their calculations at the bit level of variables. They are often used to clear and set specific bits of a variable.

See the bitwise math tutorial for more information on bitwise operators.

Contents

Compound bitwise AND (&=)

The compound bitwise AND operator `&=` is often used with a variable and a constant to force particular bits in a variable to be zero. This is often referred to in programming guides as “clearing” or “resetting” bits. In a program, writing the line `x &= y;` is equivalent to writing `x = x & y;`. That is, the value of `x` after the line will be equal to its old value bitwise ANDed with the value of `y`:

```
x &= y;   // equivalent to x = x & y;
```

You can use any integer variable for `x` (i.e., any variable of type `int`, `char`, `byte`, `long long`, etc.). You can use either an integer variable or any integer value (like `3` or `0x20`) for `y`.

Before doing an example of `&=`, let’s first review the Bitwise AND (`&`) operator:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  0  0  1    (operand1 & operand2) = result
```

v: latest ▾

As shown above, bits that are “bitwise ANDed” with 0 become 0, while bits that are “bitwise ANDed” with 1 are left unchanged. So, if `b` is a `byte` variable, then `b & B00000000` equals zero, and `b & B11111111` equals `b`.

Note: The above uses binary constants. The numbers are still the same value in other representations, they just might not be as easy to understand.

Normally, in C and C++ code, hexadecimal or octal are used when we’re interested in an integer’s bits, rather than its value as a number.

While hexadecimal and octal literals might be harder to understand at first, you should really take the time to learn them. They’re part of C, C++, and many other programming languages, while binary constants are available only for compatibility with Arduino.

Also, `B00000000` is shown for clarity, but zero in any number format is zero.

So, to clear (set to zero) bits 0 and 1 of a one-byte variable, while leaving the rest of the variable’s bits unchanged, use the compound bitwise AND operator `&=` with the constant `B11111100` (hexadecimal `0xFC`):

```
1  0  1  0  1  0  1  0    variable
1  1  1  1  1  1  0  0    mask
----------------------
1  0  1  0  1  0  0  0
^^^^^^^^^^^^^^^^  ^^^^
   unchanged      cleared
```

Here is the same representation with the variable’s bits replaced with the symbol `x`

```
x  x  x  x  x  x  x  x    variable
1  1  1  1  1  1  0  0    mask
----------------------
x  x  x  x  x  x  0  0
^^^^^^^^^^^^^^^^  ^^^^
   unchanged       cleared
```

So, using a byte variable `b`, if we say:

```
b =  B10101010; // B10101010 == 0xAA
b &= B11111100; // B11111100 == 0xFC
```

v: latest

then we will have

```
b == B10101000; // B10101000 == 0xA8
```

Compound bitwise OR (|=)

The compound bitwise OR operator |= is often used with a variable and a constant to “set” (set to 1) particular bits in a variable. In a program, writing the line `x |= y;` is equivalent to writing `x = x | y;`. That is, the value of `x` after the line will be equal to its old value bitwise ORed with the value of `y`:

```
x |= y;   // equivalent to x = x | y;
```

You can use any integer variable for `x` (i.e., any variable of type `int`, `char`, `long long` etc.). You can use either an integer variable or any integer value (like `3` or `0x20`) for `y`. (This works the same way as compound bitwise AND, `&=`).

Before doing an example of |=, let’s first review the Bitwise OR (|) operator:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  1    (operand1 | operand2) = result
```

Bits that are “bitwise ORed” with 0 are unchanged, while bits that are “bitwise ORed” with 1 are set to 1. So if `b` is a `byte` variable, then `b | B00000000` equals `b`, and `b & B11111111` equals `B11111111` (here we’ve used binary constants; see the note above).

So, to set bits 0 and 1 of a one-byte variable, while leaving the rest of the variable unchanged, use the compound bitwise OR operator (|=) with the constant `B00000011` (hexadecimal `0x3`):

```
1  0  1  0  1  0  1  0    variable
0  0  0  0  0  0  1  1    mask
----------------------
1  0  1  0  1  0  1  1
^^^^^^^^^^^^^^^^  ^^^^
    unchanged      set
```

Here is the same representation with the variable’s bits replaced with the symb

v: latest

```
x  x  x  x  x  x  x  x    variable
0  0  0  0  0  0  1  1    mask
----------------------
x  x  x  x  x  x  1  1
^^^^^^^^^^^^^^^^  ^^^^
    unchanged      set
```

So, using a byte variable `b`, if we say:

```
b  = B10101010; // B10101010 == 0xAA
b |= B00000011; // B00000011 == 0x3
```

then we will have

```
b == B10101011; // B10101011 == 0xAB
```

Compound bitwise XOR (^=)

The compound bitwise XOR operator `^=` is used with a variable and a constant to “toggle” (change 0 to 1, and 1 to 0) particular bits in a variable. In a program, writing the line `x ^= y;` is equivalent to writing `x = x ^ y;`. That is, the value of `x` after the line will be equal to its old value bitwise XORed with the value of `y`:

```
x ^= y;   // equivalent to x = x ^ y;
```

You can use any integer variable for `x` (i.e., any variable of type `int`, `char`, `long long`, etc.). You can use either an integer variable or any integer value (like `3` or `0x20`) for `y`. (This works the same way as &= and |=; in fact, these three operators all work the same in this way).

Before doing an example of `^=`, let’s first review the Bitwise XOR operator, ^:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  0    (operand1 ^ operand2) = result
```

One way to look at bitwise XOR is that each bit in the result is a 1 if the input different, or 0 if they are the same. Another way to think about it is that the result bit will be 1 when *exactly* one (no more, no less) of the input bits is 1; otherwise, it will be zero. This means that if you XOR a bit with 1, it will change (or toggle) its value, while if you XOR a bit with 0, it stays the same.

So, to toggle bits 0 and 1 of a one-byte variable, while leaving the rest of the variable unchanged, use the compound bitwise XOR operator `^=` with the constant `B00000011` (hexadecimal `0x3`; see note above):

```
1  0  1  0  1  0  1  0     variable
0  0  0  0  0  0  1  1     mask
----------------------
1  0  1  0  1  0  1  1
^^^^^^^^^^^^^^^^  ^^^^
    unchanged     toggled
```

So, using a byte variable `b`, if we say:

```
b  = B10101010; // B10101010 == 0xAA
b ^= B00000011; // B00000011 == 0x3
```

then we will have

```
b == B10101001; // B10101001 == 0xA9
```

See Also

- Boolean operations (&&, ||)
- Bitwise operators (&, |, ^, ~)

v: latest