Bitwise Techniques for Embedded C/C++ Programming

Prepared by Sean Collins
September 2018

Integer Literals

Literals are constant values that are hard-coded into a program. For example, the numbers “5” and “6” are literals in the following line of code:

```
int n = 5 + 6;
```

Prefixes:

The C language specification includes a set of prefixes that can be used to control the “radix,” or base, of an integer literal’s representation. These prefixes include “0x” for base 16 (hexadecimal), “0” for base 8 (octal), “0b” for base 2 (binary), and nothing for base 10 (decimal). Note: the “0” in these prefixes is the symbol for zero.

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
1	0b1	01	0x1
10	0b1010	012	0xA

Suffixes:

The specification also includes a set of suffixes, which can be used to specify the type of the integer literal (e.g., int, long, char, unsigned int, etc.). The following table provides some of the suffixes which are provided in C/C++. You should use these when there is some reason that you need to force the compiler to treat an integer literal as a specific type. Generally, if you do not specify a type using one of these suffixes, the integer literal will default to a signed integer type.

Suffix	Type
U (or u)	Unsigned
L (or l)	Long Integer
UL (or ul)	Unsigned Long Integer

For example, the following constant will be an unsigned int:

```
#define MAX_INDEX 31u
```

Bitwise Operations

A bitwise operation is an operation that works at the level of the binary representations of its operands.

Bitwise NOT Operator

For simplicity, let us consider the truth table for a single bit, represented by a variable called A. As the truth table illustrates, the NOT simply inverts the value of its input.

A	NOT A
1	0
0	1

To apply the bitwise NOT operator to an eight-bit number, we flip each bit in the input. In C and C++, the bitwise NOT operator is represented with a tilde symbol (~). Here is a C++ program to illustrate the effect of the bitwise NOT operator:

```
#include <iostream>
#include <bitset>   // enables easy output in binary form

int main ()
{
  unsigned char a = 0b11110000;
  std::cout << “a     = “ << std::bitset<8>(a)  << ‘\n’;
  std::cout << “NOT a = “ << std::bitset<8>(~a) << ‘\n’;
  return 0;
}

// OUTPUT:
// a     = 11110000
// NOT a = 00001111
```

Bitwise AND Operator

Here is the truth table illustrating all of the possible inputs and outputs for the AND operation applied to two bits:

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

The bitwise AND operator is a single ampersand (&). The following C++ program illustrates its effect:

```
#include <iostream>
#include <bitset>

int main ()
{
  unsigned char a = 0b10101010;
  unsigned char b = 0b00001111;
  std::cout << "a       = " << std::bitset<8>(a)     << '\n';
  std::cout << "b       = " << std::bitset<8>(b)     << '\n';
  std::cout << "a AND b = " << std::bitset<8>(a & b) << '\n';
  return 0;
}

// OUTPUT:
// a       = 10101010
// b       = 00001111
// a AND b = 00001010
```

Bitwise OR Operator

Here is the truth table illustrating all of the possible inputs and outputs for the OR operation applied to two bits:

A	**B**	**A OR B**
1	1	1
1	0	1
0	1	1
0	0	0

The bitwise OR operator is a single vertical bar (|). The following C++ program illustrates its effect:

```
#include <iostream>
#include <bitset>

int main ()
{
  unsigned char a = 0b10101010;
  unsigned char b = 0b00001111;
  std::cout << "a      = " << std::bitset<8>(a)     << '\n';
  std::cout << "b      = " << std::bitset<8>(b)     << '\n';
  std::cout << "a OR b = " << std::bitset<8>(a | b) << '\n';
  return 0;
}

// OUTPUT:
// a      = 10101010
// b      = 00001111
// a OR b = 10101111
```

Bitwise XOR Operator

The bitwise XOR (“exclusive or”) operation has the following table. Whenever I see a predicate such as “A XOR B,” I read it in my head as: “A or B, but not both.” As the truth table shows, the XOR operation evaluates to 1 only when exactly one of the inputs is 1.

A	**B**	**A XOR B**
1	1	0
1	0	1
0	1	1
0	0	0

The bitwise XOR operator is represented with a carrot symbol (^). The following C++ program illustrates its effect:

```
#include <iostream>
#include <bitset>

int main ()
{
  unsigned char a = 0b10101010;
  unsigned char b = 0b00001111;
  std::cout << "a       = " << std::bitset<8>(a)     << '\n';
  std::cout << "b       = " << std::bitset<8>(b)     << '\n';
  std::cout << "a XOR b = " << std::bitset<8>(a ^ b) << '\n';
  return 0;
}

// OUTPUT:
// a       = 10101010
// b       = 00001111
// a XOR b = 10100101
```

Bitwise SHIFT Operators

C and C++ also provide bitwise left-shift and right-shift operators. These operators work by shifting the bits in the first argument by the number of positions given by the second argument. Note: If we shift a binary number to the left, then the rightmost bit would presumably be left empty. In C and C++, this bit is automatically assigned a value of 0.

The left-shift operator is << and the right-shift operator is >>. Here's another example program:

```
#include <iostream>
#include <bitset>

int main ()
{
  unsigned char a = 0b11111111;

  // Left-Shift
  std::cout << "Shifting \'a\' to the left:\n";
  for (int i = 0; i <= 8; ++i)
  {
    unsigned char a_shifted = a << i;
    std::cout << "(a << " << i << ") = " << std::bitset<8>(a_shifted) << '\n';
  }

  // Right-Shift
  std::cout << "Shifting \'a\' to the right:\n";
  for (int i = 0; i <= 8; ++i)
  {
    unsigned char a_shifted = a >> i;
    std::cout << "(a >> " << i << ") = " << std::bitset<8>(a_shifted) << '\n';
  }

  return 0;
}

// OUTPUT:
// Shifting 'a' to the left:
// (a << 0) = 11111111
// (a << 1) = 11111110
// (a << 2) = 11111100
// (a << 3) = 11111000
// (a << 4) = 11110000
// (a << 5) = 11100000
// (a << 6) = 11000000
// (a << 7) = 10000000
// (a << 8) = 00000000
// Shifting 'a' to the right:
// (a >> 0) = 11111111
// (a >> 1) = 01111111
// (a >> 2) = 00111111
// (a >> 3) = 00011111
// (a >> 4) = 00001111
// (a >> 5) = 00000111
// (a >> 6) = 00000011
// (a >> 7) = 00000001
// (a >> 8) = 00000000
```

Techniques Using Bitwise Operations

Often in embedded systems programming, we need to manipulate or query individual bits of a register or binary value. The following sections describe the most common techniques.

Setting Bits (a.k.a. "Masking bits to 1")

Say we have an unsigned, 8-bit value. Our objective is to set a particular "target bit" to a value of 1 while leaving the other bits unchanged.

Observe the truth table for the bitwise OR operation, above. You will notice that whenever the value of B is 1, the value of the output is 1, regardless of the value of A. Additionally, whenever the value of B is 0, the value of the output matches A's initial value. These properties of the OR operation make it an ideal tool for setting a target bit to 1 while leaving the other bits unchanged.

In the following example, we use the OR operation to obtain an output value that is the same as the first operand except with the first two bits set to 1.

```
   00111100
OR 11000000  (mask)
   11111100
```

The second operand in the example above is often called a "mask." For all positions in which the mask contains a 1, the result will contain a 1. All other positions will retain the same value as the non-mask.

Clearing Bits (a.k.a. "Masking bits to 0")

Let's say that instead of setting particular bits to 1, we want to set particular bits to 0. Observe the truth table above for the AND operation. Whenever B is 0, the output is always 0, regardless of A. Also, when B is 1, the output is the same as A. We can thus use the AND operation to clear a particular bit while leaving the other bits unchanged.

Here's another example:

```
    11110011
AND 00001111  (mask)
    00000011
```

As the example illustrates, the output is 0 wherever the mask is 0. The bits in all the other positions are the same as the first input.

Toggling Bits

Similar logic reveals that we can use the XOR operation to "flip" particular bits in an unsigned integer. The truth table for XOR, above, shows that the result of XOR'ing the value 1 with some input value is always the opposite of the input value. Additionally, the result of XOR'ing the value 0 with some input value is always the same as the input value.

```
    11110011
XOR 00001111  (mask)
    11111100
```

Creating a Mask

All of the three techniques above utilized masks in order to manipulate the bits of some input value. Since masks are so ubiquitous in embedded C programming, it pays to learn how to create them. Sometimes, you can hard-code your mask as a constant because it will never change. Often, however, you will need to generate a mask programmatically. Let's go over how to do this.

As an example, let's create a mask that has a 1 in positions 4 and 6. The table below illustrates the mask we aim to create:

Position	7	6	5	4	3	2	1	0
Mask	0	1	0	1	0	0	0	0

We can create this mask using the following code:

```
unsigned char mask = (1 << 4) | (1 << 6);
// mask is now 0b01010000
```

Once we have a mask, we can easily manipulate the bits in positions 4 and 6 for some input value:

```
input = input | mask;  // Sets bits 4 and 6
input = input & ~mask; // Clears bits 4 and 6
input = input ^ mask;  // Toggles bits 4 and 6
```

Note: We can abbreviate the preceding statements by using the "combined assignment operators" as follows:

```
input |= mask;  // Sets bits 4 and 6
input &= ~mask; // Clears bits 4 and 6
input ^= mask;  // Toggles bits 4 and 6
```

Querying a Single Bit

It is often useful to test whether or not a particular bit is set to 1 inside of a register. This can be accomplished by extending the logic of the techniques above. But first, it is important to remember that in C and C++, any value that is non-zero is interpreted as TRUE when placed in a Boolean context; Zero is interpreted as FALSE.

By considering the logic discussed above, you should be able to recognize that the following expression evaluates to TRUE if and only if the bit in position 3 of the variable "input" is set to 1:

```
input & (1 << 3)
```

Summary

Technique	Description	Example
Create Mask	Shift 1 to the target positions and OR the results	`uint8_t msk = (1 << 2) | (1 << 3);`
Set Target Bits	Create a mask with 1's in target positions and then OR with input	`input |= msk;`
Clear Target Bits	Create a mask with 1's in target positions, then AND the input with NOT(mask)	`input &= ~msk;`
Toggle Target Bits	Create a mask with 1's in target positions, then XOR with input	`input ^= msk;`

Query a Single Bit	Create a mask with a 1 in the target position, then AND with input	`if (input & (1 << 3)) {`
		`    // body of this "if" statement`
		`    // only runs if bit in pos. 3`
		`    // of input is 1`
		`}`

*Note: "uint8_t" is an unsigned 8-bit type, defined in standard C header file <stdint.h>