CHAPTER X

Bits and Bytes

A bit is the smallest unit of information there is. It is a single digit in the binary number system, with the value “0” or “1”. Two aspects of a bit make it useful. First, a bit’s value can be interpreted as anything at all by the computer. That single bit might represent “yes” or “no,” or the presence or absence of a disk, or whether a mouse button is pressed. Second, the values of several bits can be concatenated to represent more complex data. Each bit that’s tacked on doubles the number of possible values that can be represented.

In other words, one bit can hold two possible values, “0” or “1”. Two bits can hold 2×2, or four, possible values, “00”, “01”, “10”, or “11”. Likewise, three bits can hold 2×2×2, or eight, possible values, and so on. This characteristic is both the greatest strength and the greatest limitation of computers. It is a strength because very complex data (such as this book) can be stored by breaking down the information to its representation in bits. It is a weakness because many things in real life have inexact values, which cannot be represented in a finite number of bits.

Programmers must be constantly aware of how many bits must be used to hold each data item. Because a bit is such a small unit, most computers are designed to handle them in more convenient chunks called bytes. A byte is the smallest addressable unit of information on most computers. That means that the computer assigns an address to each byte of information, and it can retrieve or store information only a byte at a time. The number of bits in a byte is arbitrary and can be different on different machines. The most common value is eight bits per byte, which can be store up to 256 different values. Eight bits is a convenient size for storing data that represents characters in ASCII (the American Standard Code for Information Interchange).

The following program displays the ASCII character set, starting with the space character and continuing up through the graphics character set of the PC. Note that the variable `ctr` must be an int and not a char because a char consists of 8 bits and thus can hold only the values 0 through 255 (or –128 to 127 for signed chars). If `ctr` were a char, it could never hold a value of 256 or greater, so the program would never end. If you run this program on a machine other than a PC, note that the non-ASCII characters this program prints might result in a garbled screen.

```
#include <stdio.h>

void main(void);

void main()
{
   /*  Display ASCII char set */

   unsigned char space = ' ';      /*  Start with SPACE
                                       char = 8 bits only */
   int ctr = 0;

   printf("ASCII Characters\n");
   printf("================\n");
   for (ctr = 0; ctr + space < 256; ctr++)
      printf("%c", ctr + space);

   printf("\n");
}
```

Because the computer works in chunks of bytes, most programs work this way as well. Sometimes it becomes necessary to conserve memory space because of either the number of items to be stored or the time it takes to move each bit of information. In this case, we would like to use less than one byte for storing information that has only a few possible values. That's what this chapter is all about.

X.1: What is the most efficient way to store flag values?

Answer:

A flag is a value used to make a decision between two or more options in the execution of a program. For instance, the `/w` flag on the MS-DOS `dir` command causes the command to display filenames in several columns across the screen instead of displaying them one per line. Another example of a flag can be seen in the answer to FAQ III.5, in which a flag is used to indicate which of two possible types is held in a union. Because a flag has a small number of values (often only two), it is tempting to save memory space by not storing each flag in its own `int` or `char`.

Efficiency in this case is a tradeoff between size and speed. The most memory-space efficient way to store a flag value is as single bits or groups of bits just large enough to hold all the possible values. This is because most computers cannot address individual bits in memory, so the bit or bits of interest must be extracted from the bytes that contain it.

The most time-efficient way to store flag values is to keep each in its own integer variable. Unfortunately, this method can waste up to 31 bits of a 32-bit variable, which can lead to very inefficient use of memory.

If there are only a few flags, it doesn't matter how they are stored. If there are many flags, it might be advantageous to store them packed in an array of characters or integers. They must then be extracted by a process called bit masking, in which unwanted bits are removed from the ones of interest.

Sometimes it is possible to combine a flag with another value to save space. It might be possible to use high-order bits of integers that have values smaller than what an integer can hold. Another possibility is that some data is always a multiple of 2 or 4, so the low-order bits can be used to store a flag. For instance, in FAQ III.5, the low-order bit of a pointer is used to hold a flag that identifies which of two possible types the pointer points to.

Cross Reference:

X.2: What is meant by "bit masking"?

X.3: Are bit fields portable?

X.4: Is it better to bitshift a value than to multiply by 2?

X.2: What is meant by "bit masking"?

Answer:

Bit masking means selecting only certain bits from byte(s) that might have many bits set. To examine some bits of a byte, the byte is bitwise "ANDed" with a mask that is a number consisting of only those bits of interest. For instance, to look at the one's digit (rightmost digit) of the variable `flags`, you bitwise AND it with a mask of one (the bitwise AND operator in C is `&`):

```
flags & 1;
```

To set the bits of interest, the number is bitwise "ORed" with the bit mask (the bitwise OR operator in C is `|`). For instance, you could set the one's digit of `flags` like so:

```
flags = flags | 1;
```

Or, equivalently, you could set it like this:

```
flags |= 1;
```

To clear the bits of interest, the number is bitwise ANDed with the one's complement of the bit mask. The "one's complement" of a number is the number with all its one bits changed to zeros and all its zero bits changed to ones. The one's complement operator in C is ~. For instance, you could clear the one's digit of `flags` like so:

```
flags = flags & ~1;
```

Or, equivalently, you could clear it like this:

```
flags &= ~1;
```

Sometimes it is easier to use macros to manipulate flag values. Listing X.2 shows a program that uses some macros to simplify bit manipulation.

Listing X.2. Macros that make manipulating flags easier.

```
/* Bit Masking */

/* Bit masking can be used to switch a character
   between lowercase and uppercase */

#define BIT_POS(N)            ( 1U << (N) )
#define SET_FLAG(N, F)        ( (N) |= (F) )
#define CLR_FLAG(N, F)        ( (N) &= -(F) )
#define TST_FLAG(N, F)        ( (N) & (F) )

#define BIT_RANGE(N, M)       ( BIT_POS((M)+1 - (N))-1 << (N) )
#define BIT_SHIFTL(B, N)      ( (unsigned)(B) << (N) )
#define BIT_SHIFTR(B, N)      ( (unsigned)(B) >> (N) )
#define SET_MFLAG(N, F, V)    ( CLR_FLAG(N, F), SET_FLAG(N, V) )
#define CLR_MFLAG(N, F)       ( (N) &= ~(F) )
#define GET_MFLAG(N, F)       ( (N) & (F) )

#include <stdio.h>

void main()
{

  unsigned char ascii_char = 'A';        /*  char = 8 bits only */
  int test_nbr = 10;

  printf("Starting character = %c\n", ascii_char);

  /*  The 5th bit position determines if the character is
      uppercase or lowercase.
      5th bit = 0  - Uppercase
      5th bit = 1  - Lowercase      */

  printf("\nTurn 5th bit on = %c\n", SET_FLAG(ascii_char, BIT_POS(5)) );
  printf("Turn 5th bit off = %c\n\n", CLR_FLAG(ascii_char, BIT_POS(5)) );

  printf("Look at shifting bits\n");
  printf("=====================\n");
  printf("Current value = %d\n", test_nbr);
  printf("Shifting one position left = %d\n",
         test_nbr = BIT_SHIFTL(test_nbr, 1) );
  printf("Shifting two positions right = %d\n",
         BIT_SHIFTR(test_nbr, 2) );
}
```

`BIT_POS(N)` takes an integer `N` and returns a bit mask corresponding to that single bit position (`BIT_POS(0)` returns a bit mask for the one's digit, `BIT_POS(1)` returns a bit mask for the two's digit, and so on). So instead of writing

```
#define A_FLAG  4096
#define B_FLAG  8192
```

you can write

```
#define A_FLAG  BIT_POS(12)
#define B_FLAG  BIT_POS(13)
```

which is less prone to errors.

The SET_FLAG(N, F) macro sets the bit at position F of variable N. Its opposite is CLR_FLAG(N, F), which clears the bit at position F of variable N. Finally, TST_FLAG(N, F) can be used to test the value of the bit at position F of variable N, as in

```
if (TST_FLAG(flags, A_FLAG))
        /* do something */;
```

The macro BIT_RANGE(N, M) produces a bit mask corresponding to bit positions N through M, inclusive. With this macro, instead of writing

```
#define FIRST_OCTAL_DIGIT  7       /* 111 */
#define SECOND_OCTAL_DIGIT 56      /* 111000 */
```

you can write

```
#define FIRST_OCTAL_DIGIT  BIT_RANGE(0, 2)    /* 111 */
#define SECOND_OCTAL_DIGIT BIT_RANGE(3, 5)    /* 111000 */
```

which more clearly indicates which bits are meant.

The macro BIT_SHIFT(B, N) can be used to shift value B into the proper bit range (starting with bit N). For instance, if you had a flag called C that could take on one of five possible colors, the colors might be defined like this:

```
#define C_FLAG          BIT_RANGE(8, 10)      /* 11100000000 */

/* here are all the values the C flag can take on */
#define C_BLACK         BIT_SHIFTL(0, 8)       /* 00000000000 */
#define C_RED           BIT_SHIFTL(1, 8)       /* 00100000000 */
#define C_GREEN         BIT_SHIFTL(2, 8)       /* 01000000000 */
#define C_BLUE          BIT_SHIFTL(3, 8)       /* 01100000000 */
#define C_WHITE         BIT_SHIFTL(4, 8)       /* 10000000000 */

#define C_ZERO          C_BLACK
#define C_LARGEST       C_WHITE

/* A truly paranoid programmer might do this */
#if C_LARGEST > C_FLAG
        Cause an error message. The flag C_FLAG is not
        big enough to hold all its possible values.
#endif /* C_LARGEST > C_FLAG */
```

The macro SET_MFLAG(N, F, V) sets flag F in variable N to the value V. The macro CLR_MFLAG(N, F) is identical to CLR_FLAG(N, F), except the name is changed so that all the operations on multibit flags have a similar naming convention. The macro GET_MFLAG(N, F) gets the value of flag F in variable N, so it can be tested, as in

```
if (GET_MFLAG(flags, C_FLAG) == C_BLUE)
        /* do something */;
```

NOTE

Beware that the macros `BIT_RANGE()` and `SET_MFLAG()` refer to the `N` argument twice, so the expression

```
SET_MFLAG(*x++, C_FLAG, C_RED);
```

will have undefined, potentially disastrous behavior.

Cross Reference:

X.1: What is the most efficient way to store flag values?

X.3: Are bit fields portable?

X.3: Are bit fields portable?

Answer:

Bit fields are not portable. Because bit fields cannot span machine words, and because the number of bits in a machine word is different on different machines, a particular program using bit fields might not even compile on a particular machine.

Assuming that your program does compile, the order in which bits are assigned to bit fields is not defined. Therefore, different compilers, or even different versions of the same compiler, could produce code that would not work properly on data generated by compiled older code. Stay away from using bit fields, except in cases in which the machine can directly address bits in memory and the compiler can generate code to take advantage of it and the increase in speed to be gained would be essential to the operation of the program.

Cross Reference:

X.1: What is the most efficient way to store flag values?

X.2: What is meant by "bit masking"?

X.4: Is it better to bitshift a value than to multiply by 2?

Answer:

Any decent optimizing compiler will generate the same code no matter which way you write it. Use whichever form is more readable in the context in which it appears. The following program's assembler code can be viewed with a tool such as CODEVIEW on DOS/Windows or the disassembler (usually called "dis") on UNIX machines:

Listing X.4. Multiplying by 2 and shifting left by 1 are often the same.

```
void main()
{

  unsigned int test_nbr = 300;
  test_nbr *= 2;

  test_nbr = 300;
  test_nbr <<= 1;

}
```

Cross Reference:

X.1: What is the most efficient way to store flag values?

X.5: What is meant by high-order and low-order bytes?

Answer:

We generally write numbers from left to right, with the most significant digit first. To understand what is meant by the "significance" of a digit, think of how much happier you would be if the first digit of your paycheck was increased by one compared to the last digit being increased by one.

The bits in a byte of computer memory can be considered digits of a number written in base 2. That means the least significant bit represents one, the next bit represents 2×1, or 2, the next bit represents 2×2×1, or 4, and so on. If you consider two bytes of memory as representing a single 16-bit number, one byte will hold the least significant 8 bits, and the other will hold the most significant 8 bits. Figure X.5 shows the bits arranged into two bytes. The byte holding the least significant 8 bits is called the least significant byte, or low-order byte. The byte containing the most significant 8 bits is the most significant byte, or high-order byte.

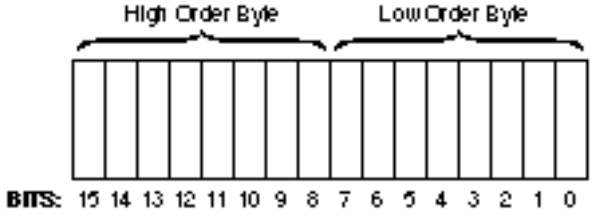

Figure X.5.

The bits in a two-byte integer.

Cross Reference:

X.6: How are 16- and 32-bit numbers stored?

X.6: How are 16- and 32-bit numbers stored?

Answer:

A 16-bit number takes two bytes of storage, a most significant byte and a least significant byte. The preceding FAQ (X.5) explains which byte is which. If you write the 16-bit number on paper, you would start with the most significant byte and end with the least significant byte. There is no convention for which order to store them in memory, however.

Let's call the most significant byte `M` and the least significant byte `L`. There are two possible ways to store these bytes in memory. You could store `M` first, followed by `L`, or `L` first, followed by `M`. Storing byte `M` first in memory is called "forward" or "big-endian" byte ordering. The term *big endian* comes from the fact that the "big end" of the number comes first, and it is also a reference to the book *Gulliver's Travels*, in which the term refers to people who eat their boiled eggs with the big end on top.

Storing byte `L` first is called "reverse" or "little-endian" byte ordering. Most machines store data in a big-endian format. Intel CPUs store data in a little-endian format, however, which can be confusing when someone is trying to connect an Intel microprocessor-based machine to anything else.

A 32-bit number takes four bytes of storage. Let's call them `Mm`, `MI`, `Lm`, and `LI` in decreasing order of significance. There are 4! (4 factorial, or 24) different ways in which these bytes can be ordered. Over the years, computer designers have used just about all 24 ways. The most popular two ways in use today, however, are (`Mm`, `MI`, `Lm`, `LI`), which is big-endian, and (`LI`, `Lm`, `MI`, `Mm`), which is little-endian. As with 16-bit numbers, most machines store 32-bit numbers in a big-endian format, but Intel machines store 32-bit numbers in a little-endian format.

Cross Reference:

X.5: What is meant by high-order and low-order bytes?