IBM

Writing endian-independent code in C

Don't let endianness "byte" you

Architectures, processors, network stacks, and communication protocols all have to define endianness at some point. This article explains how endianness affects code, how to determine endianness at run time, and how to write code that can reverse byte order and free you from being bound to a certain endian.

Harsha Adiga works in the IBM Software Group in Bangalore, India. He is heavily involved in various Linux, open source, and working groups. His primary focus includes Linux and UNIX internals, porting, compilers, and code optimization. He has been involved in software development and testing on Linux and UNIX platforms for more than six years. You can contact him at haradiga@in.ibm.com.

15 May 2007 (First published 24 April 2007)

Also available in Russian

Introduction

To understand the concept of *endianness* (see Endianness), you need to be familiar, at a highly abstract level, with memory. All you need to know about memory is that it's one large array. The array contains bytes. In the computer world, people use *address* to refer to the array locations.

Each address stores one element of the memory array. Each element is typically one byte. In some memory configurations, each address stores something besides a byte. However, those are extremely rare so, for now, let's make the broad assumption that all memory addresses store bytes.

Endianness
The attribute of a system that indicates whether integers are represented with the most significant byte stored at the lowest address (big endian) or at the highest address (little endian).

Storing bytes in memory

I refer to 32 bits, which is the same as four bytes. Integers or single-precision floating point numbers are all 32-bits long. But since each memory address can store a single byte and not four bytes, let's split the 32-bit quantity into four bytes. For example, suppose you have a 32-bit quantity written as 12345678, which is hexadecimal. Since each hex digit is four bits, you need eight hex digits to represent the 32-bit value. The four bytes are: 12, 34, 56, and 78. There are two ways to store this in memory, as shown below.

Big-endian: Store the most significant byte in the smallest address, as follows:

Table 1. Big-endian storage

Address	Value
1000	12
1001	34
1002	56
1003	78

Little-endian: Store the least significant byte in the smallest address, as follows:

Table 2. Little-endian storage

Address	Value
1000	78
1001	56
1002	34
1003	12

Notice that they are in the reverse order. To remember which is which, think of the least significant byte being stored first (little-endian), or the most significant byte being stored first (big-endian).

Registers and endianness

Endianness only makes sense when you're breaking up a multi-byte quantity and are trying to store the bytes at consecutive memory locations. However, if you have a 32-bit register storing a 32-bit value, it makes no sense to talk about endianness. The register is neither big-endian nor little-endian; it's just a register holding a 32-bit value. The rightmost bit is the least significant bit, and the leftmost bit is the most significant bit.

Some people classify a *register* as a big-endian, because it stores its most significant byte at the lowest memory address.

Importance of endianness

Endianness is the attribute of a system that indicates whether integers are represented from left to right or right to left. In today's world of virtual machines and gigahertz processors, why would a programmer care about such a minor topic? Unfortunately, endianness must be chosen every time a hardware or software architecture is designed. There isn't much in the way of natural law to help decide, so implementations vary.

All processors must be designated as either big-endian or little-endian. For example, the 80x86 processors from Intel® and their clones are little-endian, while Sun's SPARC, Motorola's 68K, and the PowerPC® families are all big-endian.

Why is endianness so important? Suppose you are storing integer values to a file, and you send the file to a machine that uses the opposite endianness as it reads in the value. This causes problems because of endianness; you'll read in reversed values that won't make sense.

Endianness is also a big issue when sending numbers over the network. Again, if you send a value from a machine of one endianness to a machine of the opposite endianness, you'll have problems. This is even worse over the network because you might not be able to determine the endianness of the machine that sent you the data.

Listing 1 shows an example of the dangers of programming while unaware of endianness.

Listing 1. Example

```
#include <stdio.h>
#include <string.h>

int main (int argc, char* argv[]) {
    FILE* fp;

    /* Our example data structure */
    struct {
        char one[4];
        int  two;
        char three[4];
    } data;

    /* Fill our structure with data */
    strcpy (data.one, "foo");
    data.two = 0x01234567;
    strcpy (data.three, "bar");

    /* Write it to a file */
    fp = fopen ("output", "wb");
    if (fp) {
        fwrite (&data, sizeof (data), 1, fp);
        fclose (fp);
    }
}
```

The above code compiles properly on all machines. However, the output is different on big-endian and little-endian machines. The program outputs, when examined using the `hexdump` utility, are shown below in Listings 2 and 3.

Listing 2. hexdump –C output on big-endian machines

```
00000000  66 6f 6f 00 12 34 56 78  62 61 72 00              |foo..4Vxbar.|
0000000c
```

Listing 3. hexdump -C output on little-endian machines

```
00000000  66 6f 6f 00 78 56 34 12  62 61 72 00              |foo.xV4.bar.|
0000000c
```

When endianness affects code

Endianness doesn't apply to everything. If you do bitwise or bitshift operations on an int, you don't notice the endianness. The machine arranges the multiple bytes, so the least significant byte is still the least significant byte, and the most significant byte is still the most significant byte.

Similarly, it's natural to wonder whether strings might be saved in some sort of strange order, depending on the machine. To understand this, let's go back to the basics of an array. A C-style string, after all, is still an array of characters. Each character requires one byte of memory, since characters are represented in ASCII. In an array, the address of consecutive array elements increases. Thus, &arr[i] is less than &arr[i+1]. Though it isn't obvious, if something is stored with increasing addresses in memory, it's going to be stored with increasing "addresses" in a file. When you write to a file, you usually specify an address in memory and the number of bytes you wish to write to the file starting at that address.

For example, suppose you have a C-style string in memory called `man`. Assume that `m` is stored at address 1000, `a` at address 1001, and `n` at address 1002. The null character \0 is at address 1003. Since C-style strings are arrays of characters, they follow the rules of characters. Unlike int or long, you can easily see the individual bytes of a C-style string, one byte at a time. You use array indexing to access the bytes (characters) of a string. But you can't easily index the bytes of an int or long without playing some pointer tricks. The individual bytes of an int are more or less hidden from you.

Now imagine writing out this string to a file using some sort of `write()` method. You specify a pointer to `m` and the number of bytes you wish to print (in this case four). The `write()` method proceeds byte by

byte in the character string and writes it to the file, starting with `m` and working to the null character.

Given this explanation, it's clear that endianness doesn't matter with C-style strings.

Endianness *does* matter when you use a type cast that depends on a certain endian being in use. One example is shown in Listing 4, but keep in mind that there are many different type casts that can cause problems.

Listing 4. Forcing a byte order

```
unsigned char endian[2] = {1, 0};
short x;

x = *(short *) endian;
```

What would be the value of `x`? Let's look at what this code is doing. You're creating an array of two bytes, and then casting that array of two bytes into a single short. By using an array, you basically forced a certain byte order, and you're going to see how the system treats those two bytes.

If this is a little-endian system, the 0 and 1 is interpreted backwards and seen as if it is 0,1. Since the high byte is 0, it doesn't matter and the low byte is 1, so `x` is equal to 1.

On the other hand, if it's a big-endian system, the high byte is 1 and the value of `x` is 256.

Determine endianness at run time

One way to determine endiannes is to test the memory layout of a predefined constant. For example, you know that the layout of a 32-bit integer variable with a value of 1 is 00 00 00 01 for big-endian and 01 00 00 00 for little-endian. By looking at the first byte of the constant, you can tell the endianness of the running platform and then take the appropriate action.

Listing 5 tests the first byte of the multi-byte integer `i` to determine if it is 0 or 1. If it is 1, the running platform is assumed to be little-endian. If it is 0, it is assumed to be big-endian.

Listing 5. Determine endianness

```
const int i = 1;
#define is_bigendian() ( (*(char*)&i) == 0 )

int main(void) {
    int val;
    char *ptr;
    ptr = (char*) &val;
    val = 0x12345678;
    if (is_bigendian()) {
        printf("%X.%X.%X.%X\n", u.c[0], u.c[1], u.c[2], u.c[3]);
    } else {
        printf("%X.%X.%X.%X\n", u.c[3], u.c[2], u.c[1], u.c[0]);
    }
    exit(0);
}
```

Another way to determine endiannes is to use a character pointer to the bytes of an int and then check its first byte to see if it is 0 or 1. Listing 6 shows an example.

Listing 6. Character pointer

```
#define LITTLE_ENDIAN 0
#define BIG_ENDIAN    1

int endian() {
    int i = 1;
    char *p = (char *)&i;

    if (p[0] == 1)
        return LITTLE_ENDIAN;
    else
        return BIG_ENDIAN;
}
```

Network byte order

Network stacks and communication protocols must also define their endianness. Otherwise, two nodes of different endianness would be unable to communicate. This is a more substantial example of endianness affecting the embedded programmer. All of the protocol layers in the Transmission Control Protocol and the Internet Protocol (TCP/IP) suite are defined to be big-endian. Any 16-bit or 32-bit value within the various layer headers (such as an IP address, a packet length, or a checksum) must be sent and received with its most significant byte first.

The multi-byte integer representation used by the TCP/IP protocols is sometimes called *network byte order*. Even if the computers at each end are little-endian, multi-byte integers passed between them must be converted to network byte order prior to transmission across the network and converted back to little-endian at the receiving end.

Assume you want to establish a TCP socket connection to a computer whose IP address is 192.0.1.2. Internet Protocol version 4 (IPv4) uses a unique 32-bit integer to identify each network host. The dotted decimal IP address must be translated into such an integer.

Suppose an 80x86-based PC is to talk to a SPARC-based server over the Internet. Without further manipulation, the 80x86 processor would convert 192.0.1.2 to the little-endian integer 0x020100C0 and transmit the bytes in the order 02 01 00 C0. The SPARC would receive the bytes in the order 02 01 00 C0, reconstruct the bytes into a big endian integer 0x020100c0, and misinterpret the address as 2.1.0.192.

If the stack runs on a little-endian processor, it has to reorder, at run time, the bytes of every multi-byte data field within the various headers of the layers. If the stack runs on a big-endian processor, there's nothing to worry about. For the stack to be portable (so it runs on processors of both types), it has to decide whether or not to do this reordering, typically at compile time.

To enable these conversions, sockets provide a set of macros to convert to and from host to network byte order, as shown below.

`htons()`
: Reorder the bytes of a 16-bit unsigned value from processor order to network order. The macro name can be read as "host to network short."

`htonl()`
: Reorder the bytes of a 32-bit unsigned value from processor order to network order. The macro name can be read as "host to network long."

`ntohs()`
: Reorder the bytes of a 16-bit unsigned value from network order to processor order. The macro name can be read as "network to host short."

`ntohl()`
: Reorder the bytes of a 32-bit unsigned value from network order to processor order. The macro name can be read as "network to host long."

Consider the C program in Listing 7.

Listing 7. Sample C program

```
#include <stdio.h>
main() {
    int i;
    long x = 0x112A380; /* Value to play with */
    unsigned char *ptr = (char *) &x; /* Byte pointer */

    /* Observe value in host byte order */
    printf("x in hex: %x\n", x);
    printf("x by bytes: ");

    for (i=0; i < sizeof(long); i++)
        printf("%x\t", ptr[i]);
    printf("\n");

    /* Observe value in network byte order */
    x = htonl(x);
    printf("\nAfter htonl()\n");
    printf("x in hex: %x\n", x);
    printf("x by bytes: ");

    for (i=0; i < sizeof(long); i++)
        printf("%x\t", ptr[i]);
    printf("\n");
}
```

This program shows how the long variable *x* with the value 112A380 (hexadecimal) is stored.

When this program is executed on a little-endian processor, it outputs the information in Listing 8.

Listing 8. Little-endian output

```
x in hex: 112a380
x by bytes: 80 a3 12 1
After htonl()
x in hex: 80a31201
x by bytes: 1 12 a3 80
```

When you look at the individual bytes of *x*, you find the least significant byte (0x80) in the lowest address. After you call `htonl( )` to convert to network byte order, you get the most significant byte (0x1) in the lowest address. Of course, if you try to print the value of *x* after converting its byte order, you get a meaningless number.

Listing 9 shows the output from executing the same program on a big-endian processor.

Listing 9. Big-endian output

```
x in hex: 112a380
x by bytes: 1 12 a3 80
After htonl()
x in hex: 112a380
x by bytes: 1 12 a3 80
```

Here you see the most significant byte (0x1) in the lowest address. Calling `htonl( )` to convert to network byte order does not change anything because network byte order is already big-endian.

Reversing the byte order

Now let's get down to writing some code that is not bound to a certain endian. There are many ways of doing this. The goal is to write code that doesn't fail, regardless of the endianness of the machine. You need to ensure that the file data is in the correct endian when read from or written to. It would also be nice to avoid having to specify conditional compilation flags and just let the code automatically determine the endianness of the machine.

Let's write a set of functions that automatically reverse the byte order of a given parameter, depending on the endian of the machine.

First, you need to deal with `short` by taking apart the two bytes of the short parameter `s` with some simple bit math and then gluing them back together in reverse order. As shown in Listing 10 below, the

function finally returns reversed short if the processor is little-endian. Otherwise, it simply returns `s`.

Listing 10. Method 1: Using bit shifting and bit ANDs

```
short reverseShort (short s) {
    unsigned char c1, c2;

    if (is_bigendian()) {
        return s;
    } else {
        c1 = s & 255;
        c2 = (s >> 8) & 255;

        return (c1 << 8) + c2;
    }
}
```

In the function below, you cast `short` to see it as an array of characters, and then assign each byte to a new array in the reverse order, if the processor is little-endian.

Listing 11. Method 2: Using pointer to an array of characters

```
short reverseShort (char *c) {
    short s;
    char *p = (char *)&s;

    if (is_bigendian()) {
        p[0] = c[0];
        p[1] = c[1];
    } else {
        p[0] = c[1];
        p[1] = c[0];
    }

    return s;
}
```

Now let's handle `int`.

Listing 12. Method 1:Using bit shifting and bit ANDs with int

```
int reverseInt (int i) {
    unsigned char c1, c2, c3, c4;

    if (is_bigendian()) {
        return i;
    } else {
        c1 = i & 255;
        c2 = (i >> 8) & 255;
        c3 = (i >> 16) & 255;
        c4 = (i >> 24) & 255;

        return ((int)c1 << 24) + ((int)c2 << 16) + ((int)c3 << 8) + c4;
}
```

This is more or less the same thing you did to reverse a `short`, but it switches around four bytes instead of two.

Listing 13. Method 2: Using pointer to an array of characters with int

```
short reverseInt (char *c) {
    int i;
    char *p = (char *)&i;

    if (is_bigendian()) {
        p[0] = c[0];
        p[1] = c[1];
        p[2] = c[2];
        p[3] = c[3];
    } else {
        p[0] = c[3];
        p[1] = c[2];
        p[2] = c[1];
        p[3] = c[0];
    }

    return i;
}
```

Again, this is exactly what you did to reverse a short, but here you swapped four bytes.

Similarly, you can write code to reverse bytes of float, long, double, and so on, but that is outside the

scope of this article.

Conclusion

There seems to be no significant advantage in using one method of endianness over the other. Both are still common and different architectures use them. Little-endian based processors (and their clones) are used in most personal computers and laptops, so the vast majority of desktop computers today are little-endian.

Endian issues do not affect sequences that have single bytes, because "byte" is considered an atomic unit from a storage point of view. On the other hand, sequences based on multi-byte are affected by endianness and you need to take care while coding.

Resources

Learn

AIX® and UNIX®: The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.

New to AIX and UNIX?: Visit the New to AIX and UNIX page to learn more about AIX and UNIX.

Popular content: See what AIX and UNIX content your peers find interesting.

AIX 5L™ Wiki: Discover a collaborative environment for technical information related to AIX.

Search the AIX and UNIX library by topic:

- System administration
- Application development
- Performance
- Porting
- Security
- Tips
- Tools and utilities
- Java™ technology
- Linux®
- Open source

Safari bookstore: Visit this e-reference library to find specific technical resources.

developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.

Podcasts: Tune in and catch up with IBM technical experts.

Get products and technologies

IBM trial software: Build your next development project with software for

Dig deeper into AIX and Unix on developerWorks

- Overview
- New to AIX and Unix
- Technical library (articles and more)
- Forums
- Community
- Downloads and products
- Open source projects
- Events

download directly from developerWorks.

Discuss

Participate in the developerWorks blogs and get involved in the developerWorks community.

Participate in the AIX and UNIX forums:

- AIX 5L—technical forum
- AIX for Developers Forum
- Cluster Systems Management
- IBM Support Assistant
- Performance Tools -- technical
- Virtualization—technical
- More AIX and UNIX forums