Understanding Big and Little Endian Byte Order

Problems with byte order are frustrating, and I want to spare you the grief I experienced. Here's the key:

- **Problem: Computers speak different languages, like people.** Some write data "left-to-right" and others "right-to-left".
  - A machine can read its own data just fine - problems happen when one computer stores data and a different type tries to read it.
- Solutions
  - Agree to a common format (i.e., all network traffic follows a single format), or
  - Always include a header that describes the format of the data. If the header appears backwards, it means data was stored in the other format and needs to be converted.

Numbers vs. Data

The most important concept is to recognize the difference between a number and the data that represents it.

A **number** is an abstract concept, such as a count of something. You have ten fingers. The idea of "ten" doesn't change, no matter what representation you use: ten, 10, diez (Spanish), ju (Japanese), 1010 (binary), X (Roman numeral)... these representations all point to the same concept of "ten".

Contrast this with data. **Data** is a physical concept, a raw sequence of bits and bytes stored on a computer. **Data has no inherent meaning** and must be interpreted by whoever is reading it.

Data is like human writing, which is simply marks on paper. There is no inherent meaning in these marks. If we see a line and a circle (like this: |O) we may interpret it to mean "ten".

But we assumed the marks referred to a number. They could have been the letters "IO", a moon of Jupiter. Or perhaps the Greek goddess. Or maybe an abbreviation for Input/Output. Or someone's initials. Or the number 2 in binary ("10"). The list of possibilities goes on.

The point is that a single piece of data (|O) can be interpreted in many ways, and the meaning is unclear until someone clarifies the intent of the author.

Computers face the same problem. They store data, not abstract concepts, and do so using a sequence of 1's and 0's. Later, they read back the 1's and 0's and try to recreate the abstract concept from the raw data. Depending on the assumptions made, the 1's and 0's can mean very different things.

Why does this problem happen? Well, there's no rule that all computers must use the same language,

just like there's no rule all humans need to. Each type of computer is internally consistent (it can read back its own data), but there are no guarantees about how **another** type of computer will interpret the data it created.

Basic concepts

- Data (bits and bytes, or marks on paper) is meaningless; it must be interpreted to create an abstract concept, like a number.
- Like humans, computers have different ways to store the same abstract concept. (i.e., we have many ways to say "ten": ten, 10, diez, etc.)

Storing Numbers as Data

Thankfully, most computers agree on a few basic data formats (this was not always the case). This gives us a common starting point which makes our lives a bit easier:

- A bit has two values (on or off, 1 or 0)
- A byte is a sequence of 8 bits
  - The "leftmost" bit in a byte is the biggest. So, the binary sequence 00001001 is the decimal number 9. 00001001 = ($2^3 + 2^0 = 8 + 1 = 9$).
  - Bits are numbered from right-to-left. Bit 0 is the rightmost and the smallest; bit 7 is leftmost and largest.

We can use these basic agreements as a building block to exchange data. If we store and read data one byte at a time, it will work on any computer. The concept of a byte is the same on all machines, and the idea of "Byte 0" is the same on all machines. Computers also agree on the order you sent them bytes -- they agree on which byte was sent first, second, third, etc. so "Byte 35" is the same on all machines.

So what's the problem -- computers agree on single bytes, right?

Well, this is fine for single-byte data, like ASCII text. However, a lot of data needs to be stored using multiple bytes, like integers or floating-point numbers. And there is no agreement on how these sequences should be stored.

Byte Example

Consider a sequence of 4 bytes, named W X Y and Z - I avoided naming them A B C D because they are hex digits, which would be confusing. So, each byte has a value and is made up of 8 bits.

```
Byte Name:     W       X       Y       Z
Location:      0       1       2       3
Value (hex):   0x12    0x34    0x56    0x78
```

For example, W is an entire byte, 0x12 in hex or 00010010 in binary. If W were to be interpreted as a number, it would be "18" in decimal (by the way, there's nothing saying we have to interpret it as a

number - it could be an ASCII character or something else entirely).

With me so far? We have 4 bytes, W X Y and Z, each with a different value.

Understanding Pointers

Pointers are a key part of programming, especially the C programming language. A pointer is a number that references a memory location. It is up to us (the programmer) to interpret the data at that location.

In C, when you cast (convert) a pointer to certain type (such as a char * or int *), it tells the computer how to interpret the data at that location. For example, let's declare

```
void *p = 0; // p is a pointer to an unknown data type
             // p is a NULL pointer -- do not dereference
char *c;     // c is a pointer to a single byte
```

Note that we can't get the data from p because we don't know its type. p could be pointing at a single number, a letter, the start of a string, your horoscope, an image -- we just don't know how many bytes to read, or how to interpret what's there.

Now, suppose we write

```
c = (char *)p;
```

Ah -- now this statement tells the computer to point to the same place as p, and interpret the data as a single character (1 byte). In this case, c would point to memory location 0, or byte W. If we printed c, we'd get the value in W, which is hex 0x12 (remember that W is a whole byte).

This example does not depend on the type of computer we have -- again, all computers agree on what a single byte is (in the past this was not the case).

The example is helpful, even though it is the same on all computers -- if we have a pointer to a single byte (char *, a single byte), we can walk through memory, reading off a byte at a time. We can examine any memory location and the endian-ness of a computer won't matter -- every computer will give back the same information.

So, What's The Problem?

Problems happen when computers try to read multiple bytes. Some data types contain multiple bytes, like long integers or floating-point numbers. A single byte has only 256 values, so can store 0 - 255.

Now problems start - when you read multi-byte data, where does the biggest byte appear?

- Big endian machine: Stores data **big-end first**. When looking at multiple bytes, the first byte (lowest address) is the biggest.
- Little endian machine: Stores data **little-end first**. When looking at multiple bytes, the first byte is **smallest**.

The naming makes sense, eh? Big-endian thinks the big-end is first. (By the way, the big-endian / little-endian naming comes from Gulliver's Travels, where the Lilliputans argue over whether to break eggs on the little-end or big-end. Sometimes computer debates are just as meaningful :-))

Again, endian-ness does not matter if you have a single byte. If you have one byte, it's the only data you read so there's only one way to interpret it (again, because computers agree on what a byte is).

Now suppose we have our 4 bytes (W X Y Z) stored the same way on a big-and little-endian machine. That is, memory location 0 is W on both machines, memory location 1 is X, etc.

We can create this arrangement by remembering that bytes are machine-independent. We can walk memory, one byte at a time, and set the values we need. This will work on any machine:

```
c = 0;     // point to location 0 (won't work on a real machine!)
*c = 0x12; // Set W's value
c = 1;     // point to location 1
*c = 0x34; // Set X's value
...        // repeat for Y and Z; details left to reader
```

This code will work on any machine, and we have both set up with bytes W, X, Y and Z in locations 0, 1, 2 and 3.

Interpreting Data

Now let's do an example with multi-byte data (finally!). Quick review: a "short int" is a 2-byte (16-bit) number, which can range from 0 - 65535 (if unsigned). Let's use it in an example:

```
short *s; // pointer to a short int (2 bytes)
s = 0;    // point to location 0; *s is the value
```

So, s is a pointer to a short, and is now looking at byte location 0 (which has W). What happens when we read the value at s?

- Big endian machine: I think a short is two bytes, so I'll read them off: location s is address 0 (W, or 0x12) and location s + 1 is address 1 (X, or 0x34). Since the first byte is biggest (I'm big-endian!), the number must be 256 * byte 0 + byte 1, or 256*W + X, or 0x1234. I multiplied the first byte by 256 (2^8) because I needed to shift it over 8 bits.

- Little endian machine: I don't know what Mr. Big Endian is smoking. Yeah, I agree a short is 2 bytes, and I'll read them off just like him: location s is 0x12, and location s + 1 is 0x34. But in my world, the first byte is the littlest! The value of the short is byte 0 + 256 * byte 1, or 256*X + W, or 0x3412.

Keep in mind that both machines start from location s and read memory going upwards. There is no confusion about what location 0 and location 1 mean. There is no confusion that a short is 2 bytes.

But do you see the problem? The big-endian machine thinks s = 0x1234 and the little-endian machine thinks s = 0x3412. The same exact data gives two different numbers. Probably not a good thing.

Yet another example

Let's do another example with 4-byte integer for "fun":

```
int *i; // pointer to an int (4 bytes on 32-bit machine)
i = 0;  // points to location zero, so *i is the value there
```

Again we ask: what is the value at i?

- Big endian machine: An int is 4 bytes, and the first is the largest. I read 4 bytes (W X Y Z) and W is the largest. The number is 0x12345678.
- Little endian machine: Sure, an int is 4 bytes, but the first is smallest. I also read W X Y Z, but W belongs way in the back -- it's the littlest. The number is 0x78563412.

Same data, different results - not a good thing. Here's an interactive example using the numbers above, feel free to plug in your own:

The NUXI Problem

Issues with byte order are sometimes called the NUXI problem: UNIX stored on a big-endian machine can show up as NUXI on a little-endian one.

Suppose we want to store 4 bytes (U, N, I and X) as two shorts: UN and IX. Each letter is a entire byte, like our WXYZ example above. To store the two shorts we would write:

```
short *s; // pointer to set shorts
s = 0;    // point to location 0
*s = UN;  // store first short: U * 256 + N (fictional code)
s = 2;    // point to next location
*s = IX;  // store second short: I * 256 + X
```

This code is not specific to a machine. If we store "UN" on a machine and ask to read it back, it had better be "UN"! I don't care about endian issues, if we store a value on one machine, we need to get the same value back.

However, if we look at memory one byte at a time (using our char * trick), the order could vary. On a big endian machine we see:

```
Byte:     U N I X
Location: 0 1 2 3
```

Which make sense. U is the biggest byte in "UN" and is stored first. The same goes for IX: I is the biggest, and stored first.

On a little-endian machine we would see:

```
Byte:     N U X I
Location: 0 1 2 3
```

And this makes sense also. "N" is the littlest byte in "UN" and is stored first. Again, even though the bytes are stored "backwards" in memory, the little-endian machine *knows* it is little endian, and interprets them correctly when reading the values back. Also, note that we can specify hex numbers such as x = 0x1234 on any machine. Even a little-endian machine knows what you mean when you write 0x1234, and won't force you to swap the values yourself (you specify the hex number to write, and it figures out the details and swaps the bytes in memory, under the covers. Tricky.).

This scenario is called the "NUXI" problem because byte sequence UNIX is interpreted as NUXI on the other type of machine. Again, this is only a problem if you exchange data -- each machine is internally consistent.

Exchanging Data Between Endian Machines

Computers are connected - gone are the days when a machine only had to worry about reading its own data. Big and little-endian machines need to talk and get along. How do they do this?

Solution 1: Use a Common Format

The easiest approach is to agree to a common format for sending data over the network. The standard network order is actually big-endian, but some people get uppity that little-endian didn't win... we'll just call it "network order".

To convert data to network order, machines call a function hton (host-to-network). On a big-endian machine this won't actually do anything, but we won't talk about that here (the little-endians might get

mad).

But it is important to use hton before sending data, even if you are big-endian. Your program may be so popular it is compiled on different machines, and you want your code to be portable (don't you?).

Similarly, there is a function ntoh (network to host) used to read data off the network. You need this to make sure you are correctly interpreting the network data into the host's format. You need to know the type of data you are receiving to decode it properly, and the conversion functions are:

```
htons() - "Host to Network Short"
htonl() - "Host to Network Long"
ntohs() - "Network to Host Short"
ntohl() - "Network to Host Long"
```

Remember that a single byte is a single byte, and order does not matter.

These functions are critical when doing low-level networking, such as verifying the checksums in IP packets. If you don't understand endian issues correctly your life will be painful - take my word on this one. Use the translation functions, and know why they are needed.

Solution 2: Use a Byte Order Mark (BOM)

The other approach is to include a magic number, such as 0xFEFF, before every piece of data. If you read the magic number and it is 0xFEFF, it means the data is in the same format as your machine, and all is well.

If you read the magic number and it is 0xFFFE (it is backwards), it means the data was written in a format different from your own. You'll have to translate it.

A few points to note. First, the number isn't really magic, but programmers often use the term to describe the choice of an arbitrary number (the BOM could have been any sequence of different bytes). It's called a byte-order mark because it indicates the byte order the data was stored in.

Second, the BOM adds overhead to all data that is transmitted. Even if you are only sending 2 bytes of data, you need to include a 2-byte BOM. Ouch!

Unicode uses a BOM when storing multi-byte data (some Unicode character encodings can have 2, 3 or even 4-bytes per character). XML avoids this mess by storing data in UTF-8 by default, which stores Unicode information one byte at a time. And why is this cool?

(Repeated for the 56th time) "Because endian issues don't matter for single bytes".

Right you are.

Again, other problems can arise with BOM. What if you forget to include the BOM? Do you assume the data was sent in the same format as your own? Do you read the data and see if it looks "backwards" (whatever that means) and try to translate it? What if regular data includes the BOM by coincidence? These situations are not fun.

Why Are There Endian Issues at All? Can't We Just Get Along?

Ah, what a philosophical question.

Each byte-order system has its advantages. Little-endian machines let you read the lowest-byte first, without reading the others. You can check whether a number is odd or even (last bit is 0) very easily, which is cool if you're into that kind of thing. Big-endian systems store data in memory the same way we humans think about data (left-to-right), which makes low-level debugging easier.

But why didn't everyone just agree to one system? Why do certain computers have to try and be different?

Let me answer a question with a question: Why doesn't everyone speak the same language? Why are some languages written left-to-right, and others right-to-left?

Sometimes communication systems develop independently, and later need to interact.

Epilogue: Parting Thoughts

Endian issues are an example of the general encoding problem - data needs to represent an abstract concept, and later the concept needs to be created from the data. This topic deserves its own article (or series), but you should have a better understanding of endian issues. More information:

- Wikipedia entry
- Endian Faq

Programming Printable version

209 Comments

1. Michael P

   Great post! Thank you!

2. angela

   how do you write 999 in little endian

3. Kalid

   Hi Angela, it depends on whether you are using a 2-byte or 4-byte integer. I'll assume you are using a 4-byte integer because they are more common.

   In hex, 999 is 0x03e7. But we need to pad it out to 4 bytes, so it becomes

   0×00 00 03 e7

   (Broken into 1-byte groups for easier reading). On a little-endian machine, the little end (e7) comes first, so it would be stored as

   e7 03 00 00

   To double-check, if you plug these numbers into the endian calculator it should give 999 for little-endian.

   Hope this helps!

4. Vivi

   This is really really helpful! Big Thanks!

5. jakob

   In Byte Example you say that 00100010 is 0×12 or dec. 18. Is that true?

6. Kalid

   Whoops, that was a typo — thanks! It should be:

   0×12 = binary 0001 0010 = decimal 18

   Putting the space helps make it more clear, I'll make the change now. Appreciate the comment.

7. shekhar

   thanks a million..a superb explanation about byte ordering…
   mr. jakob: there is no typo… its right in doc. as 00010010(0×12) and 18(dec)..

8. Kalid

   Hi Shekhar, I'm glad you found it useful. Jakob actually caught the typo, I fixed up the original article

9. Sumeash

   Thank you boss… just when i thought that Endianess is a complete waste in the world of Computer arithmatics!!!.. you proved it works…

10. Kalid

    Hi Sumeash, always happy to help!

11. Navin

    i want to know about the coding … how can we change the number present in big endian to little endian and vice-versa

12. emile

    How will following structure look in big-endian and little-endian systems?
    struct STRUCT {
    unsigned int a:12;
    unsigned int b:10;
    unsigned int c:10;
    };
    struct STRUCT myStruct= {0xFED, 0×345, 0x3AB};

13. Giridhar

    Was so very educative. Now I think i can go easy with my programming.

14. Kalid

    @emile: I'll have to take a closer look at that one and remember how padded items are laid out :).

    @Giridhar: Thanks, glad you liked it.

15. yogesh

    Awesome post!
    thank you

16. Kalid

    Thanks Yogesh!

17. Kalid

    Question from Doran: "How can 2 chars allocate to an unsigned short? It just doesn't make sense to me. I've heard about NUXI few times before, and still I can't get it. Please can you explain it for me. (even in C)

    My reply:

    On a 32-bit computer, a short is composed of 16 bits (2 bytes). In order to set the value, you an specify the short using two bytes, which is 4 hex characters (in C):

    short a = 0×1234;
    short b = 0×5678;

    So short a has 0×1234 (4,660 decimal) and b is similar. Now, instead of using the characters "0-A", let's just use U N I and X to represent each byte. For example, U could be 0×12, N could be 0×34, I could be 0×56, and X could be 0×78.

    short a = 0xUN;
    short b = 0xIX;

    On any machine, these shorts would be stored consecutively in memory. Address 0 and 1 would be "a", and address 2 and 3 would be "b". [Again, each short takes up 2 bytes].

On a big-endian machine, the data would look like this:

Addr 0: U
Addr 1: N
Addr 2: I
Addr 3: X

On a little-endian machine, we store the *smallest* part of the number first. That is, in a = 0xUN, we store “N” first, which are the low-order bits. So in memory it would look like this:

Addr 0: N
Addr 1: U
Addr 2: X
Addr 3: I

Hence the “NUXI” problem. On a big-endian machine the data looks like UNIX, on a little-endian machine the data looks like NUXI. This isn’t a problem if you stay on the same machine (each machine knows how to convert appropriately), but can be a problem if you are exchanging binary data between machines.

Hope this helps,

-Kalid

18. SHAAN

Well information is good!!!
I have one query if there is not much advantage of Big-Endian over Little Endian then why Network Byte order is Big-Endian????

19. Kalid

While there are advantages to each, I don’t think one is clearly better than the other. I think they just had choose one or the other — Big Endian may have been a more popular format at the time :).

20. Steve

Thank you so much. I’ve been lucky thus far doing high end coding, but having rolled my sleeves up to start mucking about with bit and bytes this has been very helpful indeed.

kudos!

21. Kalid

    Thanks Steve, glad you found it useful! It's fun to dip into bits & bytes every once in a while :).

22. Ramkumar

    That was damn good explanation. Thanks a lot for the post

23. Kalid

    Hi Ramkumar, you're welcome. Glad you liked it.

24. socal

    this is a great read

25. Kalid

    Thanks Socal!

26. avvy

    How can I change byte order from Big Endian to Little Endian and vice versa without breaking structure. When we are sending any structure.

27. Kalid

    Hi avvy, you can use the "host to network" and "network to host" functions to convert data (more info here: http://linux.die.net/man/3/htons). You'd have to convert each field in the structure separately.

28. Shweta

    Hi Kalid,

    I cannot get a better picture of this topic wherever i search. I was searching about endianness in a hurry as was looking for some stuff which can atleast give the details in short and i feel lucky to find your post. Your post came like an angel as i had some urgency to find about endianness faster.
    Very briefly you told whole story about it. Details perfect…Flow of explanation perfect! Kudos!
    I'll appreciate if you drop a small mail whenver you post stuff like this.

Thanks a tonn!

29. Kalid

Hi Shweta, glad you liked the article! If you'd like to receive emails when new posts appear, just enter your email address in the "subscribe" form on the upper-right of the page. Thanks for the comment.

30. Shweta

Hey thanks for info..done it
Hoping to see something informative soon.

31. Alrenous

You have a 'locaiton' in there, if you care to fix it.
I second everyone else and say that this is awesome. I didn't even know this issue existed and now I understand it well (I think).

I think you should mention explicitly that if you store 'UNIX' in little-endian it will end up as NUXI in big-endian. Not strictly necessary, but I think it would Explain it Better. (Even Better.) Or perhaps lainExp it terBet. (enEv terBet.)

Whenever I see the word endian I think first of Ender Wiggin. The enemy is down and so forth.

32. Kalid

Thanks Alrenous, glad you're enjoying it. Appreciate the tip — went through and cleaned up a bunch of typos (it's a bit embarrassing how many were in there).

Good suggestion on the explanation, I'm always looking for ways to make things clearer (that's why this isn't best explained).

I hadn't made the Ender Wiggin association, but I love Ender's game… maybe there's a way to fit him and Bean into this article somewhere.

33. Breeson

Gr8 post. really helpful.

34. Kalid

Thanks Breeson, glad it was useful.

35. Kumar

Thanks alot Kalid ur explanation about endianness is awesome. I hve one question that need to be answered Is there is a way In a mixed binary file with 4 byte Integers and single byte characters to identify whether the byte we read from the file is a character or is a part of 4 byte integer data.

It would be very helpful if you can answer me

36. Kalid

Hi Kumar, glad you enjoyed it. Offhand, I don't think there's a way, looking at the raw data, to tell whether it's supposed to be a character or integer.

I think you'd need the file format spec to figure out the structure of the data — for example, the TCP header defines the byte ranges for each field.

37. joonas

hi,
actually i know nothing about the c-language, but i have a question.
i have a riddle. it is: 71+, OÄE
i have to get numbers of this riddle, that is, i have to "translate" this riddle somehow into numbers.
and i know, that it has something to do with the c or c++ language and also something with intel byte order...
can you please help me? i would be most thankful.

38. Kalid

Hi Joonas, I'm not exactly sure of the question — you have raw data and need to extract a value?

If you have 4 bytes (for example), I would examine it as a 4-byte integer, 4-byte unsigned integer, 2 shorts, and 4 characters (all in the big and little endian varieties). I'd then try to look for some pattern in the results.

39. hema

Very nice article! crisp and to the point.....

40. Kalid

    Thanks Hema, glad you liked it.

41. Apurva

    Ahh… Finally i got it.
    What an explanation!
    Concepts, Simplicity, Analogies, Humor and Writing style – first Question then Answer.
    Really, Thankx A Lot.

42. Kalid

    Awesome Apurva, glad you enjoyed it!

43. Shaoji

    Do you have any idea what machine uses which endian? For example, how about PC, big endian or little endian? Unix, big endian or little endia?

44. barn

    excellent job. the way you explained it is the way things should be explained in this world.

45. Basheer

    Simple and interesting presentation

46. Paul Dowd

    Thanks for an excellent explanation. Just one question: you only mention integers – what about floating point values? How are they converted between endianness?

47. Kalid

    @Shaoji: Good question, there is a partial list here: http://en.wikipedia.org/wiki/Endianness#Endianness_and_hardware. The endian issue is more with the processor (x86 Pentium) than the OS (Unix).

    @Barn: Thanks, glad you liked it.

@Basheer: Thanks!

@Paul: Appreciate the kind words. Great question — I believe you would treat a floating point number as a 4-byte data type (on a 32 bit machine). Between different endian machines you'd have to reverse the bytes before reading the float (not positive but that's my guess — data is data).

48. Kalid

Here's sample code for floating point / byte array conversion. We'll start with a number (3.14159) and flip the sign bit (more info on floating point).

```
#include <stdio.h>

int main(int argc, char** argv){
  char *a;
  float f = 3.14159;  // number to start with

  a = (char *)&f;   // point a to f's location

  // print float & byte array as hex
  printf("float: %f\n", f);
  printf("byte array: %hhX:%hhX:%hhX:%hhX\n", \
    a[0], a[1], a[2], a[3]);

  // toggle the sign of f -- using the byte array
  a[3] = ((unsigned int)a[3]) ^ 128;

  //print the numbers again
  printf("float: %f\n", f);
  printf("byte array: %hhX:%hhX:%hhX:%hhX\n", \
    a[0], a[1], a[2], a[3]);

  return 0;
}
```

And it outputs this on my machine:

```
float: 3.141590
```

```
byte array: D0:F:49:40
float: -3.141590
byte array: D0:F:49:C0
```

As you can see, we can access the floating point number as a byte array and change individual portions of it. This machine is little-endian. Notice how a[3], the highest byte containing the sign bit, is last. That means the little end, a[0], comes first.

Theoretically, on a big-endian machine the order of bytes would be reversed, but some sources say IEEE 754 floats are always stored little-endian. I don't have a big-endian machine to confirm this :).

49. Ajay

Keep posting...Very good article.

50. anil

How will following structure look in big-endian and little-endian systems?
struct STRUCT {
unsigned int a:12;
unsigned int b:10;
unsigned int c:10;
};
struct STRUCT myStruct= {0xFED, 0×345, 0×3AB};

51. Sharada Chanda

Why are sequence of characters not converted into network byte order when sending them to a remote UNIX server?
Example : – 'Article' is sent without converting to 'elcitrA'.

52. Ajish

Interesting Presentation!!
Just one question.. Whether the byte order conversion for a 8-byte variable is similar to a 4-byte variable?

53. pramod

Excellent Boss!!! thnaks a billon...for presenting such a complex thing in a very lucid manner.

54. Kalid

@anil: I'm not certain how structs are laid out on big/little endian systems (in the order variables are declared or otherwise). The easiest method may be to use the byte-walking method above.

@Sharada: Most network protocols should convert everything to "network order" when transmitting — it's probably a bug if this doesn't happen! (Or if the client & server agreed not to use the standard protocols).

@Ajish: Yes, I imagine 8-byte information is stored similarly (as the jump from 2 to 4, extrapolate from 4 to 8; it always pays to double check though!).

@pramod: Thanks, glad you enjoyed it.

55. Delron

Good article. I am having trouble talking to SD cards which are little-endian. I assume that, when sending the data serially, the LSB of byte 0 goes first.

56. Kalid

Thanks Delron. When reading a stream of data for something as modern as an SD card, I'd assume that it would be given to you in 1-byte chunks, so hopefully individual bits aren't an issue. (I.e. you could read/write 1 byte at a time — just my guess).

57. Delron

Thanks for your reply. It let me take another run at the problem. It turns out to be a good example of the confusion that can result in "endians". I see now that the SD card runs on the SPI protocol which is big-endian, however, the data that the big_endian SPI is transferring to the card uses the FAT file system which is little_enidian. I just have to remember that a byte is a byte, as you so correctly emphasize.

58. Kalid

@Delron: Wow, I thought I had it rough when having to tinker around with networking :). Yeah, it helped me to realize that the endian craziness stops at the byte level, and a byte's a byte.

59. Shanx

Hello guys wassup?

I have a doubt...earlier this guy said that to fix the endianness issue we can use a magical number before

every piece of data, n if the we read the magic number n he is 0xFEFF we have a data with the same format else if 0xFFFE we have to swap memory, but i don get the idea very well

So e.g. if i have a pixel (his order is RGBA) on a little endian machine i can use a rmask n make a AND? Someone can help me to understand better? ^^

60. Shanx

Kalid, hi man!

U can explain the thing about put a magic number before every data? I don get it very well =\

If i have a vector of pixels that i pick n my machine is little endian e.g., n the format of pixels is RGBA (bing endian), like this

0 X RR GG BB AA

what is the idea? How i put a magic number before this pixel n mak a test?

Thanks! ^^

61. Kalid

Hi Shanx, good question. The "magic number" is a special header everyone agrees to write before the data.

If you're reading data and the magic number appears backwards (0xFEFF instead of 0xFFFE), you know the data was stored in a different format and therefore you must byte-swap the data.

This is the strategy used in some Unicode documents — include a header field so people can tell when someone else wrote it differently. So, you just need to check whether the header == 0xFFFE (same byte order) or header == 0xFEFF (different byte order).

62. dada

awesome is the word...plesae try to include the solution to the problem in detailed!
thanks,.

63. mike

Awesome post! Thank you for writing this!

64. Kalid

    @mike: Thanks!

65. Michael

    Wow, that explained everything so thorougly. Other articles gave the basics, yet none gave you the practice.

    Thanks!

66. Kalid

    Thanks for the note Michael! Yes, I find it helps to cement understanding with a real example of what's going on.

67. Mark

    Excellent article! Very informative with great examples, and just enough light-heartedness thrown in to make it a fun read.

    I have read many other articles on endianess and somewhat understood the topic. But this was the first article I came away from feeling like I understand it well enough to explain it to someone else (the ultimate test IMHO as to whether you truly understand a topic).

    Thank you for taking the time to write this. I can honestly say this knowledge will help me in my career; which in turn will help my earning potential. So you should feel very good knowing you have made such an impact in someones life via your efforts. 

68. Kalid

    Thanks Mark, your note made my day :). I know what you mean, it took a while for me to really get why endian problem happened, and why only for multi-byte data (single bytes and the order of bits are a common denominator that all computers agree on, otherwise we'd have the "bit ordering problem" too).

    Regarding writing, I find it's no fun for the writer or reader when done in a stuffy style. Thanks again for the comment.

69. vikas

    thnks for this explaination I always had problem with big and little endian

70. Anonymous

    many thanks for your great post.

    but where should I set the commend of “setenv/export F_UFMTENDIAN=unit”? in the program, in the compile file or ?

71. Anonymous

    many thanks for your great post.

    but where should I set the command of “setenv/export F_UFMTENDIAN=unit”? in the program, in the compile file or ?

72. Faisal Shaikh

    Great post! It was really helpful.
    Thanks a lot!

73. vijay

    great job ….. thanks

74. salaz

    this is good man..even i can only understand 1/4 of it..this should be enough..cheers man.

75. Thura

    Thanks, very nice explanation …

76. Bob

    Excellent explanation! Much Better Explained than other sources I’ve seen. Well done!

77. Bob

    Excellent explanation! Much *Better Explained* than other sources I’ve seen. Well done!

78. Kalid

@salaz, Bob, Thura: Thanks!

79. chipvang

Cảm ơn rất nhiều! / Thank You very much! / Merci beaucoup! / Danke Sehr! / …
But I still *wonder* that in Little-Endian, 'UNIX' will be stored as 'XINU', not 'NUXI' (??). Because the *number* UNIX is a double-word (4-byte): HIGH-word is 'UN' (2-byte), LOW-word is 'IX' (also 2-byte). And due to Little-Endian, I have the LOW-word stored first, then the HIGH-word second –> 'IX', 'UN'. Next, in each WORD-number, still due to Little-Endian, the *number* IX (2-byte) will be stored in usual order: X I. Same thing for the 2-byte UN *number* : N U.
So, a 4-byte 'UNIX' will be stored (in memory -of course) as: X I N U (?!?!). Do I confuse something?
Hope to hear from You soon… ^_^

80. Kalid

@chipvang: You're welcome, gracias, danke!

If UNIX was stored as a double-word (4 bytes) then it would become XINU. But, the problem refers to storing UNIX as two shorts: UN and IX. So if you switch the endian on each short, it becomes NU and XI, or NUXI.

Great question — it confused me too.

81. truonglvx

thanks a lot.

82. samasat

Great article … detailed view right from Basics !

83. David Rogoff

Great article! However (there's always a however for engineers…), I'd like to suggest a couple of additions:

1) talk a bit about mixed-endian, like some ARM (and IBM?) processors have used. They can be BE for 16-bit values, but have two 16-bit values LE as a 32-bit read (or was that 32 and 64 – I still get confused).

2) Also, your explanations and example are all software/firmware based/targeted. For us hardware designers, mostly using Verilog, LE naturally makes sense and is assumed by the language. For example, if I declare a 32-bit vector and assign a value to it, it's naturally LE:

```
wire [31:0] my32bitregister;
assign my32bitregister = 32'h12_34_56_78;
```

In this case,
my32bitregister[7: 0] is 8'h78
my32bitregister[15: 8] is 8'h56
my32bitregister[23:16] is 8'h34
my32bitregister[31:24] is 8'h12

I think most of my big design headaches over the years have been byte/word swapping to convert endianness!

84. Amey

Hi
Your approach to the confusing topic is really simple.you made it easy to understand.
Thanks

85. Anonymous

pls tell me why the word address is jumping from 0 to 4..

86. Another anonymous guy

Hey,
thank you very much for that introduction. It is way nicer to read than the respective Wikipedia entry. One thing you may want to change are the pointer initialized to 0. It might be a little confusing because they're actually NULL pointers. You mention that, but why not just let the addresses start at 1 instead of 0?

Keep up the good work
JAAG

87. rajKumar

very nice article keep up the good work

88. Lakshminarayanan

I'm trying to interact from a little endian machine to a big endian – Motorola machine. I get data from the big endian machine in bytes. I never know the type of data packed in it. So I'm don't know if I'm receiving integer or shot int or whatever type.

How can I rightly interpret the data in the little endian machine?

89. Mani

Great Article, I could understand the byte ordering better than any other source of Information. I still have one question which i am not able to understand. Say we have a binary data (a JPEG Image) created in little endian machine and its transmitted to a big endian machine. If we compute a SHA1 hash of the file in both the machine do they give same result of they would give different hash.
If i read the file as byte by byte in base64 encoded format and hash them in both the machine. Please see i am not rendering the file, i am just interested in computing the hash of the file.

Mani

90. Kalid

@Lakshminarayanan: Unfortunately, you need to know the data types in the byte stream if you want to convert from big to little endian.

@Mani: Thanks. SHA1 looks at data as a stream of bytes, so big and little endian shouldn't matter. Also, file formats like JPEG usually have a standard format they agree on, so it will be the same file on both systems.

91. Harish

AUBE FUTI LY0L PLEX NEAI 000D. 

(BEAUTIFULLY EXPLAINED.)

KUDOS.

92. neeraj

really a gud one....

93. Johanes

great post, my lecturer couldn't explain as good as you do

94. rahul

Boy… You cleared my issues which I was struggling with for two hours! trying to read a big endian based data from a file into C…. Thanks…

95. Kalid

@Harish, Neeraj, Johanes, Rahul: Thanks!

96. David Whalen-Robinson

re. floats: If they are the same kind of IEEE float on your source machine and destination, then the bytes will be compatible, but the byte order may not be the same.
Unlike 8, 4, and 2 byte integers, floats do not have an official network order, but I think common practice is to treat the bytes like a 4 byte int when converting from host to network order.

97. David Whalen-Robinson

re. floats (part 2):
You can save yourself a lot of pain by looking at how the numbers are stored in memory. So if your number's bytes looks like 01 02 03 04 in memory, and your buffer from the network has 04 03 02 01, you know you are going to have to flip the order somehow.

98. Kalid

@David: Thanks for the details!

99. MP

Here's where the NUXI analogy falls apart for me. Consider (using a 16 bit value from Kalids example):

0×03 e7 (BigE)

0xe7 03 (LittleE)

When viewed from the context of the UNIX.. I have

U = 0
N = 3
I = E
X = 7
When viewed from a NUXI perspective I get:
N = 3
U = 0
X = 7
I = E

0×30 7E is not the same as 0xE7 03. Having a hard time drawing synergy between the two

00. Kalid

@MP: Hi there, in the original NUXI analogy, the data was stored as two shorts (so "UN" "IX" became "NU" "XI" -> each short was reversed). This example was picked, I believe, because it makes UNIX turn into another readable word.

In your example, the data is is stored as a single 4-byte integer, so as you notice it would transform UNIX to XINU, not UNIX to NUXI. Hope this helps!

01. Shah

Hi Kalid! I'm rather a noob at programming and I need your help in programming for a Cryptography problem.

How to come up with the pseudocode and program(using C++ or Java) for this problem:

In its binary form of Big-Endian byte-order format, create a program that computes "sig" for a given data from a file. Whereby

sig = SHA-1(m)^d mod N

where "m" is the byte stream formed by the content of the file; "d" and "N" are given as well.

I'm not sure of converting to binary form in computer language and whether this SHA-1 function is given in library.

Appreciate your help.

02. taz

First, thanks a lot!

I have a question. Can U just tell me if I get this right?
If I have a hexadecimal number 12A4382
and I want to represent it on BigE and LittleE
and addresses start from 1000, which byte would be on each address?
I think:
BigE:
01 on 1000
2A on 1001
43 on 1002
82 on 1003

LittleE:
82 on 1000
43 on 1001
2A on 1002
01 on 1003

is this right?

03. Murugesh

Nice article!

I have a doubt. The article says that the storage is same for little and big endian machines.

But when i check the following code in Intel Pentium 4, it displays as little endian:

(snip)

#include

void main()
{
short Var = 0×1234 ;
char *ptr ;

ptr = (char *) & Var;

if(*ptr == 0×12)
{
ptr++ ;
if(*ptr == 0×34)
{

printf(“\nBig Endian\n”) ;

```
}
}
else if( *ptr == 0×34)
{
ptr++ ;
if(*ptr == 0×12)
{
printf("\nLittle Endian\n") ;
}
}

}
```

(snip end)

My doubt is, when i see the memory, the value is stored in the order "34 12". That is, the order corresponding to little endian.

As per the article, it should have been stored as "12 34".

Pls clear my doubt.

Many thanks in advance.

04. Kalid

@Murugesh: Hi, Pentium chips are little endian, so you're getting the right answer! (It should be 34 12 in this case). Hope this helps.

05. Yahya

Thanks a ton Kalid for the post! I'm also amazed at you interaction with the responders for about three years now. Keep it up man

06. Letty

Hi Kalid!

I don't know what happened to my previous post. But anyways, I am looking at my compiler/architecture for my system and found that my architecture uses big endian. I found some compiler options for bit field alignment. I don't see anything for bit order though. Should I just assume that if Byte order is Big Endian that bit order of bit fields would also be Big Endian?

Letty

07. Marc

Great post

It was fun reading and even thou I already understand und manage the difference I read it to the end – that means a lot regarding me always in a hurry

08. antaeus

Thank you so much!
I am designing a program to load .ply file into my Visual studio environment, this article really helps~~~

09. Anonymous

Wow!!! Excellent explantion.. This is the best one I found when I searched.. I have had confusing ideas about the endianness before and now it is clear as crystal.

Thanks a million.. Keep it up!!

10. Kalid

@Anon: Thank you!

11. crescent

thanks to your post, now the endian issue is not a issue to me.

12. Suxw

thank you!
Now the endian is not a problem for me^-^

13. Muta Gosh

Hi there!
I have one question:" A single character like A is encoded in binery as 01000001. Could you please help provide me with a table that gives all the characters from A – Z in binery. This will truly help me in my desire to understanding Central Processing Units.

Thanks!

Muta

14. bobbriddly

That racist little dittie I learned as a child has a whole new meaning now.

one little, two little, three little-endians…

Thanks to the author for a fun and informative article.

You should be on Wiki!

15. Tasos

Very cool article.
Keep walking

16. Sanjeev Kumar

Very nice article. Very helpful.

17. AgnelCodes

One of the best articles I've ever read. Thanks a ton!

18. Amazed…

That was an AMAZING tutorial! I learned so much. I hope you write lots and lots more!

19. g

Excellent post!! Thank you very much

20. Kalid

@g: Glad it helped!

21. Arvind SJ

    Simply Superb!!! Cheers

22. Kalid

    @Arvind: Thanks!

23. Swapan

    Thanku for this excellent post . Worth it.

    Also, clear my one doubt. Ia64 architecture works as little endian as well as big endian. (referring to the wiki Architecture and OS link) ..

    1. So, how is this switching made .. Any pointers on this..

    2. suppose ia64 m/c was running in Little endian mode and in next boot, it was made big Endian. So, does the architecture take care of converting the byte order accordingly when reading from disk for data stored as little endian at the first place..`

    Appreciate ur help

24. Amit

    This is one of the excellent post about big endian and little endian I have ever seen. Awesome clarity ! Thanks a ton for sharing.I can now never forget differences between endian-ess and how to check.

    Great Job !

25. Pooja

    Answer for bit maniptulation question by Anil:

    http://www.naic.edu/~phil/notes/bitfieldStorage.html

    Awesome post!

26. Kalid

    @Amit: Thanks, glad it helped!

27. Smoke

    It should all be stored logically.

    No debate, if you're idea goes against logic, it's illogical, and thus, should never be implemented. /End Discussion

    How this crap ever survived is beyond me..

    Also, why the hell are there so many missing conversions, there are still tons of conversions you have to write yourself? The fuck have you ppl been doing for the last 20 years?

    Seriously, I should NOT be having to write fucking code to convert between endians, and bytes\hex\string, etc, etc,.. This shit has been around for ages, and should be part of every fucking IDE by default. I should never need to do conversions on standard types that are harder than using a simple cast, or conversion.. (ie, ToString(), etc,..).

    I'm sure I'll be bashed, and blah, but, this is truth, if it isn't simple, ditch it, if it defeats logic, don't use it, and if a data type has existed for 20+ years, then there damn well better be a standard converter built in that can change it to ANY TYPE it can legally be.

    /Rant

    Now, I'll spending the next ten hours trying to do something, that should by all rights take no more than one line of code, and two seconds to implement. (Thanks)

28. Rauf

    Great article.
    You made it much more easier for me to understand the subject.
    Thanks.

29. vaishnavi

    Little and Big Endian, char ch = 'a', how will this get stored in little and big endian
    Segments in memory, char *p, where will p get stored in memory

30. Ram

    Better explained.

31. Anton

    Awesome post – thanks a lot!!!

32. casi

    Hi, great post — well explained.

    One thing:

    “XML avoids this mess by storing data in UTF-8 by default, which stores Unicode information one byte at a time.”

    UTF-8 is multibyte also, otherwise it could not represent the over 1 million Unicode code points because singlebyte only offers 2^8 = 256 different possibilities.

    cheers

33. RAJA GOPAL MURALIDHARAN

    The Best technical article, I have ever seen .. Language is also awesome !! Nice work ! Thanks !

34. Lee

    Amongst everything I have read on this topic, your explanation was like a breath of fresh air. Good job!

35. Kalid

    @Lee: Thank you!

36. sid

    A very well written article! However, could you please explain to me why in the header netinet/ip.h, the two fields ip->ip_hl and ip->ip_v interchange postions in little endian and big endian machines. You mentioned that we have to worry about byte ordering only with multibyte data. These two fields are 4 bits each.
    Similarly for the flag bits in the tcp header, which are declared in reverse order for little endian machines. Kindly explain...

37. Kalid

    @Sid: Great question. Take a look here:

    http://cboard.cprogramming.com/c-programming/95463-bitfields-bit-little-endian.html

    Apparently, some compilers will pack bits differently based on the little or big endian (which they don't need to do — endian ordering is a byte-by-byte issue). In this case, this is a compiler optimization to have bits packed in the same order. Hope this helps.

38. Ying

    Great Post! Finally makes my mind clear about this issue. Big Thanks!

39. Kalid

    @Ying: You're welcome!

40. Ralph

    Great information. Thanks.

    One question: How is data exchanged between a 32-bit big-endian machine and a 64-bit little-endian machine? You briefly mentioned 32-bit formats and I am wonder how, and if, 64-bit formats are handled differently.

41. vegard

    thanks, helped me alot in understanding endianness

42. Kalid

    @Vegard: Glad it helped.

43. Kalid

    @Ralph: Great question. I think 32- and 64- bit systems would need to know how large integers, etc. are in order to communicate. The NUXI problem assumes 16-bit shorts (2-bytes each). I don't know if 64-bit systems are handled differently in a specific way — you'd just need to know that the expected data was 8 bytes (so you'd read 8 bytes forward or backward when you did the conversion).

44. fajar

    mangstap bro!! i like you more than the article >:) just kidding

45. Me

    Wow. So many years…

    Thanks Kalid.

46. Kalid

    @Me: Glad it helped

47. Venkat

    I need to write a program to read the following.

    00000F000000202020363600000000 10
    00000020202036370000000011000000
    20202036380000000012000000202020
    36390000000013000000202020373000
    00000014000000202020373100000000
    15000000202020373200000000160000
    00202020373300000000170000002020
    20373400000000000000000000000000

    This data contains some sequence of numbers like 1,2,3 etc.

    can you help me to sort it out.

48. FGreene

    hi but utf8 isnt singlebyte

49. Swapnil

    Very well written post .. Thanks ..

50. Arun

Thanks! nicely explained..

51. Johnathan

Thanks for the great work…nice article, very easy to understand.

52. Kalid

@ Johnathan: Thanks!

53. snash

grandiosa spiegazione, ora posso dire che qualcosina la so pure io ^^ … io ho un programma in ASCII e voglio tradurre in un linguaggio comprensibile che software devo usare?

54. Nitin Tripathi

Hi,

I have a requirement where I need to read a file with few columns (Not the entire file) in Little Endian format and convert them into decimal and load in Oracle.
It would be great if someone can help me with tool/utility/script so that I can try calling that through Informatica.

Thanks,
Nitin

55. Srihari Konakanchi

superb explanation. It was amazing how VMs (eg. Java Virtual Machine – JVM) take care of all this complexity internally)

Thanks
- srihari konakanchi

56. esimov

Concise and very readable explanation. However should be nice to implement a good way to test and convert the endian-nes of a custom data.

This may be a possibility to to determine it?

```
if (buf[4] === 0x0a && buf[5] === 0x0b && buf[6] === 0x0c &&
buf[7] === 0x0d) {
isLittleEndian = false;
```

57. kalid

    @esimov: It's difficult to tell the endianness of data unless there is some known format or header. I.e., unicode has a byte-order-mark [BOM] that can be tested: 0xFFFE or 0xFEFF. So, if your mutli-byte data is designed to be self-describing you can include a similar header and test it on each machine (if it reads with bytes flipped, you know the data is stored in the other format than your native one).

58. Robert M. Badea

    Thank you, sir! This article helped me very much.

59. kalid

    @Robert: Glad it helped!

60. Prathmesh

    Dude, the above article is superbly written … Flushed out my old thinking .. I will remember this for a long time "For a single byte, endianness does not matter "…. haaaha

61. kalid

    @Prathmesh: Awesome, glad you enjoyed it!

62. Anonymous

    this is called passion of explanation,incredible!!

63. kalid

    @Anonymous: Thanks, glad the excitement came through 

64. Lucy

    Great explanation; it was a big help; thank you!

65. kalid

    @Lucy: Thanks!

66. Nikhil Sebastian

    Detailed explanation and that too with such low-level details with examples… Really appreciate your effort. Thank you. It was a big help.

67. kalid

    @Nikhil: Glad you liked it!

68. Ashok

    Hi,
    it was a good explanation. I need help here.
    I tried to compile the program in big-endian format but i am getting error. Pls help.

    user@ubu:~/ashoka$ uname -a
    Linux ubu 2.6.32-41-generic #88-Ubuntu SMP Thu Mar 29 13:08:43 UTC 2012 i686 GNU/Linux
    user@ubu:~/ashoka$ gcc endian.c
    user@ubu:~/ashoka$ gcc -mbig-endian endian.c
    cc1: error: unrecognized command line option “-mbig-endian”
    user@ubu:~/ashoka$ gcc -mbig endian.c
    cc1: error: unrecognized command line option “-mbig”
    user@ubu:~/ashoka$ gcc -mwords-big-endian endian.c
    cc1: error: unrecognized command line option “-mwords-big-endian”
    user@ubu:~/ashoka$

    user@ubu:~/ashoka$ gcc -v
    Using built-in specs.
    Target: i486-linux-gnu
    Configured with: ../src/configure -v –with-pkgversion=’Ubuntu 4.4.3-4ubuntu5.1′ –with-bugurl=file:///usr/share/doc/gcc-4.4/README.Bugs –enable-languages=c,c++,fortran,objc,obj-c++ –prefix=/usr –enable-shared –enable-multiarch –enable-linker-build-id –with-system-zlib –libexecdir=/usr/lib –without-included-gettext –enable-threads=posix –with-gxx-include-dir=/usr/include/c++/4.4 –program-suffix=-4.4 –enable-nls –enable-clocale=gnu –enable-libstdcxx-debug –enable-plugin –enable-objc-gc –enable-targets=all –disable-werror –with-arch-32=i486 –with-tune=generic –enable-checking=release –build=i486-linux-gnu –host=i486-linux-gnu –target=i486-linux-gnu

    Thread model: posix

gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5.1)
user@ubu:~/ashoka$

69. itsme

Best post I've read for Endianness overall so far! Excellent, too, are the C-examples the network reference and the attempt (philosophy) to explain why. The best post to begin to dig deeper into, if someone wants. (It would perhaps be nice to mention more the hardware-register-issue, early(iest) PC's (not mainframes) didn't have a 16b, 32b or 64b register and the LSbit (and LSByte) is still (very) important these days). Thanks.

70. kalid

@itsme: Glad you liked it! Ah, I wasn't aware of the hardware-register issues, it's a bit beyond my current understanding :).

71. Anonymous

Awesome!!! i finally understand this stuff!!

72. jj

I am a little confused w/: "… the number must be 256 * byte 0 + byte 1, or 256*W + X, or 0×1234. I multiplied the first byte by 256 (2^8) because I needed to shift it over 8 bits." Please elaborate what you mean. I know let's say you have an integer array, then each address is an offset of 4B (assuming architecture stores integral data type as 32 bit so 4B). so how is this like what you're address arithmetic is? I'm confused.

73. kalid

@jj: No problem, happy to clarify. An integer array is a list of integers, but in this case we are going "inside" a single integer to figure out what its value should be.

Suppose our short integer has 2 bytes, W (byte 0) and X (byte 1). On a big-endian machine, we know the first byte we see (W, aka byte 0) is actually the "big end" of the number. In binary terms, we would write the bytes

WX

where W was "in front" of X, even though W has the smaller memory address. In order to get W "in front" of X, we shift it to the left 8 bits. This can be accomplished with `W << 8` [a left shift of 8 bits] or more simply `W * 256` [multiply by 256, which has the same effect].

We just add X because we don't need to shift its bits at all (on a big-endian machine, where X, the last byte we read, is in the last position).

Hope this helps!

74. Paris

Bravo Kalid! Best explanation I've seen on Endianess yet!

75. Vinit

Great post, and clarifies endianness (big / little) in a great way. I also like the way all queries have been clarified and answered.
I wish all posts had similar quality and approach. Keep up the good work.

76. kalid

@Vinit: Thanks!

77. kalid

@Paris: Awesome, glad you enjoyed it!

78. Anonymous

very much informative !!!

79. Ab

Awesome post! Thanks!!

80. Valentin Radu

Hi Kalid,

Although I'm usually on your site for the absolutely wonderful math articles, this one was a very good read as well. Thank you. There is one small thing: a byte doesn't always have 8 bits. Believe it or not

there are still some specialised embedded system that don't agree on that :). It's architecture depended.

8bits==octet.

81. Valer

Hi Kalid,
Your article really helped me to better understand Big and Little Endian memory storage. There still is one thing I don't understand: why do you need to shift the first byte for Big Endian by 8 bits when the first byte is already on address 0? I saw your reply @jj comment, but I'm still confused.
Thank you!

82. kalid

Hi Valer,

Happy to help. The shifting is needed when we're accessing bytes one at a time, and want to build up the actual number.

If I tell you "the first byte is 0×55, and the second byte is 0×88", how would you compute that number?

On a big-endian machine, you know the value should be 0×5588. Step by step, this is

1) Take the first byte: 0×55
2) Shift it over: 0×5500
3) Take the next byte: 0×88
4) Add it in: 0×5500 + 0×88 = 0×5588

If we are storing & retrieving bytes on a single machine, we don't need these complications. But if we are getting a stream of individual bytes, we need to capture them, shift, and add them up to get the real number back out.

83. Valer

Now I understand how it works. Thanks for your time!

84. Subi

i have a doubt.Suppose A supports bigendian format & B supports little endian format.Now i am sending data from A to B and data is read as mutibyte.Then in the network ww will convert data to network byte order using htonl().Here A is any way sending in network byte order format.so the function gives the same ouput.But the receiving end ie B receives it and read it in the reverse order.then how you said solution as adhering to a common format,ie Bigendian(network byte order)

85. codeMonkey_111

    Excellently written article, but typo:

    ” Or 10 in base 2, aka 2 in binary” should be: ” Or 10 in base 2, aka 2 in radix-10 (base 10, decimal etc)”

86. kalid

    @codeMonkey: Thanks, I just clarified that part.

87. IG

    Brilliant!! Keep up the good work.

88. akshay

    Thank you. Nicely explained the concept of Endianness.

89. chetan

    It’s of great help, thanks a lot…

90. Prashant Sawant

    One of the best article written

91. Nitin

    Loved the explanation… Why can’t all the teachers in the world be like you ?

92. kalid

    Thanks Nitin — I wish more teachers would take an intuition-first approach too!

93. Kiran Mathews

    Good work Kalid..

94. Correction

    61. Kalid got it the WRONG way!

    In unicode:

    A byte order mark (BOM) consists of the character code U+FEFF at the beginning of a data stream.

    So if you read a 16-bit word from the start of a document, if it starts with a BOM, you should get the number 0FEFFh.

    IF you instead get 0FFFEh, you have disagreement between the data producer and the data consumer. In this case, do a byte swap of every 16-bit word.

    for more, please read:

    http://www.unicode.org/faq/utf_bom.html
    Byte Order Mark (BOM) FAQ

95. SAIFUL ABU

    great post. Really helpful. Thank you very much

96. VINAY R

    Very well informed post,Thanks a lot.

97. Vimal

    Great post!!!
    I was trying to understand it since quite a while, this article has just enlightened me.

98. GM

    Very Nice. Thanks a lot

99. Anonymous

    I am very happy by posting 200th comment that The article is written very SUPERB… Might have taken lot's of precious time of author.

    Thanks a lot for helping me in crucial time of these Endieness…. 

200. Narmada

Could you please tell me how to convert number format to big endian or little endian

201. moon

could you tell me what could be the example for network programming which uses the byte order concept?

202. gabriel

re: #201 – Narmada (and anyone else who cares),

Check out “swab” (swa*p* b*ytes*); no need to reinvent the wheel, after all

from “man 3 swab”:
#include
void swab(const void *from, void *to, ssize_t n);

203. gabriel

heh, character entities… try this again
#include &ltunistd.h&gt

204. gabriel

#include <unistd.h>

205. Herbert Eberle

Under the question “Why Are There Endian Issues at All? Can’t We Just Get Along?” you answer: “Little-endian machines let you read the lowest-byte first, without reading the others. You can check whether a number is odd or even (last bit is 0)”. I am absolutely sure that under little-endian the even-odd bit is ”not” the last bit of the first byte – it is the ”first” bit of the first byte. The bits in the byte are as reverted as the bytes in the integer. You can find that out by shifting a register by less than 8 bits.

206. engineer

helpful

107. Marxy

Hey, many thanks for your post! it is really helpful!

108. unknown

This is the most understandable explanation of big and little endian byte order existing on the net. Just great and still helpful for many beginners like me! I wish everyone could teach this way it covers basics and doesn't stupidly assume you already know them.

109. Imran Khan

My Question z why computer use 0 & 1? Why its not any other numbers like 5 & 6 if the diff z only 1.?