Types of endianness

Asked 16 years, 4 months ago Modified 2 years, 8 months ago

Viewed 17k times

18

What is the difference between the following types of endianness?

- byte (8b) invariant big and little endianness
- half-word (16b) invariant big and little endianness
- word (32b) invariant big and little endianness
- double-word (64b) invariant big and little endianness

Are there other types/variations?

endianness

Share

Improve this question

Follow

edited Jun 17, 2009 at 15:39

starblue

56.7k ● 14 ● 99 ● 153

asked Aug 21, 2008 at 23:42

Ben Lever

2,073 ● 7 ● 26 ● 34

If your system has 128-bit data types, you of course have quadword big and little endianness. // This often arises in multiprocessor code, where you have instructions like Intel's

CMPXCHG16B: imagine having a LE x86 talking to a Big Endian I/O device. – Krazy Glew Sep 14, 2016 at 22:34

9 Answers

Sorted by: Highest score (default)

33

There are two approaches to endian mapping: *address invariance* and *data invariance*.

Address Invariance

In this type of mapping, the address of bytes is always preserved between big and little. This has the side effect of reversing the order of significance (most significant to least significant) of a particular datum (e.g. 2 or 4 byte word) and therefore the interpretation of data. Specifically, in little-endian, the interpretation of data is least-significant to most-significant bytes whilst in big-endian, the interpretation is most-significant to least-significant. In both cases, the set of bytes accessed remains the same.

Example

Address invariance (also known as *byte invariance*): the byte address is constant but byte significance is reversed.

```
Addr    Memory
        7    0
        |    |    (LE)   (BE)
        |----|
 +0     | aa |    lsb    msb
        |----|
 +1     | bb |     :      :
        |----|
  +2    | cc |      :       :
        |----|
  +3    | dd |     msb     lsb
        |----|
        |    |

At Addr=0:          Little-endian          Big-
endian
Read 1 byte:                 0xaa                 0xaa
(preserved)
Read 2 bytes:              0xbbaa               0xaabb
Read 4 bytes:       0xddccbbaa           0xaabbccdd
```

Data Invariance

In this type of mapping, the relative byte significance is preserved for datum of a particular size. There are therefore different types of data invariant endian mappings for different datum sizes. For example, a 32-bit word invariant endian mapping would be used for a datum size of 32. The effect of preserving the value of particular sized datum, is that the byte addresses of bytes within the datum are reversed between big and little endian mappings.

Example

32-bit data invariance (also known as *word invariance*): The datum is a 32-bit word which always has the value `0xddccbbaa`, independent of endianness. However, for accesses smaller than a word, the address of the bytes are reversed between big and little endian mappings.

```
Addr                Memory

            | +3   +2   +1   +0 |  <- LE
            |-------------------|
+0      msb | dd | cc | bb | aa |  lsb
            |-------------------|
+4      msb | 99 | 88 | 77 | 66 |  lsb
            |-------------------|
     BE ->  | +0   +1   +2   +3 |


At Addr=0:             Little-endian
Big-endian
Read 1 byte:                 0xaa
0xdd
Read 2 bytes:              0xbbaa
0xddcc
Read 4 bytes:          0xddccbbaa
0xddccbbaa   (preserved)
Read 8 bytes:  0x99887766ddccbbaa
0x99887766ddccbbaa   (preserved)
```

Example

16-bit data invariance (also known as *half-word invariance*): The datum is a 16-bit which always has the value `0xbbaa`, independent of endianness. However, for accesses smaller than a half-word, the address of the bytes are reversed between big and little endian mappings.

```
Addr           Memory

            | +1   +0 |  <- LE
            |---------|
+0      msb | bb | aa |  lsb
            |---------|
+2      msb | dd | cc |  lsb
            |---------|
+4      msb | 77 | 66 |  lsb
            |---------|
+6      msb | 99 | 88 |  lsb
            |---------|
     BE ->  | +0   +1 |


At Addr=0:              Little-endian
Big-endian
Read 1 byte:                  0xaa
0xbb
Read 2 bytes:               0xbbaa
0xbbaa   (preserved)
Read 4 bytes:           0xddccbbaa
0xddccbbaa   (preserved)
Read 8 bytes:  0x99887766ddccbbaa
0x99887766ddccbbaa   (preserved)
```

Example

64-bit data invariance (also known as *double-word invariance*): The datum is a 64-bit word which always has the value `0x99887766ddccbbaa`, independent of endianness. However, for accesses smaller than a double-word, the address of the bytes are reversed between big and little endian mappings.

```
Addr                             Memory

            | +7   +6   +5   +4   +3   +2   +1
+0 |  <- LE
            |----------------------------------------
--|
+0      msb | 99 | 88 | 77 | 66 | dd | cc | bb |
aa |  lsb
            |----------------------------------------
--|
     BE ->  | +0   +1   +2   +3   +4   +5   +6
+7 |
At Addr=0:             Little-endian
Big-endian
Read 1 byte:                 0xaa
0x99
Read 2 bytes:              0xbbaa
0x9988
Read 4 bytes:          0xddccbbaa
0x99887766
Read 8 bytes:  0x99887766ddccbbaa
0x99887766ddccbbaa   (preserved)
```

Share Improve this answer Follow

answered Sep 14, 2008 at 23:46
Ben Lever
2,073 ● 7 ● 26 ● 34

Is data invariance used in any major modern ISA? – Benni Feb 13, 2022 at 9:45

4

There's also middle or mixed endian. See wikipedia for details.

The only time I had to worry about this was when writing some networking code in C. Networking typically uses big-endian IIRC. Most languages either abstract the whole thing or offer libraries to guarantee that you're using the right endian-ness though.

Share Improve this answer Follow

answered Aug 21, 2008 at 23:57
Erik van Brakel
23.8k ● 3 ● 53 ● 66

4

Philibert said,

> bits were actually inverted

I doubt any architecture would break byte value invariance. The order of bit-fields may need inversion when mapping structs containing them against data. Such direct mapping relies on compiler specifics which are outside the C99 standard but which may still be common. Direct mapping is faster but does not comply with the C99 standard that does not stipulate packing, alignment and byte order. C99-compliant code should use slow mapping based on values rather than addresses. That is, instead of doing this,

```
#if LITTLE_ENDIAN
  struct breakdown_t {
    int least_significant_bit: 1;
    int middle_bits: 10;
    int most_significant_bits: 21;
  };
#elif BIG_ENDIAN
  struct breakdown_t {
    int most_significant_bits: 21;
    int middle_bits: 10;
    int least_significant_bit: 1;
  };
#else
  #error Huh
#endif

uint32_t data = ...;
struct breakdown_t *b = (struct breakdown_t
*)&data;
```

one should write this (and this is how the compiler would generate code anyways even for the above "direct mapping"),

```
uint32_t data = ...;
uint32_t least_significant_bit = data &
0x00000001;
uint32_t middle_bits = (data >> 1) & 0x000003FF;
uint32_t most_significant_bits = (data >> 11) &
0x001fffff;
```

The reason behind the need to invert the order of bit-fields in each endian-neutral, application-specific data storage unit is that compilers pack bit-fields into bytes of growing addresses.

The "order of bits" in each byte does not matter as the only way to extract them is by applying masks of values and by shifting to the the least-significant-bit or most-significant-bit direction. The "order of bits" issue would only become important in imaginary architectures with the notion of bit addresses. I believe all existing architectures hide this notion in hardware and provide only least vs. most significant bit extraction which is the notion based on the endian-neutral byte values.

Share Improve this answer

Follow

edited May 23, 2017 at 11:53

Community Bot

1 ● 1

answered Aug 27, 2010 at 4:29

eel ghEEz

1,215 ● 12 ● 25

It is easy to break byte value invariance. E.g. you have an 8-bit bus, data bits 0-7, and you connect them to a device or memory backwards - i.e. you connect Dev0.Bit0-Dev1.Bit7, Dev0.Bit1-Dev1.Bit6 ... Dev0.Bit7-Dev1.Bit0. // You may not be likely to encounter this on an entire system, but it CAN happen, and often does for a narrow region of memory, and so on. // I tend to use terms such a "bit endianness" and "byte endianness", to be precise about what is being discussed.
– Krazy Glew Sep 14, 2016 at 22:26

2

Best article I read about endianness "Understanding Big and Little Endian Byte Order".

Share Improve this answer

Follow

answered Aug 22, 2008 at 1:57

jfs

16.7k ● 13 ● 63 ● 90

1

Actually, I'd describe the endianness of a machine as the order of **bytes** inside of a word, and not the order of **bits**.

By "bytes" up there I mean the "smallest unit of memory the architecture can manage individually". So, if the smallest unit is 16 bits long (what in x86 would be called a *word*) then a 32 bit "word" representing the value 0xFFFF0000 could be stored like this:

```
FFFF 0000
```

or this:

```
0000 FFFF
```

in memory, depending on endianness.

So, if you have 8-bit endianness, it means that every word consisting of 16 bits, will be stored as:

```
FF 00
```

or:

```
00 FF
```

and so on.

Share Improve this answer Follow

answered Aug 22, 2008 at 0:39

dguaraglia
6,014 ● 1 ● 27 ● 23

Sorry, but this is very confused. Endianness is (almost) always either bit-wise, or more usually byte-wise, where a byte is the standard eight bits. Your `FFFF0000` example is badly chosen as it becomes `0000FFFF` if you reverse it bit-wise, byte-wise or word-wise, so it's not possible to tell what you're trying to say! – Scott Griffiths Oct 26, 2009 at 16:18

Yup, you are absolutely right, I was just feeling lazy, LOL. Thanks for pointing out though :) – dguaraglia Nov 15, 2009 at 22:18

I say "byte endianness" or "bit endianness" when I need to distinguish. – Krazy Glew Feb 19, 2014 at 2:05

1

Practically speaking, endianess refers to the way the processor will interpret the content of a given memory location. For example, if we have memory location 0x100 with the following content (hex bytes)

```
  0x100:  12 34 56 78 90 ab cd ef

Reads     Little Endian            Big Endian
 8-bit:   12                        12
16-bit:   34 12                     12 34
32-bit:   78 56 34 12               12 34 56 78
64-bit:   ef cd ab 90 78 56 34 12   12 34 56 78 90 
ab cd ef
```

The two situations where you need to mind endianess are with networking code and if you do down casting with pointers.

TCP/IP specifies that data on the wire should be big endian. If you transmit types other than byte arrays (like pointers to structures), you should make sure to use the ntoh/hton macros to ensure the data is sent big endian. If you send from a little-endian processor to a big-endian processor (or vice versa), the data will be garbled...

Casting issues:

```
uint32_t* lptr = 0x100;
uint16_t  data;
*lptr = 0x0000FFFF

data = *((uint16_t*)lptr);
```

What will be the value of data? On a big-endian system, it would be 0 On a little-endian system, it would be FFFF

Share Improve this answer

Follow

answered Sep 16, 2008 at 1:53

Benoit

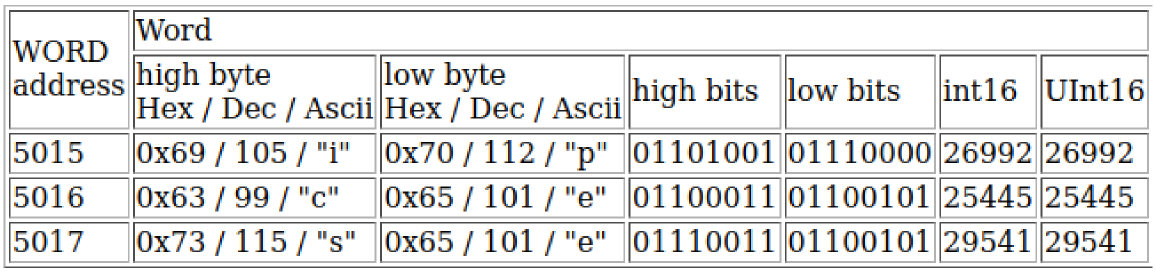
38.9k ● 24 ● 85 ● 117

1

As @erik-van-brakel answered on this post, be careful when communicating with certain PLC : **Mixed-endian still alive !**

Indeed, I need to communicate with a PLC (from a well known manufacturer) with (Modbus-TCP) OPC protocol and it seems that it returns me a mixed-endian on every half word. So it is still used by some of the larger manufacturers.

Here is an example with the "pieces" string :



WORD address	Word					
	high byte Hex / Dec / Ascii	low byte Hex / Dec / Ascii	high bits	low bits	int16	UInt16
5015	0x69 / 105 / "i"	0x70 / 112 / "p"	01101001	01110000	26992	26992
5016	0x63 / 99 / "c"	0x65 / 101 / "e"	01100011	01100101	25445	25445
5017	0x73 / 115 / "s"	0x65 / 101 / "e"	01110011	01100101	29541	29541

Share Improve this answer

Follow

answered Apr 5, 2022 at 7:44

Meloman

3,702 ● 3 ● 42 ● 52

I'm fairly certain that's just 16-bit little endian. If I had to guess the data originates from a word addressed machine. Since it doesn't support byte-level I/O it writes out the string as a series of 16-bit words, and since it's little endian the earliest character ends up in the low byte of each word.
– Michael Morris Aug 30 at 13:50

Well @michael-morris, I also use 16 booleans in one address for fault reporting; indeed, the 16 bits follow each other, you're right. However, when the PLC writes a 2-byte word, it reverses the bytes at the 8-bit level rather than at the 16-bit level. I could be mistaken, but I've developed a library that translates the different data types, and this approach works correctly with M340 Schneider Electrics PLC. – Meloman Sep 9 at 13:37

0

13 years ago I worked on a tool portable to both a DEC ALPHA system and a PC. On this DEC ALPHA the **bits were actually inverted**. That is:

```
1010 0011
```

actually translated to

```
1100 0101
```

It was almost transparent and seamless in the C code except that I had a bitfield declared like

```
typedef struct {
   int firstbit:1;
   int middlebits:10;
   int lastbits:21;
};
```

that needed to be translated to (using #ifdef conditional compiling)

```
typedef struct {
   int lastbits:21;
   int middlebits:10;
   int firstbit:1;
};
```

Share Improve this answer Follow

answered Sep 16, 2008 at 3:27

Philibert Perusse
4,096 ● 5 ● 26 ● 27

The order of assignment of bit-fields within a word is implementation defined. The first specified field may be in the MSB or LSB (most or least significant byte or bit, depending on which you want to look at). You found that out the hard way. Neither system is incorrect; the chosen implementations are simply different. – Jonathan Leffler Apr 19, 2015 at 6:05

-1

the basic concept is the ordering of bits:

```
1010 0011
```

in little-endian is the same as

```
0011 1010
```

in big-endian (and vice-versa).

You'll notice the order changes by grouping, not by individual bit. I don't know of a system, for example, where

```
1100 0101
```

would be the "other-endian" version of the first version.

Share Improve this answer Follow

answered Aug 21, 2008 at 23:49

James A. Rosen
65.2k ● 62 ● 184 ● 263

Typically endianness is byte-wise, but what you've described here is nibble-wise endianness (groups of 4 rather than 8 bits, which I doubt that anything uses). – Scott Griffiths Sep 9, 2009 at 20:41