Endianness

From Wikipedia, the free encyclopedia

In computing, **endianness** is the byte (and sometimes bit) ordering used to represent some kind of data. Typical cases are the order in which integer values are stored as bytes in computer memory (relative to a given memory addressing scheme) and the transmission order over a network or other medium. When specifically talking about bytes, endianness is also referred to simply as **byte order**. [1]

Generally speaking, endianness is a particular attribute of a representation format—which byte of a UCS-2 character would be stored at the lower address, etc. Byte order is an important consideration in network programming, since two computers with different byte orders may be communicating. Failure to account for varying endianness when writing code for mixed platforms can lead to bugs that can be difficult to detect.

Contents

- 1 Clarifying analogy
- 2 Endianness and hardware
  - 2.1 Bi-endian hardware
  - 2.2 Floating-point and endianness
- 3 Discussion, background, etymology
- 4 Method of mapping registers to memory locations
- 5 Examples of storing the value 0x0A0B0C0D in memory
  - 5.1 Big-endian
  - 5.2 Little-endian
  - 5.3 Middle-endian
- 6 Endianness in networking
- 7 Endianness in files and byte swap
- 8 "Bit endianness"
- 9 Other meanings
- 10 References and notes
- 11 Further reading
- 12 External links

Clarifying analogy

Endianness is the convention that two parties that wish to exchange information will use to send and receive this information when they need to cut the information down to pieces. Say Joe wants to send the word "SONAR" to his friend Moe across town. However, he can only do so on little cards that only fit three letters at a time. Since (for the most part) English uses big-endian order, Joe will first send *SON* and then *AR*. Moe needs to be using the same convention as Joe when receiving this information such that when he receives the first part (*SON*) he knows that this is the beginning of the word, then when he receives the other part (*AR*) he knows that it goes at the right hand (or little) end. If Moe is unaware and assumes the inverse, he ends up with the word "ARSON" and confusion ensues.

This same concept applies to computer applications which need to store all values into bytes (often breaking them apart and putting them back together). The application storing the values and the application reading the values need to be on the same page in terms of endianness.

Little-endian order is not unheard of in English, though, as in the example *four and twenty blackbirds*, with the lower order digit (four) coming first. Examples of *middle-endianness* include the U.S. date format (MM/DD/YY) or U.S. street addresses of the form 123 Any St., Suite 101, Yourtown, ST, USA.

Endianness and hardware

Most modern computer processors agree on bit ordering "inside" individual bytes (this was not always the case). This means that any single-byte value will be read the same on almost any computer one may send it to.

Integers are usually stored as sequences of bytes, so that the encoded value can be obtained by simple concatenation. The two most common of them are:

- increasing numeric significance with increasing memory addresses, known as *little-endian*, and
- its opposite, most-significant byte first, called *big-endian*.[2]

"Big-endian" as in "big end first".

Well known processor architectures that use the little-endian format include x86, 6502, Z80, VAX, and, largely, PDP-11. Processors using big-endian format are generally Motorola processors such as the 6800 and 68000 and PowerPC (which includes Apple's Macintosh line prior to the Intel switch) and System/370 also adopt big-endian. SPARC historically used big-endian, though version 9 is bi-endian (see below). The PDP-10 also uses big-endian addressing for byte oriented instructions. Network protocols are also generally in big-endian format, see further down this article.

Bi-endian hardware

Some architectures (including ARM, PowerPC except the PPC970/G5, Alpha, SPARC V9, MIPS, PA-RISC and IA64) feature switchable endianness. This feature can improve performance or simplify the logic of networking devices and software. The word *bi-endian*, said of hardware, denotes the capability to compute or pass data in either of two different endian formats.

Many of these architectures can be switched via software to default to a specific endian format (usually done when the computer starts up);

however, on some systems the default endianness is selected by hardware on the motherboard and cannot be changed via software (e.g., the DEC Alpha, which runs only in big-endian mode on the Cray T3E).

Note that "bi-endian" refers primarily to how a processor treats *data* accesses. *Instruction* accesses (fetches of instruction words) on a given processor may still assume a fixed endianness, even if *data* accesses are fully bi-endian.

Note, too, that some nominally bi-endian CPUs may actually employ internal "magic" (as opposed to really switching to a different endianness) in one of their operating modes. For instance, some PowerPC processors in little-endian mode act as little-endian from the point of view of the executing programs but they do not actually store data in memory in little-endian format (multi-byte values are swapped during memory load/store operations). This can cause problems when memory is transferred to an external device if some part of the software, e.g. a device driver, does not account for the situation.

Floating-point and endianness

On some machines, while integers were represented in little-endian form, floating-point numbers were represented in big-endian form. [3] Because there are many floating formats, and a lack of a standard "network" representation, no standard for transferring floating point values has been made. This means that floating point data written on one machine may not be readable on another, and this is the case even if both use IEEE 754 floating point arithmetic since the endian-ness of the memory representation is not part of the IEEE specification. [4]

Discussion, background, etymology

The term *big-endian* comes from Jonathan Swift's satirical novel *Gulliver's Travels*, where tensions are described in Lilliput and Blefuscu: whereas royal edict in Lilliput requires cracking open one's soft-boiled egg at the small end, inhabitants of the rival kingdom of Blefuscu crack theirs at the big end (giving them the moniker *Big-endians*).[3] The terms *little-endian* and *endianness* have a similar intent.[4]

The problem of dealing with data in different representations is sometimes termed the **NUXI problem**.[5] This terminology alludes to the issue that a value represented by the byte-string "UNIX" on a big-endian system may be stored as "NUXI" on a PDP-11 middle-endian system; UNIX was one of the first systems to allow the same code to run on, and transfer data between, platforms with different internal representations. (This is not however why the UNIX operating system is called UNIX)

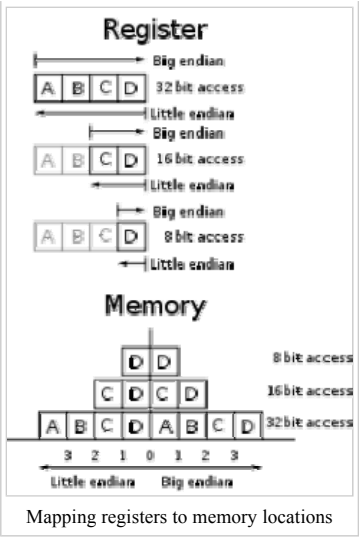
An often cited argument in favor of big-endian is that it is consistent with the ordering commonly used in natural languages;[6] that is, however, far from being universal, both in spoken and written form: spoken languages have a wide variety of organizations of numbers, and although numbers are nowadays written almost universally in the Hindu-Arabic numeral system, with the most significant digits written to the left, whether this is "big-endian" or "little-endian" depends on the direction of text flow in the writing system being used.

The little-endian system has the property that, in the absence of alignment restrictions, the same value can be read from memory at different lengths without using different addresses. For example, a 32-bit memory location with content 4A 00 00 00 can be read at the same address as either 8-bit (value = 4A), 16-bit (004A), or 32-bit (0000004A), all of which retain the same numeric value. (This example works only if the value makes sense in all three sizes, which means the value fits in just 8 bits.) This little-endian property is rarely exploited by programmers; it tends to be more used by modern compilers for higher level languages, and does not imply that little-endianness has any performance advantage in variable-width data access, as appropriate indirections are performed on big-endian platforms.

In some situations it may be useful to obtain an approximation of a value without reading its complete representation, by reading only its most-significant portion. With a big-endian representation, this approximation may be constructed from a *prefix* of the representation, whereas with a little-endian representation it is constructed from a *suffix* of the representation. Conversely, a *suffix* of a big-endian representation or a *prefix* of a little-endian representation conveys little about the overall magnitude of the value, only some of its least-significant bits.

One reason little-endian representation probably found such favor in early (small-scale) byte-addressable computers in spite of the more human-oriented naturality of a big-endian representation is that little-endian multi-byte addition may be carried out with a monotonic incrementing address sequence. That is, if a bytewide data bus processor attempted to add two sixteen-bit numbers, only incrementation of the addresses need be done, an operation that was already present in hardware. (Carry bits propagate from least-significant to most-significant bytes.) A big-endian processor's more complex bus addressing unit would have to be told how big the addition was going to be so that it could hop forward to the least significant byte, then count back down towards the most, instead of performing the operation atomically (which all modern systems do). This kind of extra complexity in such early hardware was relatively expensive and slow.

Method of mapping registers to memory locations



Mapping registers to memory locations

Using this chart, one can map an access (or, for a concrete example: "write 32 bits to address 0") from register to memory or from memory to register. To help in understanding that access, little and big endianness can be seen in the diagram as differing in their coordinate system's orientation. Big endianness's atomic units and memory coordinate system increases in the diagram from left to right, while little endianness's units increase from right to left.

Examples of storing the value `0x0A0B0C0D` in memory

> *Note: the prefix `0x` indicates hexadecimal notation.*

To further illustrate the above notions this section provides example layouts of a 32-bit number in the most common variants of endianness. There is no general guarantee that a platform will use one of these formats but in practice there are few if any exceptions.

All the examples refer to the storage in memory of the value `0x0A0B0C0D`.

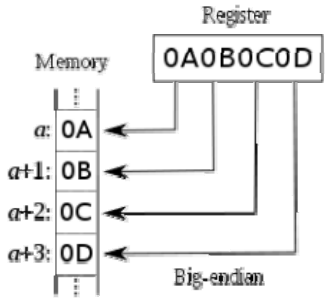
Big-endian

- *With 8-bit atomic element size and 1-byte (octet) address increment*:

increasing addresses →

...	0x0A	0x0B	0x0C	0x0D	...

The most significant byte (*MSB*) value, which is `0x0A` in our example, is stored at the memory location with the lowest address, the next byte value in significance, `0x0B`, is stored at the following memory location and so on. This is akin to Left-to-Right reading order in hexadecimal.



- *With 16-bit atomic element size*:

increasing addresses →

...	0x0A0B	0x0C0D	...

The most significant atomic element stores now the value `0x0A0B`, followed by `0x0C0D`.

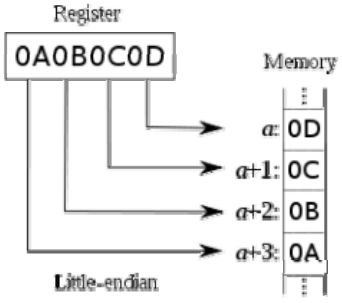
Little-endian

- *With 8-bit atomic element size and 1-byte (octet) address increment*:

increasing addresses →

...	0x0D	0x0C	0x0B	0x0A	...

The least significant byte (*LSB*) value, `0x0D`, is at the lowest address. The other bytes follow in increasing order of significance.



- *With 16-bit atomic element size*:

increasing addresses →

...	0x0C0D	0x0A0B	...

The least significant 16-bit unit stores the value `0x0C0D`, immediately followed by `0x0A0B`.

- *With byte addresses increasing from right to left*:

The 16-bit atomic element byte ordering may look backwards as written above, but this is because little-endian is best written with addressing increasing towards the left. If we write the bytes this way then the ordering makes slightly more sense:

← *increasing addresses*

...	0x0A	0x0B	0x0C	0x0D	...

The least significant byte (*LSB*) value, `0x0D`, is at the lowest address. The other bytes follow in increasing order of significance.

← *increasing addresses*

...	0x0A0B	0x0C0D	...

The least significant 16-bit unit stores the value `0x0C0D`, immediately followed by `0x0A0B`.

However, if one displays memory with addresses increasing to the left like this, then the display of Unicode (or ASCII) text is reversed from the normal display (for left-to-right languages). For example, the word "XRAY" displayed in the "little-endian-friendly" manner just described is:

← *increasing addresses*

...	"Y"	"A"	"R"	"X"	...

This conflict between the memory arrangements of binary data and text is intrinsic to the nature of the little-endian convention, but is a conflict only for languages written left-to-right (such as Indo-European languages like English(Roman), French(Roman), Russian(Cyrillic) and Hindi (Devanagari)). For right-to-left languages such as Arabic or Hebrew, there is no conflict of text with binary, and the preferred display in both cases would be with addresses increasing to the left. (On the other hand, right-to-left languages have a complementary intrinsic conflict in the big-endian system.)

Middle-endian

Still other architectures, generically called *middle-endian* or *mixed-endian*, may have a more complicated ordering; PDP-11, for instance, stored some 32-bit words, counting from the most significant, as: 2nd byte first, then 1st, then 4th, and finally 3rd.

- *storage of a 32-bit word on a PDP-11*

increasing addresses →

...	0x0B	0x0A	0x0D	0x0C	...

Note that this can be interpreted as storing the most significant "half" (16-bits) followed by the less significant half (as if big-endian) but with each half stored in little-endian format. This ordering is known as *PDP-endianness*.

The ARM architecture can also produce this format when writing a 32-bit word to an address 2 bytes from a 32-bit word alignment.

Endianness in networking

Networks generally use big-endian order, and thus it is called **network order** when sending information over a network in a common format. The telephone network, historically and presently, uses a big-endian order; doing so allows routing while a telephone number is being composed. In fact, the Internet Protocol defines a standard big-endian *network byte order*. This byte order is used for all numeric values in the packet headers and by many higher level protocols and file formats that are designed for use over IP. The Berkeley sockets API defines a set of functions to convert 16- and 32-bit integers to and from network byte order: the `htonl` (host-to-network-long) and `htons` (host-to-network-short) functions convert 32-bit and 16-bit values respectively from machine (*host*) to network order; whereas the `ntohl` and `ntohs` functions convert from network to host order.

While the lowest network protocols may deal with sub-byte formatting, all the layers above them usually consider the *byte* (mostly meant as *octet*) as their atomic unit.

Endianness in files and byte swap

Endianness is a problem when a binary file created on a computer is read on another computer with different endian. Some compilers have built-in facilities to deal with data written in other endians, for example the Intel Fortran compiler supports the non-standard `CONVERT` specifier, so a file can be opened as

```
OPEN(unit,CONVERT='BIG_ENDIAN',...)
```

or

```
OPEN(unit,CONVERT='LITTLE_ENDIAN',...)
```

If the compiler does not support such conversion, the programmer needs to swap the bytes by an ad hoc code.

Fortran sequential unformatted files created with one endian usually cannot be even read on a system using the other endian because Fortran usually implements a record (defined as the data written by a single Fortran statement) as data preceded and succeeded by count fields, which are integers equal to the number of bytes in the data. An attempt to read such file on the other endian system then results in a run-time error, because the count fields are incorrect.

Application binary data formats, such as for example Matlab .mat files, or the .BIL data format, used in topography, are usually endian-independent. This is achieved by storing the data always in one fixed endian, or carrying with the data a switch to indicate which endian the data was written with. When reading the file, the application converts the endian, transparently to the user.

Note that since the required byte swap depends on the length of the variables stored in the file (two 2 byte integers require a different swap than one 4 byte integer), a general utility to convert endian in binary files cannot exist.

"Bit endianness"

The terms *bit endianness* or *bit-level endianness* are seldom used when talking about the representation of a stored value, as they are only meaningful for the rare computer architectures which support addressing of individual bits. They are used however to refer to the transmission order of bits over a serial medium. Most often that order is transparently managed by the hardware and is the bit-level analogue of little-endian (low-bit first), although protocols exist which require the opposite ordering (e.g. I²C). In networking, the decision about the order of transmission of bits is made in the very bottom of the data link layer of the OSI model.

Other meanings

Some authors extend the usage of the word "endianness", and of related terms, to entities such as street addresses, date formats and others. It should be noted however that such usages—basically reducing *endianness* to a mere synonym of *ordering of the parts*—are non-standard usage (e.g., ISO 8601:2004 talks about "descending order year-month-day", not about "big-endian format"), do not have widespread usage, and are generally (other than for date formats) employed in a metaphorical sense.

"Endianness" is sometimes used to describe the order of the components of an internet site address, e.g. 'www.wikipedia.org' (the usual modern 'little-endian' form) versus the reverse-DNS 'org.wikipedia.www' ('big-endian', used for naming components, packages, or types in computer systems, for example Java packages, Macintosh ".plist" files, etc.). Also file paths: Linux `"~user/My_documents"`, Windows `"C:\Documents and Settings\user\My documents"`

References and notes

1. ^ For hardware, the Jargon File also reports the less common expression *byte sex* [1]. It is unclear whether this terminology is also used when more than two orderings are possible. Similarly, the manual for the ORCA/M assembler refers to a field indicating the order of the bytes in a number field as `NUMSEX`, and the Mac OS X operating system refers to "byte sex" in its compiler tools [2].
2. ^ Note that, in these expressions, the term "end" is meant as "extremity", not as "last part"; and that *big* and *little* say which extremity is written *first*.
3. ^ Jonathan Swift (1726). *Gulliver's Travels*. http://en.wikisource.org/wiki/Gulliver%27s_Travels/Part_I/Chapter_IV. "Which two mighty powers have, as I was going to tell you, been engaged in a most obstinate war for six-and-thirty moons past. (...) the primitive way of breaking eggs, before we eat them, was upon the larger end; (...) the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. (...) Many hundred large volumes have been published upon this controversy: but the books of the Big-endians have been long forbidden (...)"
4. ^ David Cary. "Endian FAQ". http://www.rdrop.com/~cary/html/endian_faq.html. Retrieved on 2008-12-20.
5. ^ "NUXI problem". *The Jargon File*. http://catb.org/jargon/html/N/NUXI-problem.html. Retrieved on 2008-12-20.
6. ^ *Cf.* entries 539 and 704 of the Linguistic Universals Database

Further reading

- Danny Cohen (1980-04-01). "On Holy Wars and a Plea for Peace". *Internet Experiment Note 137*. http://www.ietf.org/rfc/ien/ien137.txt. Retrieved on 2008-12-20. — also published at *IEEE Computer*, October 1981 issue.
- David V. James (June 1990). "Multiplexed buses: the endian wars continue". *IEEE Micro* **10** (3): 9–21. doi:10.1109/40.56322. ISSN 0272-1732. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=56322. Retrieved on 2008-12-20.
- Bertrand Blanc, Bob Maaraoui (December 2005). "*Endianness or Where is Byte 0?*". Retrieved on 2008-12-21.

External links

- Understanding big and little endian byte order
- The Layout of Data in Memory
- Byte Ordering PPC
- Writing endian-independent code in C
- How to convert an integer to little endian or big endian

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.