Endianness

From Wikipedia, the free encyclopedia

In computing, **endian** and **endianness** in the most common cases, refers to how bytes are ordered within computer memory. Computer memory is organized just the same as words on the page of a book or magazine, with the first words located in the upper left corner and the last in the lower right corner. The same is also true in mathematics or computer programming when diagramming matrices or arrays on a page; with the first element in the upper left corner and the last in the lower right corner, with the indexing of elements being described in similar terms in math and computer programming.

Big endian systems are systems whose memory is organized with the most significant digits or bytes of a number or series of numbers in the upper left corner of a memory page and the least significant in the lower right, just as in a normal spreadsheet. In contrast, little endian systems are organized with the least significant digits or bytes of a number or series of numbers in the upper left corner of a memory page and the most significant in the lower right. There are many examples of both types of systems, with the principal reasons for implementing either format being the underlying operation of the given system. Mixed forms are possible, for instance the ordering of bytes within a 16-bit word may differ from the ordering of 16-bit words within a 32-bit word. Such cases are rare and are sometimes referred to as **mixed-endian** or **middle-endian**.

Endianness is important as a low-level attribute of a particular data format. Failure to account for varying endianness across architectures when writing code for mixed platforms can lead to failures and bugs. The term *big-endian* originally comes from Jonathan Swift's satirical novel *Gulliver's Travels* by way of Danny Cohen in 1980.[1]

Contents

Etymology

In 1726, Jonathan Swift described in his satirical novel *Gulliver's Travels* tensions 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*).[2][1] The terms *little-endian* and *endianness* have a similar intent.[3]

Danny Cohen's "On Holy Wars and a Plea for Peace" published in 1980[1] ends with: "Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way. We agree that the difference between sending eggs with the little- or the big-end first is trivial, but we insist that everyone must do it in the same way, to avoid anarchy. Since the difference is trivial we may choose either way, but a decision must be made."

This trivial difference was the reason for a hundred-years war between the fictional kingdoms. It is widely assumed that Swift was either alluding to the historic War of the Roses or — more likely — parodying through oversimplification the religious discord in England and Scotland brought about by the conflicts between the Roman Catholics (Big Endians) on the one side and the Anglicans and Presbyterians (Little Endians) on the other.

History

Historically, byte order distinction was born out of mainframe vs. microprocessor approach. Until 1970s virtually all processors were big-endian. The introduction of microprocessors using initially simpler logic and byte level computations led to the little-endian approach.[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.

Endianness and hardware

Computer memory consists of a sequence of cells, usually bytes, and each cell has a number called its address that programs use to refer to it. If total number of bytes in memory is `n` then bytes addresses would be enumerated `0` to `n-1`. Multi-byte CPU registers are stored in memory as a simple concatenation of bytes. The simple forms are:[6]

- increasing numeric significance with increasing memory addresses (or increasing time), known as *little-endian*, and
- decreasing numeric significance with increasing memory addresses (or increasing time), known as *big-endian*[7]

The Intel x86 and x86-64 series of processors use the little-endian format, and for this reason, the little-endian format is also known as the **Intel convention**. Other well-known little-endian processor architectures are the 6502 (including 65802, 65C816), Z80 (including Z180, eZ80 etc.), MCS-48, 8051, DEC Alpha, Altera Nios II, Atmel AVR, SuperH, VAX, and, largely, PDP-11.

The Motorola 6800 and 68k series of processors use the big-endian format, and for this reason, the

big-endian format is also known as the **Motorola convention**. Other well-known processors that use the big-endian format include the Xilinx Microblaze, IBM POWER, and System/360 and its successors such as System/370, ESA/390, and z/Architecture. The PDP-10 also used big-endian addressing for byte-oriented instructions.

SPARC historically used big-endian until version 9, which is bi-endian, similarly the ARM architecture was little-endian before version 3 when it became bi-endian, and the PowerPC and Power Architecture descendants of POWER are also bi-endian (see below).

Bi-endian hardware

Some architectures (including ARM versions 3 and above, PowerPC, Alpha, SPARC V9, MIPS, PA-RISC, SuperH SH-4 and IA-64) feature a setting which allows for switchable endianness in data segments, code segments or both. This feature can improve performance or simplify the logic of networking devices and software. The word *bi-endian* or *bytesexual*, when said of hardware, denotes the capability of the machine to compute or pass data in either endian format.

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 Alpha, which runs only in big-endian mode on the Cray T3E).

Note that the term "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, though this is not always the case, such as on Intel's IA-64-based Itanium CPU, which allows both.

Note, too, that some nominally bi-endian CPUs require motherboard help to fully switch endianness. For instance, the 32-bit desktop-oriented PowerPC processors in little-endian mode act as little-endian from the point of view of the executing programs but they require the motherboard to perform a 64-bit swap across all 8 byte lanes to ensure that the little-endian view of things will apply to I/O devices. In the absence of this unusual motherboard hardware, device driver software must write to different addresses to undo the incomplete transformation and also must perform a normal byte swap.

Some CPUs, such as many PowerPC processors intended for embedded use, allow per-page choice of endianness.

Floating-point and endianness

Although the ubiquitous x86 of today use little-endian storage for all types of data (integer, floating point, BCD), there have been a few historical machines where floating point numbers were represented in big-endian form while integers were represented in little-endian form.[8] There are old ARM processors that have half little-endian, half big-endian floating point representation. Because there have been many floating point formats with no "network" standard representation for them, there is no formal standard for transferring floating point values between heterogeneous systems. It may therefore appear strange that the widespread IEEE 754 floating point standard does not specify endianness.[9] Theoretically, this means that even standard IEEE floating point data written by one machine might not be readable by another. However, on modern standard computers (i.e., implementing IEEE 754), one may in practice safely assume that the endianness is the same for floating point numbers as for integers, making the conversion straightforward regardless of data type. (Small embedded systems using special floating point formats may be another matter however.)

Optimization

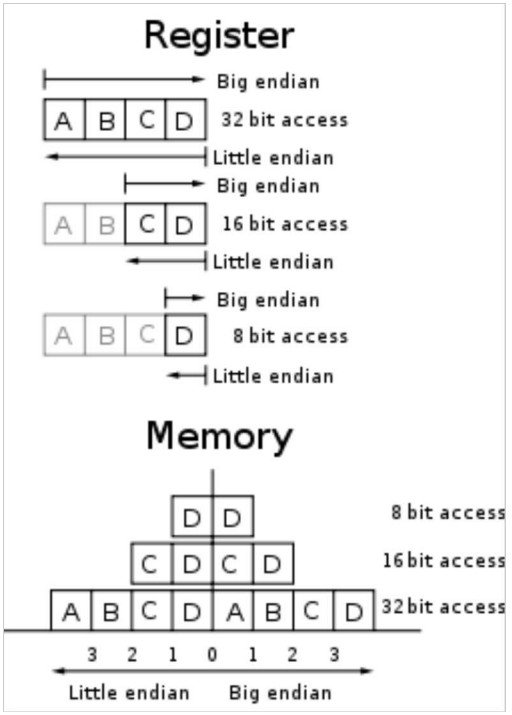
The little-endian system has the property that the same value can be read from memory at different lengths without using different addresses (even when alignment restrictions are imposed). For example, a 32-bit memory location with content 4A 00 00 00 can be read at the same address as either 8-bit (value = 4A), 16-bit (004A), 24-bit (00004A), or 32-bit (0000004A), all of which retain the same numeric value. Although this little-endian property is rarely used directly by high-level programmers, it is often employed by code optimizers as well as by assembly language programmers.

On the other hand, in some situations it may be useful to obtain an approximation of a multi-byte or multi-word value by reading only its most-significant portion instead of the complete representation; a big-endian processor may read such an approximation using the same base-address that would be used for the full value.

Calculation order

Little-endian representation simplifies hardware in processors that add multi-byte integral values a byte at a time, such as small-scale byte-addressable processors and microcontrollers. As carry propagation must start at the least significant bit (and thus byte), multi-byte addition can then be carried out with a monotonic incrementing address sequence, a simple operation already present in hardware. On a big-endian processor, its addressing unit has to be told how big the addition is going to be so that it can hop forward to the least significant byte, then count back down towards the most significant. However, high performance processors usually perform these operations as a single operation, fetching multi-byte operands from memory in a single operation, so that the complexity of the hardware is not affected by the byte ordering.

Diagram for 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 bit value 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 (in this example the atomic unit is the byte) and memory coordinate system increases in the diagram from left to right, while little endianness's units increase from right to left.

A simple way to remember is "In **L**ittle Endian, The **L**east significant byte goes into the **L**owest value slot".

So in the above example, D, the least significant byte, goes into slot 0.

If you are writing in a western language the hex value 0x0a0b0c0d you are writing the bytes from *left to right*, you are implicitly writing Big-Endian style. 0x0a at 0, 0x0b at 1, 0x0c at 2, 0x0d at 3. On the other hand the output of memory is normally also printed out bytewise from left to right, first memory address 0, then memory address 1, then memory address 2, then memory address 3. So on a Big-Endian system when you write a 32-bit value (from a register) to an address in memory and after that output the memory, you "see what you have written" (because you are using the left to right coordinate system for the output of values in registers as well as the output of memory). However on a Little-Endian system the logical 0 address of a value in a register (for 8-bit, 16-bit and 32-bit) is the *least significant byte*, the one to the right. 0x0d at 0, 0x0c at 1, 0x0b at 2, 0x0a at 3. If you write a 32 bit register value to a memory location on a Little-Endian system and after that output the memory location (with growing addresses from left to right), then the output of the memory will appear *reversed* (byte-swapped). You have 2 choices now to synchronize the output of what you are seeing as values in registers and what you are seeing as memory: You can swap the output of the register values (0x0a0b0c0d => 0x0d0c0b0a) or you can swap the output of the memory (print from right to left). Because the values of registers are interpreted as numbers, which are, in western languages, written from left to right, it is natural to use the second approach, to display the memory from right to left. The above diagram does exactly that, when visualizing memory (when "thinking memory") on a Little-Endian system the memory should be seen growing to the *left*.

Examples of storing the value $\mathtt{0A0B0C0D}_{h}$ in memory

Note that hexadecimal notation is used.

To illustrate the notions this section provides example layouts of the 32-bit number $\mathtt{0A0B0C0D}_{h}$ in the most common variants of endianness. There exist several digital processors that use other formats, but these two are the most common in general processors. That is true for typical embedded systems as well as for general computer CPU(s). Most processors used in non CPU roles in typical computers (in storage units, peripherals etc.) also use one of these two basic formats, although not always 32-bit of course.

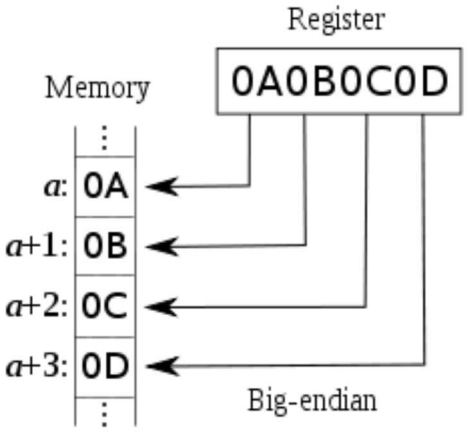
All the examples refer to the storage in memory of the value.

Big-endian

Atomic element size 8-bit, address increment 1-byte (octet)

increasing addresses →

…	$\mathtt{0A}_{h}$	$\mathtt{0B}_{h}$	$\mathtt{0C}_{h}$	$\mathtt{0D}_{h}$	…



The most significant byte (*MSB*) value, which is $\mathtt{0A}_{h}$ in our example, is stored at the memory location with the lowest address, the next byte value in significance, $\mathtt{0B}_{h}$, is stored at the following memory location and so on. This is akin to Left-to-Right reading in hexadecimal order.

Atomic element size 16-bit

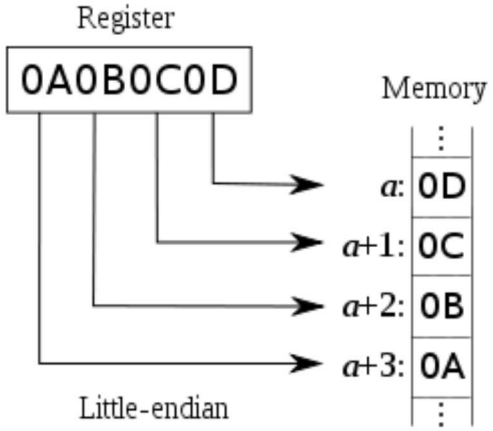
increasing addresses →

…	$0A0B_h$	$0C0D_h$	…

The most significant atomic element stores now the value $0A0B_h$, followed by $0C0D_h$.

Little-endian

Atomic element size 8-bit, address increment 1-byte (octet)



increasing addresses →

…	$0D_h$	$0C_h$	$0B_h$	$0A_h$	…

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

Atomic element size 16-bit

increasing addresses →

…	$0C0D_h$	$0A0B_h$	…

The least significant 16-bit unit stores the value $0C0D_h$, immediately followed by $0A0B_h$. Note that $0C0D_h$ and $0A0B_h$ represent integers, not bit layouts (see bit numbering).

Byte addresses increasing from right to left

Visualising memory addresses from left to right makes little-endian values appear backwards. If the addresses are written increasing *towards* the left instead, each individual little-endian value will appear forwards. However strings of values or characters appear reversed instead.

With 8-bit atomic elements:

← *increasing addresses*

…	$0A_h$	$0B_h$	$0C_h$	$0D_h$	…

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

With 16-bit atomic elements:

← *increasing addresses*

…	$0A0B_h$	$0C0D_h$	…

The least significant 16-bit unit stores the value $0C0D_h$, immediately followed by $0A0B_h$.

The display of text is reversed from the normal display of languages such as English that read from left to right. For example, the word "XRAY" displayed in this manner, with each character stored in an 8-bit atomic

element:

← *increasing addresses*					
…	"Y"	"A"	"R"	"X"	…

If pairs of characters are stored in 16-bit atomic elements (using 8 bits per character), it could look even stranger:

← *increasing addresses*			
…	"AY"	"XR"	…

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 English. For right-to-left languages such as Arabic and 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

Numerous other orderings, generically called *middle-endian* or *mixed-endian*, are possible. On the PDP-11 (16-bit little-endian) for example, the compiler stored 32-bit values with the 16-bit halves swapped from the expected little-endian order. This ordering is known as *PDP-endian*.

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

increasing addresses →					
…	$0B_h$	$0A_h$	$0D_h$	$0C_h$	…

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

Segment descriptors on Intel 80386 and compatible processors keep a base 32-bit address of the segment stored in little-endian order, but in four nonconsecutive bytes, at relative positions 2,3,4 and 7 of the descriptor start.

Endianness in networking

Many IETF RFCs use the term *network order*, meaning the order of transmission for bits and bytes *over the wire* in network protocols. Among others, the historic RFC 1700 (also known as Internet standard STD 2) has defined its network order to be big endian, though not all protocols do.[10]

The telephone network, historically and presently, sends the most significant part first, the area code; doing so allows routing while a telephone number is being composed.

The Internet Protocol defines big-endian as the standard *network byte order* 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-bit and 32-bit integers to and from network byte order: the `htons` (host-to-network-short) and `htonl` (host-to-network-long) functions convert 16-bit and 32-bit values respectively from machine (*host*) to network order; the `ntohs` and `ntohl` functions convert from network to host order. These functions may be a no-op on a big-endian system.

In CANopen multi-byte parameters are always sent least significant byte first (little endian). The same is true for Ethernet Powerlink.[11]

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 endianness. Some compilers have built-in facilities to deal with data written in other formats. 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',...)
```

Some compilers have options to generate code that globally enables the conversion for all file IO operations. This allows one to reuse code on a system with the opposite endianness without having to modify the code itself. If the compiler does not support such conversion, the programmer needs to swap the bytes via ad hoc code.

Fortran sequential unformatted files created with one endianness usually cannot be read on a system using the other endianness 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 a system of the other endianness then results in a run-time error, because the count fields are incorrect. This problem can be avoided by writing out sequential binary files as opposed to sequential unformatted.

Unicode text can optionally start with a byte order mark (BOM) to signal the endianness of the file or stream. Its code point is U+FEFF. In UTF-32 for example, a big-endian file should start with `00 00 FE FF`. In a little-endian file these bytes are reversed.

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

This is the case of TIFF image files, which instructs in its header about endianness of their internal binary integers. If a file starts with the signature "`MM`" it means that integers are represented as big-endian, while "`II`" means little-endian. Those signatures need a single 16-bit word each, and they are palindromes (that is, they read the same forwards and backwards), so they are endianness independent. "`I`" stands for Intel and "`M`" stands for Motorola, the respective CPU providers of the IBM PC compatibles (Intel) and Apple Macintosh platforms (Motorola) in the 1980s. Intel CPUs are little-endian, while Motorola 680x0 CPUs are big-endian. This explicit signature allows a TIFF reader program to swap bytes if necessary when a given file was generated by a TIFF writer program running on a computer with a different endianness.

The LabVIEW programming environment, though most commonly installed on Windows machines, was first developed on a Macintosh, and uses Big Endian format for its binary numbers, while most Windows programs use Little Endian format.[12]

Note that since the required byte swap depends on the size of the numbers stored in the file (two 2-byte

integers require a different swap than one 4-byte integer), the file format must be known to perform endianness conversion.

"Bit endianness"

Main article: bit numbering

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 where each individual bit has a unique address. 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. Teletext, I²C, and SONET and SDH[13]). 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. As bit ordering is usually only relevant on a very low level, terming transmissions "LSB first" or "MSB first" is more descriptive than assigning an endianness to bit ordering.

Bit endianness is also used when referring to certain image formats, particularly bitonal images, which store a series of pixels as individual bits within a byte. If the bit order is incorrect, every group of eight pixels in the image will appear backwards.

References and notes

1. ^ **a b c** Danny Cohen (1980-04-01). *On Holy Wars and a Plea for Peace* (http://www.ietf.org/rfc/ien/ien137.txt). IEN 137. http://www.ietf.org/rfc/ien/ien137.txt. "…which bit should travel first, the bit from the little end of the word, or the bit from the big end of the word? The followers of the former approach are called the Little-Endians, and the followers of the latter are called the Big-Endians." Also published at *IEEE Computer*, October 1981 issue (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1667115).
2. ^ 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 (…)"
3. ^ David Cary. "Endian FAQ" (http://david.carybros.com/html/endian_faq.html). Retrieved 2010-10-11.
4. ^ Ken Lunde (13 January 2009). *CJKV Information Processing* (http://books.google.com/books?id=SA92uQqTB-AC&pg=PA29). O'Reilly Media, Inc. p. 29. ISBN 978-0-596-51447-1. Retrieved 21 May 2013.
5. ^ "NUXI problem" (http://catb.org/jargon/html/N/NUXI-problem.html). *The Jargon File*. Retrieved 2008-12-20.
6. ^ Andrew S. Tanenbaum; Todd M. Austin (4 August 2012). *Structured Computer Organization* (http://books.google.com/books?id=m0HHygAACAAJ). Prentice Hall PTR. ISBN 978-0-13-291652-3. Retrieved 18 May 2013.
7. ^ 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*.
8. ^ "Floating point formats" (http://www.quadibloc.com/comp/cp0201.htm).
9. ^ "pack – convert a list into a binary representation" (http://www.perl.com/doc/manual/html/pod/perlfunc/pack.html).
10. ^ Reynolds, J.; Postel, J. (October 1994). "Data Notations" (https://tools.ietf.org/html/rfc1700#page-3). *Assigned Numbers* (https://tools.ietf.org/html/rfc1700). IETF. p. 3. STD 2. RFC 1700. https://tools.ietf.org/html/rfc1700#page-3. Retrieved 2012-03-02.
11. ^ Ethernet POWERLINK Standardisation Group (2012), *EPSG Working Draft Proposal 301: Ethernet POWERLINK Communication Profile Specification Version 1.1.4*, chapter 6.1.1.
12. ^ read write binary files with LabVIEW (http://digital.ni.com/public.nsf/allkb/97332426D63630EE862565070049FFBB)
13. ^ Cf. Sec. 2.1 Bit Transmission of http://tools.ietf.org/html/draft-ietf-pppext-sonet-as-00

Further reading

- Danny Cohen (1980-04-01). *On Holy Wars and a Plea for Peace* (http://www.ietf.org/rfc/ien/ien137.txt). IEN 137. http://www.ietf.org/rfc/ien/ien137.txt. Also published at *IEEE Computer*, October 1981 issue (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1667115).
- David V. James (June 1990). "Multiplexed buses: the endian wars continue" (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=56322). *IEEE Micro* **10** (3): 9–21. doi:10.1109/40.56322 (http://dx.doi.org/10.1109%2F40.56322). ISSN 0272-1732 (//www.worldcat.org/issn/0272-1732). Retrieved 2008-12-20.
- Bertrand Blanc, Bob Maaraoui (December 2005). *Endianness or Where is Byte 0?* (http://3bc.bertrand-blanc.com/endianness05.pdf). Retrieved 2008-12-21.

External links

- Understanding big and little endian byte order (http://betterexplained.com/articles/understanding-big-and-little-endian-byte-order/)
- Byte Ordering PPC (http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/Concepts/ByteOrdering.html)
- Writing endian-independent code in C (http://www.ibm.com/developerworks/aix/library/au-endianc/index.html?ca=drs-)
- How to convert an integer to little endian or big endian (http://techforb.blogspot.com/2007/10/how-to-convert-integer-to-little-endian.html)
- C-Level Code Illustration (http://sites.google.com/site/insideoscore/endianness)
- xlong/xshort data-types, the Big-Endian, Little-Endian Rosetta stone (http://www.stanford.edu/dept/its/support/uspires/xlong/)
- Predef endianness recommendations (http://sourceforge.net/p/predef/wiki/Endianness/)

This article is based on material taken from the Free On-line Dictionary of Computing prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.

Retrieved from "http://en.wikipedia.org/w/index.php?title=Endianness&oldid=570147843"
Categories: Computer memory | Data transmission | Metaphors
