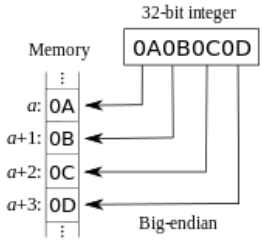
Endianness

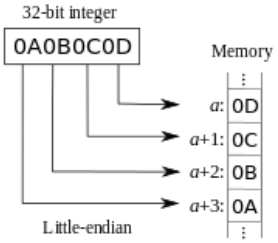
From Wikipedia, the free encyclopedia

The terms **endian** and **endianness** refer to the convention used to interpret the bytes making up a data word when those bytes are stored in computer memory. In computing, memory commonly stores binary data by organizing it into 8-bit units called bytes. When reading or writing a data word consisting of multiple such units, the order of the bytes stored in memory determines the interpretation of the data word.



Each byte of data in memory has its own address. **Big-endian** systems store the *most significant byte* of a word in the *smallest address* and the least significant byte is stored in the largest address (also see Most significant bit). **Little-endian** systems, in contrast, store the *least significant byte* in the *smallest address*.[1]

The illustration to the right shows an example using the data word "0A 0B 0C 0D" (a set of 4 bytes written out using left-to-right positional, hexadecimal notation) and the four memory locations with addresses *a*, *a*+1, *a*+2 and *a*+3; then, in big-endian systems, byte 0A is stored in *a*, 0B in *a*+1, 0C in *a*+2 and 0D in *a*+3. In little-endian systems, the order is reversed with 0D stored in memory address *a*, 0C in *a*+1, 0B in *a*+2, and 0A in *a*+3.



Both forms of endianness are in widespread use in computing and networking. The initial endianness design choice was (is) mostly arbitrary, but later technology revisions and updates perpetuate the same endianness (and many other design attributes) to maintain backward compatibility. As examples, the Intel x86 processor represents a common *little-endian* architecture, and IBM z/Architecture mainframes are all *big-endian* processors. The designers of these two processor architectures fixed their endiannesses in the 1960s and 1970s with their initial product introductions to the market. Big-endian is the most common convention in data networking (including IPv6), hence its pseudo-synonym *network byte order*, and little-endian is popular (though not universal) among microprocessors in part due to Intel's significant historical influence on microprocessor designs. Mixed forms also exist, 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 sometimes referred to as mixed-endian or middle-endian. There are also some *bi-endian* processors which can operate either in little-endian or big-endian mode.

Endianness is important as a low-level attribute of a particular data format. Failure to account for varying endianness across architectures when writing software code for mixed platforms and when exchanging certain types of data might lead to failures and bugs, though these issues have been understood and properly handled for many decades.

Contents

History

Before the advent of microprocessors, most computers used a big-endian approach (a notable exception being the PDP-11). Given the limitations of the chip technology of the time, the first microprocessor, for the Datapoint 2200, was designed using simpler bit-serial logic where little-endian address and data formats facilitate carry propagation. Datapoint's initial specification was big-endian but to save transistors, they acquiesced to Intel's request for a design change to little-endian. When byte-parallel computation was implemented in later processors (ex: 8080), Intel left in the little-endian format as a compromise for consistency with Datapoint's earlier bit-serial microprocessor. Datapoint never used the 8008 chip (using an MSI equivalent implementation as Intel was unable to deliver it in time), however, and the little-endian format was never actually necessary.[2][3]

The problem of dealing with data in different endianness format is sometimes termed the *NUXI problem*.[4] This terminology alludes to the problems due to byte order conflicts encountered while adapting Unix (which ran on the little-endian PDP-11) to a big-endian computer (the IBM Series/1). One of the first programs converted was supposed to print out "Unix" but on the Series/1 instead printed out "nUxi" due to the differing endianness of the machines.[5] UNIX was one of the first systems to allow the same code to run on, and transfer data between, platforms with different internal representations.

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*).[6][7] The terms *little-endian* and *endianness* have a similar intent.[8]

Wikisource has original text related to this article: **Gulliver's Travels (Part I, Chapter IV)**

Danny Cohen's "On Holy Wars and a Plea for Peace" published in 1980[7] 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, Ireland 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.

Endianness and hardware

An example of endianness is to think of how a decimal number is written and read in place-value notation. Assuming a writing system where numbers are written left to right, the leftmost position is analogous to the smallest address of memory used, and rightmost position the largest. For example, the number one hundred twenty three is written "1 2 3", with the hundreds digit 1 left-most. Anyone who reads this number also knows that the leftmost digit has the biggest place value. This is an example of a big-endian convention followed in daily life. The little-endian way of writing the same number would be to put the ones digit 3 in the left-most position: "3 2 1". A person following place-value notation who is not alerted of this special ordering would assume the number was three hundred and twenty one. Endianness in computing is similar, but it applies to the ordering of bytes in memory (groups of typically 8 base 2 digits) instead of the ordering of decimal (base 10) digits on paper.

Computer memory consists of a sequence of cells, usually bytes, and each cell has a number called its address that programs (more exactly: machine instructions of compiled programs) use to refer to it. If the total number of bytes in memory is `n` then bytes addresses would be enumerated `0` to `n-1`. Computer programs deal with fields which may consist of more than one byte. For the purpose of this article where its use as an operand of an instruction is relevant, a field consists of a consecutive sequence of bytes and represents a simple data value. In addition to that, it has to be of numeric type in some positional number system (mostly base-10 or base-2 — or base-256 in case of 8-bit bytes).[9] In such a number system the "value" of a digit is determined not only by its value as a single digit, but also by the position it holds in the complete number, its "significance". These positions can be mapped to memory mainly in two ways:[10]

- 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*[11]

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, DEC Alpha, Altera Nios II, Atmel AVR, VAX, and, largely, PDP-11. The Intel 8051, contrary to other Intel processors, expects 16-bit addresses in big-endian format, except for the LCALL instruction whose target address is stored in little-endian format.[12]

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, SuperH, IBM POWER, Atmel AVR32, 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.

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*, 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.[13] There are old ARM processors that have half little-endian, half big-endian floating point representation for double-precision numbers: both 32-bit words are stored in little-endian like integer registers, but the most significant one first. 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 diverse systems. It may therefore appear strange that the widespread IEEE 754 floating point standard does not specify endianness.[14] 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 monotonically-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 simultaneously, fetching multi-byte operands from memory as a single operation, so that the complexity of the hardware is not affected by the byte ordering.

Mapping multi-byte binary values to memory

C-type	*name*	*initial value of variable*
`int32_t`	longVar	= 168496141;

Big-Endian

memory at offset

+0	+1	+2	+3
$0A_h$	$0B_h$	$0C_h$	$0D_h$

Little-Endian

memory at offset

+0	+1	+2	+3
$0D_h$	$0C_h$	$0B_h$	$0A_h$

Let us agree to understand the orientation *left to right* in memory as increasing memory addresses − as in the table to the left. Furthermore, the hex value 0x0a0b0c0d is defined to be the value 168496141 in the usual (and Big-Endian style) decimal notation. If you map this value as a binary value to a sequence of 4 bytes in memory in Big-Endian style, you are writing the bytes from *left to right*

`int16_t shortVar = 3085;`

A simple way to remember is "In *L*ittle Endian, the *L*east significant byte goes into the *L*owest-addressed slot". So in the example in the table, $0D_h$, the least significant byte, in a Little-Endian system goes into slot +0.

in ***de****creasing* significance: $0A_h$ at +0, $0B_h$ at +1, $0C_h$ at +2, $0D_h$ at +3. However, on a Little-Endian system, the bytes are written from *left to right* in ***in****creasing* significance, starting with the one's byte: $0D_h$ at +0, $0C_h$ at +1, $0B_h$ at +2, $0A_h$ at +3. If you write a 32-bit binary 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 is *reversed* (byte-swapped) compared to usual Big-Endian notation. This is the way a hexdump is displayed: Because the dumping program is unable to know what kind of data it is dumping, the only orientation it can observe is monotonically increasing addresses. The human reader, however, who knows that he is reading a hexdump of a Little-Endian system and who knows what kind of data he is reading, reads the byte sequence $0D_h$,$0C_h$,$0B_h$,$0A_h$ as the 32-bit binary value 168496141, or 0x0a0b0c0d in hexadecimal notation. (Of course, this is *not* the same as the number $0D0C0B0A_h$ = 0x0d0c0b0a = 218893066.)

Example: Interpretation of a Hexdump

The following example shows how a piece of a hexdump is to be interpreted containing two consecutive bytes (4 half-bytes) with hexadecimal a732.

	hex-dump			2 unsigned 8-bit binary numbers				1 unsigned 16-bit binary number			Shift Right 16-bit by 1 bit (>> 1)		
	half-bytes		text	byte0: bits	dec	byte1: bits	dec	bits	hex	dec	bits	hex	dec
offset	0	1		0123 4567		0123 4567		0123 4567 89ab cdef			0123 4567 89ab cdef		
				Big-Endian									
value	a7	32	§2	1010 0111	167	0011 0010	50	1010 0111 0011 0010	a732	42802	0101 0011 1001 1001	5399	21401
				Little-Endian									
internal bit sequence				1110 0101		0100 1100		1110 0101 0100 1100			1100 1010 1001 1000		
bit pattern				7654 3210		7654 3210		fedc ba98 7654 3210			fedc ba98 7654 3210		
value	a7	32	§2	1010 0111	167	0011 0010	50	0011 0010 1010 0111	32a7	12967	0001 1001 0101 0011	1953	6483

If a field consists of only one byte, e. g. a 8-bit binary number, or of a sequence of 1-byte characters e. g. in ISO 8859 code, then the interpretation of the 2 bytes does not differ between the two styles Big- or Little-Endian. Although in Little-Endian systems the internal sequence of the bits in the byte is swapped the same way as the bytes in the integer, the bit pattern 7654 3210 which is to be found in the manuals (and which has nothing to do with the internal order of the bits) always shows the bits in the usual base-2 numeral system in descending significance with the one's position at bit 0 at the far right. In short: the reordering of the bits in the byte cannot be noticed in the hexdump.

However, if the data field is larger than a single byte, the reversed sequence of the bytes in Little-Endian systems becomes visible. In case of a 16-bit binary number the long bit pattern fedc ba98 7654 3210 exchanges the 2 bytes. In the table the byte with the lower address is marked by an overstrike.

The 3 columns to the right in the table show a shift operation. With shift operations, left and right is independent of Big- or Little-Endian style and is oriented solely by the usual base-2 numeral system: a left-shift multiplies and a right-shift divides by a power of 2. The (both) shift operations uniquely induce an internal order from the bytes to the bits. The example in the table shows a (logical) right shift (division by 2) of an (unsigned) 16-bit binary number: in Big-Endian systems a one-bit is moved into the adjacent byte, so it must have been „in the middle" of the integer; in Little-Endian systems a one-bit is moved out of the 16-bit number, so it must have been „at the edge" of the integer. A subroutine for the concatenation of 2 bit strings s_1 and s_2 of lengths l_1 resp. l_2 (counted in bits) has to shift the second bit string by the bit offset $b := l_1 \bmod 8$, in order to combine it with the last byte of the first bit string. On Big-Endian systems this means a right and on Little-Endian systems a left shift by b bits.

Examples of storing the value $0A0B0C0D_h$ in memory

Note that hexadecimal notation is used.

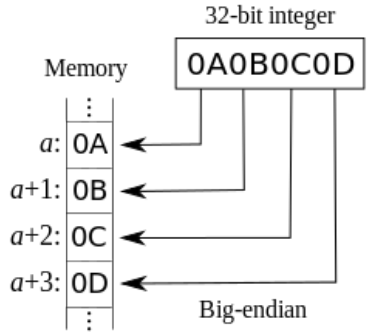
To illustrate the notions this section provides example layouts of the 32-bit number $0A0B0C0D_h$ in the most common variants of endianness. There exist several digital processors that use other formats, but these two are the most common in general processors. That is true for typical embedded systems 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.

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 →

…	$0A_h$	$0B_h$	$0C_h$	$0D_h$	…

The most significant byte (*MSB*) value, which is $0A_h$ in our example, is stored at the memory location with the lowest address, the next byte value in significance, $0B_h$, 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 →

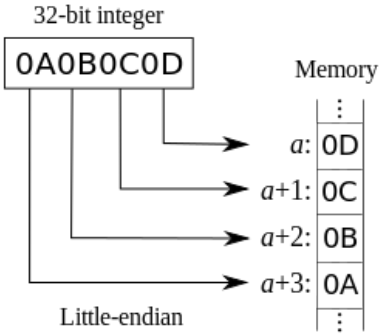
…	$0A0B_h$	$0C0D_h$	…

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.

When organized by Byte addresses

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 (hexadecimal 0A0B0C0D) 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

The IBM 1400 series has characteristics of both little- and big-endian. Integers are stored as a series of decimal digits, coded in the same form used for decimal digits in a character string, one per addressed memory location. Strings are stored in "natural" order, the first character being stored in the lowest-addressed position. For a multi-digit integer, the lowest-addressed character contains the most significant decimal digit, which is characteristic of big-endian. However, an integer's *address* is the address of its *least* significant digit; this is characteristic of little-endian. To perform most arithmetic the machine starts with the least significant digits of the operands and works toward the most significant digits (the same sequence as usually used by people in manual calculation), i.e. from the higher memory address to the lower.

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.[15]

The telephone network has always sent the most significant part first, the area code; doing so allows routing to begin while a telephone number is still being keyed or dialed.

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.[16]

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 programmers 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`; a little-endian should start with `FF FE 00 00`.

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:

1. storing the data always in one fixed endianness, or
2. carrying with the data a switch to indicate which endianness the data was written with.

When reading the file, the application converts the endianness, invisibly from the user. An example of the first case is the binary XLS file format that is portable between Windows and Mac systems and always little endian, leaving the Mac application to swap the bytes on load and save.[17]

TIFF image files are an example of the second strategy, whose header instructs the application 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.

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.

```c
/* C function to change endianness for byte swap in an unsigned 32-bit integer */

uint32_t ChangeEndianness(uint32_t value)
{
    uint32_t result = 0;
    result |= (value & 0x000000FF) << 24;
    result |= (value & 0x0000FF00) << 8;
    result |= (value & 0x00FF0000) >> 8;
    result |= (value & 0xFF000000) >> 24;
    return result;
}
```

"Bit endianness"

The terms *Bit endianness* and *bit-level endianness* often refer to the transmission order of bits over a serial medium. Usually that order is transparently managed by the hardware and is the bit-level analogue of little-endian (low-bit first), as in RS-232 and Ethernet. Some protocols use the opposite ordering (e.g. Teletext, I²C, and SONET and SDH[18]). In networking, the order of transmission of bits is specified 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, terms like "LSB first" and "MSB first" are more descriptive than the word endianness to bit ordering.

Bit endianness is also important when handling some image file formats, especially 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. Also for hardware that uses data stored in bytes, but displays individual bits on the screen (one bit per pixel).

The terms *bit endianness* and *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.

References and notes

1. University of Maryland - Definitions (http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/endian.html). Accessed 26 Sept 2014
2. House, David; Fagin, Federico; Feeney, Hal; Gelbach, Ed; Hoff, Ted; Mazor, Stan; Smith, Hank (2006-09-21). "Oral History Panel on the Development and Promotion of the Intel 8008 Microprocessor" (http://archive.computerhistory.org/resources/text/Oral_History/Intel_8008/Intel_8008_1.oral_history.2006.102657982.pdf#page=5). Computer History Museum. p. 5. Retrieved 23 April 2014. "Mazor: And lastly, the original design for Datapoint... what they wanted was a [bit] serial machine. And if you think about a serial machine, you have to process all the addresses and data one-bit at a time, and the rational way to do that is: low-bit to high-bit because that's the way that carry would propagate. So it means that [in] the jump instruction itself, the way the 14-bit address would be put in a serial machine is bit-backwards, as you look at it.

because that's the way you'd want to process it. Well, we were gonna built a byte-parallel machine, not bit-serial and our compromise (in the spirit of the customer and just for him), we put the bytes in backwards. We put the low- byte [first] and then the high-byte. This has since been dubbed "Little Endian" format and it's sort of contrary to what you'd think would be natural. Well, we did it for Datapoint. As you'll see, they never did use the [8008] chip and so it was in some sense "a mistake", but that [Little Endian format] has lived on to the 8080 and 8086 and [is] one of the marks of this family."

3. 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.
4. "NUXI problem" (http://catb.org/jargon/html/N/NUXI-problem.html). *The Jargon File*. Retrieved 2008-12-20.
5. Jalics, Paul J.; Heines, Thomas S. (1 December 1983). "Transporting a portable operating system: UNIX to an IBM minicomputer". *Communications of the ACM* **26** (12): 1066–1072. doi:10.1145/358476.358504 (https://dx.doi.org/10.1145%2F358476.358504).
6. 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 …"
7. Danny Cohen (1980-04-01). *On Holy Wars and a Plea for Peace* (http://www.ietf.org/rfc/ien/ien137.txt). IETF. 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).
8. David Cary. "Endian FAQ" (http://david.carybros.com/html/endian_faq.html). Retrieved 2010-10-11.
9. When character (text) strings are compared with one another, this is done lexicographically where a single positional element (character) also has a positional value. Lexicographical comparison means almost everywhere: first character ranks highest — as in the telephone book. This would have the consequence that almost every machine would be big-endian or at least mixed-endian. Therefore, for the criterion below to apply, the data type in question has to be *numeric*.
10. 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.
11. Note that, in these expressions, the term "end" is meant as "extremity", not as "last part"; and that (the extremity with) *big* resp. *little* significance is written *first*.
12. http://www.keil.com/support/man/docs/c51/c51_xe.htm
13. "Floating point formats" (http://www.quadibloc.com/comp/cp0201.htm).
14. "pack – convert a list into a binary representation" (http://www.perl.com/doc/manual/html/pod/perlfunc/pack.html).
15. 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.
16. Ethernet POWERLINK Standardisation Group (2012), *EPSG Working Draft Proposal 301: Ethernet POWERLINK Communication Profile Specification Version 1.1.4*, chapter 6.1.1.
17. "Microsoft Office Excel 97 - 2007 Binary File Format Specification (*.xls 97-2007 format)" (http://download.microsoft.com/download/0/B/E/0BE8BDD7-E5E8-422A-ABFD-4342ED7AD886/Excel97-2007BinaryFileFormat(xls)Specification.xps). Microsoft Corporation. 2007.
18. 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). IETF. 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 (https://dx.doi.org/10.1109%2F40.56322). ISSN 0272-1732 (https://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.
- Wikipedia (May 2014). "Mark William Co." (http://en.wikipedia.org/wiki/Mark_Williams_Company). Retrieved 2014-05-30.

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-)

Categories: Computer memory | Data transmission | Metaphors