Endianness

Endianness refers to the sequential order in which bytes are arranged into larger numerical values when stored in memory or when transmitted over digital links. Endianness is of interest in computer science because two conflicting and incompatible formats are in common use: words may be represented in **big-endian** or **little-endian** format, depending on whether bits or bytes or other components are ordered from the big end (most significant bit) or the little end (least significant bit).

In big-endian format, whenever addressing memory or sending/storing words bytewise, the most significant byte — the byte containing the most significant bit — is stored first (has the lowest address) or sent first, then the following bytes are stored or sent in decreasing significance order, with the least significant byte — the one containing the least significant bit — stored last (having the highest address) or sent last.

Little-endian format reverses this order: the sequence addresses/sends/stores the least significant byte first (lowest address) and the most significant byte last (highest address). Most computer systems prefer a single format for all its data; using the system's native format is automatic. But when reading memory or receiving transmitted data from a different computer system, it is often required to process and translate data between the preferred native endianness format to the opposite format.

The order of bits within a byte or word can also have endianness (as discussed later); however, a byte is typically handled as a single numerical value or character symbol and so bit sequence order is obviated.

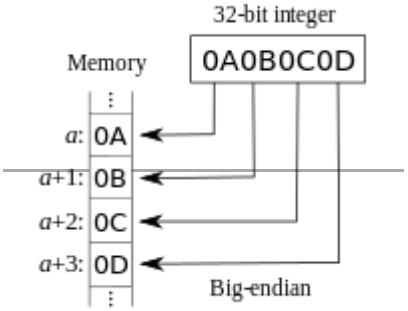
Both big and little forms of endianness are widely used in digital electronics. The choice of endianness for a new design is often arbitrary, but later technology revisions and updates perpetuate the existing endianness and many other design attributes to maintain backward compatibility. As examples, the IBM z/Architecture mainframes and the Motorola 68000 series use big-endian while the Intel x86 processors use little-endian. The designers of System/360, the ancestor of z/Architecture, chose its endianness in the 1960s; the designers of the Motorola 68000 and the Intel 8086, the first members of the 68000 and x86 families, chose their endianness in the 1970s.

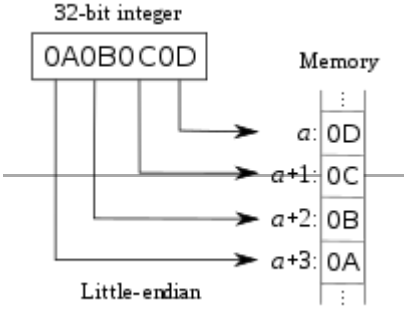
Big-endian is the most common format in data networking; fields in the protocols of the Internet protocol suite, such as IPv4, IPv6, TCP, and UDP, are transmitted in big-endian order. For this reason, big-endian byte order is also referred to as **network byte order**. Little-endian storage is popular for microprocessors, in part due to significant influence on microprocessor designs by Intel Corporation. Mixed forms also exist, for instance the ordering of bytes in 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 that operate in either little-endian or big-endian mode.

Compare also to the Head-initial vs. head-final languages in linguistics.

Contents

Illustration





Big-endianness may be demonstrated by writing a decimal number, say one hundred twenty-three, on paper in the usual positional notation understood by a numerate reader: *123*. The digits are written starting from the left and to the right, with the most significant digit, *1*, written first. This is analogous to the lowest address of memory being used first. This is an example of a big-endian convention taken from daily life.

The little-endian way of writing the same number, one hundred twenty-three, would place the hundreds-digit *1* in the right-most position: *321*. A person following conventional big-endian place-value order, who is not aware of this special ordering, would read a different number: three hundred and twenty one. Endianness in computing is similar, but it usually applies to the ordering of bytes, rather than of digits.

The illustrations to the right, where *a* is a memory address, show big-endian and little-endian storage in memory.

Etymology

Danny Cohen introduced the terms Little-Endian and Big-Endian for byte ordering in an article from 1980.[1][2] In this technical and political examination of byte ordering issues, the "endian" names were drawn from Jonathan Swift's 1726 satire, *Gulliver's Travels*, in which civil war erupts over whether the big end or the little end of a boiled egg is the proper end to crack open, which is analogous to counting from the end that contains the most significant bit or the least significant bit.[3][4]

Hardware

Computer memory consists of a sequence of storage cells. Each cell is identified in hardware and software by its memory address. If the total number of storage cells in memory is *n*, then addresses are enumerated from *0* to *n-1*. Computer programs often use data structures of fields that may consist of more data than is stored in one memory cell. 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).[5] 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:[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]

History

While the Intel microprocessor product line (most notable amongst others) has become a popular architecture, many historical and extant processors use a big-endian memory representation, commonly referred to as network order, as used in the Internet protocol suite, either exclusively or as a design option; others use yet another scheme called "middle-endian", "mixed-endian" or "PDP-11-endian".

The IBM System/360 uses big-endian byte order, as do its successors System/370, ESA/390, and z/Architecture. The PDP-10 also uses big-endian addressing for byte-oriented instructions. The IBM Series/1 minicomputer also use big-endian byte order.

Dealing with data of different endianness is sometimes termed the *NUXI problem*.[8] This terminology alludes to the byte order conflicts encountered while adapting UNIX, which ran on the mixed-endian PDP-11, to a big-endian IBM Series/1 computer. Unix was one of the first systems to allow the same code to be compiled for platforms with different internal representations. One of the first programs converted was supposed to print out *Unix*, but on the Series/1 it printed *nUxi* instead.[9]

The Datapoint 2200 uses simple bit-serial logic with little-endian to facilitate carry propagation. When Intel developed the 8008 microprocessor for Datapoint, they used little-endian for compatibility. However, as Intel was unable to deliver the 8008 in time, Datapoint used a medium scale integration equivalent, but the little-endianness was retained in most Intel designs.[10][11] Intel MCS-48 is also little-endian, as are the well-known DEC Alpha, Atmel AVR, VAX and many more.

The Motorola 6800 / 6801, the 6809 and the 68000 series of processors used the big-endian format, and for this reason, it is also known as the "*Motorola convention*".[12][13]

The Intel 8051, contrary to other Intel processors, expects 16-bit addresses for LJMP and LCALL in big-endian format; however, xCALL instructions store the return address onto the stack in little-endian format.[14]

SPARC historically used big-endian until version 9, which is bi-endian; similarly early IBM POWER processors were big-endian, but now the PowerPC and Power Architecture descendants are bi-endian. The ARM architecture was little-endian before version 3 when it became bi-endian.

Other well-known little-endian processor architectures include MOS Technology 6502 (including Western Design Center 65802 and 65C816), Zilog Z80 (including Z180 and eZ80) and Altera Nios II.

Current architectures

The Intel x86 and also AMD64 / x86-64 series of processors use the **little-endian** format, and for this reason, it is also known in the industry as the "*Intel convention*".[12][13]

The few current **big-endian** architectures include the IBM z/Architecture, Freescale ColdFire (which is Motorola 68000 series-based), Xilinx Microblaze, SuperH, Atmel AVR32.

As a consequence of its original implementation on the Intel x86 platform, the operating system-independent FAT file system is defined to use little-endian byte ordering, even on platforms using other endiannesses natively

Bi-endianness

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 fetches and stores, instruction fetches, 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 and almost all SPARC processors, allow per-page choice of endianness.

SPARC processors since the late 1990s ("SPARC v9" compliant processors) allow data endianness to be chosen with each individual instruction that loads from or stores to memory

Many processors have instructions to convert a word in a register to the opposite endianness, that is, they swap the order of the bytes in a 16-, 32- or 64-bit word. All the individual bits are not reversed though.

Recent Intel x86 and x86-64 architecture CPUs have a MOVBE instruction (Intel Core since generation 4, after Atom),[15] which fetches a big-endian format word from memory or writes a word into memory in big-endian format. These processors are otherwise thoroughly little-endian. They also already had a range of swap instructions to reverse the byte order of the contents of registers, such as when words have already been fetched from memory locations where they were in the 'wrong' endianness.

ZFS/OpenZFS combined file system and logical volume manager is known to provide adaptive endianness and to work with both big-endian and little-endian systems.[16]

Floating point

Although the ubiquitous x86 processors of today use little-endian storage for all types of data (integer, floating point, BCD), there are a number of hardware architectures where floating-point numbers are represented in big-endian form while integers are represented in little-endian form.[17] There are 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, the XDR standard uses big-endian IEEE 754 as its representation. It may therefore appear strange that the widespread IEEE 754 floating-point standard does not specify endianness.[18] 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 onl 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 byte (MSB). On the other hand, arithmetic division is done starting from the MSB, so it is more natural for big-endian processors. However, high-performance processors usually fetch typical multi-byte operands from memory in the same amount of time they would have fetched a single byte, so the complexity of the hardware is not affected by the byte ordering.

Mapping multi-byte binary values to memory

			Big-Endian				**Little-Endian**			
C-type	*name of variable*	*initial value*	*memory at offset*				*memory at offset*			
			+0	+1	+2	+3	+0	+1	+2	+3
`int32_t`	longVar	= 0x0a0b0c0d;	$0A_h$	$0B_h$	$0C_h$	$0D_h$	$0D_h$	$0C_h$	$0B_h$	$0A_h$
`int16_t`	shortVar	= 0x0c0d;	$0C_h$	$0D_h$			$0D_h$	$0C_h$		

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.

We can assume that as we write text *left to right*, we are increasing the 'address' on paper, as a processor would write bytes with increasing memory addresses − as in the adjacent table. On paper, the hex value 0a0b0c0d (written 168496141 in usual decimal notation) is big-endian style since we write the most significant digit first and the rest follow ***de****creasing* significance. Mapping this number as a binary value to a sequence of 4 bytes in memory in big-endian style also writes the bytes from *left to right* in ***de****creasing* significance: $0A_h$ at +0, $0B_h$ at +1, $0C_h$ at +2, $0D_h$ at +3.

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. Writing a 32-bit binary value to a memory location on a little-endian system and outputting the memory location (with growing addresses from left to right) shows that the order 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 or she is reading a hexdump of a little-endian system and who knows what kind of data he or she 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.)

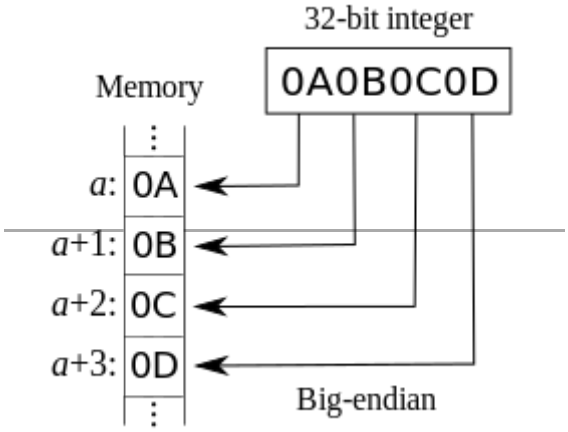
Examples

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. That is true for typical embedded systems as well as for general computer CPUs. 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.

The examples refer to the storage in memory of the value. It uses hexadecimal notation.

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, $0A_h$, is at the lowest address. The other bytes follow in decreasing order of significance. This is akin to left-to-right reading in hexadecimal order

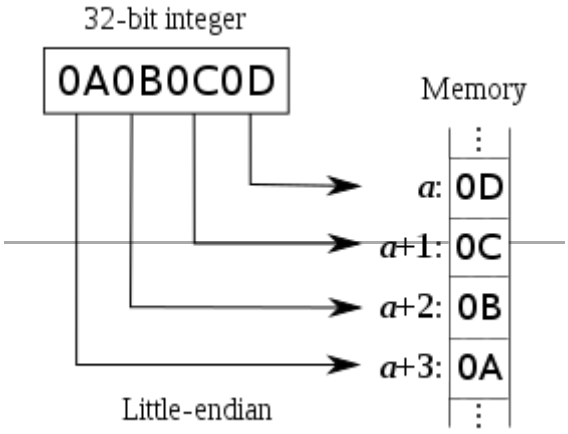
Atomic element size 16-bit

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. This is akin to right-to-left reading in hexadecimal order

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 instructions to convert between floating-point and integer values in the optional floating-point processor on the PDP-11/45[19] and PDP-11/70, and in some later processors, stored 32-bit "double precision integer long" values with the 16-bit halves swapped from the expected little-endian order, and the UNIX C compiler used the same format for 32-bit long integers. 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.

Segment descriptors on Intel 80386 and compatible processors keep a 32-bit base 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.

An example of middle-endianness is the American date format

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 CPU instruction sets provide native support for endian byte swapping, such as *bswap*[20] (x86 - 486 and later), and *rev*[21] (ARMv6 and later).

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 when running on a big-endian Motorola 68K or PowerPC processor.[22]

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.

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;
}
```

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 the network order for protocols in the Internet protocol suite to be big-endian, hence the use of the term "network byte order" for big-endian byte order; however, not all protocols use big-endian byte order as the network order.[23]

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.[24]

While the high-level network protocols usually consider the byte (mostly meant as *octet*) as their atomic unit, the lowest network protocols may deal with ordering of bits within a byte.

Bit endianness

Bit numbering is a concept similar to endianness, but on a level of bits, not bytes. *Bit endianness* or *bit-level endianness* refers to the transmission order of bits over a serial medium. The bit-level analogue of little-endian (least significant bit goes first) is used in RS-232, Ethernet, and USB. Some protocols use the opposite ordering (e.g. Teletext, I²C, SMBus, PMBus, and SONET and SDH[25]). Usually, there exists a consistent view to the bits irrespective of their order in the byte, such that the latter becomes relevant only on a very low level. One exception is caused by the feature of some cyclic redundancy checks to detect *all* burst errors up to a known length, which would be spoiled if the bit order is different from the byte order on serial transmission.

Apart from serialization, the terms *bit endianness* and *bit-level endianness* are seldom used, as computer architectures where each individual bit has a unique address are rare. Individual bits or bit fields are accessed via their numerical value or, in high-level programming languages, assigned names, the effects of which, however, may be machine dependent or lack software portability. The natural numbering is that the arithmetic left shift $1<<n$ yields a mask for the bit of position n, a rule which exhibits the machine's (byte) endianness at least if $n \geq 8$, e.g. if used for indexing a sufficiently large bit array. Other numberings do occur in various documentations.

References

1. 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)
2. "Internet Hall of Fame Pioneer" (http://internethalloffame.org/inductees/danny-cohen). *Internet Hall of Fame*. The Internet Society.
3. Jonathan Swift (1726). *Gulliver's Travels* (http://en.wikisource.org/wiki/Gulliver%27s_Travels/Part_I/Chapter_IV)
4. David Cary. "Endian FAQ" (http://david.carybros.com/html/endian_faq.html) Retrieved 2010-10-11.

5. 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*.
6. Andrew S. Tanenbaum; Todd M. Austin (4 August 2012). *Structured Computer Organization* (https://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 (the extremity with) *big* resp. *little* significance is written *first*.
8. "NUXI problem" (http://catb.org/jargon/html/N/NUXI-problem.html) *The Jargon File*. Retrieved 2008-12-20.
9. 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://doi.org/10.1145%2F358476.358504)
10. House, David; Faggin, 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) (PDF). 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"
11. Ken Lunde (13 January 2009). *CJKV Information Processing* (https://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.
12. Küveler, Gerd; Schwoch, Dietrich (2013) [1996]. *Arbeitsbuch Informatik - eine praxisorientierte Einführung in die Datenverarbeitung mit Projektaufgabe* (https://books.google.com/books?id=b8-dBgAAQBAJ) (in German). Vieweg-Verlag, reprint: Springer-Verlag. doi:10.1007/978-3-322-92907-5 (https://doi.org/10.1007%2F978-3-322-92907-5) ISBN 978-3-528-04952-2. 9783322929075. Retrieved 2015-08-05.
13. Küveler, Gerd; Schwoch, Dietrich (2007-10-04). *Informatik für Ingenieure und Naturwissenschaftler: PC- und Mikrocomputertechnik, Rechnernetze* (https://books.google.com/books?id=xQbvPYxceY0C) (in German). **2** (5 ed.). Vieweg, reprint: Springer-Verlag. ISBN 3834891916. 9783834891914. Retrieved 2015-08-05.
14. "Cx51 User's Guide: E. Byte Ordering" (http://www.keil.com/support/man/docs/c51/c51_xe.htm). *keil.com*.
15. "How to detect New Instruction support in the 4th generation Intel® Core™ processor family" (https://software.intel.com/sites/default/files/article/405250/how-to-detect-new-instruction-support-in-the-4th-generation-intel-core-processo-family.pdf) (PDF). Retrieved 2 May 2017.
16. Matt Ahrens (2016). "FreeBSD Kernel Internals: An Intensive Code Walkthrough" (http://open-zfs.org/wiki/Documentation/Read_Write_Lecture). OpenZFS Documentation/Read Write Lecture.
17. "Floating point formats" (http://www.quadibloc.com/comp/cp0201.htm).
18. "pack – convert a list into a binary representation" (http://www.perl.com/doc/manual/html/pod/perlfunc/pack.html)
19. *PDP-11/45 Processor Handbook* (http://bitsavers.org/pdf/dec/pdp11/handbooks/PDP1145_Handbook_1973.pdf) (PDF). Digital Equipment Corporation 1973. p. 165.
20. "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z" (http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf) (PDF). Intel. September 2016. p. 3–112 Retrieved 2017-02-05.
21. "ARMv8-A Reference Manual" (http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.k_10775/index.html) ARM Holdings.
22. "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.

23. 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
24. Ethernet POWERLINK Standardisation Group (2012). *EPSG Working Draft Proposal 301: Ethernet POWERLINK Communication Profile Specification Version 1.1.4*, chapter 6.1.1.
25. Cf. Sec. 2.1 Bit Transmission of draft-ietf-pppext-sonet-as-00 "Applicability Statement for PPP over SONET/SDH" (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*. IETF. IEN 137. http://www.ietf.org/rfc/ien/ien137.txt. 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. Retrieved 2008-12-20.
- Bertrand Blanc, Bob Maaraoui (December 2005). "Endianness or Where is Byte 0?" (PDF). Retrieved 2008-12-21.

External links

- Understanding big and little endian byte order
- Byte Ordering PPC
- Writing endian-independent code in C