WIKIPEDIA

Endianness

In computing, **endianness** is the ordering or sequencing of bytes of a word of digital data in computer memory storage or during transmission. Endianness is primarily expressed as *big-endian* or *little-endian*. A **big-endian** system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest. A **little-endian** system, in contrast, stores the least-significant byte at the smallest address.

Internally, any given computer will work equally well regardless of what endianness it uses, since its hardware will consistently use the same endianness to both store and load its data. For this reason, programmers and computer users normally ignore the endianness of the computer they are working with. However, endianness can become an issue when moving data external to the computer – as when transmitting data between different computers, or a programmer investigating internal computer bytes of data from a memory dump – and the endianness used differs from expectation. In these cases, the endianness of the data must be understood and accounted for.

Both types of endianness are in widespread use in digital electronic engineering. The initial choice of endianness of 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. Big-endianness is the dominant ordering in networking protocols, such as in the internet protocol suite, where it is referred to as **network order**, transmitting the most significant byte first. Conversely, little-endianness is the dominant ordering for processor architectures (x86, most ARM implementations, base RISC-V implementations) and their associated memory. File formats can use either ordering; some formats use a mixture of both.

The term may also be used more generally for the internal ordering of any representation, such as the digits in a numeral system or the sections of a date. Numbers in place-value notation are written with their digits in big-endian order, even in right-to-left scripts. Similarly, programming languages use big-endian digit ordering for numeric literals as well as big-endian way of speaking for bit-shift operations (namely “left” [to be associated with low address] shifts towards the MSB), regardless of the endianness of the target architecture.

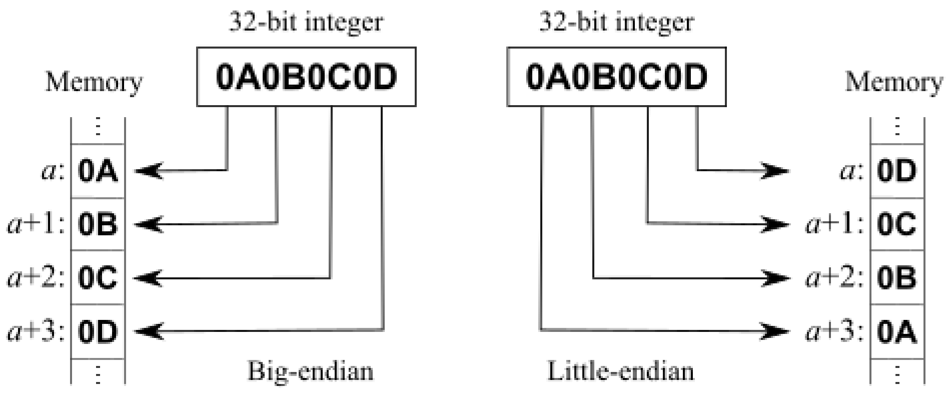
Bit-endianess is sometimes used in serialization of data transmission, but is seldom used in other contexts.

Bi-endianess is a feature supported by numerous computer architectures that feature switchable endianness in data fetches and stores or for instruction fetches.

Contents

Classical example

A big-endian architecture stores the most-significant byte of a multi-byte object, here: a 32-bit unsigned integer, at the lowest memory address of the storage location, and increases the address for each less significant byte. This is illustrated in the following graphic (left side), where the address of the first storage location is represented as *a*, the next one as *a + 1*, and so forth, and the data to be stored has the value 168496141_{dec} = `0x0A0B0C0D` in both cases:



The right-side graphic shows the little-endian representation. With increasing memory address (*a*), the numerical significance of each digit increases, so that the most-significant byte is stored at the highest address of the storage region. A valid notation for the data in storage could be `0x0D,0x0C,0x0B,0x0A` = $\texttt{0D}_{\texttt{h}}\texttt{,0C}_{\texttt{h}}\texttt{,0B}_{\texttt{h}}\texttt{,0A}_{\texttt{h}}$.

Etymology

The adjective *endian* has its origin in the writings of 18th century Anglo-Irish writer Jonathan Swift. In the 1726 novel *Gulliver's Travels*, he portrays the conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the big end or from the little end. He called them the *"Big-Endians"* and the *"Little-Endians"*.[1]

Danny Cohen introduced the terms *big-endian* and *little-endian* into computer science for data ordering, in an article published by the Internet Engineering Task Force (IETF) in 1980.[2][3][4]

Basics

Computer memory consists of a sequence of storage cells (smallest addressable units), most commonly called *bytes*. Each byte is identified and accessed in hardware and software by its memory address. If the total number of bytes 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 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 which – at least potentially – can be manipulated by *one* hardware instruction. The address of such a field is mostly the address of its first byte.[note 1] 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).[note 2] 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:[5]

- decreasing numeric significance with increasing memory addresses (or increasing time), known as *big-endian* and
- increasing numeric significance with increasing memory addresses (or increasing time), known as *little-endian*.[note 3]

Hardware

History

Many historical and extant processors use a big-endian memory representation, either exclusively or as a design option. Big-endian memory representation is commonly referred to as network order, as used in the Internet protocol suite. Other processor types use little-endian memory representation; 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*.[6] 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.[7]

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, including the MCS-48 and the 8086 and its x86 successors.[8][9]

The DEC Alpha, Atmel AVR, VAX, the MOS Technology 6502 family (including Western Design Center 65802 and 65C816), the Zilog Z80 (including Z180 and eZ80), the Altera Nios II, and many other processors and processor families are also little-endian.

The Motorola 6800 / 6801, the 6809 and the 68000 series of processors used the big-endian format.

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.[10]

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

Current architectures

The Intel x86 and AMD64 / x86-64 series of processors use the little-endian format. Other instruction set architectures that follow this convention, allowing only little-endian mode, include Nios II, Andes Technology NDS32, and Qualcomm Hexagon.

Some instruction set architectures allow running software of either endianness on a bi-endian architecture. This includes ARM AArch64, C-Sky, Power ISA, and RISC-V.

Solely big-endian architectures include the IBM z/Architecture, Freescale ColdFire (which is Motorola 68000 series-based), Atmel AVR32, and OpenRISC. The IBM AIX and Oracle Solaris operating systems on bi-endian Power ISA and SPARC run in big-endian mode; some distributions of Linux on Power have moved to little-endian mode.

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.

The ARM architecture supports two big-endian modes, called **BE-8** and **BE-32**.[11] CPUs up to ARMv5 only support BE-32 or **Word-Invariant** mode. Here any naturally aligned 32-bit access works like in little-endian mode, but access to a byte or 16-bit word is redirected to the corresponding address and unaligned access is not allowed. ARMv6 introduces BE-8 or **Byte-Invariant** mode, where access to a single byte works as in little-endian mode, but accessing a 16-bit, 32-bit or (starting with ARMv8) 64-bit word results in a byte swap of the data. This simplifies unaligned memory access as well as memory mapped access to registers other than 32 bit.

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),[12] 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.

Floating point

Although the ubiquitous x86 processors of today use little-endian storage for all types of data (integer, floating point), 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.[13] 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.[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.)

Variable length data

Most instructions considered so far contain the size (lengths) of its operands within the operation code. Frequently available operand lengths are 1, 2, 4, 8, or 16 bytes. But there are also architectures where the length of an operand may be held in a separate field of the instruction or with the operand itself, e. g. by means of a word mark. Such an approach allows operand lengths up to 256 bytes or even full memory size. The data types of such operands are character strings or BCD.

Machines being able to manipulate such data with one instruction (e. g. compare, add) are e. g. IBM 1401, 1410, 1620, System/3x0, ESA/390, and z/Architecture, all of them of type big-endian.

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.

In more concrete terms, such optimizations are the equivalent of the following C code returning true on most little-endian systems:

```c
union {
  uint8_t u8; uint16_t u16; uint32_t u32; uint64_t u64;
} u = { .u64 = 0x4A };
puts(u.u8 == u.u16 && u.u8 == u.u32 && u.u8 == u.u64 ? "true" : "false");
```

While not allowed by C++, such type punning code is allowed as "implementation-defined" by the C11 standard[15] and commonly used[16] in code interacting with hardware.[17]

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.

Optimizations of this kind are not portable across systems of different endianness.

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 and comparison 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.

Special cases

Other orderings, generically called *middle-endian* or *mixed-endian*, are possible.

PDP-endian

The PDP-11 is in principle is a 16-bit little-endian system. The instructions to convert between floating-point and integer values in the optional floating-point processor of the PDP-11/45, 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. The UNIX C compiler used the same format for 32-bit long integers. This ordering is known as *PDP-endian*.[18]

A way to interpret this endianness is that it stores a 32-bit integer as two 16-bit words in big-endian, but the words themselves are little-endian:

Storage of a 32-bit integer, `0x0A0B0C0D`, on a PDP-11

increasing addresses →

…	0B$_h$	0A$_h$	0D$_h$	0C$_h$	…
…	0A0B$_h$		0C0D$_h$		…

The 16-bit values here refer to their numerical values, not their actual layout.

Honeywell Series 16

The Honeywell Series 16 16-bit computers, including the Honeywell 316, are the opposite of the PDP-11 in storing 32-bit words in C. It stores each 16-bit word in big-endian order, but joins them together in little-endian manner:

storage of a 32-bit integer, `0x0A0B0C0D`, on a Honeywell 316

increasing addresses →

…	0C$_h$	0D$_h$	0A$_h$	0B$_h$	…
…	0C0D$_h$		0A0B$_h$		…

Intel IA-32 segment descriptors

Segment descriptors of IA-32 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.

Byte addressing

Little-endian representation of integers has the significance increasing from left to right, if it is written e. g. in a storage dump. In other words, it appears backwards when visualized, an oddity for programmers.

This behavior is mainly a concern for programmers utilizing FourCC or similar techniques that involve packing characters into an integer, so that it becomes a sequences of specific characters in memory. Let's define the notation `'John'` as simply the result of writing the characters in hexadecimal ASCII and appending `0x` to the front, and analogously for shorter sequences (a C multicharacter literal, in Unix/MacOS style):

```
    '  J  o  h  n  '
hex  4A 6F 68 6E
----------------
    -> 0x4A6F686E
```

On big-endian machines, the value appears left-to-right, coinciding with the correct string order for reading the result:

increasing addresses →

…	$4A_h$	$6F_h$	68_h	$6E_h$	…
…	'J'	'o'	'h'	'n'	…

But on a little-endian machine, one would see:

increasing addresses →					
…	$6E_h$	68_h	$6F_h$	$4A_h$	…
…	'n'	'h'	'o'	'J'	…

Middle-endian machines like the Honeywell 316 above complicate this even further: the 32-bit value is stored as two 16-bit words `'hn' 'Jo'` in little-endian, themselves with a big-endian notation (thus `'h' 'n' 'J' 'o'`).

This conflict between the memory arrangements of binary data and text is intrinsic to the nature of the little-endian convention.

Byte swapping

Byte-swapping consists of masking each byte and shifting them to the correct location. Many compilers provide built-ins that are likely to be compiled into native processor instructions (`bswap`/`movbe`), such as `__builtin_bswap32`. Software interfaces for swapping include:

- Standard network endianness functions (from/to BE, up to 32-bit).[19] Windows has a 64-bit extension in `winsocks2.h`.
- BSD and Glibc `endian.h` functions (from/to BE and LE, up to 64-bit).[20]
- macOS `OSByteOrder.h` macros (from/to BE and LE, up to 64-bit).

Files and filesystems

The recognition of endianness is important when reading a file or filesystem that was created on a computer with different endianness.

Some CPU instruction sets provide native support for endian byte swapping, such as `bswap`[21] (x86 - 486 and later), and `rev`[22] (ARMv6 and later).

Some compilers have built-in facilities for byte swapping. For example, the Intel Fortran compiler supports the non-standard `CONVERT` specifier when opening a file, e.g.: OPEN(unit, CONVERT='BIG_ENDIAN', …).

Some compilers have options for generating code that globally enable the conversion for all file IO operations. This permits the reuse of code on a system with the opposite endianness without code modification.

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 storing the data always in one fixed endianness, or carrying with the data a switch to indicate the endianness.

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.[23]

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.

As a consequence of its original implementation on the Intel 8080 platform, the operating system-independent File Allocation Table (FAT) file system is defined with little-endian byte ordering, even on platforms using another endiannes natively, necessitating byte-swap operations for maintaining the FAT.

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.[24]

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.[25]

However, not all protocols use big-endian byte order as the network order. The Server Message Block (SMB) protocol uses little-endian byte order. In CANopen, multi-byte parameters are always sent least significant byte first (little-endian). The same is true for Ethernet Powerlink.[26]

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.

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, HDLC, Ethernet, and USB. Some protocols use the opposite ordering (e.g. Teletext, I^2C, SMBus, PMBus, and SONET and SDH[27]). 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.

Notes

1. An exception to this rule is e. g. the Add instruction of the IBM 1401 which addresses variable-length fields at their low-order (highest-addressed) position with their lengths being defined by a word mark set at their high-order (lowest-addressed) position. When an operation such as addition was performed, the processor began at the low-order position of the two fields and worked its way to the high-order.
2. 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. Almost all machines which can do this using *one* instruction only (see section #Variable length data) are anyhow of type big-endian or at least mixed-endian. Therefore, for the criterion below to apply, the data type in question is assumed to be *numeric*.
3. Note that, in these expressions, the term "end" is meant as the extremity where the *big* resp. *little* significance is written *first*, namely where the object *starts*.

References

Citations

1. Swift, Jonathan (1726). *Gulliver's Travels* (https://en.wikisource.org/wiki/Gulliver%27s_Travels/Part_I/Chapter_IV).
2. Cohen, Danny (1980-04-01). *On Holy Wars and a Plea for Peace* (http://www.ietf.org/rfc/ien/ien137.txt). IETF. IEN 137. "…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 (https://ieeexplore.ieee.org/document/1667115).
3. "Internet Hall of Fame Pioneer" (http://internethalloffame.org/inductees/danny-cohen). *Internet Hall of Fame*. The Internet Society.
4. Cary, David. "Endian FAQ" (http://david.carybros.com/html/endian_faq.html). Retrieved 2010-10-11.
5. Tanenbaum, Andrew S.; Austin, Todd M. (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.
6. "NUXI problem" (http://catb.org/jargon/html/N/NUXI-problem.html). *The Jargon File*. Retrieved 2008-12-20.

7. 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).
8. 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. b5. 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."
9. Lunde, Ken (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.
10. "Cx51 User's Guide: E. Byte Ordering" (http://www.keil.com/support/man/docs/c51/c51_xe.htm). *keil.com*.
11. "Differences between BE-32 and BE-8 buses" (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0290g/ch06s05s01.html).
12. "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-processor-family.pdf) (PDF). Retrieved 2 May 2017.
13. Savard, John J. G. (2018) [2005], "Floating-Point Formats" (http://www.quadibloc.com/comp/cp0201.htm), *quadibloc*, archived (https://web.archive.org/web/20180703001709/http://www.quadibloc.com/comp/cp0201.htm) from the original on 2018-07-16, retrieved 2018-07-16
14. "pack – convert a list into a binary representation" (http://www.perl.com/doc/manual/html/pod/perlfunc/pack.html).
15. "C11 standard" (https://www.iso.org/standard/57853.html). ISO. Section 6.5.2.3 "Structure and Union members", §3 and footnote 95. Retrieved 15 August 2018. "95) If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called “type punning”)."
16. "3.10 Options That Control Optimization: -fstrict-aliasing" (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Type-punning). *GNU Compiler Collection (GCC)*. Free Software Foundation. Retrieved 15 August 2018.
17. Torvalds, Linus (5 Jun 2018). "[GIT PULL] Device properties framework update for v4.18-rc1" (https://lkml.org/lkml/2018/6/5/769). *Linux Kernel* (Mailing list). Retrieved 15 August 2018. "The fact is, using a union to do type punning is the traditional AND STANDARD way to do type punning in gcc. In fact, it is the *documented* way to do it for gcc, when you are a f*cking moron and use "-fstrict-aliasing" …"
18. *PDP-11/45 Processor Handbook* (http://bitsavers.org/pdf/dec/pdp11/handbooks/PDP1145_Handbook_1973.pdf) (PDF). Digital Equipment Corporation. 1973. p. 165.
19. `byteorder(3)` (http://man7.org/linux/man-pages/man3/byteorder.3.html) – Linux Programmer's Manual – Library Functions

20. endian(3) (http://man7.org/linux/man-pages/man3/endian.3.html) – Linux Programmer's Manual – Library Functions
21. "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.
22. "ARMv8-A Reference Manual" (http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.k_10775/index.html). ARM Holdings.
23. "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.
24. Matt Ahrens (2016). *FreeBSD Kernel Internals: An Intensive Code Walkthrough* (http://open-zfs.org/wiki/Documentation/Read_Write_Lecture). OpenZFS Documentation/Read Write Lecture.
25. 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. doi:10.17487/RFC1700 (https://doi.org/10.17487%2FRFC1700). STD 2. RFC 1700 (https://tools.ietf.org/html/rfc1700). Retrieved 2012-03-02.
26. Ethernet POWERLINK Standardisation Group (2012), *EPSG Working Draft Proposal 301: Ethernet POWERLINK Communication Profile Specification Version 1.1.4*, chapter 6.1.1.
27. 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)

Sources

- 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.

Further reading

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

External links

- Understanding big and little endian byte order (https://betterexplained.com/articles/understanding-big-and-little-endian-byte-order/)
- Byte Ordering PPC (https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/Concepts/ByteOrdering.html#//apple_ref/doc/uid/20001150-CJBEJBHH)
- Writing endian-independent code in C (https://developer.ibm.com/articles/au-endianc/)