The Origin and Anatomy of the Endian Wars

by Thomas Walker Lynch

2022-02-12

Abstract

Positive place value represented integers are fundamental to computing, because they are used for instruction counters, array indexes, and memory addresses, among other things. Place value representation and the accompanying arithmetic came to Europe from the Middle East during the middle ages; however, it arrived with a wrinkle. Arabic writing for text is right-to-left, while European writing is left-to-right. Translators chose to transcribe numbers 'as is' without reversing their digit order, thus in effect they reversed the digits of numbers with respect to the default scanning order used by the reader. As a direct consequence of using this convention for over a thousand years, when contemporary computer architects of European cultural heritage read computer dumps of memory, they find it more natural for numbers to print most significant digit first. However, as other architects point out, it is more consistent to have the weights of digits go from low to high at the same time addresses go from low to high, so these architects prefer that the least significant digit comes first. The debate goes on from there. This has lead to some subtle but profound compatibility issues among computers, standards, and even documentation. The analysis in this paper does seem to indicate that for positive integers that the least significant digit first convention does have advantages. However, the picture becomes more murkey when we add more complex number types, such as floating-point.

Introduction

The first computing machines which performed discrete state computation, as a opposed to analog computation, made use of ten symbols for digits. Examples include Pascal's calculator, Babbage's machine, the many mechanical calculators that came after, the Mark computers, and ENIAC. Hence the term *digital* initially carried with it the sense of both being discrete, and of making use of ten symbols.

The Atanasoff-Berry machine that was developed in the late 1930s used two state switch logic instead of ten index mark gear logic (or circular shift registers made from vacuum tubes). By the 1950s almost all discrete state computers used two state switch logic, yet the term *digital* continued to be used to describe them. This lead to some contradictions in terminology. For example, the company *Digital Equipment Corporation* only built two state switch logic computers. As another example a person who studies *digital electronics* will most likely only study two state switch logic, and thus never even see computation done using wheels or circular registers. What has happened is that the terms *digit* and *digital* continue to refer to computing with discrete states, but they have lost the connotation that there must be *ten* of those states.

Because there are two states in switch logic it is convenient to use a base two number system for arithmetic. In such a system each binary digit will have either the value zero, or the value one. Note, that the term *binary digit* is often shortened to *bit*. Today computing based on ten state digits, such as we see on calculators, is known as *decimal* computing. If we had a machine that made use of 256 state digits it would be neither decimal nor binary. Because we do not have a special name for the number 256, we would say that such a computer uses *base 256* digits and that it is a base 256 computer. Decimal computing, binary computing, and even base 256 computing are all examples of digital computing.

Decimal Place Value Representation

Now lets consider a special set of vectors. For each vector in this set, each component of said vector will be a single decimal digit. An example of such a vector is <7, 8, 9>, which has as its index zero component the digit 7, has as its index one component the digit 8, and of course has as its index two component the digit 9.

We may interpret such a vector as a place value number by using this function:

$\sum_{i=0}^{n} \boldsymbol{a}_i \cdot ten^i$; *gives meaning to a decimal place value represented number*

The symbol $\boldsymbol{a}$ is a digit vector where $\boldsymbol{a}_i$ is *i* th component of the vector, also known as the *i* th digit. If all the components of a given digit vector are zero, then the number is said to be zero. Here the *base* is the number *ten*. It is fortunate that *ten* is so well known that it has a name, because otherwise we might have been tempted to do what many other authors have done, and write the base while using the very same representation that we are trying to define.

Although this function gives numeric meaning to our digit vectors, actually performing the suggested computation is pointless, because the result would be a number, and we would have to represent that number, and that representation would be the very same vector that was input into the function in the first place.

Within a context where all the numbers will be digit vectors, we may, without causing ambiguity, drop the decoration we have been using to signify a vector. It is conventional that when dropping the decoration that we also reverse the order of the vector components. So the example vector from the prior paragraph, <7, 8, 9>, becomes 987. We call this writing convention *most-significant-digit-first*. This is because the first digit we write, the 9, is the one that is given the largest weight in the representation sum. Just to make sure we are on the same page, please note that the number 987 is only 13 away from a thousand.

We usually continue on from this definition to talk about place value numbers in *normal form*. For normal form we consider that zero has a special symbol of its own. Then for numbers that are not zero, we require that the index n be the largest index for which $\boldsymbol{a}_n$ is not zero. This is the same as saying we drop all the 'leading' zeros. When we have a number in normal form, $\boldsymbol{a}_0$ is called the *least-significant-digit*, because it is the digit with the smallest weight in the representation function. The digit $\boldsymbol{a}_n$ is then the *most-significant-digit* because it is the digit with greatest weight in the representation function. It follows that $\boldsymbol{a}_0$ can be zero, but $\boldsymbol{a}_n$ can not be zero. It is possible that n turns out to be zero, in which case the least-significant-digit will be the same digit as the most-significant-digit. Such a number would have only one digit in its normal form.

The One True Order

Had I written this paper in Arabic the text would have been written right-to-left. When writing from right-to-left our initial vector with its decorations would reversed from the prior example, but as per our more than thousand year old convention, the digits in the number would not be reversed, so we would get something like this:

.987 ot seifilpmis <9 ,8 ,7> rotcev eht ,tfel ot thgir morf gnitirw nehW

Now imagine, we start with this, as the early translators did, but instead of making an exception for numbers, we literally reverse the entire string. Then our vector becomes <7, 8, 9> and without decoration it becomes 789. We call this writing convention *least-significant-digit-first*. Note, this convention has the nice property that the digits appear in the same order with or without the decoration that signify it is a vector. Independent of our

writing conventions we are still talking about the same number. Hence, while using the least-significant-digit-first writing convention, we notice that the number 789 is still only 31 away from a thousand.

Least-significant-digit-first representation of numbers is what the inventors of the place value number system intended. Arabs today are still right-to-left writers and thus *they are still writing numbers least-significant-digit-first*. When an Arabic reader scans across a line on a page and sees the first digit of a number, the reader immediately knows this first digit is in the one's place.

In contrast, when a European reader scans across a line of text the first digit of a number he or she sees will be of unknown weight. The European reader then must continue scanning to the right to find the last digit scanned, and then he or she will know that this digit at the far right will be in the one's place. The European reader may then read the number right-to-left just as an Arab reader would, while knowing the weights of the digits. Essentially, out of practical necessity European readers acknowledge that numbers are written backwards for them, and that the only true digit order is least-significant-digit-first.

The same happens when European readers perform the most common arithmetic operations such as addition or multiplication. Fact is, starting with the least-significant-digit first is *an artifact of the place value number system*, so it was never an option for the translators to change this.

If per chance we desired to fix this error and scan numbers least-significant-digit-first as originally envisioned and practiced, we European culture writers would either have to start writing from right-to-left like the Arabs do, or we would have to reverse the order of digits in numbers.

Given the burden of over a thousand years of convention you might wonder why anyone would go through the trouble of fixing this now? Why even talk about this at all? Well as it turns out, internally many *computers* follow the Arabic style least-significant-digit-first convention of writing numbers. These computers scan text characters and numbers in their internal memory in the same address order while going from small addresses to larger addresses. In this way the one's digit is consistently found at the lower address, independent of the precision of the number. I.e. computers do this for the same reason that the inventors of place value representation originally put least significant digits first.

However not all computers do this. Some computers do as European left-to-right authors do, and index digit vectors in the opposite direction from how they index other objects in memory. These machines do this for a couple of reasons, among them, however, is that printouts of byte streams are more natural for European readers to make sense of. For

example, when doing a hex dump of an IP packet starting with the first byte transmitted on the left, then the next byte, etc. working right, the source and destination IP numbers may be read directly from the hex dump without having to reverse the four quads.

It is not just hardware that is a mixed up, so are standards specifications and explanatory documents.

Binary, Octal, Hexadecimal, BCD

We may interpret a vector of n bits as a number by using this function:

$\sum_{i=0}^{n} \boldsymbol{a}_i \cdot two^i$; *gives meaning to a binary place value represented number*

Just as for vectors of decimal digits we may drop the vector decorations on bit vectors to give us strings of bits. Just as for decimal digit strings, we have two options for writing the string, either most-significant-digit-first, or least-significant-digit-first. These are also known as most-significant-*bit*-first and least-significant-*bit*-first.

Although both strings of binary digits and strings of decimal digits grow logarithmic against a count, binary strings will grow more than twice as fast. The expansion to a length two string will occur immediately at the count of two. Then length expands to three at a count of four. Then to string length four at a count of eight. Hence, while incrementing to eight, the binary digit string has already expanded to length four yet the decimal digit string hasn't yet budged from being one digit long.

So as to make working with binary strings shorter and more convenient, we typically treat bits in groups. When we group bits in threes, we will then actually be working in base eight. This is called octal. We use one of the symbols 0, 1, 2, ... 7 for each octal digit.

octal	binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

In the 1960s it was not uncommon to find computer panels with switches and lights organized in threes, and for coding forms to be filled out in octal. However, today almost all documents that must show bit strings will use groups of fours. This is the hexadecimal system. It has sixteen base symbols. After reaching 9, we just use letters.

octal	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
a	1010
b	1011
c	1100
d	1101
e	1110
f	1111

It is not a coincidence that the table of hexidecimal digits is twice as long as the table of octal digits. Each time we add another bit to the grouping, the table will double in size.

The memory of a computer can be thought of as a very large array of bits. All contemporary computers will present an architectural model to the programmer where groups of bits called *bytes* are addressable. Bytes are most commonly groups of eight bits; however, computers of the past have used other values, such as seven or twelve bits. The designers of UTF-8 wanted to make clear that they considered groups of eight, so they named such a group an *octet* rather than a byte. When a group of eight bits, i.e. an octet, is considered as a digit of a number, we will be working in base 256 (here the number 256 is given in decimal representation. In hexadecimal representation it would be written as 100.).

Groups get larger than this. The organization of early RISC microprocessors was such that memory was always moved in groups of 32 bits, called *words*. Today it is common that address variables will be 64 bits while integer variables will be either 32 or 64 bits. Groups found on internal buses can be larger yet.

We can also use groups of bits to create numbers that have bases that are not powers of two. For example, in the BCD codes we group bits in fours to create decimal digits.

BCD	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

In BCD some encodings for the group of 4 binary bits are simply not used. Although this code is less efficient, it still grows logarithmically with a count. At least when using this encoding our numbers will be represented in base ten, so no number conversation has to be done to print such numbers for human readers to make sense of them. They also divide by ten without creating infinite fractions. It was not long ago that programmers who wrote business programs preferred this encoding scheme. Note, Computers that support BCD arithmetic are still implemented using binary switch logic.

Allocation

In most text documents a number is written down once and that is it. In contrast, while computing we often come back and change a given value many times. This would be equivalent to going to a paper document and repeatedly erasing the old value, and then writing a new value in the same space.

Consider the case of recording a count in real time, where at any moment we might have to erase the current count value, and then to write a new one. Suppose we begin the count with a single digit of 0. We then increment to 1. We still have a single digit. Our count will grow to two digits in length upon reaching the count of *ten*. As we continue the count will grow by another digit in length when we reach a *hundred*. It grows again when we reach a *thousand*, etc. It is probably not a coincidence that we have names for the numbers at these boundaries. Thus a digit vector that represents the count grows in a discrete manner against the log of the count value. The log function grows without bound, but it does so quite slowly, so the net effect is that relatively short digit vectors can be used to represent rather large numbers. This observation turns out to be key.

If we had a paper document and only left space for a single digit count, then we would run into a problem when the count grows to ten, and an even a bigger problem when it eventually grows to a thousand. If we knew in advance that we would want work our way up to counts in the thousands, we could simply *allocate* enough space for four digits to start with. Such an allocation by itself is not much of a waste of space because the digit vectors for very large numbers are not much longer than those for small numbers. Yet, for any choice we make for amount of space to allocate, the possibility remains that a number will come along that is so large that its representation doesn't fit in the allocation, i.e. that the number will *overflow* the allocation. As any child knows who has tried to count to a million, at some point this issue becomes moot. Basically to make reasonable allocations we have to know how much numbers grow with the operations applied to them, and how many operations will be applied. This latter question could well be related to how long we plan to keep working on a given problem.

Just as numbers require space to be written on a page of print, they also require space to be written into computer memory. The conventional solution is to assume that all numbers will fit into fixed length allocations called *words .* Then a map is made of the location of all the words in memory even before the program begins. Because the locations of the words are known in advance, these locations may be placed directly into the instructions of the program by the compiler, the linker, or even the loader.

As we know the initial values of the numbers the program uses, when the program begins running all numbers fit in the words they have been allocated into. However, as the program runs, it is possible that larger and larger values will need to be stored. This might not happen, but depending on the details of the program, it can happen.

In general it is not possible to write a program that can analyze another program and determine how long it runs. Even in specific cases where it is possible to succeed at such an analysis, the analysis itself might be so difficult and time consuming that the programmer does not see it as a worth while exercise. In point of fact, nobody does such an analysis unless they plan to put the code into a life critical situation, and often times, not even then. Most programmers would not even know how to go about doing it anyway. Consequently computer users live under the sword of Damocles while not knowing if an allocation will overflow and cause an error, and really, not knowing there is even such an issue. When this problem does come up, it is simply reported as a bug. In a sense, programmers let the users do the analysis for them.

Each fixed length word of allocation will contain multiple bytes of data, where each byte has an address. Hence there will be a byte with the smallest address, and a byte with the largest address. All memory managers to date use the smallest address as the address of the allocation itself.

Now we come to the question: what order will the digits of a number be written into a word? Consider the convention where the smallest address byte of a word is written on the left, with addresses growing as we move right. The two most common conventions are then, you guessed it, least-significant-digit on the left, or least-significant-digit on the right.

Let us say that 'a numbers starts' with its least-significant-digit. This nomenclature is justified based on our observation that independent of the writing scheme, we will have to scan a number starting with its least-significant-digit. Accordingly, the allocation scheme where the number starts on the left, and runs to the right, is called *little-endian* because the number starts at the smallest address of the word. In contrast, the allocations scheme where the number starts on the right, and runs leftward, is called *big-endian* because the numbers starts at the biggest address of the word. [Cohen]

In Figure 1 we see a word that is addressed from hexidecimal *c0* to *c3.* In decimal those addresses would be written as 192, 193, 194, 195; however, we almost always use hexadecimal notation for addresses. The address of the byte before c*0* is *bf*. The address after *c3* is *c4,* as shown. The address for the word itself is taken to be *c0*, because the address for an allocation is always that of the smallest address appearing in the allocation. If we consider that a digit of the number is a byte, and bytes are octets, the binary encoding for the least significant digit of this number is 0001 1000. The most significant digit is 1010 1110.

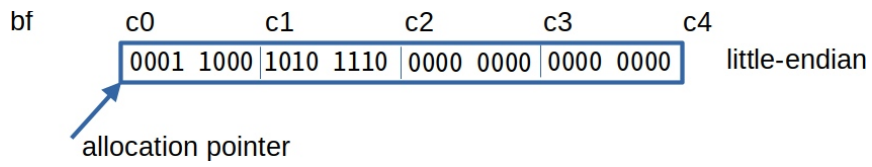

Figure 1: Number in word using little-endian digit order

For big-endian digit order the number is placed into the word going in the opposite direction. For all but very large numbers, the digit being pointed at by the allocation pointer will be zero. We will continue to scan zeros until either reaching the end of the allocation or upon reaching the most significant digit. If we reach the end of the allocation, the contained number is taken to be zero. The following figure shows the same number as in the prior figure, but using big-endian byte order. Because it is the same number, it also has the same least significant digit and the same most significant digit.

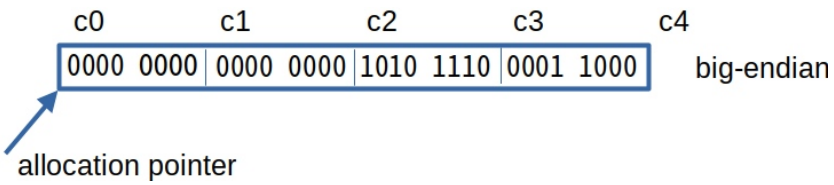

Figure 2: Same number in word using big-endian digit order

Now suppose our word is holding a count. This is the same as saying that the word is a counter. When counting with the little-endian convention, a number will grow into larger addresses as the count carries into new digits. In contrast, with big-endian, counting will carry into ever smaller memory addresses.

Now consider what happens when a number grows and overflows the allocation. In the little-endian case fixing an overflow when incrementing would conceptually require expanding the allocation the number is held in. Something becoming too long for its allocation is a common enough problem that some heap libraries include a function that attempts to expand an allocation. In contrast, in the big-endian case an overflow will run over the allocation pointer, so we can not fix this problem by just calling a library routine to expand the allocation. One trick we might try would be to expand the allocation and then slide the number right to make space for another digit. When using such a scheme the program that loads the number will have to take into account the size of the adjusted allocation to know the weight of each digit.

Hence, the little-endian allocation convention fits our thinking about data and allocation better than does the big-endian convention. As we noted earlier little-endian words will print least-significant-digit first, which to European readers will see as being backwards. If perchance a clever dump program prints data from large address to small address going left-to-right (i.e. backwards), digits will appear in a natural order for European readers, but text would be backwards. If instead we use the big-endian allocation scheme, a print out of memory will show everything in the expected order for the European reader. Though, is

‘looking natural’ in memory dumps a good metric for basing architecture decisions on? After all, the end user will never look at a memory dump, and architects should be capable of the required mental gymnastics.

This issue with memory allocation and overflow is not generally discussed today because when a number gets too large for its allocation, no matter the digit order convention, the overflow will either be ignored thus leading to run time errors, or result in an exception being signaled. Either way, the programmer will not be aware of the direction of growth in the number that lead up to the allocation overflow event.

Conventionally processor registers are also one word in length, and a processor will load the entire contents of a word of memory into a register as a single operation. Consequently the contemporary processors do not care how many digits are in a number held within said word. However, the ALU would impose an order on the digits. Take for example, when a register is incremented, the increment must be inserted at the least-significant-digit end.

Suppose we had an unconventional processor, or perhaps a future processor, that loaded numbers as digit streams. The load instruction would have to have a means of detecting the end of a number being loaded, or it would somehow have to know a-priori how long the number is. Such a problem is nearly identical to the already existing problem of loading strings of characters. Both length counts and end terminators have been used for solving that problem of loading character streams.

Bit Order Within Bytes

Data is transported between different points within a processor or computer system over bundles of wires called buses. Buses have specifications that include the order of bits in bytes, and for all contemporary machines bytes are octets. All compute processors, channel processors, and other devices plugged into a bus must conform to the bus’s specifications. Interface devices that bridge between the system bus and a storage device will arrange to write the storage devices as per its specifications. Typically this is done by first bridging to a standard storage device bus, then later bridging to the storage device itself. In which case the processor interface can be designed against a storage bus standard, while the storage devices will be also be designed against the storage bus standard.

Hence bit order issues occur only at the layer below the standards and specifications for major components of the computer. Unless a programmer is involved in hardware design, perhaps by programming firmware, he or she will not even be aware of the order of bits within a byte.

This is not to say there can't be differences. In fact some processors do store bits into memory bytes in different order than others. However, the values that are read and written into the respective memories must be placed on a bus, and at that point the bit order is described by the relevant specification.

On all contemporary computers it is possible to perform arithmetic on bytes. Carries always travel from the lower significant bits to more significant bits, consequently ALUs also impose a bit order within bytes. However, that order will conform to the order documented for the processor's bus.

I would challenge any programmer who is not involved in hardware design to try and write a C program that detects the physical order of bits in bytes of memory, or to identify a difference in bit order within bytes between computers.

Byte Order Within Words

Common word sizes include 2, 4, 8, and even 16 bytes. A given processor architecture might simultaneously have native support for byte data and various lengths of words. The most common word lengths being 16, 32, and 64 bits, aka 4, 8, and 16 bytes, where each byte is an octet, i.e. a group of 8 bits.

Most all communication channels and storage devices have payloads organized as octets, *but they have no built in support for words*. Hence when we desire to store a word or to communicate a word, we will be forced to first serialize the word into a series of bytes, then transfer or store the data, then to read the data back while deserializing the byte stream back into words.

Place value numbers consist of vectors of digits. Words of allocation consist of consecutive addresses bytes. Bytes are handled atomically by all modern machines. Hence, any bit encoding for the digits of a number must pack cleanly into bytes, or the digits themselves might be split apart. Such a clean packing might require padding with zeros to make the length of the number to be even by 8 bits. For example, a hexadecimal string of digits might be 12 bits long. Another 4 zeros must be appended to make it a multiple of 8 bits. When this criteria is met, we suffer no loss of generalization by considering that a number is a series of bytes as digits. It is because of this fact that little-endian and big-endian are often referred to as byte orders. When this is done we are considering the bytes within a number, i.e. thinking in terms of byte digits.

The following figure shows a stream of bytes arriving as data, and then that data being copied into a word. In this case the digits of the word, the bytes, arrive in little-endian order and they are received on a little-endian machine, so they are just copied in the order they are scanned off the channel.

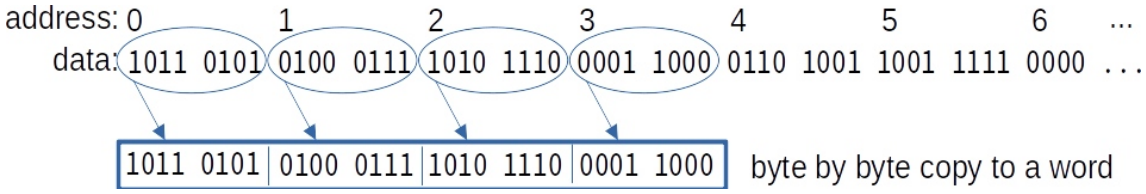

Figure 3: In address order byte by byte copy

In this second case the same stream of data arrives with words serialized as bytes in little-endian order, but this time the receiving machine is big-endian. The bytes will have to be reversed on a word by word basis.

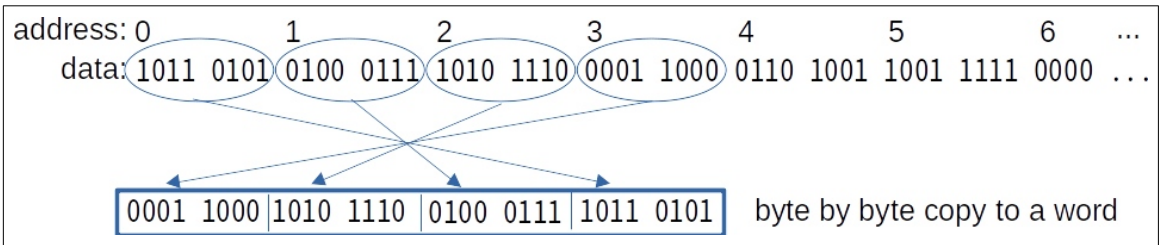

Figure 4: Reverse order byte copy for words

Here is the problem, when the data arrives there is no way to know where the words are. That information was lost when we serialized. If we do not know where the words are we can not know when to do the byte order reversal. Hence this problem can not be solved using the same approach that was used for solving the problem of bit order in bytes.

At some level of abstraction all applications must be designed to be able to make sense of their data. It is just too bad that byte order within words has to be an application level design consideration, because it has nothing to do with applications.

In the case of something like the Internet Protocol, IP, the specification dictates the offset as to where data is to be placed into the packet header. Coding is done in a stable and efficient manner. However IP does not know where the words are located in the data payload it is carrying, so IP just passes the payload, byte order and all, up the abstraction stack.

JSON is a standard for expressing tagged structured data built from primitive types while using character only data. It comes with a specification saying how numeric character strings are to be interpreted. This makes it so that both little-endian and big-endian machines can share JSON character encoded numeric data. JSON is not a very stable means of encoding data and it is inefficient, but it has one great thing going for it. Namely, it works.

There are many file formats and data communications standards that serve various classes of applications by making it clear where words that need to be reversed are located when data is transferred between machines.

IP Packet Example

This choice because it looks natural to European readers.

Fractions

fp we still have to read the exponent

other operations

Survey

something for people not familiar with left-to-right right-to-left writing conventions

ENAIC using 10 values in a circle would be a good reference.

https://link.springer.com/book/10.1007/978-3-030-66599-9

who is responsible for shortening binary digit to bit? Mention Shannon for switch logic. Really we had to wait for Atanasoff to connect switch logic to arithmetic?

In 1980 this situation lead Danny Cohen to write the missive, “ON HOLY WARS AND A PLEA FOR PEACE” where he begged the industry to adopt one convention or the other. The terms little-endian and big-endian refer to the Swifts Gulliver’s travels war on which end of the egg is broken.

At the level of bits within bytes Danny Cohen has gotten his wish.

Something with machine endian conventions

https://www.stanislavs.org/helppc/byte_ordering.html

Architecture	16 Bit Integer	32 Bit Format	Floating Point
MC68000	MSB	MSB	MSB
Intel	LSB	LSB	LSB
PDP-11	LSB	MSW...LSW	MSW...LSW
VAX	LSB	LSB	MSW...LSW
IBM 360/370	MSB	MSB	MSB

MSB means Most Significant Byte first or a byte order of 3210
LSB means Least Significant Byte first or a byte order of 0123
MSW...LSW means a byte order of 3201 or 67452301

See BIBLIO reference "Computer Language Magazine", April, 1987,
P.J. Plauger for more information

https://devopedia.org/byte-ordering

~~Conclusion

Cohen called for peace. Let me propose a way to get that peace. Let us stop reproducing an error made by scribes in the 9th Century AD, and simply ditch big-endian and go back to the original vision for place value represented integers. That will move the problem of making numbers look pretty for European readers to the print function, but that function already has this purpose, and it will be possible to do it in the print function because that function will know what order the bytes are in. When we do this numbers that grow under the application of the most common operations such as incrementing, adding, and multiplying, will bump into the ends of their allocation blocks, not the beginnings of them. Characters and numbers will both scan going from little addresses to big addresses. And most importantly there will be no need to bind ‘where the words are’ information back into the low level reading of data from storage or communication channels. The net result will be that byte order in words will become a non-issue just like bit order in bytes.