Understanding Endianness

A Comprehensive Guide

Mathematical and Practical Analysis

For Computer Architecture and Cryptography

July 31, 2025

This document provides a comprehensive understanding of endianness for educational purposes.
For cryptographic implementations, always refer to official specifications and test vectors.

Contents

Introduction

Endianness is a fundamental concept in computer architecture and cryptography that determines how multi-byte data is stored and interpreted in memory. This document provides a mathematical and practical understanding of endianness, its implications, and real-world applications.

> **Endianness** refers to the order in which bytes are stored in memory for multi-byte data types (like 32-bit integers, 64-bit floats, etc.). The term originates from Jonathan Swift's "Gulliver's Travels," where the Lilliputians argued over which end of a boiled egg to crack first.

Mathematical Definition

For a multi-byte value stored in memory, endianness determines the mapping between:

- **Logical position**: The significance of each byte in the value
- **Physical position**: The actual memory address where each byte is stored

Types of Endianness

Big-Endian (Most Significant Byte First)

> **Definition**: The most significant byte (MSB) is stored at the lowest memory address.

Mathematical representation

For a 32-bit integer with value `0x12345678`:

Memory Address 0x1003	0x1000	0x1001	0x1002
Byte Value 0x78	0x12	0x34	0x56

Formula:
For a value V with bytes b_0, b_1, b_2, b_3 where b_0 is the MSB:
$V = b_0 \times 2^{24} + b_1 \times 2^{16} + b_2 \times 2^8 + b_3 \times 2^0$

Little-Endian (Least Significant Byte First)

> **Definition**: The least significant byte (LSB) is stored at the lowest memory address.

Mathematical representation

For a 32-bit integer with value `0x12345678`:

Memory Address 0x1003	0x1000	0x1001	0x1002
Byte Value 0x12	0x78	0x56	0x34

Formula:
For a value V with bytes b_0, b_1, b_2, b_3 where b_0 is the LSB:
$V = b_3 \times 2^{24} + b_2 \times 2^{16} + b_1 \times 2^8 + b_0 \times 2^0$

Step-by-Step Examples

Example 1: 32-bit Integer Conversion

Value: `0xDEADBEEF`

Big-Endian Representation:

Memory Layout:

Address: 0x1003	0x1000	0x1001	0x1002
Bytes: 0xEF	0xDE	0xAD	0xBE

Calculation:
$V = 0xDE \times 2^{24} + 0xAD \times 2^{16} + 0xBE \times 2^8 + 0xEF \times 2^0$
$V = 222 \times 16,777,216 + 173 \times 65,536 + 190 \times 256 + 239 \times 1$
$V = 3,723,914,752 + 11,337,728 + 48,640 + 239$
$V = 3,735,301,359 = 0xDEADBEEF$

Little-Endian Representation:

Memory Layout:

Address: 0x1003	0x1000	0x1001	0x1002
Bytes: 0xDE	0xEF	0xBE	0xAD

Calculation:
$V = 0xEF \times 2^{24} + 0xBE \times 2^{16} + 0xAD \times 2^8 + 0xDE \times 2^0$
$V = 239 \times 16,777,216 + 190 \times 65,536 + 173 \times 256 + 222 \times 1$
$V = 4,009,754,624 + 12,451,840 + 44,288 + 222$
$V = 4,022,249,974 = 0xEFBEADDE \neq 0xDEADBEEF$

Example 2: 16-bit Integer Conversion

Value: `0x1234`

Big-Endian:
Memory: `0x12 0x34`
Value: $0x12 \times 256 + 0x34 \times 1 = 4,660 = 0x1234$

Little-Endian:
Memory: `0x34 0x12`
Value: $0x34 \times 256 + 0x12 \times 1 = 13,330 = 0x3412 \neq 0x1234$

Why Endianness Matters

Data Portability

Problem: Different architectures use different endianness:

- **Big-Endian**: Motorola 68000, IBM PowerPC, most network protocols
- **Little-Endian**: Intel x86, ARM (configurable), most modern processors

Example: A file created on a big-endian system may be unreadable on a little-endian system.

Network Communication

Problem: Network protocols must standardize byte order for interoperability.
Solution: Network byte order is typically big-endian (also called "network byte order").

Cryptographic Applications

Critical Issue: Cryptographic algorithms are sensitive to byte order. Incorrect endianness can:

- Generate incorrect keys
- Produce wrong ciphertext
- Break security properties

Pros and Cons

Big-Endian
Pros:

- **Human-readable**: Memory dumps match written representation
- **Network standard**: Most network protocols use big-endian
- **Mathematical consistency**: Left-to-right reading matches significance

Cons:

- **Less efficient**: On little-endian hardware, requires byte swapping
- **Counter-intuitive**: For arithmetic operations on little-endian processors

Little-Endian
Pros:

- **Arithmetic efficiency**: Addition/subtraction can start from LSB
- **Hardware compatibility**: Matches most modern processor architectures
- **Memory efficiency**: No byte swapping needed on little-endian hardware

Cons:

- **Less intuitive**: Memory dumps don't match written representation
- **Network overhead**: Requires conversion for network communication

Practical Applications

File Format Design

Example: PNG image format uses big-endian for all multi-byte values:

```
uint32_t length = (bytes[0] << 24) | (bytes[1] << 16) |
                  (bytes[2] << 8) | bytes[3];
```

Listing 1: PNG chunk length (4 bytes

Network Programming

Example: Converting between host and network byte order:

```
// Host to network (little-endian to big-endian)
uint32_t htonl(uint32_t hostlong);

// Network to host (big-endian to little-endian)
uint32_t ntohl(uint32_t netlong);
```

Listing 2: Host and Network Byte Order Functions

Cryptographic Implementations

Critical: Cryptographic algorithms must specify byte order explicitly.

Example: SHA-256 specification defines big-endian representation:

```
uint32_t word = (data[0] << 24) | (data[1] << 16) |
                (data[2] << 8) | data[3];
```

Listing 3: Correct SHA-256 implementation (big-endian)

Endianness Detection

Runtime Detection

```
bool isLittleEndian() {
    uint16_t test = 0x0001;
    return (*(uint8_t*)&test == 0x01);
}
```

Listing 4: Runtime Endianness Detection

Compile-time Detection

```
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    #define IS_LITTLE_ENDIAN 1
#else
    #define IS_LITTLE_ENDIAN 0
#endif
```

Listing 5: Compile-time Endianness Detection

Conversion Functions

Manual Conversion

```
uint32_t swapEndian32(uint32_t value) {
    return ((value & 0xFF000000) >> 24) |
           ((value & 0x00FF0000) >> 8) |
           ((value & 0x0000FF00) << 8) |
           ((value & 0x000000FF) << 24);
}
```

Listing 6: Manual Endianness Conversion

Using Built-in Functions

```
// GCC/Clang built-in
uint32_t swapped = __builtin_bswap32(value);

// Windows
uint32_t swapped = _byteswap_ulong(value);
```

Listing 7: Built-in Endianness Conversion

Real-World Example: Cryptographic Key Loading

Problem Scenario

Loading a 256-bit key for Serpent cipher:

```
// Key: 2BD6459F82C5B300952C49104881FF48
// Expected 32-bit words (big-endian):
// W[0] = 0x2BD6459F
// W[1] = 0x82C5B300
// W[2] = 0x0952C491
// W[3] = 0x04881FF4
```

Listing 8: Key Loading Example

Correct Implementation

```
UInt32Vector loadKeyBigEndian(const BytesVector& key) {
    UInt32Vector words;
    for (int i = 0; i < key.size(); i += 4) {
        uint32_t word = (key[i] << 24) | (key[i+1] << 16) |
                        (key[i+2] << 8) | key[i+3];
        words.push_back(word);
    }
    return words;
}
```

Listing 9: Correct Big-Endian Key Loading

Incorrect Implementation (Little-Endian)

```
UInt32Vector loadKeyLittleEndian(const BytesVector& key) {
    UInt32Vector words;
    for (int i = 0; i < key.size(); i += 4) {
        uint32_t word = key[i] | (key[i+1] << 8) |
                        (key[i+2] << 16) | (key[i+3] << 24);
        words.push_back(word);
    }
    return words;
}
```

Listing 10: Incorrect Little-Endian Key Loading

Result: The little-endian version produces completely different words, leading to incorrect cryptographic operations.

Best Practices

Always Specify Endianness

```
// Good: Explicit endianness
uint32_t readBigEndian32(const uint8_t* data) {
    return (data[0] << 24) | (data[1] << 16) |
           (data[2] << 8) | data[3];
}

// Bad: Platform-dependent
uint32_t read32(const uint8_t* data) {
    return *(uint32_t*)data; // Endianness depends on platform
}
```

Listing 11: Explicit vs Platform-dependent Endianness

Use Standard Functions

```
// Network byte order functions
uint32_t networkValue = htonl(hostValue);
uint32_t hostValue = ntohl(networkValue);
```

Listing 12: Network Byte Order Functions

Document Assumptions

```
// Document expected endianness
/**
 * Loads a 256-bit key in big-endian format.
 * Key bytes are interpreted as: [MSB][MSB-1]...[LSB+1][LSB]
 */
```

Listing 13: Documenting Endianness Assumptions

Common Architectures and Their Endianness

Architecture	Endianness	Notes
Intel x86/x64	Little-Endian	Most common desktop/server processors
ARM	Configurable	Usually little-endian by default
Motorola 68000	Big-Endian	Classic Macintosh processors
IBM PowerPC	Big-Endian	Older Macintosh and some servers
MIPS	Configurable	Can be either endian
Network Protocols	Big-Endian	Standard for interoperability

Table 1: Common Architectures and Their Endianness

Mathematical Analysis

Binary Representation

For a 32-bit integer, the binary representation shows the significance of each bit:

Bit Position	31	30	29	...	2	1	0
Value	2^{31}	2^{30}	2^{29}	...	2^2	2^1	2^0

Byte-Level Analysis

In big-endian, the most significant byte contains the highest-order bits:

$$Value = byte_0 \times 2^{24} + byte_1 \times 2^{16} + byte_2 \times 2^8 + byte_3 \times 2^0$$

In little-endian, the least significant byte is stored first:

$$Value = byte_3 \times 2^{24} + byte_2 \times 2^{16} + byte_1 \times 2^8 + byte_0 \times 2^0$$

Performance Considerations

Memory Access Patterns

Little-endian systems can be more efficient for:

- Arithmetic operations starting from least significant digits
- Variable-length integer operations
- Memory-mapped I/O where byte order matters

Big-endian systems are more efficient for:

- Network packet processing
- Human-readable debugging
- String comparisons

Conclusion

Endianness is a critical concept that affects data interpretation, network communication, and cryptographic implementations. Understanding the differences between big-endian and little-endian representations is essential for:

- Writing portable code
- Implementing network protocols
- Developing cryptographic algorithms
- Debugging cross-platform issues

> **Key Takeaway**: The key is to always be explicit about byte order and use appropriate conversion functions when necessary. In cryptographic applications, incorrect endianness can lead to security vulnerabilities, making this understanding particularly important for security-critical code.

This document provides a comprehensive understanding of endianness for educational purposes. For cryptographic implementations, always refer to official specifications and test vectors.