Home > Tutorials > C/C++ > Endian Byte Order

Endianness: Big and Little Endian Byte Order

Big and little endian hardware store in memory their Most Significant Bytes (MSB) and Least Significant Byte (LSB) in an order opposite from each other. Thus data exchange between big and little endian systems, including translation to the network big endian byte order, often requires endian conversion achieved by byte swapping the data. This only applies to binary data values and not to text strings.

Tutorial Table of Contents:

- # Processor Endianness and Data Representation
- # Conversion and Swapping Bytes
- # Byte Swapping Libraries
- # Bit Field Conversions
- # Tests and Examples
- # Compilers and Programming Languages
- # Tools

Search | Home Page | Linux Tutorials | Terms | Privacy Policy | Advertising | Contact |

Related YoLinux Tutorials:

°Linux and C++
°C++ Unions & Structures
°C++ Templates
°C++ STL
°C++ STL Map
°C++ String Class
°C++ Singleton
°C++ Coding Style
°C++ XML Beans
°C/C++ Dynamic Memory
°C++ Memory Corruption
°C/C++ Signal Handling
°C++ CGI
°Google C++ Unit Test
°Jenkins CI
°Jenkins Plugins for C++
°YoLinux Tutorials Index

Processor Endianness and Data Representation:

Computer processors store data in either large (big) or small (little) endian format depending on the CPU processor architecture. The Operating System (OS) does not factor into the endianess of the system. Big endian byte ordering is considered the standard or neutral "Network Byte Order". Big endian byte ordering is in a form for human interpretation and is also the order most often presented by hex calculators.

Processor	Endianness
Motorola 68000	Big Endian
PowerPC (PPC)	Big Endian
Sun Sparc	Big Endian
IBM S/390	Big Endian
Intel x86 (32 bit)	Little Endian
Intel x86_64 (64 bit)	Little Endian
Dec VAX	Little Endian
Alpha	Bi (Big/Little) Endian
ARM	Bi (Big/Little) Endian
IA-64 (64 bit)	Bi (Big/Little) Endian
MIPS	Bi (Big/Little) Endian

Bi-Endian processors can be run in either mode, but only one mode can be chosen for operation, there is no bi-endian byte order. Byte order is either big or little endian.

Network Byte Order:

Big endian byte ordering has been chosen as the "neutral" or standard for network data exchange and thus Big Endian byte ordering is also known as the "Network Byte Order". Thus Little Endian systems will convert their internal Little Endian representation of data to Big Endian byte ordering when writing to the network via a socket. This also requires Little Endian systems to swap the byte ordering when reading from a network connection. Languages such as Java manage this for you so that Java code can run on any platform and programmers do not have to manage byte ordering.

It is important to observe Network Byte Order not just to support heterogeneous hardware but also to support heterogeneous languages.

Byte order and data representation in memory:

Big endian refers to the order where the most significant bytes come first. This means that the bytes representing the largest values come first. Regular integers are printed this way. The number "1025" shows the numeral one first which represents "1000". This is a representation most comfortable to

humans. This most significant value first is represented in bytes for computer memory representation. The number 1025 is represented in hex as 0x0401 where 0x0400 represents 1024 and 0x0001 represents the numeral 1. The sum is 1025. The most significant (larger value) byte is listed first in this big endian representation.
One can see that the word size is a factor as well as it determines how many bytes are used to represent the number.

Endian byte ordering affects integer and floating point data but does not affect character strings as they maintain the string order as viewed and intended by the programmer.

- Decimal: 1025
  16 bit representation in memory:
  - **Big Endian:**
    Hex: 0x0401
    binary: 00000100 00000001
  - **Little Endian:**
    Hex: 0x0104
    binary: 00000001 00000100

  32 bit representation in memory:
  - **Big Endian:**
    Hex: 0x00000401
    binary: 00000000 00000000 00000100 00000001
  - **Little Endian:**
    Hex: 0x01040000
    binary: 00000001 00000100 00000000 00000000
- Decimal: 133124
  32 bit representation in memory:
  - **Big Endian:**
    Hex: 0x00020804
    binary:

00000000	00000010	00001000	00000100
bits 31-25	bits 24-16	bits 15-8	bits 7-0

  - **Little Endian:**
    Hex: 0x04080200
    binary:

00000100	00001000	00000010	00000000
bits 7-0	bits 15-8	bits 24-16	bits 31-25

- Decimal: 1,099,511,892,096
  64 bit representation in memory:
  - **Big Endian:**
    Hex: 0x0000010000040880
    binary: 00000000 00000000 00000001 00000100 00000000 00000100 00001000 10000100
  - **Little Endian:**
    Hex: 0x8008040000010000
    binary: 10000100 00001000 00000100 00000000 00000100 00000001 00000000 00000000

Note that the bytes representing the entire number are swapped. Also note that only the bytes are reversed and the bits within the byte are NOT reversed.

Conversion and Swapping Bytes:

Byte swapping to convert the endianness of binary data can be achieved using the following macros, routines or libraries. The Java virtual machine operates in big endian mode on all platforms and thus is often immune from processor architecture effects. Data files however often suffer the effects of hardware. Even file format standards can be affected, for example, data files such as JPEG are stored in Big Endian format while GIF and BMP are stored in Little Endian format.

Integers, floating point data (modern systems) and bit field data are byte swapped to convert between big and little endian systems. ASCII text is not. This is due to the internal representation of numerical data and the method in which it is processed. Note that floating point data representation between an older system and a newer IEEE floating point based processor requires format conversion before byte order is even a consideration (eg float transfer between System 390 mainframes and Intel based systems).

C macros to swap bytes on little endian systems:

```
#include <endian.h>

#if __BYTE_ORDER == __BIG_ENDIAN
// No translation needed for big endian system
#define Swap2Bytes(val) val
#define Swap4Bytes(val) val
#define Swap8Bytes(val) val
```

```
#else
// Swap 2 byte, 16 bit values:

#define Swap2Bytes(val) \
 ( (((val) >> 8) & 0x00FF) | (((val) << 8) & 0xFF00) )

// Swap 4 byte, 32 bit values:

#define Swap4Bytes(val) \
 ( (((val) >> 24) & 0x000000FF) | (((val) >>  8) & 0x0000FF00) | \
   (((val) <<  8) & 0x00FF0000) | (((val) << 24) & 0xFF000000) )

// Swap 8 byte, 64 bit values:

#define Swap8Bytes(val) \
 ( (((val) >> 56) & 0x00000000000000FF) | (((val) >> 40) & 0x000000000000FF00) | \
   (((val) >> 24) & 0x0000000000FF0000) | (((val) >>  8) & 0x00000000FF000000) | \
   (((val) <<  8) & 0x000000FF00000000) | (((val) << 24) & 0x0000FF0000000000) | \
   (((val) << 40) & 0x00FF000000000000) | (((val) << 56) & 0xFF00000000000000) )
#endif
```

Bytes can also be swapped programmatically but this is slower than that of the macro operation shown but applicable to other word sizes, in this case 128 bits:

File: `swapbytes128.cpp`

```
char *swapbytes128(char *val)
{
    char *swp[16];

    swp[0]  = val[15];
    swp[1]  = val[14];
    swp[2]  = val[13];
    swp[3]  = val[12];
    swp[4]  = val[11];
    swp[5]  = val[10];
    swp[6]  = val[9];
    swp[7]  = val[8];
    swp[8]  = val[7];
    swp[9]  = val[6];
    swp[10] = val[5];
    swp[11] = val[4];
    swp[12] = val[3];
    swp[13] = val[2];
    swp[14] = val[1];
    swp[15] = val[0];

    return swp;
}
```

A generic "in-place" byte swapping endian conversion routine for a user specified number of bytes:

```
#include <netdb.h>

const int bsti = 1;  // Byte swap test integer
#define is_bigendian() ( (*(char*)&bsti) == 0 )

/**
  In-place swapping of bytes to match endianness of hardware

  @param[in/out] *object : memory to swap in-place
  @param[in]     _size   : length in bytes
*/
void swapbytes(void *_object, size_t _size)
{
   unsigned char *start, *end;

   if(!is_bigendian())
   {
       for ( start = (unsigned char *)_object, end = start + _size - 1; start < end; ++start, --end )
       {
          unsigned char swap = *start;
          *start = *end;
          *end = swap;
       }
   }
}
```

Byte Swapping Libraries:

There are byte swapping libraries which are included with most C/C++ libraries. The most commonly used routines are htons() and ntohs() used for network byte order conversions. The host to Big/Little Endian routines (htobe16()/be16toh(), etc) are more complete as they handle swaps of 2, 4 and 8 bytes. These routines are platform independent and know that a swap is only required on Little Endian systems. No swapping is applied to the data when run on a Big Endian host computer as the data is already in "network byte order".

- htons()/ntohs() and htonl()/ntohl(): convert values between host and network byte order
  - uint16_t htons(uint16_t): converts the unsigned "short" integer from host byte order to network byte order.
  - uint16_t ntohs(uint16_t): converts the unsigned "short" integer from network byte order to host byte order.
  - uint32_t htonl(uint32_t): converts the unsigned "long" integer from host byte order to network byte order.
  - uint32_t ntohl(uint32_t): converts the unsigned "long" integer from network byte order to host byte order.
- htobe16()/be16toh(), also 32 and 64 bit: convert values between host and big-/little-endian byte order
  - uint16_t htobe16(uint16_t host_16bits): Host to Big Endian swap 16 bits
  - uint16_t htole16(uint16_t host_16bits): Host to Little Endian swap 16 bits
  - uint16_t be16toh(uint16_t big_endian_16bits): Big Endian to Host swap 16 bits
  - uint16_t le16toh(uint16_t little_endian_16bits): Little Endian to Host swap 16 bits

  - uint32_t htobe32(uint32_t host_32bits);
  - uint32_t htole32(uint32_t host_32bits);
  - uint32_t be32toh(uint32_t big_endian_32bits);
  - uint32_t le32toh(uint32_t little_endian_32bits);

  - uint64_t htobe64(uint64_t host_64bits);
  - uint64_t htole64(uint64_t host_64bits);
  - uint64_t be64toh(uint64_t big_endian_64bits);
  - uint64_t le64toh(uint64_t little_endian_64bits);

Example:

```
#include <endian.h>
#include <stdint.h>
#include <stdio.h>

main()
{
    uint16_t val16 = 4;
    printf("val16 = %d       swapped val16 = %d\n",val16, htobe16(val16));
    printf("val16 = 0x%04x   swapped val16 = 0x%04x\n\n",val16,
htobe16(val16));

    uint32_t val32 = 4;
    printf("val32 = %d           swapped val32 = %d\n",val32,
htobe32(val32));
    printf("val32 = 0x%08x   swapped val32 = 0x%08x\n\n",val32,
htobe32(val32));

    val32 = 1024;
    printf("val32 = %d        swapped val32 = %d\n",val32, htobe32(val32));
    printf("val32 = 0x%08x   swapped val32 = 0x%08x\n\n",val32,
htobe32(val32));
}
```

output when run on a "Little Endian" system:

```
val16 = 4       swapped val16 = 1024
val16 = 0x0004  swapped val16 = 0x0400

val32 = 4           swapped val32 = 67108864
val32 = 0x00000004  swapped val32 = 0x04000000

val32 = 1024        swapped val32 = 262144
val32 = 0x00000400  swapped val32 = 0x00040000
```

Other less worthy options:

- In GCC (not portable) for you can directly call:
  - int32_t __builtin_bswap32 (int32_t x)
  - int64_t __builtin_bswap64 (int64_t x)
- Two macros in `byteswap.h`
  - int32_t bswap_32(int32_t x)
  - int64_t bswap_64(int64_t x)

Additional endian conversion libraries, functions and macros:

- Gnome byte order macros
- Qt byte order functions
- See macros in include files `linux/kernel.h` and `asm/byteorder.h` (`/usr/include/linux/byteorder/little_endian.h` has actual macros): le16_to_cpu(), cpu_to_le16(), be16_to_cpu() and cpu_to_be16() for 16, 32 and 64 bit variables. Also see the versions which take a pointer as its' argument. Versions of the macro have function return values as well as argument return values.
- Boost endian macros - See `boost/detail/endian.hpp`

Bit Field Conversions:

Bit Field data structures are represented by the compiler in the opposite order on Big and Little Endian systems.

Note the use of the macro to determine the endianess of the system and thus which ordering of the bit field to use. The structure is written for cross platform use and can be used on Big and Little Endian systems.

The following macros will handle the differences in bit field representation between big and little endian systems. The bit field struct then requires byte swapping to handle the endian translation.

File: `myData.h`

```
#ifndef MY_DATA_H__
#define MY_DATA_H__
#include <endian.h>
#include <stdint.h>

#if __BYTE_ORDER == __BIG_ENDIAN
   #define BIG_ENDIAN 1
//    #error "machine is big endian"
#elif __BYTE_ORDER == __LITTLE_ENDIAN
//    #error "machine is little endian"
#else
   #error "endian order can not be determined"
#endif

typedef struct {
#ifdef BIG_ENDIAN
        unsigned int Val15:1;              /// [15] MSB
        unsigned int Val14:1;              /// [14]
        unsigned int Val13:1;              /// [13]
        unsigned int Val12:1;              /// [12]
        unsigned int Val11:1;              /// [11]
        unsigned int Val10:1;              /// [10]
        unsigned int Val09:1;              ///  [9]
        unsigned int Val08:1;              ///  [8]
        unsigned int Val07:1;              ///  [7]
        unsigned int Val06:1;              ///  [6]
        unsigned int Val05:1;              ///  [5]
        unsigned int Val04:1;              ///  [4]
        unsigned int Val03:1;              ///  [3]
        unsigned int Val02:1;              ///  [2]
        unsigned int Val01:1;              ///  [1]
        unsigned int Val00:1;              ///  [0] LSB
#else
        unsigned int Val00:1;
        unsigned int Val01:1;
        unsigned int Val02:1;
        unsigned int Val03:1;
        unsigned int Val04:1;
        unsigned int Val05:1;
        unsigned int Val06:1;
        unsigned int Val07:1;
        unsigned int Val08:1;
        unsigned int Val09:1;
        unsigned int Val10:1;
        unsigned int Val11:1;
        unsigned int Val12:1;
        unsigned int Val13:1;
        unsigned int Val14:1;
        unsigned int Val15:1;
#endif
} Struct_A;

// Struct Size: 42 bytes
typedef struct {
    uint16_t Reserved;
    Struct_A struct_A;
    uint16_t Time_3_4;
    uint16_t DataIDs;
#ifdef BIG_ENDIAN
    unsigned int Word_1:24;
    unsigned int Word_2:8;
#else
    unsigned int Word_2:8;
    unsigned int Word_1:24;
#endif
    uint32_t Spare;
} DataStruct;
#endif
```

Although this use of macro #define statements will adjust for bit field nuances, it does not swap any bytes which still need to occur when exchanging data.

```
#include <string.h>
#include "myData.h"

Struct_A hton_Struct_A(Struct_A _struct_A)
{
    uint16_t tmpVar;
    // Use memcpy() because we can't cast a bit field struct to uint16_t
    memcpy((void *)&tmpVar, (void *)&_struct_A, 2);
    tmpVar = htobe16(tmpVar);
    memcpy((void *)&_struct_A, (void *)&tmpVar, 2);
```

```
    return _struct_A;
}

DataStruct hton_DataStruct(DataStruct _dataStruct)
{
    _dataStruct.Reserved = htobe16(_dataStruct.Reserved);
    _dataStruct.struct_A = hton_Struct_A(_dataStruct.struct_A);
    _dataStruct.Time_3_4 = htobe16(_dataStruct.Time_3_4);
    _dataStruct.DataIDs  = htobe16(_dataStruct.DataIDs);
    // Swap the bit fields which do not lie on regular variable size boundaries
    // but collectively they lie on a 32 bit boundary.
    // Use struct pointer plus offset.
    size_t offset = sizeof(uint16_t) + sizeof(Struct_A) + sizeof(uint16_t) + sizeof(uint16_t);
    int *pIntVal = (int *)((char *) &_dataStruct + offset);
    int IntVal = *pIntVal;
    IntVal = htobe32(IntVal);
    memcpy(pIntVal,&IntVal,4);  // Location in structure updated with swapped value
    _dataStruct.Spare = htobe32(_dataStruct.Spare);

    return _dataStruct;
}

main()
{
    DataStruct B;
    bzero((void *) &B, sizeof(DataStruct));
    B.struct_A.Val09 = 1;
    B.struct_A.Val11 = 1;
    B.Time_3_4       = 5;
    B.DataIDs        = 104;
    B.Word_2         = 'A';

    DataStruct C = hton_DataStruct(B);
}
```

One can use macros to define bits in opposite order on big and little endian systems or one can reverse them programmatically.

File: `reverse32Bits.c`

```
unsigned long reverse32Bits(unsigned long x)
{
    unsigned long h = 0;
    int i = 0;

    for(h = i = 0; i < 32; i++)
    {
        h = (h << 1) + (x & 1);
        x >>= 1;
    }

    return h;
}
```

Tests and Examples:

Test bit field positions:

File: `bitOrderStruct.cpp`

```
/** Test most significant bit vs least significant bit of a data structure
 */
#include <iostream>
#include <stdint.h>
// SunOS requires include <inttypes.h>

typedef struct AAAA {
    unsigned int a0:1;
    unsigned int a1:1;
    unsigned int a2:1;
    unsigned int a3:1;
    unsigned int a4:1;
    unsigned int a5:1;
    unsigned int a6:1;
    unsigned int a7:1;
    unsigned int a8:1;
    unsigned int a9:1;
    unsigned int a10:1;
    unsigned int a11:1;
    unsigned int a12:1;
    unsigned int a13:1;
    unsigned int a14:1;
    unsigned int a15:1;
} AAA;

typedef struct BBBB {
    unsigned int b15:1;
    unsigned int b14:1;
    unsigned int b13:1;
    unsigned int b12:1;
    unsigned int b11:1;
```

```
    unsigned int b10:1;
    unsigned int b9:1;
    unsigned int b8:1;
    unsigned int b7:1;
    unsigned int b6:1;
    unsigned int b5:1;
    unsigned int b4:1;
    unsigned int b3:1;
    unsigned int b2:1;
    unsigned int b1:1;
    unsigned int b0:1;
} BBB;

typedef union {
   AAA aa;
   uint16_t a;
   char ca[2];
} UA;

typedef union {
   BBB bb;
   uint16_t b;
   char cb[2];
} UB;

using namespace std;

main()
{
    UA ua;
    UB ub;

    // 83: 10000011
    // MSB 76543210 LSB
    // C1: 11000001
    // MSB 76543210 LSB
    ua.a = 0x0083;
    ub.b = 0x00C1;
    cout << " pos a0: " << ua.aa.a0 << endl;
    cout << " pos a1: " << ua.aa.a1 << endl;
    cout << " pos a6: " << ua.aa.a6 << endl;
    cout << " pos a7: " << ua.aa.a7 << endl;
    cout << " pos b0: " << ub.bb.b0 << endl;
    cout << " pos b1: " << ub.bb.b1 << endl;
    cout << " pos b6: " << ub.bb.b6 << endl;
    cout << " pos b7: " << ub.bb.b7 << endl;
    cout << endl;

    ua.a = 0x8300;
    ub.b = 0xC100;
    cout << " pos a0: " << ua.aa.a0 << endl;
    cout << " pos a1: " << ua.aa.a1 << endl;
    cout << " pos a6: " << ua.aa.a6 << endl;
    cout << " pos a7: " << ua.aa.a7 << endl;
    cout << " pos b0: " << ub.bb.b0 << endl;
    cout << " pos b1: " << ub.bb.b1 << endl;
    cout << " pos b6: " << ub.bb.b6 << endl;
    cout << " pos b7: " << ub.bb.b7 << endl;
}
```

Compile: `g++ bitOrderStruct.cpp`

Run: `./a.out`

Results for Intel x86_64 architecture little endian:

```
pos a0: 1
pos a1: 1
pos a6: 0
pos a7: 1
pos b0: 0
pos b1: 0
pos b6: 0
pos b7: 0

pos a0: 0
pos a1: 0
pos a6: 0
pos a7: 0
pos b0: 1
pos b1: 1
pos b6: 0
pos b7: 1
```

G4 PPC and SunOS 5.8 sun4 sparc sun-blade 2500:

```
pos a0: 0
pos a1: 0
pos a6: 0
pos a7: 0
pos b0: 1
pos b1: 0
pos b6: 1
pos b7: 1
```

```
pos a0: 1
pos a1: 0
pos a6: 1
pos a7: 1
pos b0: 0
pos b1: 0
pos b6: 0
pos b7: 0
```

Compilers and Programming Languages:

GNU Compiler Notes:

The endian byte ordering of binary data is not the only data inconsistency that you may encounter. When using the GNU C compiler "gcc" or the C++ compiler "g++", you may find that the compiler is optimizing structures by padding bytes to byte-align variables. This will be problematic when transfering binary data between computers or between programs written in different languages or with different compilers. The value calculated for "`sizeof(`*`struct-name`*`)`" may be larger than the value expected due to the byte padding. One can turn off this "optimization" by using the compiler flag `-fpack-struct`.

Java:

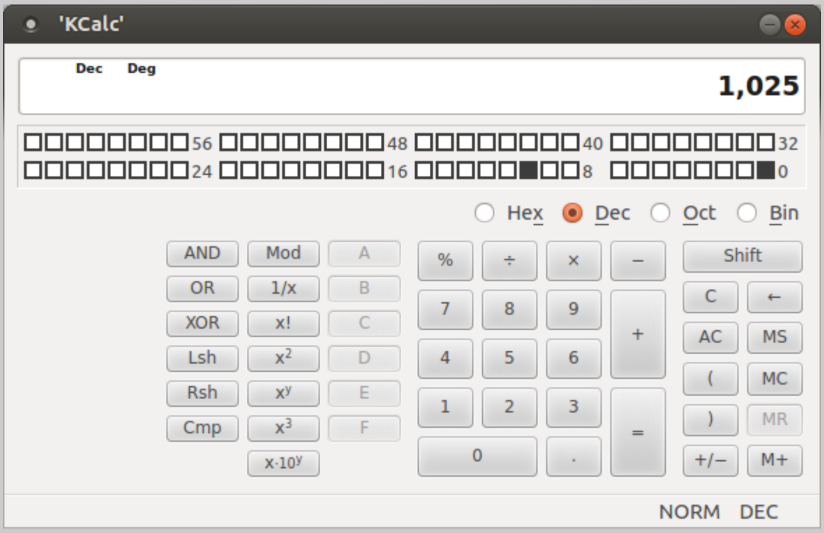
Java internally stores all variables in big endian byte order. If exporting binary data to be consumed for little endian processing or formatting try using the following helper library methods:

# bytes	Method	Description
8	`long n = Long.reverseBytes(lnum)`	Will take a Java big endian long integer and convert it to a little endian integer
4	`int n = Integer.reverseBytes(inum)`	Will take a Java big endian integer and convert it to a little endian integer
2	`short n = Short.reverseBytes(snum)`	Will take a Java big endian short integer and convert it to a little endian integer

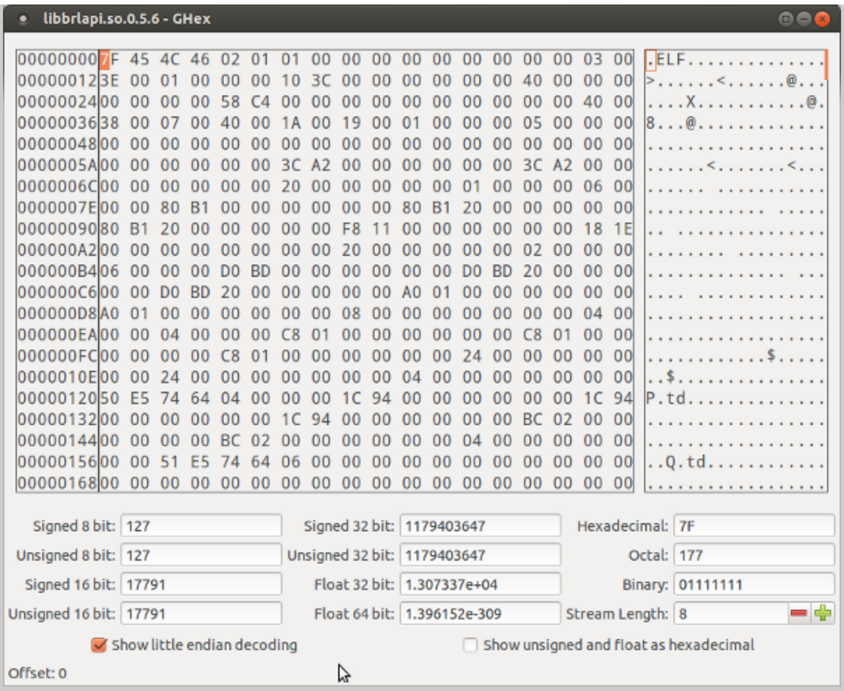
Note that a single byte is not swapped as there is not another byte to swap with.

Tools:

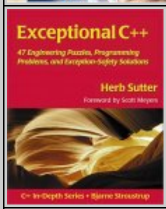
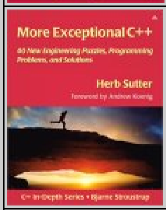
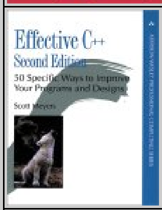
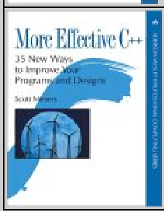
My most commonly used tool when working with bit manipulations and viewing bits is kcalc the KDE calculator which can view data as bits, hex bytes, octal and as decimal numbers.
Installation (Ubuntu): `apt-get install kcalc`



When working with larger quantities of data, I find the Gnome hex editor to be useful.
Installation (Ubuntu): `apt-get install ghex`



Books:

	C++ How to Program by Harvey M. Deitel, Paul J. Deitel ISBN #0131857576, Prentice Hall Fifth edition. The first edition of this book (and Professor Sheely at UTA) taught me to program C++. It is complete and covers all the nuances of the C++ language. It also has good code examples. Good for both learning and reference.	Buy it now! amazon.com
	Exceptional C++: 47 Engineering Puzzles, Programming Problems and Solutions by Herb Sutter ISBN #0201615622, Addison-Wesley Professional Advanced C++ features and STL.	Buy it now! amazon.com
	More Exceptional C++ by Herb Sutter ISBN #020170434X, Addison-Wesley Professional	Buy it now! amazon.com
	Effective C++: 50 Specific Ways to Improve Your Programs and Design (2nd Edition) by Scott Meyers ISBN #0201924889, Addison-Wesley Professional	Buy it now! amazon.com
	More Effective C++: 35 New Ways to improve your Programs and Designs by Scott Meyers ISBN #020163371X, Addison-Wesley Professional	Buy it now! amazon.com

Cal Turney · 9 months ago

I'm a network protocol analyst and frequently see 32-bit numbers received in **"Mid-Little Endian"** order. A 32-bit number is converted from Big Endian to Mid-Little Endian as follows: divide the 4 bytes in half and convert the halves to Little Endian by reversing their order. For example, 01 02 03 04 in Mid-Little Endian is 03 04 01 02. The same number is converted to **Mid-Big Endian** as follows: (1) convert the number to LE (reverse the order of the 4 bytes), (2) divide the bytes in two, and (3) reverse the order of the halves. For example, 01 02 03 04 in Mid-Big Endian is 02 01 04 03.

Reply · Share ›

Sivaramakrishna Devarapalli · 2 years ago

i have a problem with the endianness conversion and printiing , when value converted to

```
#define ENDIAN_SWAP_4_BYTE(_i) \
((((uint32_t)(_i)) & 0xff000000) >> 24) | \
((((uint32_t)(_i)) & 0x00ff0000) >> 8) | \
((((uint32_t)(_i)) & 0x0000ff00) << 8) | \
((((uint32_t)(_i)) & 0x000000ff) << 24)
#endif
```

So after this is done the value is getting printed as -1 with a %d print.and the variable is uint32_t type. Not sure which case this is getting the value

Reply · Share ›