Byte Order & Bit Order

1980 年 4 月 1 日，Danny Cohen，一位网络协议的开创者，在其撰写的《ON HOLY WARS AND A PLEA FOR PEACE》说道：“This is an attempt to stop a war. I hope it is not too late and that somehow, magically perhaps, peace will prevail again.“。看到文章日期(没有去深入查证，据说在 IEEE 收录的这篇文章也是类似的日期)，不禁心存疑问：在这样的一个节日，号称要阻止一场战争，也不知烽烟还会不会再起？：)。那么，到底 Danny Cohen 想阻止一场怎样的战争呢？The latecomers into the arena believe that the issue is:

"What is the proper byte order in messages?".

Byte Order

当我们通讯时，消息是由连续的单元组成的。若想保证正确的结果，则必须关心这些连续单元的序列。一种序列是先从这连续单元的高位起开始发送，这种方式称作大端序列(Big-endian)；另一种便是先从连续单元的低位开始发送，这种方式称作小端序列(Little-endian)。而关于这两种方式的起源及哪种更为正确，Swift, Jonathan，早在其1726所著的《Gulliver's Travel》(《格利佛游记》)中有着详细的描述：

> “……我下面要告诉你的是，Lilliput和Blefuscu这两大强国在过去三十六个月里一直在苦战。战争开始是由于以下的原因：我们大家都认为，吃鸡蛋前，原始的方法是打破鸡蛋较大的一端，可是当今皇帝的祖父小时候吃，一次按古法打鸡蛋时碰巧将一个手指弄破了，因此他的父亲，当时的皇帝，就下了一道敕令，命令全体臣民吃鸡蛋时打破鸡蛋较小的一端，违令者重罚。老百姓们对这项命令极为反感。历史告诉我们，由此曾发生过六次叛乱，其中一个皇帝送了命，另一个丢了王位。这些叛乱大多都是由Blefuscu的国王大臣们煽动起来的。叛乱平息后，流亡的人总是逃到那个帝国去寻救避难。据估计，先后几次有一万一千人情愿受死也不肯去打破鸡蛋较小的一端。关于这一争端，曾出版过几百本大部著作，不过大端派的书一直是受禁的，法律也规定该派的任何人不得做官。”(此段译文摘自网上蒋剑锋译的《格利佛游记》第一卷第4章。)

很容易可以看出，Lilliput和Little-Endians都是以”L”开始，而Blefuscu和Big-Endians均是以字母”B”开始。Danny Cohen，首次使用这两个术语来指代字节顺序，后来这个术语被广泛接受了，这便是Little-endian、Big-endian的来源。

在某些计算机系统中，主要是IBM、Motorola、Sun Microsystems的大多数[1]机器，一个多字节的单元在其存储空间起始地址(低地址)存储的字节是其最高的字节，而其 他的计算机系统，大多数源于以前的Digital Equipment公司(现在是Compaq公司一部分)的机器以及Intel的机器，在低地址存放的是其最低位字节。沿用”格列佛游记”中的方法，IBM/Motoral的方法便叫做大端法(big-endian)，而DEC/Intel的方法叫做小端法(little-endian)。

[1] 注意，是大多数。如IBM制造的PC使用的是Intel兼容的处理器，则其不再采用大端而采用小端法。

很多年来，为两种机制的优缺点引起了强烈的争论。在实际应用中，对两种字节顺序的选择主要基于对旧的系统的兼容性的考虑，因为在使用相同字节顺序的机器之间移植程序要比在不同字节顺序的机器之间容易的多。近来很多芯片都设计成可以支持两种字节顺序，可以很容易地通过电路板的走线或这在引导的时候通过程序选择，有些情况下，甚至可以由应用程序选择。(在这些即时切换的芯片中，只有提取和存贮的指令的数据的字节顺序发生变化，但作为常量编入指令的数据的字节不变化)。许多微处理器芯片，包括 Alpha 和 Motorola 的 PowerPC，也能够运行在任一种模式中，其取决于芯片加电启动时确定的字节顺序规则。

在哪种字节顺序是合适的这个问题上，人们表现得非常情绪化。实际上，如同上面鸡蛋的争论最终演变为社会政治问题的口角一样，对于字节顺序规则的选择应该也是没有技术原因的。

能够由机器一次就完成处理的数据被称为字(Word)。这和我们在文档中用字符和页来计量数据是相似的。字是指位的数目，常用的有16、32或64等。由于目前32位机仍是主流，所以在本文中，字(Word)代表32位，用字母W来表示；字节(Byte)代表8位，用字母C来表示；位(Bit)代表1位，用字母B来表示。

Little-endian & Big-endian

在开始讨论Little-endian和Big-endian之前，我想最好提一下一些我们习以为常的思维定式：

通常我们的阅读习惯都是从左到右(left to right)，因此我们写数字时也是从左到右。如，我们写数字“127”，在现代美语或英语中称作”One Hundred and Twenty-Seven”，在现代汉语中则称作“一百二十七”。这种书写方式是这么的平常以致于多数人从来没想过是不是还有其它的书写方式能达到同样的效果。如同很多人写程序时从来都没考虑过Byte Order一样，当他第一次遇到Little-endian、Big-endian问题时一定会大吃一惊一样，当你阅读老版本的伯希来人(Hebrew)圣经时，你也一定会大吃一惊，因为，127在其中居然被书写为"Seven and Twenty and Hundred"，显然要得到正确信息需要从右到左(right to left)阅读。

可知，日常我们的阅读习惯是一种Big-endian，而Hebrew人的阅读习惯则是一种Little-endian。《Intel Architecture Software Developer's Manual》中对数据在Intel机器内存中存储的bit and byte order(little-endian)图示如下：

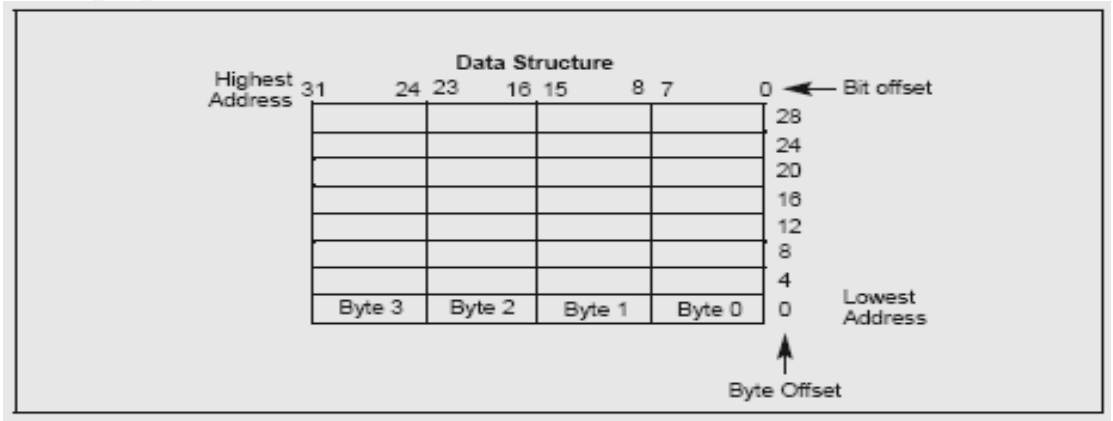


Figure 5. Bit and Byte Order in Little-endian

由图可见，数据在存储的形式是低字节、低位存储在低地址处。可以看到，存储是由右至左的，这样显示主要是为了符合我们前面所说的阅读习惯。如前所述，你若想广为推广古版伯希来人的圣经，而且一样与伯希来人有着从右到左的书写习惯，那么你在书写圣经时最好也象上面Intel图示一样从右把低字节放在低地址处。这样，即使阅读古版伯希来人圣经中的数字“127”时，你从左至右也将看到″Hundred and Twenty and Seven ″而不再是″Seven and Twenty and Hundred″。

同理，应该想得到大端机器中，数据存储应该是：

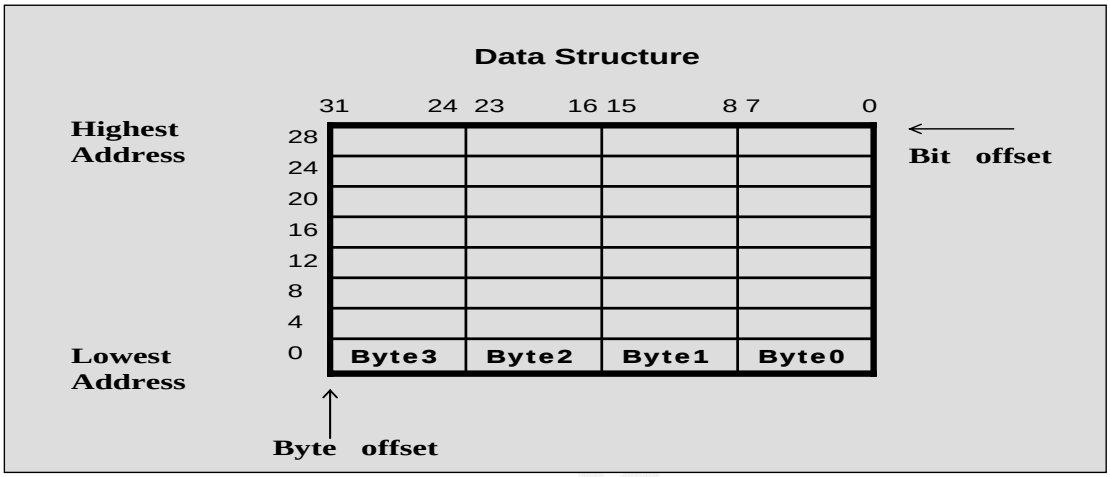


Figure 6. Bit and Byte Order

可见，大端机器存储顺序是和我们的习惯一致的。高位、高位字节存储在地址起始处。

那么，到底什么是Big-endian，什么是Little-endian呢？对于字节序，简单地来说就是：

Big-Endian ： 一个Word中的高位Byte放在内存中这个Word区域的低地址处。

Little-Endian：一个Word中的低位Byte放在内存中这个Word区域的低地址处。

举例说明，一个整型int变量x，为4个字节，Byte0～Byte3，高字节为Byte3，低字节为Byte0。位于地址0x100处，其值为十六进制数0x1234567，显而易见，其高位字节十六进制数为0x01,低位字节值为0x67。那么地址范围0x100 ～ 0x103的字节顺序依赖于机器的类型：

int x = 0x1234567;

Big endian

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

Little endian

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

Figure 1: **0x1234567 store in little & big endian system.**

由上图可见，对于一个四字节整型数据，同样的低地址处，对于大小端机器，其存储的字节却是不同的。大端在其低地址处存储的是其高位字节，而小端在其低地址却存储着其低位字节。对于 int 类型数据，除了字节序以外，我们在所有的机器上都得到相同的结果。如下图所示：

int ival = 12345; /* 0x3039 */

Machine	Value	Type	Bytes (Hex)
Linux	12,345	int	39 30 00 00
NT	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Alpha	12,345	int	39 30 00 00
Linux	12,345.0	float	00 e4 40 46
NT	12,345.0	float	00 e4 40 46
Sun	12,345.0	float	46 40 e4 00
Alpha	12,345.0	float	00 e4 40 46
Linux	&ival	int *	3c fa ff bf
NT	&ival	int *	1c ff 44 02
Sun	&ival	int *	ef ff fc e4
Alpha	&ival	int *	80 fc ff 1f 01 00 00 00

Figure 2: **Byte Representations of Different Data Values.**[1]

在知道了何为大端何为小端，我们便很容易可以知道我们的机器是大端还是小端。W. Richar Stevens在其所著《UNIX Network Programming》中给出一个小小的例子[2]：

```
UNPv1 : UNIX Network Programming, Volume 1 [Stevens 1998] (W. Richar Stevens)

#include  "unp.h"
int main(int argc, char **argv)
{
      union {
         short    s;
         char      c[sizeof(short)];
      } un;

      un.s = 0x0102;
      printf("%s: ", CPU_VENDOR_OS);
      if (sizeof(short) == 2) {
            if (un.c[0] == 1 && un.c[1] == 2)
                  printf("big-endian\n");
            else if (un.c[0] == 2 && un.c[1] == 1)
                  printf("little-endian\n");
            else
                  printf("unknown\n");
      } else
            printf("sizeof(short) = %d\n", sizeof(short));

      exit(0);
}
```

[1]可以看到，尽管浮点和整型数据都对数值 12345，但是它们使用不同的编码方法，二者之间也有一定规则。此不属于本文讨论范围，不再赘述

[2] 这个例子稍加修改才可以运行(把头文件和CPU_VENDOR_OS去掉)，此处引用主要是借此纪念一下Richard Stevens大师。

当然，你自己也可以简单地写一个小函数来确定本机的字节序：

```
int x = 1;

if (*(char *) &x == 1)
    /* 低位优先 */
else
    /* 高位优先 */
```

Middle-endian

在讲述大端小端战争时，一位同事同大家一样觉得这个故事很好笑，但她又同时说道，为什么非要从大端或小端打开鸡蛋，从中间打开不也可以嘛。是啊，鸡蛋从中间打开也是可以的，于是，除了大小端字节序，还有一种中端字节序(Middle-endian)。如前所述，数字“127”，在 King James 版本的英语的书写为"Hundred and Seven and Twenty"。

Linux 内核中对三种字节序的定义如下：

```
#define _LITTLE_ENDIAN    1234    /* least-significant byte first (vax) */
#define _BIG_ENDIAN       4321    /* most-significant byte first (IBM, net) */
#define _PDP_ENDIAN       3412    /* LSB first in word, MSW first in long (pdp) */
```

十六进制数 0x04030201，对于大端机器，在起始位置起将按 0x04030201 依次存储，对于小端机器，从起始位置起将看到 0x01020304；而对中端机器，我们将在起始位置起看到 0x03040102。如下图所示。不过，由于 Middle-endian 是一种不常用的字节序，手头也没有资料对此做进一步的说明，所以在此不再多加讨论。

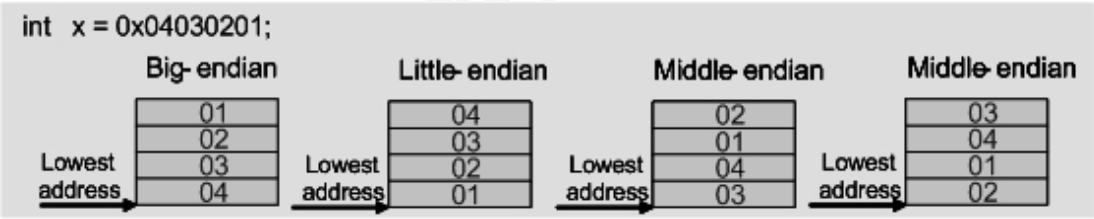


Figure 3: **Endian Performance**

Bit Order

上面讨论的基本是在不同机器上字节的顺序，但有的时候通讯时最小单位并不是字节而是位。所以研究一下位序也是很有必要的。对于大小端机器来说，数据结构在内存中存放的位序其实也遵循着字节序相应的特性，即小端机器低地址处存放着其低位字节中的低位bit，而大端机器低地址处存放着其高位字节中的高位bit。所以在位的顺序中，Big-endian与Little-endian为：

Big-endian就是指起始放置的是MSB(Most Significant Bit 最高有效位)[1]。

Little-endian就是指起始位置放置的是LSB(Least Significant Bit 最低有效位)。

因此，对于一个32位处理器来说，若Bit0存放的是MSB，Bit31存放的是LSB，它应该是

[1] MSB: 一般是最左侧位即最高有效位；LSB: 最右侧位也即最低有效位。好象也有资料把MSB译作最重要位，LSB为最次要或最不重要位。比如说：对于 111$，若一定让你丢掉一个 1，那么你一定会选择丢掉最右边的 1，让 111$变为 110$，而不是丢掉最重要位让它变成 11$。

Big-endian的，反之，若Bit0存放的是LSB，Bit31存放的是MSB，该处理器则应该是Little-endian的。

下面以整型数 0x0a0b0c0d 分别在大端、小端机器上存储来说明位序的不同。

1. 大端机器：

低地址起开始存储整型数的高位。

```
byte  addr       0         1        2        3
bit   offset  01234567 01234567 01234567 01234567
      binary  00001010 00001011 00001100 00001101
         hex     0a       0b       0c       0d
```

2. 小端机器

低地址起开始存储整型数的低位。

```
byte  addr       3         2        1        0
bit   offset  76543210 76543210 76543210 76543210
      binary  00001010 00001011 00001100 00001101
         hex     0a       0b       0c       0d
```

从上面可以看到在这两种情况下，我们均可以从左到右读到整数 0x0a0b0c0d。当然，为了符合我们的阅读习惯，小端机器的开始地址在图示中是从右开始的。如果换成我们习惯的起始地址从左开始，则整数 0x0a0b0c0d 从起始地址开始我们将看到的是 0d0c0b0a，与大端的顺序是相反的。

同时，我们可以注意到对于如 0a 这个字节来说，在大端中 bit0~bit7 位的序列为分别为 00001010，但在小端中 bit0~bit7 的序列却为 01010000。也就是说对于同一字节大端中的位序和小端中的位序是不同的。

说到这里，可能已经有同学忍不住想说了，好象哪儿有点不对劲吧：对小端存储若从左到右按位序展开表示，将看到如下情形，并不是前面所说的 0d0c0b0a，看起来好象是 b030d050 啊？

```
byte  addr       0         1        2        3
bit   offset  01234567 01234567 01234567 01234567
      binary  10110000 00110000 11010000 01010000
```

是啊。如果简单按我们阅读习惯看来，按位展开的结果是 b030d050。可也许也有人会说：注意!这是小端的世界，低地址存储可是低位，从左到右的习惯可是大端的习惯，不能直接这样读小端的数据。读小端二进制数要么从右至左读要么把二进制数转成大端习惯再读。如读小端 0 字节的二进制数 10110000 时，应该知道最前面的 1 其实是这个二进制数的最低位，应该放在大端习惯的二进制数的最右面。这样，这个二进制其实应该是 00001101，也即是 0d。那么，整数在小端机器存储应该为 0d0c0b0a，而不是 b030d050。

绕来绕去，可见这个从小端打开鸡蛋的敕令是多么的令人不习惯！

Order HOWTO

Software Solution

Byte Order

对于大多数应用程序员来说，他们机器的字节顺序或位序是完全不可见的。无论为哪种类型的机器年编译的程序都会得到相同的结果。不过有时候字节顺序会成为问题：

1. 当阅读表示整数数据的字节序列时。

这通常发生在检查机器级程序时。如：

```
80483bd: 01 05 64 94 04 08            add %eax, 0x8049464
```

当阅读此例中小端法机器生成的机器级程序表示时，经常会将字节按照相反的顺序显示，这和书写数字时最高有效位在左侧最低有效位在右侧的习惯是相反的。

2. 在不同的机器之间通过网络传送二进制数据时。

一个常见的问题是当小端法机器产生的数据被发送到大端法机器，或反之时，接收程序会发现，字里的字节成了反序的。为了这类问题，网络应用程序的代码必须遵守已建立的关于字节顺序的规则，以确保发送方机器大头针 它的内部表示转换成网络标准，而接收方机器则将网络标准转换为内部表示。

3. 当编写规避正常的类型系统的程序时。

在 C 中，可以使用强制类型转换(cast)来允许以一种不同于它被创造时的数据类型来引用一个对象。

第 1 种情形在小端机器将代码随意进行一下反汇编即可看到。所以下面主要讨论第 2 第 3 种情形中的字节序影响。

首先讨论网络应用程序中的字节序的影响。由于大小端的影响，同一数值的数据，在不同的机器可能得到的值却不相同。如对于十六进制数 0x01020304 从大端发送到小端机器，从小端机器该值的起始内存中得到的 01020304，其值其实是 0x04030201，并不是大端发送的 0x01020304。若想正确地在小端得到这个值，则小端从内存起始处该值存储为 04030201。

通常，网际协议在处理多字节整数时，都使用的是大端字节序。因此，我们在编程必须考虑主机字节序和网络字节序间的相互转换。一般，转换时用到下图中的四个宏(或函数)。

当使用这些宏或函数时，我们不关心主机字节序和网络字节序的真实值(大端或小端)。由于网际协议采用的大端字节序，所以在大端机器上，这四个宏(或函数)常常定义为空宏(或空函数)，不做任何转换。但在小端机器上，则会进行大、小端之间的转换。转换处理就是交换接到 4 字节的顺序；转换的结果就是大端机器向小端机器发送十六进制数 0x01020304，小端机器能正确的接收到这个数值。

```
/*
 * Macros for number representation conversion.
 */
#if   _BYTE_ORDER==_BIG_ENDIAN
#define   ntohl(x)   (x)
#define   ntohs(x)   (x)
#define   htonl(x)   (x)
```

```
#define    htons(x)  (x)

#define NTOHL(x) (x) = ntohl((u_long)(x))
#define NTOHS(x) (x) = ntohs((u_short)(x))
#define HTONL(x) (x) = htonl((u_long)(x))
#define HTONS(x) (x) = htons((u_short)(x))

#endif     /* _BYTE_ORDER==_BIG_ENDIAN */

#if   _BYTE_ORDER==_LITTLE_ENDIAN

#define ntohl(x)      ((((x) & 0x000000ff) << 24) | \
                      (((x) & 0x0000ff00) <<    8) | \
                      (((x) & 0x00ff0000) >>    8) | \
                      (((x) & 0xff000000) >> 24))

#define htonl(x)      ((((x) & 0x000000ff) << 24) | \
                      (((x) & 0x0000ff00) <<    8) | \
                      (((x) & 0x00ff0000) >>    8) | \
                      (((x) & 0xff000000) >> 24))

#define ntohs(x)      ((((x) & 0x00ff) << 8) | \
                      (((x) & 0xff00) >> 8))

#define htons(x)      ((((x) & 0x00ff) << 8) | \
                      (((x) & 0xff00) >> 8))
#define    NTOHL(x)      (x) = ntohl((u_long)(x))
#define    NTOHS(x)      (x) = ntohs((u_short)(x))
#define    HTONL(x)      (x) = htonl((u_long)(x))
#define    HTONS(x)      (x) = htons((u_short)(x))

#endif     /* _BYTE_ORDER==_LITTLE_ENDIAN */
```

另一种处理方法就是使用字符串。字符串中每个字符都由某个标准编码来表示，最常见的就是 ASCII 字符码，而在使用 ASCII 码的任何系统上，字符串都将得到相同的结果，与字节顺序和字大小规则无关。因而，文本数据比二进制数据具有更强的平台独立性。

第 3 种情况，在 C 中，可以使用强制类型转换(cast)来允许以一种不同于它被创造时的数据类型来引用一个对象时，字节序问题可能会给你带来困扰。举个小例子说明：

假定一个程序在一个源文件中包含了声明：[1]

```
long foo;
```

而在另一个源文件中包含了：

```
extern short foo;
```

又进一步假设，如果给long类型的foo赋一个较小的值，如37，那么short型的foo就同时获得了一个值37。我们能够对运行该程序的硬件做出什么样的推断呢？如果short类型的foo得到的值不是37而0，我们又能得到什么样的推断呢？

如果把值37赋给long型的foo，相当于同时把值赋给short型的foo，就意味着short型的foo与long 型的foo中包含了值37的有效位的部分，两者在内存中占用的是同一区域。这有可能是因为long型和short型被实现为同一类型，但很少有C实现会这样做。更有可能的是，

[1] 引自《C Traps and Pit falls》

long型的foo的低位部分与short型的foo共享了相同的内存空间，一般情况下，这个部分所处的内存地址较低；因此我们的一个推论就是，运行该程序的硬件是一个低位优先(little-endian)的机器。同样的道理，如果在long型的foo中存储了值37，而short型的foo的值却是0，我们所用的硬件可能是一个高位优先(big-endian)的机器。

而另一个能说明问题的小例子：

```
int dw = 2;
short w = *(short *)&dw;
```

结果:
XScalce(Big-endian): w = 0;
X86(Little-endian): w = 2;

Bit Order

在前面的 Bit Order 一节中已经对位序做了一定的说明，下面再举一个例子做进一步阐述。如有一个结构体，大小是一个字节：

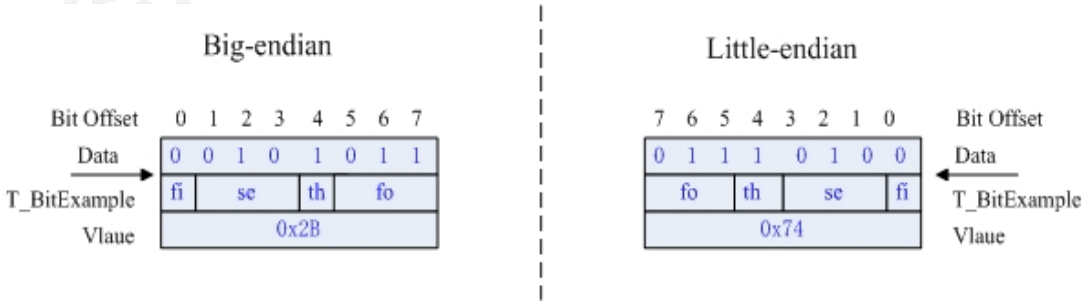
```
Typedef struct
{
    unsigned fi:1;
    unsigned se:3;
    unsigned th:1;
    unsigned fo:3;
}T_BitExample;
```

现在给位段赋值： fi＝0，se＝2，th＝1，fo＝3。

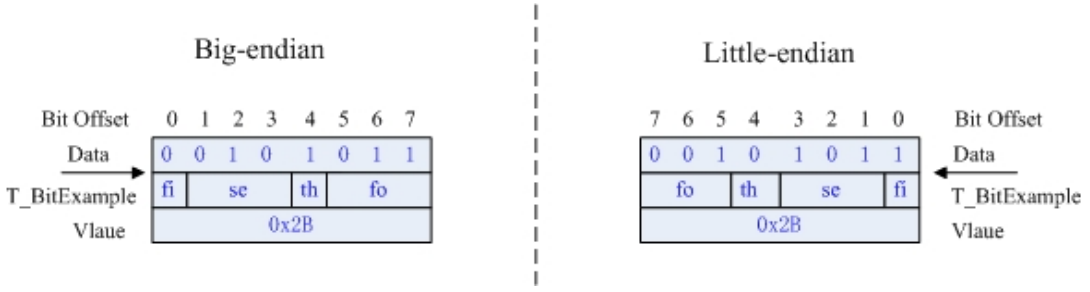
在 windows 操作系统下，这个结构体的值＝0x74，在 Solario 操作系统下，这个结构体的值＝0x2B。

0x74＝0111 0100＝011 1 010 0＝(Msb)3 1 2 0(Lsb) 结构体的高位放在内存区域大数端 Little endian

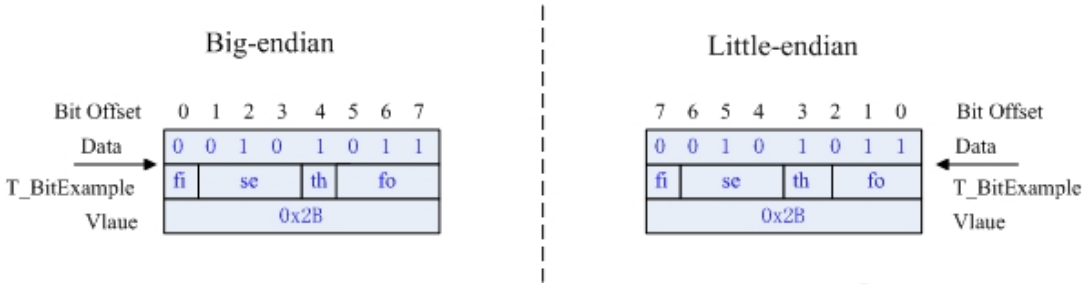
0x2B＝0010 1011＝0 010 1 011＝(Msb)0 2 1 3(Lsb) 结构体的高位放在内存区域小数端 Big endian。如下图所示：



如果在大端、小端机器上定义同样的结构，然后把 0x2B 从大端机器传送到小端机器，则稍加考虑便可知，我们在小端机器上将得不到正确的结果。如下图所示：



若想大端、小端传送数据，能得到正确的位域值。则必须调整位域在结构中的顺序。如下图所示：



由上图可知，在小端机器上，结构 T_BitExample 将被定义为：

```
Typedef struct
{
    unsigned fo:3;
    unsigned th:1;
    unsigned se:3;
    unsigned fi:1;
}T_BitExample;
```

这样，大小端机器时通讯时，便可正确接收对方发送的数据。

同样的道理，我们可以看到，网络通讯协议如标准的 TCP/IP 协议中，为了保证数据在大端小端机器的正确传送， TCP 头在结构中也对位序针对大小端做了相应的处理，结构定义如下：

TCP 首部的数据结构：

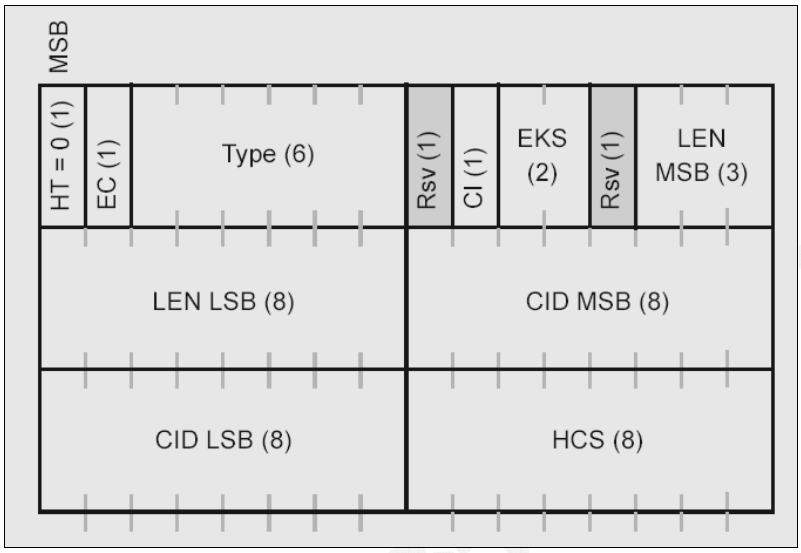
```
struct tcphdr {
       u_short th_sport;            /* source port */
       u_short th_dport;            /* destination port */
       tcp_seq th_seq;              /* sequence number */
       tcp_seq th_ack;              /* acknowledgement number */
#if BYTE_ORDER == LITTLE_ENDIAN
       u_char  th_x2:4,             /* (unused) */
               th_off:4;            /* data offset */
#endif
#if BYTE_ORDER == BIG_ENDIAN
       u_char  th_off:4,            /* data offset */
               th_x2:4;             /* (unused) */
#endif
       u_char  th_flags;
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04
```

```
#define TH_PUSH 0x08
#define TH_ACK  0x10
#define TH_URG  0x20
        u_short th_win;                 /* window */
        u_short th_sum;                 /* checksum */
        u_short th_urp;                 /* urgent pointer */
};
```

下面，再以 802.16d协议中MAC头在不同机器的处理为例来说明位序较为复杂的处理方式 [1]。

1) MAC 头格式：

以下是协议中的 MAC 头格式。



2) MAC 头结构定义

```
typedef unsigned char    BYTE;
typedef unsigned short   WORD16;
#if   _BYTE_ORDER==_BIG_ENDIAN
#pragma pack(1)
typedef struct
{
     BYTE          bHT:1;              /*  头类型  */
     BYTE          bEC:1;              /*  加密标识  */
     BYTE          bMesh:1;            /* Mesh 子头标识  */
     BYTE          bARQ:1;              /* ARQ 子头标识  */
     BYTE          bExtend:1;          /*  扩展子头标识  */
     BYTE          bFragment:1;        /*  分段子头标识  */
     BYTE          bPacking:1;         /*  组包子头标识  */
     BYTE          bFastFeedBack:1;    /* Fast FeedBack 分配子头  */

     WORD16        bRsv1:1;            /*  保留  */
     WORD16        bCI:1;              /* CRC 标识  */
     WORD16        bEKS:2;              /* EKS */
     WORD16        bRsv2:1;            /*  保留  */
```

[1] 引自《MAC帧头BIT位在不同CPU上的研究》，来传远著，本例仅是站在Endian的角度来说此数据结构，不保证协议中此结构字段赋值的意义的正确。

```
    WORD16          wLen:11;              /*  长度  */

    WORD16          wCID;                  /* CID */
    BYTE            bHCS;                 /* HCS */
}   T_GenericMacHead;
#pragma pack(0)
#endif
```

3) 结构赋值

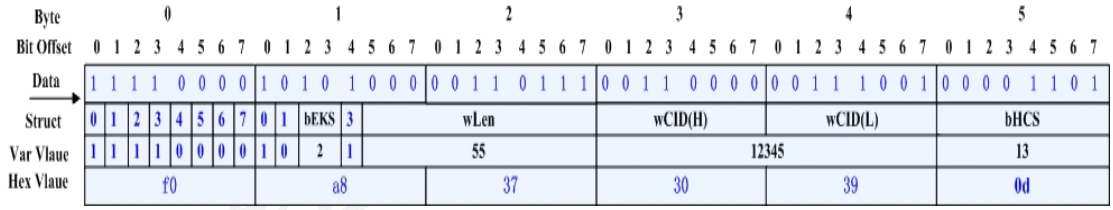
```
    bHT             = 1;
    bEC             = 1;
    bMesh           = 1;
    bARQ            = 1;
    bExtend         = 0;
    bFragment       = 0;
    bPacking        = 0;
    bFastFeedBack   = 0;

    bRsv1           = 1;
    bCI             = 0;
    bEKS            = 2;
    bRsv2           = 1;
    wLen            = 55;       /* 110111*/
    wCID            = 12345;    /*11000000111001*/
    bHCS            = 13;
```

4)大端小端之间的转换

根据上述赋值我们可以得知在大端机器上，该MAC头结构的存储具体如下图[1]所示：

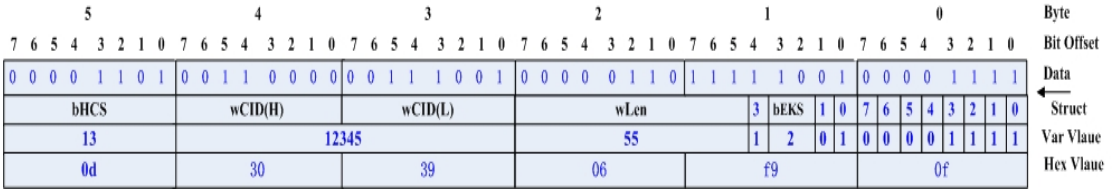
```
0xf0a83730390d
11110000   10101000   00110111   00110000   00111001   00001101
```



同样的结构，同样的赋值，在小端机器上的存储却显示为：

```
0x0ff90639300d
00001111   11111001   00000110   00111001   00110000   00001101
00001111   00010101   11101100   00001100   10011100   10110000
```

详细如下图所示：



如果你此时又感到有点迷糊，那么你最好口中念念有词：“大端是指 MSB（最高有效位）存储在低地址处，小端是指 LSB（最低有效位）存储在低地址处”。

结构的第 1 个元素的地址就是结构的起始地址。对于本结构来说，大端机器上第 1 个元素即位 bHT 是结构的 MSB，存储在结构的起始地址处；小端机器上同样是第 1 个元素的 bHT 却是结构的 LSB，存储在结构的起始地址处。所以对于结构的第 1 字节，同样的元素同样

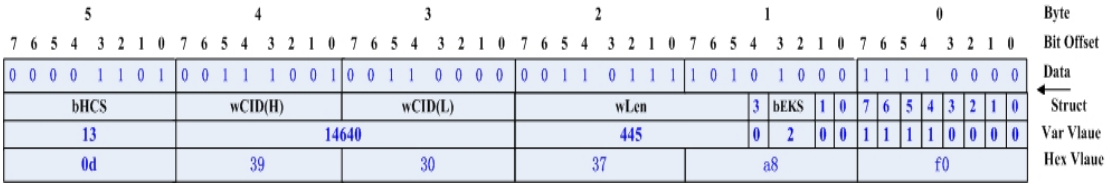
[1] 由于位域变量名较长而图示中位的空间又小，写不下变量名，所以下面图示结构中位域变量名的位置以变量在结构中的序号代替，具体可对照前面结构定义。

的顺序同样的赋值，在大小端机器却分别存储为 0xf0 及 0x0f。而结构的第 2 第 3 字节在大小端机器上值就更为不同了，分别为 0xa837 及 0xf906。具体原因相信经过这么多论述，大家已经可以自行分析了。

至此，经过再三重复啰嗦的游说，应该可以虚幻地认为 1980 年 4 月 1 日 Danny Cohen 的目的此时已经达到。

但认识到两者差异并不是我们的目的，能让两者之间尽可能透明、代价小的实现通讯才是我们的目的。

下面我们分析小端如何组织该 MAC 头结构，才能正确解析接收大端发送的数据 0xf0a83730390d。



从上图可以看出结构的第 1 字节的 8 个元素如本小节的第 1 个例子一样，调整一下相互顺序即可得到正确的值；字段 wCID 类型为 unsigned short，根据前面所述，调用 ntohs 后即可得到正确的值；字段 bHCS 的值没有变化。最后只剩下结构的第 2 第 3 字节，即包括位域 wLen 的两个字节。稍加对比可知很难通过简单的位置调整，让小端结构中的

```
    bRsv1               = 1;
    bCI                 = 0;
    bEKS                = 2;
    bRsv2               = 1;
    wLen                = 55;       /* 110111*/
```

问题便出在 wLen : 11 上！

翻阅位域规则，其中有一条规则：

> 一个位段必须存储在同一存储单元中，不能跨两个单元。如果第一个单元空间不能容纳下一个位段，则该空间不用，而从下一个单元起存放该位段。

从上例及位域规则，在设计跨平台结构用到位域时，可能可以得出这样的结论：

不要使用超过 8 位的位域；如果必须使用超过 8 位的位域时，最好将该数据类型重新分为若干个 8 位的位域。

根据这条规则，我们可以把上例中 wLen:11 分为 bLenHigh:3 和 bLenLow 两个位段。

这样，小端机器上 MAC 头结构设计如下：

```
#if   _BYTE_ORDER==_LITTLE_ENDIAN
#pragma pack(1)
typedef struct
{
    BYTE        bFastFeedBack:1;    /* Fast FeedBack 分配子头  */
    BYTE        bPacking:1;         /*  组包子头标识  */
    BYTE        bFragment:1;        /*  分段子头标识  */
    BYTE        bExtend:1;          /*  扩展子头标识  */
    BYTE        bARQ:1;               /* ARQ 子头标识  */
    BYTE        bMesh:1;            /* Mesh 子头标识  */
    BYTE        bEC:1;              /*  加密标识  */
    BYTE        bHT:1;              /*  头类型  */

    BYTE        bLenHigh:3;         /*  长度的高 3 位  */
    BYTE        bRsv2:1;            /*  保留位 2 */
    BYTE        bEKS:2;             /* EKS        */
```

```
    BYTE        bCI:1;              /* CRC 标识 */
    BYTE        bRsv1:1;            /* 保留位 1 */
    BYTE        bLenLow;            /* 长度的低 8 位 */

    WORD16      wCID;               /* CID */
    BYTE        bHCS;               /* HCS */
}   T_GenericMacHead[1];
#pragma pack(0)
#endif
```

其中结构中位段 bLenHigh 和 bLenLow 拼到一起便是大端结构中的字段 wLen，其值为 55。同时，也得注意 WORD16 的字节序。

也许上述结构符合了位域的规定，但却不符合你自己的习惯。被别人改变自己的习惯总是一件令人恼火的事情。你也许还是认为 wLen:11 这种习惯好而且确定想使用它。其实这样也是可以的。那么结构可以定义如下：

```
#if  _BYTE_ORDER==_LITTLE_ENDIAN
#pragma pack(1)
typedef struct
{
    BYTE        bFastFeedBack:1;    /* Fast FeedBack 分配子头 */
    BYTE        bPacking:1;         /* 组包子头标识 */
    BYTE        bFragment:1;        /* 分段子头标识 */
    BYTE        bExtend:1;          /* 扩展子头标识 */
    BYTE        bARQ:1;             /* ARQ 子头标识 */
    BYTE        bMesh:1;            /* Mesh 子头标识 */
    BYTE        bEC:1;              /* 加密标识 */
    BYTE        bHT:1;              /* 头类型 */

    WORD16      wLen:11;            /* 长度 */
    WORD16      bRsv2:1;            /* 保留 */
    WORD16      bEKS:2;             /* EKS */
    WORD16      bCI:1;              /* CRC 标识 */
    WORD16      bRsv1:1;            /* 保留 */

    WORD16      wCID;               /* CID */
    BYTE        bHCS;               /* HCS */
}   T_GenericMacHead;
#pragma pack(0)
#endif
```

只不过这样使用时，仍是如同 wCID 需要注意字节序一样，wLen 所处的 WORD16 也需要进行字节序的处理，然后才可以直接使用该结构。

假设在 X86（little-endian）机器上收到 Xscale(big-endian)的 MAC 头，在 X86 上 MAC 头使用上面定义的结构，则代码示意如下：

```
    T_GenericMacHead   tMacHead ;
    /* simulate the data come from big-endian mechian */
    BYTE   abuf[6]={0xf0,0xa8,0x37,0x30,0x39,0xd},abuf1[6];
    char   *p,*pa;
```

[1] 引自崔巍源码。

```
abuf1[0] = abuf[0];
abuf1[1] = abuf[2]; /* exchage WORD16's byte order */
abuf1[2] = abuf[1];
abuf1[3] = abuf[4]; /* exchage WORD16's byte order */
abuf1[4] = abuf[3];
abuf1[5] = abuf[5];

pa = abuf1;
tMacHead = *(T_GenericMacHead    *)pa;

printf("bHT(%d),bEC(%d),bMesh(%d),bARQ(%d),bExtend(%d),bFragment(%d),\
        bPacking(%d),bFastFeedBack(%d),bRsv1(%d),bCI(%d),bEKS(%d),bRsv2(%d),\
        wLen(%d),wCID(%d),bHCS(%d)\n",\
        tMacHead.bHT,tMacHead.bEC,tMacHead.bMesh,tMacHead.bARQ,\
        tMacHead.bExtend,tMacHead.bFragment,tMacHead.bPacking,\
        tMacHead.bFastFeedBack,tMacHead.bRsv1,tMacHead.bCI,tMacHead.bEKS,\
        tMacHead.bRsv2,tMacHead.wLen,tMacHead.wCID,tMacHead.bHCS);
```

打印结构结果如下：

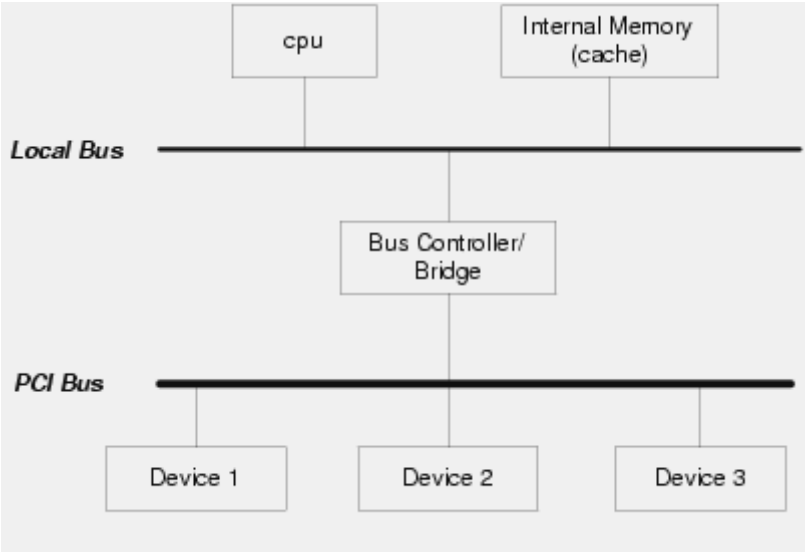
```
bHT(1),bEC(1),bMesh(1),bARQ(1),bExtend(0),bFragment(0),bPacking(0),bFastFeedBack(0),bRsv1(1),bCI(0),bEKS(2),bRsv2(1),wLen(55),wCID(12345),bHCS(13)
```

可见和前面大端上结构的赋值完全一样。

位交换的代价是昂贵的，我们总是通过在结构定义中位域顺序的调整来完成大端、小端机器数据单位为位时的数据的交换。所以在设计位域结构时要仔细小心，考虑到大小端机器上的移植问题。

Hardware Solution

过多涉及细节会让程序员神经紧张，而位序的转换是昂贵乏味的。最好是能通过硬件方式，给上层提供一个高效透明的环境，不需为这随时可能出错的转换提心吊胆。在下文中，我们首先介绍一下通用的计算机体系结构，然后讨论一下当 CPU 与外部设备总线端性不一致时的硬件处理。



在这里，我们把图中 CPU，Interna Memory、Local Bus 都统称为 CPU，并认为 CPU 寄存器、内存字长、总线宽度都是 32 位的，它们一般都拥有同样的端性。讨论 bus endianness，主要是指外部总线的 endianness。外部总结是连接 CPUs、Devices 及其它组件的媒介，总线的端性是由总线协议确定且和设备、组件共同遵循的。

通过前几节论述，我们可知当总线为 Little-endian 时，意味着对于 32 位地址/数据总线 AD[31:0]，连接 AD31 的是设备的 Most significant data line，连接 AD0 的是 least significant data line。对于 Big-endian 的机器，行为刚好相反。对于 AD[0:31]，连接 AD0 的是最高有效数据线，而连接 AD31 是最低有效数据线。

对于一个 partial word 设备，如 8 位设备，小端总线如 PCI 将会把 8 位数据线连到 AD[7:0]上；而大端总线则会把 8 位数据线连到 AD[24:31]上。

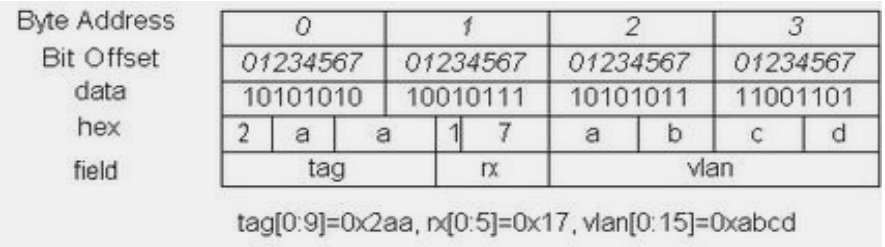
连接到总线的设备都应该有相同的字节/位序，包括 CPU。当 CPU 的 endianness 与 bus 的 endianness 不一致时，总线控制器(Bus Controller/Bridge)将会进行二者间的转换。

下面介绍两种通过配线来完成这种转换的方法

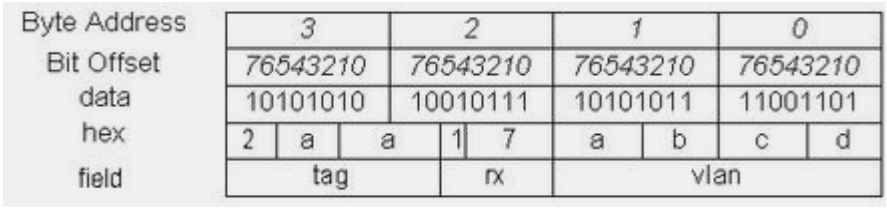
Word Consistent Approach

对于 AD[31:0]，这种方法主要是交换设备数据线 D[0:31]的次序。D0 存储 MSB，D31 存储 LSB。也就是配线 D(i)到 AD(i)，i = 0,….,31。

举例说明。如下是一个大端 NIC 卡的 32 位寄存器描述符：

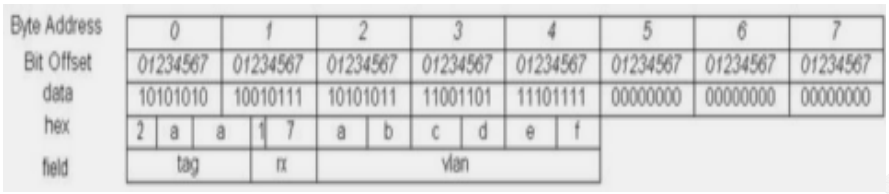


在进行了字一致性(Word Consistent)交换(配线 D[0:31]到 AD[31:0])后，CPU/bus 的结果为：

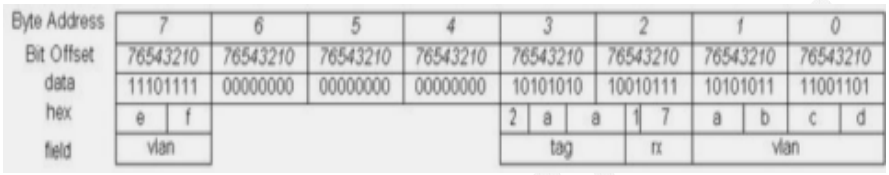


注意到，大端总线数据毋需软件的字节/位交换处理，自动地转换成了小端数据。

不过，这种 Word Consistent 方法仅是简单地适用于不跨 Word(32bit)边界的数据的处理，一旦数据跨越了 Word 边界，这种方法便不再能正确转换数据了。如下图中 vlan[0:24]跨越了 Word 边界，通过这种方法转换便出错了。



转换后：

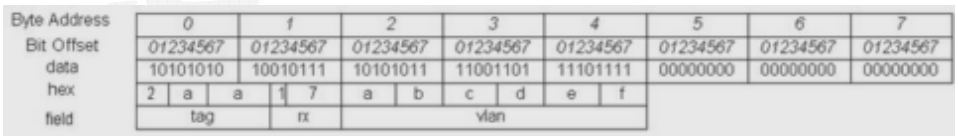


由图可以看出 vlan 在内存中被分为了不连续两部分。我们不可能定义一个在内存中不连接的 C 结构中的变量。

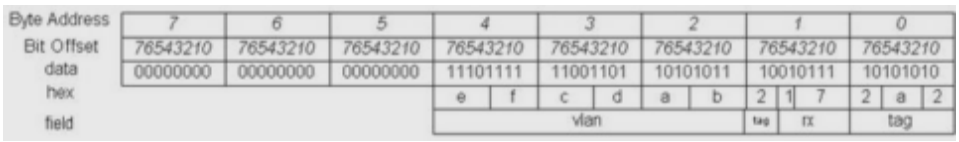
因此，此种方法仅适合在一个 Word 边界内数据的转换，不适合那些跨越 Word 边界数据的转换。跨越 Word 边界的数据的转换，我们用第二种方法。

Byte Consistent Approach

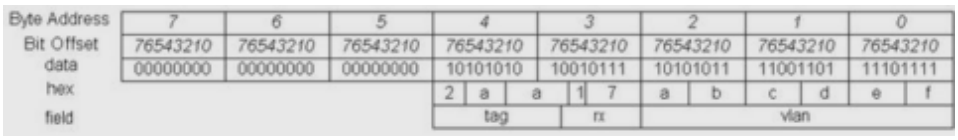
在这种方法中，我们并不是交换字节而是交换每个字节中的位序，也就是通过硬件配线把字节中的位 i 交换到位(7-i)处。



通过这种方法，大端 NIC 设备的 CPU/bus 上的值为：



这样，3 字节的 vlan 在内存中的连续的而且字节内容也是正确的。由于数据占据连续的内存空间，所以我们很容易采用软件方法将这 5 字节内容进行交换，得到如下结果：



字节的软件交换的代价是可以接受的，不象位的软件交换。这样，我们就可以看到，对于 NIC 描述符的结构可以定义如下：

```
struct nic_tag_reg {
        uint64_t vlan:24 __attribute__((packed));
        uint64_t rx    :6    __attribute__((packed));
        uint64_t tag :10 __attribute__((packed));
};
```

References