**大小端模式**

<http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq>

大端模式，是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；这和我们的阅读习惯一致。

小端模式，是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低。

**大端模式**

所谓的大端模式（Big-endian），是指数据的高字节，保存在内存的低地址中，而数据的低字节，保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；

例子：

0000430: e684 6c4e 0100 1800 53ef 0100 0100 0000

0000440: b484 6c4e 004e ed00 0000 0000 0100 0000

在大端模式下，前32位应该这样读: e6 84 6c 4e ( 假设int占4个字节)

记忆方法: 地址的增长顺序与值的增长顺序相反

**大小端模式小端模式**

所谓的小端模式（Little-endian），是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低，和我们的逻辑方法一致。

例子：

0000430: e684 6c4e 0100 1800 53ef 0100 0100 0000

0000440: b484 6c4e 004e ed00 0000 0000 0100 0000

在小端模式下，前32位应该这样读: 4e 6c 84 e6( 假设int占4个字节)

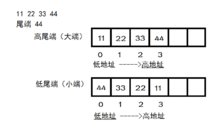
记忆方法: 地址的增长顺序与值的增长顺序相同

**大小端模式大小端模式**

为什么会有大小端模式之分呢？这是因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为 8bit。但是在C语言中除了8bit的char之外，还有16bit的short型，32bit的long型（要看具体的[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)），另外，对于位数大于 8位的处理器，例如16位或者32位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。例如一个16bit的short型x，在内存中的地址为0x0010，x的值为0x1122，那么0x11为高字节，0x22为低字节。对于 大端模式，就将0x11放在低地址中，即0x0010中，0x22放在高地址中，即0x0011中。小端模式，刚好相反。我们常用的X86结构是小端模式，而KEIL C51则为大端模式。很多的ARM，DSP都为小端模式。有些ARM处理器还可以随时在程序中(在ARM Cortex 系列使用REV、REV16、REVSH指令[1]  )进行大小端的切换。

**大小端模式图解**

对于0x11223344 储存如下

[](http://baike.baidu.com/pic/%E5%A4%A7%E5%B0%8F%E7%AB%AF%E6%A8%A1%E5%BC%8F/6750542/0/b2de9c82d158ccbfba5e4c721ed8bc3eb035419c?fr=lemma&ct=single)

**大小端模式名字由来**

在[乔纳森·斯威夫特](http://baike.baidu.com/item/%E4%B9%94%E7%BA%B3%E6%A3%AE%C2%B7%E6%96%AF%E5%A8%81%E5%A4%AB%E7%89%B9)的著名讽刺小说《格列夫游记》中，小人国内部分裂成Big-endian和Little-endian两派，区别在于一派要求从鸡蛋的大头把鸡蛋打破，另一派要求从鸡蛋的小头把鸡蛋打破。斯威夫特借以讽刺英国的政党之争，在计算机工业中指数据储存顺序的分歧。

**大小端模式编辑器模式**

下面这段代码可以用来测试一下你的编译器是大端模式还是小端模式：

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2  3  4  5 | short int x；  char x0,x1;  x=0x1122;  x0=((char\*)&x)[0]; //低地址单元  x1=((char\*)&x)[1]; //高地址单元 |

若x0=0x11,则是大端; 若x0=0x22,则是小端......

上面的程序还可以看出，数据寻址时，用的是低位字节的地址。

**大小端模式简单大小端转换的宏**

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2  3  4 | #definesw16(x)\  　((short)(\  　(((short)(x)&(short)0x00ffU)<<8)|\  　(((short)(x)&(short)0xff00U)>>8))) |

**大小端模式操作系统**

[编辑](javascript:;)

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2 | static union { char c[4]; unsigned long l; }endian\_test = { { 'l', '?', '?', 'b' } };  #define ENDIANNESS ((char)endian\_test.l) |

(如果ENDIANNESS=’l’表示系统为little endian,为’b’表示big endian )。

**大小端模式判断处理器**

[编辑](javascript:;)

通过下列的程序可以确认在某个硬件平台上的某个操作系统是大端还是小端：

**VB6：**

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | ' 注意这个CopyMemory的声明与一般的不一样，  ' 一般的都是(pDst As Any, pSrc As Any, ByVal ByteLen As Long)  ' 为了能够逐字节访问u，所以前面两个参数改成了按值传递，配合VarPtr函数获取变量地址  Private Declare Sub CopyMemory Lib "kernel32" Alias \_      "RtlMoveMemory" (ByVal pDst As Long, ByVal pSrc As Long, ByVal ByteLen As Long)    Private Sub Form\_Load()      ' VB的Integer占用2个字节！Long才是4个字节      ' 32位应用程序的指针是4个字节      Dim uptr As Long      Dim aptr As Long      Dim bptr As Long      Dim cptr As Long      Dim dptr As Long        Dim u As Long        ' 储存u从低地址到高地址的4个字节      Dim a As Byte      Dim b As Byte      Dim c As Byte      Dim d As Byte        u = 367328153 ' 十六进制数：15 E4 FB 99        uptr = VarPtr(u) ' VarPtr函数是内置函数，但是msdn不说，作用是获取变量的地址        aptr = VarPtr(a)      bptr = VarPtr(b)      cptr = VarPtr(c)      dptr = VarPtr(d)        CopyMemory aptr, uptr + 0, 1 ' 将u逐字节按顺序写入a,b,c,d      CopyMemory bptr, uptr + 1, 1      CopyMemory cptr, uptr + 2, 1      CopyMemory dptr, uptr + 3, 1        ' Windows系统，英特尔处理器：最后输出的是99 FB E4 15      MsgBox Hex(a) & " " & Hex(b) & " " & Hex(c) & " " & Hex(d)  End Sub |

在英特尔处理器，Windows10操作系统上，对话框显示的结果是99 FB E4 15，与直接求出来的16进制值15 E4 FB 99正好相反，所以是小端的。

**C++语言（VS2013下，控制台项目）：**

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | #include <iostream>  using namespace std;    typedef unsigned char byte;    // 转换char（视为整数类型）为16进制字符串  void ChtoHex(byte Val, char\* dest)  {      // 辗转相除法，倒序看得到结果        byte tmp = Val % 16;      if (tmp >= 0 && tmp <= 9)      {          dest[1] = '0' + tmp;      }      else if (tmp >= 10 && tmp <= 15)      {          dest[1] = 'A' + tmp - 10;      }        tmp = (Val/16) % 16;      if (tmp >= 0 && tmp <= 9)      {          dest[0] = '0' + tmp;      }      else if (tmp >= 10 && tmp <= 15)      {          dest[0] = 'A' + tmp - 10;      }        // 设置'\0'      dest[2] = '\0';  }    // 主函数  int main()  {      int u = 367328153; // 原始数据，8位16进制为15 E4 FB 99      byte a, b, c, d;   // u从低地址到高地址的四个字节      // a~d对应的16进制字符串，预留3个字符      char Sa[3], Sb[3], Sc[3], Sd[3];        byte\* k = (byte\*)&u;      a = k[0];      b = k[1];      c = k[2];      d = k[3];        // 转成16进制字符串      ChtoHex(a, Sa);      ChtoHex(b, Sb);      ChtoHex(c, Sc);      ChtoHex(d, Sd);        cout << Sa << " " << Sb << " " << Sc << " " << Sd << endl;        system("pause");      return 0;  } |

在英特尔处理器，Windows10操作系统上，控制台显示的结果是99 FB E4 15，与直接求出来的16进制值15 E4 FB 99正好相反，所以也证明是小端的。

**C语言（VC++6.0，控制台工程）：**

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | #include <stdio.h>  typedef unsigned char byte;    // 转换char（视为整数类型）为16进制字符串  void ChtoHex(byte Val, char\* dest)  {      // 辗转相除法，倒序看得到结果        byte tmp = Val % 16;      if (tmp >= 0 && tmp <= 9)      {          dest[1] = '0' + tmp;      }      else if (tmp >= 10 && tmp <= 15)      {          dest[1] = 'A' + tmp - 10;      }        tmp = (Val/16) % 16;      if (tmp >= 0 && tmp <= 9)      {          dest[0] = '0' + tmp;      }      else if (tmp >= 10 && tmp <= 15)      {          dest[0] = 'A' + tmp - 10;      }        // 设置\0      dest[2] = '\0';  }    // 主函数  void main()  {      int u = 367328153; // 原始数据，8位16进制为15 E4 FB 99      byte a, b, c, d;   // u从低地址到高地址的四个字节      // a~d对应的16进制字符串，预留3个字符      char Sa[3], Sb[3], Sc[3], Sd[3];        byte\* k = (byte\*)&u;      a = k[0];      b = k[1];      c = k[2];      d = k[3];        // 转成16进制字符串      ChtoHex(a, Sa);      ChtoHex(b, Sb);      ChtoHex(c, Sc);      ChtoHex(d, Sd);        printf("%s %s %s %s\n", Sa, Sb, Sc, Sd);        scanf\_s("%d", &a);  } |

在英特尔处理器，Windows10操作系统上，结果跟上面是一样的，输出的16进制数是反序的，证明是小端系统。

**C#（VS2013下，控制台项目）：**

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | using System;    namespace ConsoleApplication1  {      class Program      {          static void Main(string[] args)          {             int u = 367328153;  // 原始数据，8位16进制为15 E4 FB 99             byte[] bytes;       // u从低地址到高地址的四个字节                // 获取              bytes = System.BitConverter.GetBytes(u);                Console.WriteLine(                  bytes[0].ToString("X") + " " +                  bytes[1].ToString("X") + " " +                  bytes[2].ToString("X") + " " +                  bytes[3].ToString("X")                  );                Console.ReadLine();          }      }  } |

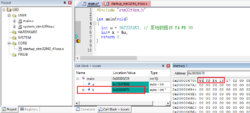
在英特尔处理器，Windows10操作系统上，结果跟上面还还是一样的，输出的16进制数是反序的，证明是小端系统。

**MDK（Keil5，STM32F407）C语言：**

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #include "stm32f4xx.h"    int main(void)  {      int u = 367328153; // 原始数据15 E4 FB 99      int\* k = &u;      return 0;  } |

编译连接然后下载到开发板上，然后启动调试，通过监视窗口可以看到u的地址，然后在内存窗口可以看到字节序是反序的，所以说明STM32F407是小端的。据某些资料说ARM内核是可以设置大小端的，但是STM32是外设自动进入了小端，似乎是无法调整的。

[](http://baike.baidu.com/pic/%E5%A4%A7%E5%B0%8F%E7%AB%AF%E6%A8%A1%E5%BC%8F/6750542/0/55e736d12f2eb938bca91afbd2628535e4dd6ff6?fr=lemma&ct=single)

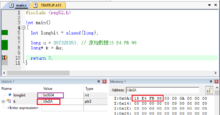
**89C52（Keil5）C语言：**

最后来一个大端的例子。手头上没有51的开发板，所以用的是软件仿真。

[?](http://baike.baidu.com/link?url=VM6ZkZPPXzgG_gsgsS-4WxZz4TIgads67RE3q8JnUwz9DY7oZC53ZRdvoFAUGH5pL32M0rYtt2iaaUG8dOJLnE3sCqjjxfATWoM5-VlAUnZ-PamfYTwqHPN0R_JRoAz_9-guZJ2BkZn_2Qpw-Bq3yq)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include <reg52.h>    int main()  {      int longbit = sizeof(long);        long u = 367328153; // 原始数据15 E4 FB 99      long\* k = &u;        return 0;  } |

注意看了，C52是8位的处理器，long才是4个字节的，看监视窗口longbit的值就知道了（紫色框）。然后再看内存窗口，就会发现u的存储是跟原始数据给的顺序是一样的，所以C51和C52是大端的！！

[](http://baike.baidu.com/pic/%E5%A4%A7%E5%B0%8F%E7%AB%AF%E6%A8%A1%E5%BC%8F/6750542/0/b7003af33a87e950ca639e6517385343fbf2b447?fr=lemma&ct=single)

**大小端模式大小端现阶段状况**

[编辑](javascript:;)

目前Intel的80x86系列芯片是唯一还在坚持使用小端的芯片，ARM芯片默认采用小端，但可以切换为大端；而MIPS等芯片要么采用全部大端的方式储存，要么提供选项支持大端——可以在大小端之间切换。另外，对于大小端的处理也和编译器的实现有关，在C语言中，默认是小端（但在一些对于单片机的实现中却是基于大端，比如Keil 51C），Java是平台无关的，默认是大端。在网络上传输数据普遍采用的都是大端。[