我学习 CRC32、CRC16、CRC 原理和算法的总结（与 WINRAR 结果一致）

wxleasyland(wxlwww@gmail.com)
2010 年 9 月 2 日

比较愚钝，学了 CRC 校验好几天，很痛苦的过程，现终于有眉目了，总结一下。

国外版的“轻松无痛苦学习 CRC 指南”，在

http://www.repairfaq.org/filipg/LINK/F_crc_v31.html

（为什么好的资料都是老外写的？）我的英文有限，这种专业性太强的文章，很多都看不太明白，所以没办法翻译，靠参考国内的翻译和自己瞎琢磨的。

国内的翻译比较不全，而且有点误导，能看英文的还是看英文吧，国内版资料比较零散，可参考：

http://www.q.cc/2001/12/08/10190.html

http://www.360doc.com/content/10/0703/12/1317564_36621098.shtml

http://www.yuanma.org/data/2006/1010/article_1637.htm

我结合国内资料和英文原版进行总结，达到和 WINRAR 一样的 CRC32 计算结果。

一、 CRC 原理

可参考 http://www.luocong.com/articles/show_article.asp?Article_ID=15

计算 CRC 的过程，就是用一个特殊的“除法”，来得到余数，这个余数就是 CRC。

它不是真正的算术上的除法！过程和算术除法过程一样，只是加减运算变成了 XOR（异或）运算！

算术上的除法：

120÷9＝13 余 3，120 是被除数，9 是除数，13 是商，3 是余数。念作 120 除以 9，或者 9 除 120，或者 9 去除 120！（除法的过程就不写了）

这个除法计算机当然会做，但是做起来很麻烦，因为减法有借位，很耗时间和指令！

所以，计算 CRC 也是除法，但是用 XOR 来代替减法，这就简单多了！

CRC 的除法：

120÷9＝14 余 6，商、余数和算术除法不一定相同!! 因为除法用的是 XOR，而不是真正的减法。

以二进制模拟这个计算过程：

```
       1110          商为 1110，即 14，商有 4 位，表示进行了 4 次 XOR
    ________
1001/1111000         被除数 120 是 1111000，除数 9 是 1001
    1001    ^
    ----
     1100      第一次 XOR 后得到 011，加入下一位 0。最高位的 0 可以消掉了，这样最高位是 1,所以下个商是 1
     1001    ^
     ----
      1010     第二次 XOR 后得到 0101，加入下一位 0。最高位的 0 可以消掉了，这样最高位是 1,所以下个商是 1
      1001    ^
      ----
       0110    第三次 XOR 后得到 0011，加入下一位 0。最高位的 0 可以消掉了，这样最高位是 0,所以下个商是 0
       0000    ^
       ----
        110  ->  最后一次 XOR 后得到 0110，最高位的 0 可以消掉了，得到余数为 110，即 6
                 注意，余数不是 0110，而是 110，因为最前面那个 0 已经被 XOR 后消掉了！
```

可见，除法（XOR）的目的是逐步消掉最高位的 1 或 0！

由于过程是 XOR 的，所以商是没有意义的，我们不要。我们要的是余数。

余数 110 是 1111000 的 CRC 吗？不是！

余数 110 是 1111（即十进制 15）的 CRC！！！

为什么？因为 CRC 是和数据一起传送的，所以数据后面要加上 CRC。

数据 1111 加上 CRC110 后，变成 1111110，再传送。接收机收到 1111110 后，除以除数 1001，余数为 000，正确；如果余数不为 0，则说明传送的数据有误！这样完成 CRC 校验。

即发送端要发送 1111，先在 1111 后加 000，变成 1111000，再除以 1001 得到余数 110，这个 110 就是 CRC，将 110 加到数据后面，变成 1111110，发送出去。

接收端收到 1111110，用它除以 1001，计算得余数为 000，就说明收到的数据正确。

所以原始数据后面要先扩展出 3 位 0，以容纳 CRC 值！

会发现，在上面的除法过程中，这 3 位 0，能保证所有的 4 个数据位在除法时都能够被处理到！不然做一次除法就到结果了，那是不对的。这个概念后面要用到。

所以，实际上，数据是 1111，CRC 是 110。

对于除数 1001，我们叫它生成多项式，即生成项，或 POLY，即 g(x)。

数据 1111 根据 POLY1001，计算得到 CRC110。

如果 POLY 不是 1001，而是 1011，那得到的 CRC 也是不同的！

所以生成项不同，得到的 CRC 也不同。要预先定义好 POLY，发送端和接收端要用一样的 POLY！

二、 生成项

上面例子中，生成项是 **1001**，共 **4** 位比特，最高位的 **1**，实际上在除法的每次 **XOR** 时，都要消掉，所以这个 **1** 可不做参考，后 **3** 位 **001** 才是最重要的！**001** 有 **3** 位，所以得到的余数也是 **3** 位，因为最后一次除法 **XOR** 时，最高位消掉了。所以 **CRC** 就是 **3** 位比特的。

CRC 是 **3** 比特，表示它的宽度 **W＝3**。也就是说，原始数据后面要加上 **W=3** 比特的 **0** 进行扩展！

生成项的最低位也必须是 **1**，这是规定的。

生成项 **1001**，就等效于 **g(x)=x2+1**

生成项也可以倒过来写，即颠倒过来，写成 **1001**，这里倒过来的值是一样的。

再如 **CRC32** 的生成项是：

1 0000 0100 1100 0001 0001 1101 1011 0111 （**33** 个比特）

即 **g(x)= x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x+1**

颠倒过来，就可以写成 **1110 1101 1011 1000 1000 0011 0010 0000 1**

一般生成项简写时不写最高位的 **1**，故生成项是 **0x04C11DB7**，颠倒后的生成项是 **0xEDB88320**

CRC32 的生成项是 **33** 比特，最高位是消掉的，即 **CRC** 值是 **32** 比特（**4** 个字节），即宽度 **W=32**，就是说，在计算前，原始数据后面要先扩展 **W=32** 个比特 **0**，即 **4** 个 **0x00** 字节。

注意：我看到网上 **CRC32** 的 **POLY** 有 **0x04C10DB7** 这个值的，它和正规的 **POLY** 值不同，需要注意！

颠倒过来，即是镜像，为什么要颠倒，后述。

三、 直接计算法 Straightforward CRC Implementation

“直接计算法”就是**直接模拟上面的除法的过程**，来得到余数即 **CRC**！

上面的例子中，除数是 **4** 位，但最高位是要一直消掉的，所以我们只需要一个 **3** 位的寄存器就好了。

计算过程：

待测数据后扩展 **W=3** 个比特 **0**，变成 **1111000**；

寄存器初始化置 **0**；

先在寄存器中移入数据 **111**；

寄存器左移一位，并且右边移入下一位数据 **1**。这样最高位 **1** 移出，由于最高位是 **1**，故本次的商是 **1**，要用除数 **1001** 来进行 **XOR**，最高位肯定 **XOR** 得 **0**，故不管它，只要用低 **3** 位 **001** 来进行 **XOR** 就可以，即 **001** 对寄存器进行 **XOR**，寄存器中得到 **110**，即第一次 **XOR** 后的结果（相当于是数据 **1111** 与生成项 **1001** 进行了一次 **XOR**，并把最高位 **0** 消掉了）。 如果移出的最高位是 **0**，则用 **0000** 来进行 **XOR**（**0** **XOR** 后，得到的还是原值）。

一直重复这个过程，就能得到最后余数了。

总共处理次数＝商的位数＝待测数据的位数－生成项位数＋**1**＋宽度 **W**＝待测数据的位数＝**4** 次。

我们假设待测数据是 **1101 0110 11**，生成项是 **10011**，假设有一个 **4 bits** 的寄存器，通过反复的移位和进行 **CRC** 的除法，最终该寄存器中的值就是我们所要求的余数。

```
             3   2   1   0   Bits
           +---+---+---+---+
 Pop   <-- |   |   |   |   | <----- Augmented message（已加 0 扩张的原始数据）
           +---+---+---+---+
         1   0   0   1   1   = The Poly 生成项
```

依据这个模型，我们得到了一个最最简单的算法：

```
把 register 中的值置 0.
把原始的数据后添加 w 个 0.
While (还有剩余没有处理的数据)
   Begin
   把 register 中的值左移一位，读入一个新的数据并置于 register 最低位的位置。
   If (如果上一步的左移操作中的移出的一位是 1)
      register = register XOR Poly.
   End
```

实际上就是模拟 **XOR** 除法的过程，即被测数据一位一位放到寄存器中来做除法。

比如生成项是 **10011**，则生成的余数是 **4** 位 **XXXX**，所以寄存器是 **4** 位。

待测数据是 **1101 0110 11**，后面加上 **0000**，即扩张 **4** 位，以容纳余数。

只要与生成项的 **0011** 做 **XOR** 就好了，最高位经过 **XOR** 肯定出 **0**，可不用最高位。

过程如下：

待测数据先移 **4** 位即 **1101** 到寄存器中，准备开始除法。

第 **1** 次除法：寄存器中是 **1101**，先从寄存器移出最高位 **1**，移进下一位待测数据位 **0**，则寄存器中是 **1010**，由于移出的位是 **1**，则需要与生成项的 **0011** 做 **XOR**，得到 **1001**，即做了第 **1** 次除法后，寄存器中是 **1001**，这个就是余数。

第 **2** 次除法：寄存器中是 **1001**，从寄存器移出最高位 **1**，移进下一位待测数据位 **1**，则寄存器中是 **0011**，由于移出的位是 **1**，则需要与生成项的 **0011** 做 **XOR**，得到 **0000**，即做了第 **2** 次除法后，寄存器中是 **0000**，这个就是余数。

第 **3** 次除法：寄存器中是 **0000**，从寄存器移出最高位 **0**，移进下一位待测数据位 **1**，则寄存器中是 **0001**，由于移出的位是 **0**，则需要不做 **XOR**，直接下一步移位。也可以等同于：本次的商是 **0,0***生成项＝**0**，即是 **0000** 与寄存器做 **XOR**，得到寄存器的数不变，还是 **0001**，即做了第 **3** 次除法后，寄存器中是 **0001**，这个就是余数。

第 **4** 次除法：寄存器中是 **0001**，从寄存器移出最高位 **0**，移进下一位待测数据位 **0**，则寄存器中是 **0010**，由于移出的位是 **0**，则需要不做 **XOR**，直接下一步移位。

第 **5** 次除法：移位，不用做 **XOR**，得到寄存器中是 **0101**

第 **6** 次除法：移位，不用做 **XOR**，得到寄存器中是 **1011**

第 **7** 次除法：移位，移出的位是 **1**，又要与生成项做 **XOR** 了

一直做下去。。。。。。直到最后，寄存器中的就是整个计算后的余数了。即 **CRC** 值。

注意：

这个算法，计算出的 **CRC32** 值**,**与 **WINRAR** 计算出来的不一样，为什么？算法是正确的，不用怀疑！只是 **CRC32** 正式算法还涉及到数据颠倒和初始化预置值等，后述。

程序实现：

程序 1：

（注：网上下的程序是有错的，我有修改了，这里是正确的）

```
    //网上的程序经修改
BYTE POLY=0x13;                         //生成项，13H=10011，这样 CRC 是 4 比特
unsigned short data = 0x035B;     //待测数据是 35BH，12 比特，注意，数据不是 16 比特
unsigned short regi = 0x0000;     // load the register with zero bits

// augment the data by appending W(4) zero bits to the end of it.
//按 CRC 计算的定义，待测数据后加入 4 个比特 0，以容纳 4 比特的 CRC；
```

```
//这样共有16比特待测数据，从第5比特开始做除法，就要做16－5＋1＝12次XOR
data <<= 4;

// we do it bit after bit
for ( int cur_bit = 15; cur_bit >= 0; -- cur_bit )   //处理16次，前4次实际上只是加载数据
{
        // test the highest bit which will be poped later.
        ///     in fact, the 5th bit from right is the hightest bit here
    if ( ( ( regi >> 4 ) & 0x0001 ) == 0x1 )     regi = regi ^ POLY;

    regi <<= 1;   // shift the register
        // reading the next bit of the augmented data
    unsigned short tmp = ( data >> cur_bit ) & 0x0001;  //加载待测数据1比特到tmp中，tmp只有1比特
    regi |= tmp;        //这1比特加载到寄存器中
}
if ( ( ( regi >> 4 ) & 0x0001 ) == 0x1 )   regi = regi ^ POLY;        //做最后一次XOR
//这时， regi中的值就是CRC
```

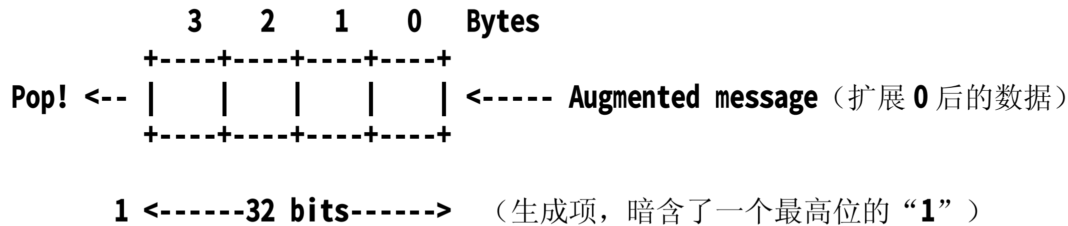
程序2：我做的通用CRC计算程序：

```
_int64  POLY = 0x104C11DB7;       //生成项，需要含有最高位的"1"，这样CRC是32比特
int crcbitnumber=32;              //crc是32比特

_int64  data = 0x31323334;              //待测数据，为字串"1234"
int  databitnumber=32;            //数据是32比特

_int64  regi = 0x0;                     // load the register with zero bits

// augment the data by appending W zero bits to the end of it.
//按CRC计算的定义，加入32个比特0，以容纳32比特的CRC；
//这样共有64比特待测数据，从第33比特开始做除法，就要做64－33＋1＝32次XOR
data <<= crcbitnumber;

// we do it bit after bit
for ( int cur_bit = databitnumber+crcbitnumber-1; cur_bit >= 0; -- cur_bit )   //处理64次（32比特待测数据＋32比特扩展0），前32次是加载数据
{
        // test the highest bit which will be poped later.
        ///     in fact, the 5th bit from right is the hightest bit here
    if ( ( ( regi >> crcbitnumber ) & 0x0001 ) == 0x1 )     regi = regi ^ POLY;

    regi <<= 1;   // shift the register
        // reading the next bit of the augmented data
    unsigned short tmp = ( data >> cur_bit ) & 0x0001;       //加载待测数据1比特到tmp中，tmp只有1比特
    regi |= tmp;        //这1比特加载到寄存器中
}
if ( ( ( regi >> crcbitnumber ) & 0x0001 ) == 0x1 )   regi = regi ^ POLY;       //做最后一次XOR
//这时， regi中的值就是CRC
```

四、 驱动表法 Table-Driven Implementation

上面的“**直接计算法**”很直观，却非常的低效。为了加快它的速度，我们使它一次能处理大于 **4 bit** 的数据。一次能处理一个字节的数据的话，那就方便多了。

我们想要实现的 **32 bit** 的 **CRC** 校验。我们还是假设有和原来一样的一个 **4 "bit"**的 **register**，但它的每一位是一个 **8 bit** 的字节。



（生成项，暗含了一个最高位的“**1**”）

根据同样的原理我们可以得到如下的算法：

While (还有剩余没有处理的数据**)**

Begin

检查 **register** 头字节，并取得它的值

求不同偏移处多项式的 **XOR**

register 左移一个字节，最右处存入新读入的一个字节

把 **register** 的值 和 多项式的 **XOR** 结果 进行 **XOR** 运算

End

可是为什么要这样作呢？ 同样我们还是以一个简单的例子说明问题：

为了简单起见，我们假设一次只移出 **4** 个比特！ 而不是 **8** 个比特。

生成多项式为： **1 0101 1100**，即宽度 **W=8**，即 **CRC8**，这样寄存器为 **8** 位

待测数据是 **1011 0100 1101**

按正常的算法做：

将 **1011 0100** 放入寄存器中，然后开始计算 **CRC**。

先将高 **4** 位移出寄存器：

当前 **register** 中的值： **0100 1101**

4 bit 应该被移出的值： **1011**

生成多项式为： **1010 1110 0**

第一步：

```
Top  Register    （top 指移出的数据）
---- --------
1011 0100 1101                待测数
1010 1110 0    + (CRC XOR)    POLY
-------------
0001 1010 1101                第一次 XOR 后的值
```

第二步：

这时，首 **4 bits** 不为 **0** 说明没有除尽，要继续除：

```
0001 1010 1101
   1 0101 1100 + (CRC XOR)     将 POLY 右移 3 位后，再做 XOR
-------------
0000 1111 0001                 第二次 XOR 后的值
^^^^
```

这时，首 **4 bits** 全 **0** 说明不用继续除了，结果满足要求了。

也就是说：待测数据与 **POLY** 相 **XOR**，得到的结果再与 **POLY** 相 **XOR**，**POLY** 要适当移位，以消掉 **1**。重复进行，直到结果满足要求。

下面，我们换一种算法，来达到相同的目的：

POLY 与 **POLY** 自己先进行 **XOR**，当然 **POLY** 要进行适当移位。使得得到的结果值的高 **4** 位与待测数据相同。

第一步：

```
1010 1110 0             POLY
   1 0101 1100 +        右移 3 位后的 POLY
-------------
1011 1011 1100          POLY 与 POLY 自己进行 XOR 后得到的值
```

第二步：

```
1011 1011 1100        POLY 相 XOR 后得到的值
1011 0100 1101+       待测数据
-------------
0000 11110001         得到的结果值和上面是一样的（说明可以先把 POLY 预先 XOR 好，再与待
测数据 XOR，就能得到结果）
```

结论：

现在我们看到，这二种算法计算的结果是一致的！这是基于 **XOR** 的交换律，即 **(a XOR b) XOR c = a XOR (b XOR c)**。而后一种算法可以通过查表来快速完成，叫做 **“驱动表法”** 算法。

也就是说，根据 4 bit 被移出的值 1011，我们就可以知道要用 POLY 自身 XOR 后得到的 1011 1011 1100 来对待测数据 1011 0100 1101 进行 XOR，这样一次就能消掉 4BIT 待测数据。（注意蓝色的最高 4 位要一样，这样 XOR 后才能得 0000，就能消掉了）

即 1011 对应 1011 1011 1100，实际只需要用到后 8 位，即 1011 对应 1011 1100

用查表法来得到，即 1011 作为索引值，查表，得到表值 1011 1100。

表格可以预先生成。

这里是每次移出 4 位，则 POLY 与 POLY 进行 XOR 的组合有 2^4＝16 种，即从 0000 到 1111。

注意，POLY 自身与自身相 XOR 时，要先对齐到和寄存器一样的长度，再 XOR。相当于有 12 位进行 XOR。

组合后的结果有 16 种：（黑色的 0 表示对齐到和寄存器一样的长度）

1. 0000 0000 0000　　即表示待测数据移出的 4 位都是 0，不需要与 POLY 相 XOR，即相当于待测数据移出的 4 位后，与 0000 0000 0000 相 XOR
2. 0001 0101 1100　即表示待测数据移出的 4 位是 0001，需要与右移过 3 位的 POLY 相 XOR
3. 0010 1011 1000
4. 0010 1011 1000 与 0001 0101 1100 相 XOR，XOR 后前 4 位为 0011 即表示待测数据移出的 4 位后，需要与 POLY 进行二次相 XOR，结果才能满足要求。
5. 0101 0111 0000 与 0001 0101 1100 相 XOR，　XOR 后前 4 位为 0100
6. 0101 0111 0000 ，　前 4 位为 0101
7. 0101 0111 0000 与 0010 1011 1000、0001 0101 1100 相 XOR，　XOR 后前 4 位为 0110
8. 0101 0111 0000 与 0010 1011 1000，　XOR 后前 4 位为 0111
9. 1010 1110 0000 与 0010 1011 1000 相 XOR，　XOR 后前 4 位为 1000
10. 1010 1110 0000 与 0010 1011 1000、0001 0101 1100 相 XOR，　XOR 后前 4 位为 1001
11. 1010 1110 0000，　前 4 位为 1010
12. 1010 1110 0000 与 0001 0101 1100 相 XOR，　XOR 后前 4 位为 1011
13. 1010 1110 0000 与 0101 0111 0000、0010 1011 1000、0001 0101 1100 相 XOR，　XOR 后前 4 位为 1100
14. 1010 1110 0000 与 0101 0111 0000、0010 1011 1000 相 XOR，　XOR 后前 4 位为 1101
15. 1010 1110 0000 与 0101 0111 0000、0001 0101 1100 相 XOR，　XOR 后前 4 位为 1110
16. 1010 1110 0000 与 0101 0111 0000 相 XOR，　XOR 后前 4 位为 1111

以 XOR 后得到的结果的前 4 位做为索引值，以 XOR 后得到的结果的后 8 位做为表值，生成一张表，即：

```
TABLE[0]=0000 0000B;
TABLE[1]=0101 1100B;
TABLE[2]=1011 1000B;
TABLE[3]=[(0010 1011 1000B ^ 0001 0101 1100B) >> 4 ] & 0xff
....
```

这张表我叫它为“**直接查询表**”。

就是说，一次移出的待测数据的 4 位 bit，有 2^4 个值，即 0000,0001,0010,....,1111，根据这个值来查表，找到相应的表值，再用表值来 XOR 寄存器中的待测数据。

所以，如果一次移出待测数据的 8 位 bit，即一次进行一个字节的计算，则表格有 2^8＝256 个表值。CRC16 和 CRC32 都是一次处理一个字节的，所以它们的查询表有 256 个表值。

“驱动表法”算法为：

```
               3    2    1    0   Bytes
            +----+----+----+----+
     +-----<|    |    |    |    | <----- Augmented message（扩展 0 后的数据）
     |      +----+----+----+----+
     |      MSB       ^         LSB
     |                |
     |               XOR
     |                |
     |     0+----+----+----+----+
  查表 v     +----+----+----+----+
     |      +----+----+----+----+
     |      +----+----+----+----+
     |      +----+----+----+----+
     |      +----+----+----+----+
     |      +----+----+----+----+
     +----->+----+----+----+----+
            +----+----+----+----+
            +----+----+----+----+
            +----+----+----+----+
            +----+----+----+----+
         255+----+----+----+----+
```

描述：

1：register 左移一个字节,从原始数据中读入一个新的字节.

2：利用刚从 register 移出的字节作为下标定位 table 中的一个 32 位的值

3：把这个值 XOR 到 register 中。

4：如果还有未处理的数据则回到第一步继续执行。

用 C 可以写成这样：

```
r=0;                //r 是寄存器，先初始化为 0
while (len--)       //len 是已扩展 0 之后的待测数据的长度
  {
   byte t = (r >> 24) & 0xFF;
   r = (r << 8) | *p++;        //p 是指向待测数据的指针（待测数据需已经扩展过 0）
   r^=table[t];             //table 是查询表
  }
```

这个代码可以优化为：

```
r=0;            //r 是寄存器，先初始化为 0
 while (len--) //len 是已扩展 0 之后的待测数据的字节长度
       r = ((r << 8) | *p++) ^ t[(r >> 24) & 0xFF]; //p 是指向待测数据的指针（待测数据需已经扩展过 0）,t 是查询表
```

注意：

这个**“驱动表法”**算法和**“直接计算法”**是完全一样的，不仅结果完全一样，处理方式也是完全一样的，所以**“驱动表法”可以完全替代“直接计算法”**！

原始数据都需要先用 0 扩展 W 位；最开始的几次循环的实质都只是先将待测数据移动到寄存器中去而已；

会发现，这个算法用到的“**直接查询表**”的表值，和网上公开的查询表（我叫它“**正规查询表**”）的表值不一样！为什么？因为网上公开的正规查询表是用于“颠倒”算法的！后述。

会发现，这个算法，计算出的 CRC32 值，同样与 WINRAR 计算出来的不一样。

生成的“直接查询表”的内容是：

CRC16 直接查询表

00H	0000	8005	800F	000A
04H	801B	001E	0014	8011
08H	8033	0036	003C	8039
0CH	0028	802D	8027	0022
10H	8063	0066	006C	8069
14H	0078	807D	8077	0072
18H	0050	8055	805F	005A
1CH	804B	004E	0044	8041
20H	80C3	00C6	00CC	80C9
24H	00D8	80DD	80D7	00D2
28H	00F0	80F5	80FF	00FA
2CH	80EB	00EE	00E4	80E1
30H	00A0	80A5	80AF	00AA
34H	80BB	00BE	00B4	80B1
38H	8093	0096	009C	8099
3CH	0088	808D	8087	0082
40H	8183	0186	018C	8189
44H	0198	819D	8197	0192
48H	01B0	81B5	81BF	01BA
4CH	81AB	01AE	01A4	81A1
50H	01E0	81E5	81EF	01EA
54H	81FB	01FE	01F4	81F1
58H	81D3	01D6	01DC	81D9
5CH	01C8	81CD	81C7	01C2
60H	0140	8145	814F	014A
64H	815B	015E	0154	8151

68H	8173	0176	017C	8179
6CH	0168	816D	8167	0162
70H	8123	0126	012C	8129
74H	0138	813D	8137	0132
78H	0110	8115	811F	011A
7CH	810B	010E	0104	8101
80H	8303	0306	030C	8309
84H	0318	831D	8317	0312
88H	0330	8335	833F	033A
8CH	832B	032E	0324	8321
90H	0360	8365	836F	036A
94H	837B	037E	0374	8371
98H	8353	0356	035C	8359
9CH	0348	834D	8347	0342
A0H	03C0	83C5	83CF	03CA
A4H	83DB	03DE	03D4	83D1
A8H	83F3	03F6	03FC	83F9
ACH	03E8	83ED	83E7	03E2
B0H	83A3	03A6	03AC	83A9
B4H	03B8	83BD	83B7	03B2
B8H	0390	8395	839F	039A
BCH	838B	038E	0384	8381
C0H	0280	8285	828F	028A
C4H	829B	029E	0294	8291
C8H	82B3	02B6	02BC	82B9
CCH	02A8	82AD	82A7	02A2
D0H	82E3	02E6	02EC	82E9
D4H	02F8	82FD	82F7	02F2
D8H	02D0	82D5	82DF	02DA
DCH	82CB	02CE	02C4	82C1
E0H	8243	0246	024C	8249
E4H	0258	825D	8257	0252
E8H	0270	8275	827F	027A
ECH	826B	026E	0264	8261
F0H	0220	8225	822F	022A
F4H	823B	023E	0234	8231
F8H	8213	0216	021C	8219
FCH	0208	820D	8207	0202

CRC32 直接查询表

00H	00000000	04C11DB7	09823B6E	0D4326D9
04H	130476DC	17C56B6B	1A864DB2	1E475005
08H	2608EDB8	22C9F00F	2F8AD6D6	2B4BCB61
0CH	350C9B64	31CD86D3	3C8EA00A	384FBDBD
10H	4C11DB70	48D0C6C7	4593E01E	4152FDA9
14H	5F15ADAC	5BD4B01B	569796C2	52568B75
18H	6A1936C8	6ED82B7F	639B0DA6	675A1011
1CH	791D4014	7DDC5DA3	709F7B7A	745E66CD

20H	9823B6E0	9CE2AB57	91A18D8E	95609039
24H	8B27C03C	8FE6DD8B	82A5FB52	8664E6E5
28H	BE2B5B58	BAEA46EF	B7A96036	B3687D81
2CH	AD2F2D84	A9EE3033	A4AD16EA	A06C0B5D
30H	D4326D90	D0F37027	DDB056FE	D9714B49
34H	C7361B4C	C3F706FB	CEB42022	CA753D95
38H	F23A8028	F6FB9D9F	FBB8BB46	FF79A6F1
3CH	E13EF6F4	E5FFEB43	E8BCCD9A	EC7DD02D
40H	34867077	30476DC0	3D044B19	39C556AE
44H	278206AB	23431B1C	2E003DC5	2AC12072
48H	128E9DCF	164F8078	1B0CA6A1	1FCDBB16
4CH	018AEB13	054BF6A4	0808D07D	0CC9CDCA
50H	7897AB07	7C56B6B0	71159069	75D48DDE
54H	6B93DDDB	6F52C06C	6211E6B5	66D0FB02
58H	5E9F46BF	5A5E5B08	571D7DD1	53DC6066
5CH	4D9B3063	495A2DD4	44190B0D	40D816BA
60H	ACA5C697	A864DB20	A527FDF9	A1E6E04E
64H	BFA1B04B	BB60ADFC	B6238B25	B2E29692
68H	8AAD2B2F	8E6C3698	832F1041	87EE0DF6
6CH	99A95DF3	9D684044	902B669D	94EA7B2A
70H	E0B41DE7	E4750050	E9362689	EDF73B3E
74H	F3B06B3B	F771768C	FA325055	FEF34DE2
78H	C6BCF05F	C27DEDE8	CF3ECB31	CBFFD686
7CH	D5B88683	D1799B34	DC3ABDED	D8FBA05A
80H	690CE0EE	6DCDFD59	608EDB80	644FC637
84H	7A089632	7EC98B85	738AAD5C	774BB0EB
88H	4F040D56	4BC510E1	46863638	42472B8F
8CH	5C007B8A	58C1663D	558240E4	51435D53
90H	251D3B9E	21DC2629	2C9F00F0	285E1D47
94H	36194D42	32D850F5	3F9B762C	3B5A6B9B
98H	0315D626	07D4CB91	0A97ED48	0E56F0FF
9CH	1011A0FA	14D0BD4D	19939B94	1D528623
A0H	F12F560E	F5EE4BB9	F8AD6D60	FC6C70D7
A4H	E22B20D2	E6EA3D65	EBA91BBC	EF68060B
A8H	D727BBB6	D3E6A601	DEA580D8	DA649D6F
ACH	C423CD6A	C0E2D0DD	CDA1F604	C960EBB3
B0H	BD3E8D7E	B9FF90C9	B4BCB610	B07DABA7
B4H	AE3AFBA2	AAFBE615	A7B8C0CC	A379DD7B
B8H	9B3660C6	9FF77D71	92B45BA8	9675461F
BCH	8832161A	8CF30BAD	81B02D74	857130C3
C0H	5D8A9099	594B8D2E	5408ABF7	50C9B640
C4H	4E8EE645	4A4FFBF2	470CDD2B	43CDC09C
C8H	7B827D21	7F436096	7200464F	76C15BF8
CCH	68860BFD	6C47164A	61043093	65C52D24
D0H	119B4BE9	155A565E	18197087	1CD86D30
D4H	029F3D35	065E2082	0B1D065B	0FDC1BEC
D8H	3793A651	3352BBE6	3E119D3F	3AD08088
DCH	2497D08D	2056CD3A	2D15EBE3	29D4F654
E0H	C5A92679	C1683BCE	CC2B1D17	C8EA00A0
E4H	D6AD50A5	D26C4D12	DF2F6BCB	DBEE767C

E8H	E3A1CBC1	E760D676	EA23F0AF	EEE2ED18
ECH	F0A5BD1D	F464A0AA	F9278673	FDE69BC4
F0H	89B8FD09	8D79E0BE	803AC667	84FBDBD0
F4H	9ABC8BD5	9E7D9662	933EB0BB	97FFAD0C
F8H	AFB010B1	AB710D06	A6322BDF	A2F33668
FCH	BCB4666D	B8757BDA	B5365D03	B1F740B4

“驱动表法”的程序：

```
// 注意：因生成项 POLY 最高位一定为“1”，故略去最高位的"1"，
unsigned short cnCRC_16 = 0x8005;        // CRC-16 = X16 + X15 + X2 + X0
unsigned short cnCRC_CCITT = 0x1021; // CRC-CCITT = X16 + X12 + X5 + X0，据说这个 16 位 CRC 多项式比上一个要好
unsigned long cnCRC_32 = 0x04C11DB7; //采用正规的 CRC32 的 POLY
unsigned long Table_CRC16[256];          // CRC16 表
unsigned long Table_CRC32[256];          // CRC32 表


// 构造 16 位 CRC 表 "直接查询表"
unsigned short i16, j16;
unsigned short nData16;
unsigned short nAccum16;
for ( i16 = 0; i16 < 256; i16++ )
{
    nData16 = ( unsigned short )( i16 << 8 );
    nAccum16 = 0;
    for ( j16 = 0; j16 < 8; j16++ )
    {
        if ( ( nData16 ^ nAccum16 ) & 0x8000 )
        nAccum16 = ( nAccum16 << 1 ) ^ cnCRC_16;   //也可以用 cnCRC_CCITT
        else
        nAccum16 <<= 1;
        nData16 <<= 1;
    }
    Table_CRC16[i16] = ( unsigned long )nAccum16;
}

// 构造 32 位 CRC 表 "直接查询表"
unsigned long i32, j32;
unsigned long nData32;
unsigned long nAccum32;
for ( i32 = 0; i32 < 256; i32++ )
{
    nData32 = ( unsigned long )( i32 << 24 );
    nAccum32 = 0;
    for ( j32 = 0; j32 < 8; j32++ )
    {
        if ( ( nData32 ^ nAccum32 ) & 0x80000000 )
            nAccum32 = ( nAccum32 << 1 ) ^ cnCRC_32;
        else
            nAccum32 <<= 1;
        nData32 <<= 1;
    }
    Table_CRC32[i32] = nAccum32;
```

```
}

unsigned char aData[512]={0x31,0x32,0x33,0x34};              //待测数据，为字串"1234"
unsigned long aSize;
unsigned long i;
unsigned char *point;

// 计算 16 位 CRC 值，CRC-16 或 CRC-CCITT
//Table-Driven 驱动表法，需要用到“直接查询表”（不能用“正规查询表”）；待测数据需扩展 0
unsigned short CRC16_1;
aSize=4;                //数据长度字节（不包含扩展 0）
CRC16_1 = 0;            //寄存器归 0
point=aData;
while (aSize--)
    CRC16_1 = ((CRC16_1 << 8) | *point++) ^ Table_CRC16[(CRC16_1 >> 8) & 0xFF];
for ( i = 0; i < 2; i++ )
    CRC16_1 = ((CRC16_1 << 8) ) ^ Table_CRC16[(CRC16_1 >> 8) & 0xFF];  //加入 2 字节的扩展 0
//这时， CRC16_1 中的值就是 CRC


// 计算 32 位 CRC-32 值
//Table-Driven 驱动表法，需要用到“直接查询表”（不能用“正规查询表”）；待测数据需扩展 0
unsigned long CRC32_1;
aSize=4;                //数据长度字节（不包含扩展 0）
CRC32_1=0x0;            //寄存器归 0
point=aData;
while (aSize--)
    CRC32_1 = ((CRC32_1 << 8) | *point++) ^ Table_CRC32[(CRC32_1 >> 24) & 0xFF];
for ( i = 0; i < 4; i++ )   CRC32_1 = ((CRC32_1 << 8) ) ^ Table_CRC32[(CRC32_1 >> 24) & 0xFF];//加入 4 字节的扩展 0
//这时， CRC32_1 中的值就是 CRC
```

打印查询表的语句：（在 TC 中实现）

```
for ( i16 = 0; i16 < 256; i16++ )
{
printf("%02xh    %04x  %04x  %04x  %04x\n",i16,( unsigned short)Table_CRC16[i16],( unsigned
short)Table_CRC16[i16+1],( unsigned short)Table_CRC16[i16+2],( unsigned short)Table_CRC16[i16+3]);
i16++;
i16++;
i16++;
}

for ( i16 = 0; i16 < 256; i16++ )
{
printf("%02xh    %08lx  %08lx  %08lx
%08lx\n",i16,Table_CRC32[i16],Table_CRC32[i16+1],Table_CRC32[i16+2],Table_CRC32[i16+3]);
i16++;
i16++;
i16++;
}
```

五、 直驱表法 DIRECT TABLE ALGORITHM

对于上面的算法：

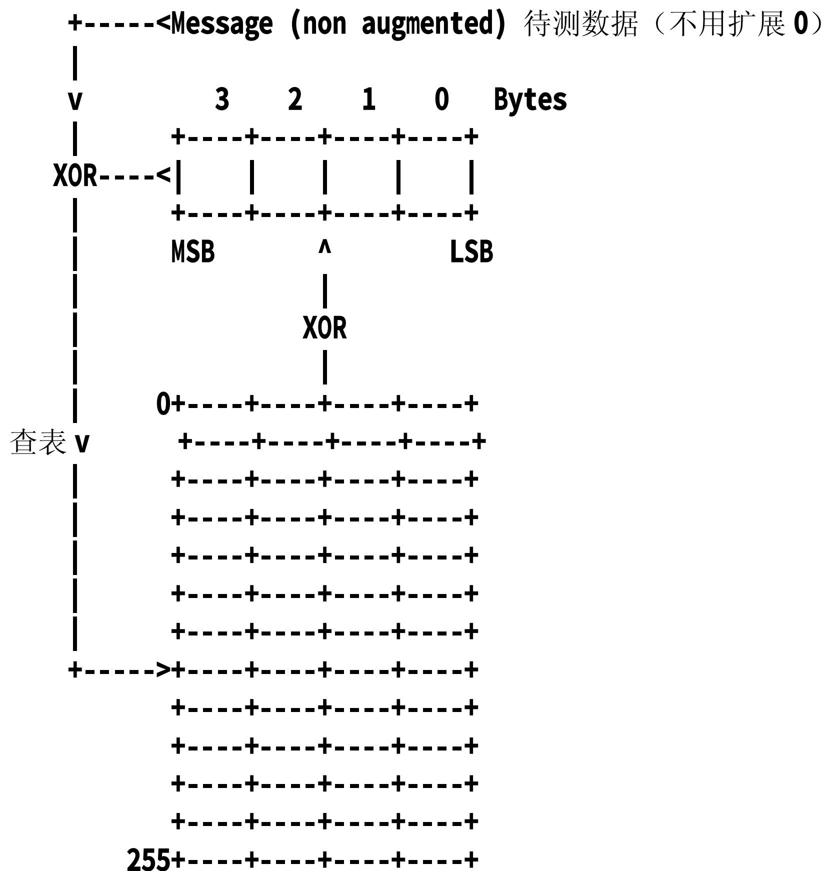
1．对于尾部的 **w/8** 个扩展 **0** 字节，事实上它们的作用只是确保所有的原始数据都已被送入 **register**，并且被算法处理。

2．如果 **register** 中的初始值是 **0**，那么开始的 **4** 次循环，作用只是把原始数据的头 **4** 个字节送入寄存器，而没有进行真正的除法操作。就算初始值不是 **0**（我注：这里并没有说是“寄存器的初始值”，而是指算法开始时的“初始化值”），开始的 **4** 次循环也只是把原始数据的头 **4** 个字节移入到 **register** 中，然后再把它们和一个特定常数相 **XOR**（我注：即这时进行初始化操作，后述）。

3．因为有交换律：**(A xor B) xor C = A xor (B xor C)**

这些信息意味着，上面提到的算法可以被优化，待测数据不需要先循环几次进入到寄存器中后再进行处理，而是数据直接就可以处理到寄存器中。

可以这样：数据可以先与刚从寄存器移出的字节相 **XOR**，用得到的结果值进行查表，再用表值 **XOR** 寄存器。这引出了以下**“直驱表法”**算法：



“直驱表法”算法：

1. **Shift the register left by one byte, reading in a new message byte.**（我注：老外的这句话有问题，应只是 **Shift the register left by one byte**，而不将新的信息字节读入 **register** 中。所以翻译为：寄存器左移一个字节）

2. **XOR the top byte just rotated out of the register with the next message byte to yield an index into the table ([0,255]).** 将刚从 **register** 移出的字节与新的信息字节相 **XOR**，结果值作为定位索引，从查询表中取得相应的表值。

3. **XOR the table value into the register.** 把表值 **XOR** 到 **register** 中

4. **Goto 1 iff more augmented message bytes.** 如果还有未处理的数据则回到第一步继续执行。

用 **C** 可以写成这样：

```
r=0;                    //r 是寄存器，先初始化为 0
while (len--)                   //len 是待测数据（不用扩展 0）的字节长度
        r = (r<<8) ^ t[(r >> 24) ^ *p++];   //p 是指向待测数据的指针,t 是查询表
```

算法相当于：

寄存器左移出 **1** 字节，右边补 **0**；

移出的字节与待测信息字节进行 **XOR**，根据结果值查表，得表值

表值与寄存器进行 **XOR**

注意：

这个 **“直驱表法”** 算法的数学原理我也不明白，但它肯定是从 **“驱动表法”** 算法推导出来的，得到的 **CRC** 结果值是完全一样的！只是它们的处理方式是不一样的。

这个算法很方便，原始数据不需要先用 **0** 扩展 **W** 位；

这个算法很方便，第一次循环就能够处理待测数据并在寄存器中生成结果，而不单纯只是将数据移动到寄存器中去而已；

这个算法，用的是和 **“驱动表法”** 同样的“直接查询表”，而不是“正规查询表”。

正是由于处理方式不一样，所以如果寄存器初始化预置值不为 **0**，那么本算法可不受影响，而 **“驱动表法”** 则需要将预置值再另外 **XOR** 到寄存器中。后述。

“直驱表法”的程序：

```
// 注意：因生成项 POLY 最高位一定为“1”，故略去最高位的"1"，
unsigned short cnCRC_16 = 0x8005;         // CRC-16 = X16 + X15 + X2 + X0
unsigned short cnCRC_CCITT = 0x1021;  // CRC-CCITT = X16 + X12 + X5 + X0，据说这个 16 位 CRC 多项式比上一个要好
unsigned long cnCRC_32 = 0x04C11DB7;  //采用正规的 CRC32 的 POLY
unsigned long Table_CRC16[256];           // CRC16 表
unsigned long Table_CRC32[256];           // CRC32 表


// 构造 16 位 CRC 表 "直接查询表"
unsigned short i16, j16;
unsigned short nData16;
unsigned short nAccum16;
for ( i16 = 0; i16 < 256; i16++ )
{
    nData16 = ( unsigned short )( i16 << 8 );
    nAccum16 = 0;
    for ( j16 = 0; j16 < 8; j16++ )
    {
        if ( ( nData16 ^ nAccum16 ) & 0x8000 )
        nAccum16 = ( nAccum16 << 1 ) ^ cnCRC_16;   //也可以用 cnCRC_CCITT
        else
        nAccum16 <<= 1;
        nData16 <<= 1;
    }
    Table_CRC16[i16] = ( unsigned long )nAccum16;
}

// 构造 32 位 CRC 表 "直接查询表"
unsigned long i32, j32;
unsigned long nData32;
unsigned long nAccum32;
for ( i32 = 0; i32 < 256; i32++ )
{
    nData32 = ( unsigned long )( i32 << 24 );
    nAccum32 = 0;
    for ( j32 = 0; j32 < 8; j32++ )
    {
        if ( ( nData32 ^ nAccum32 ) & 0x80000000 )
            nAccum32 = ( nAccum32 << 1 ) ^ cnCRC_32;
        else
            nAccum32 <<= 1;
        nData32 <<= 1;
    }
    Table_CRC32[i32] = nAccum32;
}


unsigned char aData[512]={0x31,0x32,0x33,0x34};              //待测数据，为字串"1234"
unsigned long aSize;
unsigned long i;
unsigned char *point;

// 计算 16 位 CRC 值，CRC-16 或 CRC-CCITT
//DIRECT TABLE 直驱表法，需要用到“直接查询表”（不能用“正规查询表”）；待测数据不需要扩展 0
unsigned short CRC16_2;
aSize=4;                  //数据长度字节（数据不用扩展 0 了）
CRC16_2 = 0;              //寄存器中预置初始值
point=aData;
```

```
for ( i = 0; i < aSize; i++ )
    CRC16_2 = ( CRC16_2 << 8 ) ^ ( unsigned short ) Table_CRC16[( CRC16_2 >> 8 ) ^ *point++];
//这时， CRC16_2 中的值就是 CRC


// 计算 32 位 CRC-32 值
//DIRECT TABLE 直驱表法，需要用到“直接查询表”（不能用“正规查询表”）；待测数据不需要扩展 0
unsigned long CRC32_2;
aSize=4;                //数据长度字节（数据不用扩展 0 了）
CRC32_2 = 0x0;                //寄存器中预置初始值
point=aData;
for ( i = 0; i < aSize; i++ )
    CRC32_2 = ( CRC32_2 << 8 ) ^ Table_CRC32[( CRC32_2 >> 24 ) ^ *point++];
//这时， CRC32_2 中的值就是 CRC
```

六、 CRC 的参数模型

实际上，这时已经可以计算出与 **WINRAR** 相同的 **CRC32** 值了。但是会发现，算出的结果和 **WINRAR** 是不一样的，为什么？因为不仅仅是生成项 **POLY** 会影响到 **CRC** 值，还有很多参数会影响到最终的 **CRC** 值！

CRC 计算，需要有 **CRC** 参数模型，比如 **CRC32** 的参数模型是：

```
Name   : "CRC-32"
Width  : 32
Poly   : 04C11DB7
Init   : FFFFFFFF
RefIn  : True
RefOut : True
XorOut : FFFFFFFF
Check  : CBF43926
```

解释：

NAME

名称

WIDTH

宽度，即 **CRC** 比特数

POLY

生成项的简写。以 **16** 进制表示，即是 **0x04C11DB7**。忽略了最高位的**"1"**，即完整的生成项是 **0x104C11DB7**。

重要的一点是，这是“未颠倒”的生成项！前面说过，“颠倒的”生成项是 **0xEDB88320**。

INIT

这是算法开始时寄存器的初始化预置值，十六进制表示。

这个值可以直接赋值给“直驱表法”算法中的寄存器，作为寄存器的初始值！

而对于“驱动表法”算法及“直接计算法”，寄存器的初始值必须是 **0**！前面几次循环先将待测数据移入到寄存器中，当寄存器装满后，再用这个初始化预置值去 **XOR** 寄存器，这样寄存器就被这个值初始化了！

这点很重要！！如果在“驱动表法”算法开始时，寄存器的初始值不为 **0**，那么寄存器中的值就会相当于是待测数据了，这样算出的 **CRC** 结果就不对了！我们的目的是用预置值去初始化寄存器，而不是将预置值作为待测数据去处理！

REFIN

这个值是真 **TRUE** 或假 **FALSE**。

如果这个值是 **FALSE**，表示待测数据的每个字节都不用“颠倒”，即 **BIT7** 仍是作为最高位，**BIT0** 作为最低位。

如果这个值是 **TRUE**，表示待测数据的每个字节都要先“颠倒”，即 **BIT7** 作为最低位，**BIT0** 作为最高位。

REFOUT

这个值是真 **TRUE** 或假 **FALSE**。

如果这个值是 **FALSE**，表示计算结束后，寄存器中的值直接进入 **XOROUT** 处理即可。

如果这个值是 **TRUE**，表示计算结束后，寄存器中的值要先“颠倒”，再进入 **XOROUT** 处理。注意，这是将**整个寄存器的值颠倒**，因为寄存器的各个字节合起来表达了一个值，如果只是对各个字节各自颠倒，那结果值就错误了。

XOROUT

这是 **W** 位长的 **16** 进制数值。

这个值与经 **REFOUT** 后的寄存器的值相 **XOR**，得到的值就是最终正式的 **CRC** 值！

CHECK

这不是定义值的一部分，这只是字串**"123456789"**用这个 **CRC** 参数模型计算后得到的 **CRC** 值，作为参考。

我们发现，**CRC32** 模型的 **Init=0xFFFFFFFF**，就是说寄存器要用 **0xFFFFFFFF** 进行初始化，而不是 **0**。

为什么？因为待测数据的内容和长度是随机的，如果寄存器初始值为 **0**，那么，待测字节是 **1** 字节的 **0x00**，与待测字节是 **N** 字节的 **0x00**，计算出来的 **CRC32** 值都是 **0**，那 **CRC** 值就没有意义了！所以寄存器用 **0xFFFFFFFF** 进行初始化，就可以避免这个问题了！

RefIn=True，表示输入数据的每个字节需要“颠倒”！为什么要“颠倒”，因为很多硬件在发送时是先发送最低位 **LSB** 的！比如 **UART** 等。

字节顺序不用颠倒，只是每个字节内部的比特进行颠倒。例如待测的字串是**"1234"**，这时也是一样先处理**"1"**，再处理**"2"**，一直到处理**"4"**。处理字符**"1"**时，它是 **0x31**，即 **0011 0001**，需要先将它颠倒，变成低位在前，即 **1000 1100**，即 **0x8C**，再进行处理。

也就是说，待处理的数据是 **0x31 32 33 34**，颠倒后就变成 **0x8C 4C CC 2C**，再进行 **CRC** 计算。

RefOut=True，表示计算完成后，要将寄存器中的值再颠倒。注意，这是将**整个寄存器**的值颠倒，即如果寄存器中的值是 **0x31 32 33 34**，颠倒后就变成 **0x2C CC 4C 8C**！

XorOut=FFFFFFFF，表示还需要将结果值与 **0xffffffff** 进行 **XOR**，这样就得到最终的 **CRC32** 值了！

我们直接用“**直驱表法**”，计算字串**"1234"**的 **CRC32** 值。

程序如下：

要先做一个颠倒比特的子程序：

```
unsigned long int Reflect(unsigned long int ref, char ch)
{
unsigned long int value=0;
    // 交换 bit0 和 bit7，bit1 和 bit6，类推
for(int i = 1; i < (ch + 1); i++)
{
    if(ref & 1)
        value |= 1 << (ch - i);
    ref >>= 1;
}
return value;
}
```

在主程序中的程序：

```
// 注意：因生成项 POLY 最高位一定为“1”，故略去最高位的"1"，
unsigned long cnCRC_32 = 0x04C11DB7; //采用正规的 CRC32 的 POLY
unsigned long Table_CRC32[256];          // CRC32 表

// 构造 32 位 CRC 表 "直接查询表"
unsigned long i32, j32;
unsigned long nData32;
unsigned long nAccum32;
for ( i32 = 0; i32 < 256; i32++ )
{
    nData32 = ( unsigned long )( i32 << 24 );
    nAccum32 = 0;
    for ( j32 = 0; j32 < 8; j32++ )
    {
        if ( ( nData32 ^ nAccum32 ) & 0x80000000 )
            nAccum32 = ( nAccum32 << 1 ) ^ cnCRC_32;
        else
            nAccum32 <<= 1;
        nData32 <<= 1;
    }
    Table_CRC32[i32] = nAccum32;
}

unsigned char aData[512]={0x31,0x32,0x33,0x34};            //待测数据，为字串"1234"
unsigned long aSize;
unsigned long i;
unsigned char *point;
unsigned char chtemp;
// 计算 32 位 CRC-32 值
//Table-Driven 驱动表法，需要用到“直接查询表”（不能用“正规查询表”）；待测数据需扩展 0
unsigned long ii;
unsigned long CRC32_1;
aSize=4;                //数据长度字节（不包含扩展 0）
CRC32_1=0x0;            //寄存器归 0
point=aData;
ii=0;
while (aSize--)
{
    chtemp=*point++;
    chtemp=(unsigned char)Reflect(chtemp, 8);      //将数据字节内部的比特进行颠倒
    CRC32_1 = ((CRC32_1 << 8) | chtemp) ^ Table_CRC32[(CRC32_1 >> 24) & 0xFF];
    ii++;
    if (ii==4) CRC32_1=CRC32_1^0xffffffff;//当寄存器装满 4 个字节后，用预置值 0xffffffff 去 XOR 寄存器，这样寄存器就
```

```
被这个值初始化了！
}
for ( i = 0; i < 4; i++ )
{
    CRC32_1 = ((CRC32_1 << 8) ) ^ Table_CRC32[(CRC32_1 >> 24) & 0xFF];  //加入 4 字节的扩展 0
    ii++;
    if (ii==4) CRC32_1=CRC32_1^0xffffffff;//如果待测数据小于 4 字节，则只有在这里寄存器才会装满 4 个字节，才进行初始化
}
CRC32_1=Reflect(CRC32_1, 32);    //颠倒寄存器的值
CRC32_1=CRC32_1^0xffffffff;      //寄存器的值与 0xffffffff 异或
//这时， CRC32_1 中的值就是 CRC


//DIRECT TABLE 直驱表法，需要用到“直接查询表”（不能用“正规查询表”）；待测数据不需要扩展 0
unsigned long CRC32_2;
aSize=4;                //数据长度字节（数据不用扩展 0 了）
CRC32_2 = 0xffffffff;             //寄存器中直接预置初始值 0xffffffff 即可
point=aData;
for ( i = 0; i < aSize; i++ )
{
    chtemp=*point++;
    chtemp=(unsigned char)Reflect(chtemp, 8);      //将数据字节内部的比特进行颠倒
    CRC32_2 = ( CRC32_2 << 8 ) ^ Table_CRC32[( CRC32_2 >> 24 ) ^ chtemp];
}

CRC32_2=Reflect(CRC32_2, 32);    //颠倒寄存器的值
CRC32_2=CRC32_2^0xffffffff;     //寄存器的值与 0xffffffff 异或
//这时， CRC32_2 中的值就是 CRC
```

得到的结果与 WINRAR 的计算结果是完全一样的！成功了！

其它的 CRC 参数模型：

```
Name   : "CRC-16"
Width  : 16
Poly   : 8005
Init   : 0000
RefIn  : True
RefOut : True
XorOut : 0000
Check  : BB3D

Name   : "CRC-16/CITT"
Width  : 16
Poly   : 1021
Init   : FFFF
RefIn  : False
RefOut : False
XorOut : 0000
Check  : ?
```

```
Name   : "XMODEM"
Width  : 16
Poly   : 8408
Init   : 0000
RefIn  : True
RefOut : True
XorOut : 0000
Check  : ?
```

```
Name   : "ARC"
Width  : 16
Poly   : 8005
Init   : 0000
RefIn  : True
RefOut : True
XorOut : 0000
Check  : ?
```

七、 最后的战斗—“颠倒的直驱表法”算法 "Reflected" Table-Driven Implementations

颠倒，也就是镜像！

CRC32 要求输入的字节要颠倒，那么在程序中，在对每个字节处理前，还要先把这个字节先颠倒一下，再处理，那不是超级麻烦！

所以就把“**直驱表法**”算法颠倒一下（查询表颠倒），那么算法就可以直接处理不颠倒的字节了，就方便多了。

我们把算法照镜子：

```
          “直驱表法”                      镜子                 “颠倒的直驱表法”

  +-----<Message (non augmented) 待测数据（要颠倒） | 待测数据（不颠倒）Message (non augmented) >-----+
  |                                              |                                                  |
  v        3    2    1    0   Bytes              |                 Bytes   3    2    1    0       v
  |     +----+----+----+----+                    |                      +----+----+----+----+     |
XOR----<|    |    |    |    |                    |                      |    |    |    |    |>----XOR
  |     +----+----+----+----+                    |                      +----+----+----+----+     |
  |     MSB        ^        LSB                  |                      MSB        ^        LSB     |
  |                |                             |                                 |                |
  |               XOR                            |                                XOR               |
  |                |                             |                                 |                |
  |   00H +----+----+----+----+                  |                      +----+----+----+----+ 00H   |
查表 v  01H +----+----+----+----+ 04C11DB7H       |            EDB88320H +----+----+----+----+ 80H   v 查表
  |   02H +----+----+----+----+                  |                      +----+----+----+----+ C0H   |
  |   03H +----+----+----+----+                  |                      +----+----+----+----+ 20H   |
  |       +----+----+----+----+                  |                      +----+----+----+----+ A0H   |
  |       +----+----+----+----+                  |                      +----+----+----+----+       |
  |       +----+----+----+----+                  |                      +----+----+----+----+       |
  +-----> +----+----+----+----+                  |                      +----+----+----+----+<-----+
          +----+----+----+----+                  |                      +----+----+----+----+
      80H +----+----+----+----+ 690CE0EEH        |            77073096H +----+----+----+----+ 01H
          +----+----+----+----+                  |                      +----+----+----+----+
          +----+----+----+----+                  |                      +----+----+----+----+
      FFH +----+----+----+----+                  |                      +----+----+----+----+255
          索引值和表值都不颠倒                                               索引值和表值都颠倒
```

“颠倒的直驱表法”用的查询表，是网上可以找到的查询表，我叫它“**正规查询表**”，实际就是“**颠倒的直接查询表**”。对应关系是：

```
“直接查询表”                                      “正规查询表”（颠倒的查询表）
0000 0000（00H）                                  0000 0000（00H）
0000 0001（01H）表值 04C11DB7H                    1000 0000（80H）表值 EDB88320H
0000 0010（02H）                  ──────────►     0100 0000（C0H）
0000 0011（03H）                                  1100 0000（20H）
0000 0100（04H）                  ──────────►     0010 0000（A0H）
...
1000 0000（80H）表值 690CE0EEH    ──────────►     0000 0001（01H）表值 77073096H
...
1111 1111（FFH）                                  FFFF FFFF（FFH）
```

也就是说，将“直接查询表”的索引值和表值直接镜像就是“正规查询表”。

比如，直接查询表的**[01H]= 04C11DB7H**，因为 **01H** 镜像后是 **80H**，**04C11DB7H** 镜像后是 **EDB88320H**，就得到正规查询表的**[80H]= EDB88320H**。

举例来说，假设待测的原始数据是 **10H**，简单起见，不考虑寄存器移出的字节的影响（即假设它是 **00H**）：

“直驱表法”，原始数据先颠倒为 **01H**，根据 **01H** 查表得 **04C11DB7H**，寄存器移出的字节是向左移。

“颠倒的直驱表法”，　　直接根据原始数据 **10H** 查表得 **EDB88320H**，寄存器移出的字节是向右移。

可见，这时这二个方法本质上是一样的。

对于“直驱表法”，颠倒的数据用不颠倒的表索引值、得到不颠倒的表值寄存器进入寄存器，得到的寄存器结果值是不颠倒的，还要再颠倒，变成“颠倒的 **CRC**”。

对于“颠倒的直驱表法”，不颠倒的数据用颠倒的表索引值、得到颠倒的表值寄存器进入寄存器，得到的寄存器结果值就已经是“颠倒的 **CRC**”，不用再颠倒了。

可见，对于 **REFIN=TRUE** 并且 **REFOUT=TRUE** 的 **CRC** 模型来说（注意这个先决条件），就可以直接用“颠倒的直驱表法”来代替“直驱表法”，这样原始数据的比特不用镜像，处理起来就很简单。

那是不是说：不颠倒的数据用不颠倒的表索引值和表值，把得到的寄存器结果值颠倒一下，就能得到和颠倒的数据一样的计算结果？不可能的。颠倒的数据和不颠倒的数据是完全不一样的数据，得到的结果完全两码事。

注意：

先决条件是 **REFIN=TRUE** 并且 **REFOUT=TRUE** 的 **CRC** 参数模型。

“**颠倒的直驱表法**”用的是“**正规查询表**”。

寄存器的初始化预置值也要颠倒。

待测数据每个字节的比特不用颠倒，因为算法的其他部分都做过颠倒处理了。

待测数据串肯定不用颠倒，即待测的字串是**"1234"**，这时也是一样先处理**"1"**，再处理**"2"**，一直到处理**"4"**。

算法如下：

1. 将寄存器向右移动一个字节。

2. 将刚移出的那个字节与待测数据中的新字节做 **XOR** 运算，得到一个指向查询表的索引值。

3. 将索引所指的表值与寄存器做 **XOR** 运算。

4. 如数据没有全部处理完，则跳到步骤 **1**。

用 C 可以写成这样：

```
        r=0;                //r 是寄存器，先初始化为 0
    for(i=0;  i <len;  i++)   //len 是待测数据（不用扩展 0）的字节长度
    {
        r = t[( r^(*(p+i)) ) & 0xff] ^ (r >> 8);  //p 是指向待测数据的指针,t 是查询表
    }
```

“正规查询表”的内容是：

CRC16 正规查询表

00h	0000	C0C1	C181	0140	C301	03C0	0280	C241
08h	C601	06C0	0780	C741	0500	C5C1	C481	0440
10h	CC01	0CC0	0D80	CD41	0F00	CFC1	CE81	0E40
18h	0A00	CAC1	CB81	0B40	C901	09C0	0880	C841
20h	D801	18C0	1980	D941	1B00	DBC1	DA81	1A40
28h	1E00	DEC1	DF81	1F40	DD01	1DC0	1C80	DC41
30h	1400	D4C1	D581	1540	D701	17C0	1680	D641
38h	D201	12C0	1380	D341	1100	D1C1	D081	1040
40h	F001	30C0	3180	F141	3300	F3C1	F281	3240
48h	3600	F6C1	F781	3740	F501	35C0	3480	F441
50h	3C00	FCC1	FD81	3D40	FF01	3FC0	3E80	FE41
58h	FA01	3AC0	3B80	FB41	3900	F9C1	F881	3840
60h	2800	E8C1	E981	2940	EB01	2BC0	2A80	EA41
68h	EE01	2EC0	2F80	EF41	2D00	EDC1	EC81	2C40
70h	E401	24C0	2580	E541	2700	E7C1	E681	2640
78h	2200	E2C1	E381	2340	E101	21C0	2080	E041
80h	A001	60C0	6180	A141	6300	A3C1	A281	6240
88h	6600	A6C1	A781	6740	A501	65C0	6480	A441
90h	6C00	ACC1	AD81	6D40	AF01	6FC0	6E80	AE41
98h	AA01	6AC0	6B80	AB41	6900	A9C1	A881	6840
A0h	7800	B8C1	B981	7940	BB01	7BC0	7A80	BA41
A8h	BE01	7EC0	7F80	BF41	7D00	BDC1	BC81	7C40
B0h	B401	74C0	7580	B541	7700	B7C1	B681	7640
B8h	7200	B2C1	B381	7340	B101	71C0	7080	B041
C0h	5000	90C1	9181	5140	9301	53C0	5280	9241
C8h	9601	56C0	5780	9741	5500	95C1	9481	5440
D0h	9C01	5CC0	5D80	9D41	5F00	9FC1	9E81	5E40
D8h	5A00	9AC1	9B81	5B40	9901	59C0	5880	9841
E0h	8801	48C0	4980	8941	4B00	8BC1	8A81	4A40
E8h	4E00	8EC1	8F81	4F40	8D01	4DC0	4C80	8C41
F0h	4400	84C1	8581	4540	8701	47C0	4680	8641
F8h	8201	42C0	4380	8341	4100	81C1	8081	4040

CRC32 正规查询表

00h	00000000	77073096	EE0E612C	990951BA
04h	076DC419	706AF48F	E963A535	9E6495A3
08h	0EDB8832	79DCB8A4	E0D5E91E	97D2D988
0Ch	09B64C2B	7EB17CBD	E7B82D07	90BF1D91
10h	1DB71064	6AB020F2	F3B97148	84BE41DE
14h	1ADAD47D	6DDDE4EB	F4D4B551	83D385C7
18h	136C9856	646BA8C0	FD62F97A	8A65C9EC
1Ch	14015C4F	63066CD9	FA0F3D63	8D080DF5
20h	3B6E20C8	4C69105E	D56041E4	A2677172
24h	3C03E4D1	4B04D447	D20D85FD	A50AB56B
28h	35B5A8FA	42B2986C	DBBBC9D6	ACBCF940
2Ch	32D86CE3	45DF5C75	DCD60DCF	ABD13D59
30h	26D930AC	51DE003A	C8D75180	BFD06116
34h	21B4F4B5	56B3C423	CFBA9599	B8BDA50F
38h	2802B89E	5F058808	C60CD9B2	B10BE924
3Ch	2F6F7C87	58684C11	C1611DAB	B6662D3D
40h	76DC4190	01DB7106	98D220BC	EFD5102A
44h	71B18589	06B6B51F	9FBFE4A5	E8B8D433
48h	7807C9A2	0F00F934	9609A88E	E10E9818
4Ch	7F6A0DBB	086D3D2D	91646C97	E6635C01
50h	6B6B51F4	1C6C6162	856530D8	F262004E
54h	6C0695ED	1B01A57B	8208F4C1	F50FC457
58h	65B0D9C6	12B7E950	8BBEB8EA	FCB9887C
5Ch	62DD1DDF	15DA2D49	8CD37CF3	FBD44C65
60h	4DB26158	3AB551CE	A3BC0074	D4BB30E2
64h	4ADFA541	3DD895D7	A4D1C46D	D3D6F4FB
68h	4369E96A	346ED9FC	AD678846	DA60B8D0
6Ch	44042D73	33031DE5	AA0A4C5F	DD0D7CC9
70h	5005713C	270241AA	BE0B1010	C90C2086
74h	5768B525	206F85B3	B966D409	CE61E49F
78h	5EDEF90E	29D9C998	B0D09822	C7D7A8B4
7Ch	59B33D17	2EB40D81	B7BD5C3B	C0BA6CAD
80h	EDB88320	9ABFB3B6	03B6E20C	74B1D29A
84h	EAD54739	9DD277AF	04DB2615	73DC1683
88h	E3630B12	94643B84	0D6D6A3E	7A6A5AA8
8Ch	E40ECF0B	9309FF9D	0A00AE27	7D079EB1
90h	F00F9344	8708A3D2	1E01F268	6906C2FE
94h	F762575D	806567CB	196C3671	6E6B06E7
98h	FED41B76	89D32BE0	10DA7A5A	67DD4ACC
9Ch	F9B9DF6F	8EBEEFF9	17B7BE43	60B08ED5
A0h	D6D6A3E8	A1D1937E	38D8C2C4	4FDFF252
A4h	D1BB67F1	A6BC5767	3FB506DD	48B2364B
A8h	D80D2BDA	AF0A1B4C	36034AF6	41047A60
ACh	DF60EFC3	A867DF55	316E8EEF	4669BE79
B0h	CB61B38C	BC66831A	256FD2A0	5268E236
B4h	CC0C7795	BB0B4703	220216B9	5505262F
B8h	C5BA3BBE	B2BD0B28	2BB45A92	5CB36A04

BCh	C2D7FFA7	B5D0CF31	2CD99E8B	5BDEAE1D
C0h	9B64C2B0	EC63F226	756AA39C	026D930A
C4h	9C0906A9	EB0E363F	72076785	05005713
C8h	95BF4A82	E2B87A14	7BB12BAE	0CB61B38
CCh	92D28E9B	E5D5BE0D	7CDCEFB7	0BDBDF21
D0h	86D3D2D4	F1D4E242	68DDB3F8	1FDA836E
D4h	81BE16CD	F6B9265B	6FB077E1	18B74777
D8h	88085AE6	FF0F6A70	66063BCA	11010B5C
DCh	8F659EFF	F862AE69	616BFFD3	166CCF45
E0h	A00AE278	D70DD2EE	4E048354	3903B3C2
E4h	A7672661	D06016F7	4969474D	3E6E77DB
E8h	AED16A4A	D9D65ADC	40DF0B66	37D83BF0
ECh	A9BCAE53	DEBB9EC5	47B2CF7F	30B5FFE9
F0h	BDBDF21C	CABAC28A	53B39330	24B4A3A6
F4h	BAD03605	CDD70693	54DE5729	23D967BF
F8h	B3667A2E	C4614AB8	5D681B02	2A6F2B94
FCh	B40BBE37	C30C8EA1	5A05DF1B	2D02EF8D

“颠倒的直驱表法”的程序：

同样要先做一个颠倒比特的子程序：

```
unsigned long int Reflect(unsigned long int ref, char ch)
{
unsigned long int value=0;
    // 交换 bit0 和 bit7，bit1 和 bit6，类推
for(int i = 1; i < (ch + 1); i++)
{
    if(ref & 1)
        value |= 1 << (ch - i);
    ref >>= 1;
}
return value;
}
```

在主程序中的程序：

```
unsigned long int crc32_table[256];
unsigned long int ulPolynomial = 0x04c11db7;
unsigned long int crc,temp;

for(int i = 0; i <= 0xFF; i++)    // 生成 CRC32“正规查询表”
{
    temp=Reflect(i, 8);
    crc32_table[i]= temp<< 24;
    for (int j = 0; j < 8; j++)
    {
        unsigned long int t1,t2;
        unsigned long int flag=crc32_table[i]&0x80000000;
        t1=(crc32_table[i] << 1);
        if(flag==0)
            t2=0;
```

```
        else
            t2=ulPolynomial;
        crc32_table[i] =t1^t2 ;
    }
    crc=crc32_table[i];
    crc32_table[i] = Reflect(crc32_table[i], 32);
}


   //计算 CRC32 值
unsigned   long   CRC32;
BYTE  DataBuf[512]={0x31,0x32,0x33,0x34};      //待测数据，为字串"1234"
unsigned   long   len;
unsigned   long   ii;
unsigned   long   m_CRC = 0xFFFFFFFF;      //寄存器中预置初始值
BYTE   *p;

len=4;              //待测数据的字节长度
p = DataBuf;
for(ii=0;  ii <len;  ii++)
{
    m_CRC = crc32_table[( m_CRC^(*(p+ii)) ) & 0xff] ^ (m_CRC >> 8);  //计算
}
CRC32= ~m_CRC;      //取反。经 WINRAR 对比，CRC32 值正确!!
//这时， CRC32 中的值就是 CRC
```

八、 结束

以上程序均在 VC6 中测试成功，很多程序是直接抄网上的再修改。

CRC 计算其实很简单，也就才 4 个算法，移来移去、优化来优化去而已。

但是就是这么简单的东西，国内网上好像没有文章能说得完全明白。搞得我这种笨人花了整整 5 天才整理出来。

书上的都是理论一大堆，这个式子那个式子的，一头雾水，我这个非专业人士更看不懂。

应该说，用程序实现和优化算法是容易的，创造出 CRC 方法的人才是真正的强人。