校验原理

1、循环校验码（CRC 码）：是数据通信领域中最常用的一种差错校验码，其特征是信息字段和校验字段的长度可以任意选定。

2、生成 CRC 码的基本原理：任意一个由二进制位串组成的代码都可以和一个系数仅为‘0’和‘1’取值的多项式一一对应。例如：代码 1010111 对应的多项式为 $x^6+x^4+x^2+x+1$，而多项式为 $x^5+x^3+x^2+x+1$ 对应的代码 101111。

3、CRC 码集选择的原则：若设码字长度为 N，信息字段为 K 位，校验字段为 R 位(N=K+R)，则对于 CRC 码集中的任一码字，存在且仅存在一个 R 次多项式 g(x)，使得

$$V(x)=A(x)g(x)=x^Rm(x)+r(x);$$

其中：　m(x)为 K 次信息多项式，　r(x)为 R-1 次校验多项式，

g(x)称为生成多项式：

$$g(x)=g_0+g_1x+g_2x^2+...+g_{(R-1)}x^{(R-1)}+g_Rx^R$$

发送方通过指定的 g(x)产生 CRC 码字，接收方则通过该 g(x)来验证收到的 CRC 码字。

4、CRC 校验码软件生成方法：

借助于多项式除法，其余数为校验字段。

例如：信息字段代码为：1011001；对应 $m(x)=x^6+x^4+x^3+1$

假设生成多项式为：$g(x)=x^4+x^3+1$；则对应 g(x)的代码为：11001

$x^4m(x)=x^{10}+x^8+x^7+x^4$ 对应的代码记为：10110010000；

采用多项式除法：得余数为：1010　(即校验字段为：1010)

发送方：发出的传输字段为：**1 0 1 1 0 0 1 1 0 10**

信息字段　　校验字段

接收方：使用相同的生成码进行校验:接收到的字段/生成码（二进制除法）

如果能够除尽，则正确，

CRC 校验源码分析

这两天做项目，需要用到 CRC 校验。以前没搞过这东东，以为挺简单的。结果看看别人提供的汇编源程序，居然看不懂。花了两天时间研究了一下 CRC 校验，希望我写的这点东西能够帮助和我有同样困惑的朋友节省点时间。

先是在网上下了一堆乱七八遭的资料下来，感觉都是一个模样，全都是从 CRC 的数学原理开始，一长串的表达式看的我头晕。第一次接触还真难以理解。这些东西不想在这里讲，随便找一下都是一大把。我想根据源代码来分析会比较好懂一些。

费了老大功夫，才搞清楚 CRC 根据"权"(即多项表达式)的不同而相应的源代码也有稍许不同。以下是各种常用的权。

CRC8=X8+X5+X4+1

CRC-CCITT=X16+X12+X5+1

CRC16=X16+X15+X5+1

CRC12=X12+X11+X3+X2+1

CRC32=X32+X26+X23+X22+X16+X12+X11+X10+X8+X7+X5+X4+X2+X1+1

以下的源程序全部以 CCITT 为例。其实本质都是一样，搞明白一种，其他的都是小菜。

图 1，图 2 说明了 CRC 校验中 CRC 值是如何计算出来的，体现的多项式正是 X16+X12+X5+1。 Serial Data 即是需要校验的数据。从把数据移位开始计算，将数据位（从最低的数据位开始）逐位移入反向耦合移位寄存器(这个名词我也不懂，觉得蛮酷的，就这样写了，嘿)。当所有数据位都这样操作后，计算结束。此时，16 位移位寄存器中的内容就是 CRC 码。

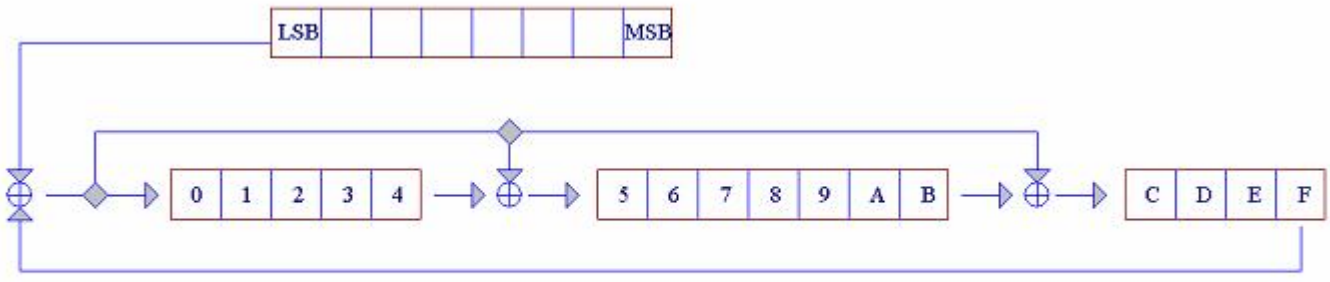

图 1　生成 CRC-CCITT 的移位寄存器的作用原理

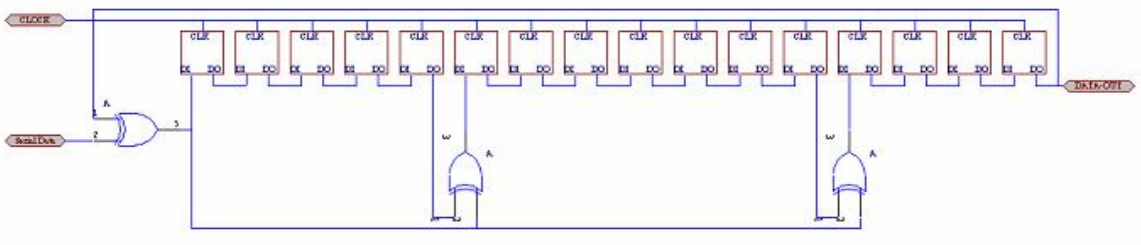

图 2　用于计算 CRC_CCITT 的移位寄存器的电路配置

图中进行 XOR 运算的位与多项式的表达相对应。

X5 代表 Bit5，X12 代表 Bit12，1 自然是代表 Bit0，X16 比较特别，是指移位寄存器移出的数据，即图中的 DATA OUT。可以这样理解，与数据位做 XOR 运算的是上次 CRC 值的 Bit15。

根据以上说明，可以依葫芦画瓢的写出以下程序。(程序都是在 keil C 7.10 下调试的)

```
typedef     unsigned char      uchar;
typedef     unsigned int       uint;

code uchar crcbuff [] = { 0x00,0x00,0x00,0x00,0x06,0x0d,0xd2,0xe3};
uint crc;                   // CRC 码
void main(void)
{
  uchar *ptr;
  crc = 0;                  // CRC  初值
  ptr = crcbuff;                //  指向第一个 Byte 数据
  crc = crc16l(ptr,8);
  while(1);
}

uint crc16l(uchar *ptr,uchar len)        // ptr 为数据指针，len 为数据长度
{
  uchar i;
  while(len--)
  {
     for(i=0x80; i!=0; i>>=1)
    {
```

```
        if((crc&0x8000)!=0) {crc<<=1; crc^=0x1021;}         1-1
          else crc<<=1;                        1-2
       if((*ptr&i)!=0) crc^=0x1021;                          1-3
     }
     ptr++;
   }
   return(crc);
}
```

执行结果 crc = 0xdbc0;
程序 1-1,1-2,1-3 可以理解成移位前 crc 的 Bit15 与数据对应的 Bit(*ptr&i)做 XOR 运算，根据此结果来决定是否执行 crc^=0x1021。只要明白两次异或运算与原值相同，就不难理解这个程序。

很多资料上都写了查表法来计算，当时是怎么也没想通。其实蛮简单的。假设通过移位处理了 8 个 bit 的数据，相当于把之前的 CRC 码的高字节(8bit)全部移出，与一个 byte 的数据做 XOR 运算，根据运算结果来选择一个值(称为余式)，与原来的 CRC 码再做一次 XOR 运算，就可以得到新的 CRC 码。

不难看出，余式有 256 种可能的值，实际上就是 0~255 以 X16+X12+X5+1 为权得到的 CRC 码，可以通过函数 crc16l 来计算。以 1 为例。

```
code test[]={0x01};
crc = 0;
ptr = test;
crc = crc16l(ptr,1);
```

执行结果 crc = 1021，这就是 1 对应的余式。

进一步修改函数，我这里就懒得写了，可得到 X16+X12+X5+1 的余式表。

```
code uint crc_ta[256]={                 // X16+X12+X5+1  余式表
    0x0000, 0x1021,   0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
  0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
    0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
```

```
    0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
    0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
    0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
    0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
    0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
    0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
    0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
    0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
     0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
    0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
    0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
    0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
    0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
    0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
    0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
    0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
     0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
    0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
    0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,

     0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
     0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};
```

根据这个思路，可以写出以下程序：

```
uint table_crc(uchar *ptr,uchar len)       //  字节查表法求 CRC
{
  uchar da;
    while(len--!=0)
  {
```

```
    da=(uchar) (crc/256);        // 以 8 位二进制数暂存 CRC 的高 8 位
    crc<<=8;                     //  左移 8 位
    crc^=crc_ta[da^*ptr];        //  高字节和当前数据 XOR 再查表
    ptr++;
  }
   return(crc);
}
```

本质上 CRC 计算的就是移位和异或。所以一次处理移动几位都没有关系，只要做相应的处理就好了。
下面给出半字节查表的处理程序。其实和全字节是一回事。

```
code uint crc_ba[16]={
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
  0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
};

uint ban_crc(uchar *ptr,uchar len)
{
  uchar da;
  while(len--!=0)
  {
    da = ((uchar)(crc/256))/16;
    crc <<= 4;
    crc ^=crc_ba[da^(*ptr/16)];
    da = ((uchar)(crc/256)/16);
    crc <<= 4;
    crc ^=crc_ba[da^(*ptr&0x0f)];
    ptr++;
  }
  return(crc);
}
```

crc_ba[16]和 crc_ta[256]的前 16 个余式是一样的。

其实讲到这里，就已经差不多了。反正当时我以为自己是懂了。结果去看别人的

源代码的时候，也是说采用 CCITT，但是是反相的。如图 3

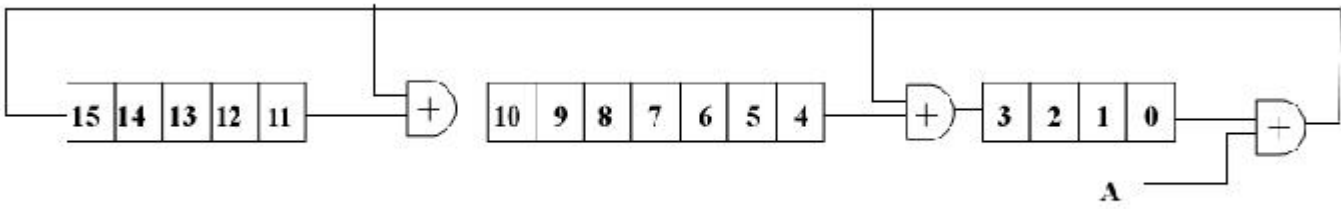

图 3 CRC-CCITT 反相运算

反过来，一切都那么陌生，faint.吐血，吐血。
仔细分析一下，也可以很容易写出按位异或的程序。只不过由左移变成右移。

```
uint crc16r(unsigned char *ptr, unsigned char len)
{
  unsigned char i;
  while(len--!=0)
  {
    for(i=0x01;i!=0;i <<= 1)
    {
      if((crc&0x0001)!=0) {crc >>= 1; crc ^= 0x8408;}
      else crc >>= 1;
      if((*ptr&i)!=0) crc ^= 0x8408;
    }
      ptr++;
  }
  return(crc);
}
```

0x8408 就是 CCITT 的反转多项式。
套用别人资料上的话
“反转多项式是指在数据通讯时，信息字节先传送或接收低位字节，如重新排位影响 CRC 计算速度，故设反转多项式。”

如

```
code uchar crcbuff [] = { 0x00,0x00,0x00,0x00,0x06,0x0d,0xd2,0xe3};
```

反过来就是

```
  code uchar crcbuff_fan[] = {0xe3,0xd2,0x0d,0x06,0x00,0x00,0x00,0x00};
```

```
  crc = 0;
  ptr = crcbuff_fan;
  crc = crc16r(ptr,8);
```

执行结果 crc = 0x5f1d;

如想验证是否正确，可改

```
  code uchar crcbuff_fan_result[] =
{0xe3,0xd2,0x0d,0x06,0x00,0x00,0x00,0x00,0x1d,0x5f};
  ptr = crcbuff_fan_result;
  crc = crc16r(ptr,10);
```

执行结果 crc = 0; 符合 CRC 校验的原理。

请注意 0x5f1d 在数组中的排列中低位在前，正是反相运算的特点。不过当时是把我搞的晕头转向。

在用半字节查表法进行反相运算要特别注意一点，因为是右移，所以 CRC 移出的 4Bit 与数据 XOR 的操作是在 CRC 的高位端。因此余式表的产生是要以下列数组通过修改函数 crc16r 产生。

```
  code uchar ban_fan[]=
{0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe
0,0xf0};
```

得出余式表

```
code uint fan_yushi[16]={
    0x0000, 0x1081, 0x2102, 0x3183,
    0x4204, 0x5285, 0x6306, 0x7387,
    0x8408, 0x9489, 0xa50a, 0xb58b,
    0xc60c, 0xd68d, 0xe70e, 0xf78f
};

uint ban_fan_crc(uchar *ptr,uchar len)
{
  uchar da;
  while(len--!=0)
  {
```

```
    da = (uchar)(crc&0x000f);
    crc >>= 4;
    crc ^= fan_yushi [da^(*ptr&0x0f)];
    da = (uchar)(crc&0x000f);
    crc >>= 4;
    crc ^= fan_yushi [da^(*ptr/16)];
    ptr++;
  }
  return(crc);
}
```

主程序中

```
  crc = 0;
  ptr = crcbuff_fan;
```

crc = ban_fan_crc(ptr,8);

执行结果 crc = 0x5f1d;

反相运算的**算法简单实现 crc 校验**

前一段时间做协议转换器的时间用到 CRC-16 校验，查了不少资料发现都不理想。查表法要建表太麻烦，而计算法觉得那些例子太罗嗦。最后只好自己写了，最后发现原来挺简单嘛：）

两个子程序搞定。这里用的多项式为：

CRC-16　　= X16 + X12 + X5 + X0 = 2^0+2^5+2^12+2^16=0x11021

因最高位一定为“1”，故略去计算只采用 0x1021 即可

CRC_Byte：计算单字节的 CRC 值

CRC_Data：计算一帧数据的 CRC 值

CRC_High　CRC_Low：存放单字节 CRC 值

CRC16_High　CRC16_Low：存放帧数据 CRC 值

```
;<>-------------------------------------------------------------
```

```
;       Function:       CRC one byte
;       Input:             CRCByte
;       Output:          CRC_High CRC_Low
;<>-------------------------------------------------------------

CRC_Byte:
       clrf         CRC_Low
       clrf         CRC_High
       movlw           09H
       movwf           v_Loop1
       movf              CRCByte, w
       movwf           CRC_High
CRC:
       decfsz            v_Loop1                                ; 8 次循
环，每一位相应计算
       goto        CRC10
       goto        CRCend
CRC10
       bcf                STATUS, C
       rlf                  CRC_Low
       rlf                  CRC_High

       btfss              STATUS, C
       goto        CRC                                          ; 为 0
不需计算
       movlw           10H                                     ;若多项
式改变，这里作相应变化
       xorwf            CRC_High, f
       movlw           21H                                     ;若多项
式改变，这里作相应变化
       xorwf            CRC_Low, f
       goto        CRC
CRCend:
       nop
       nop
       return
;<>-------------------------------------------------------------
```

```
;       CRC one byte end
;<>-------------------------------------------------------------
;<>-------------------------------------------------------------
;       Function:       CRC date
;       Input:             BufStart(A,B,C)（一帧数据的起始地
址） v_Count （要做 CRC 的字节数）
;       Output:          CRC16_High CRC16_Low（结果）
;<>-------------------------------------------------------------
CRC_Data:

       clrf         CRC16_High
       clrf         CRC16_Low

CRC_Data10

       movf              INDF, w
       xorwf            CRC16_High,w

       movwf           CRCByte
       call         CRC_Byte
       incf         FSR
       decf        v_Count                          ;需计算的字节数

       movf              CRC_High, w
       xorwf            CRC16_Low, w
       movwf           CRC16_High

       movf              CRC_Low, w
       movwf           CRC16_Low

       movf              v_Count, w
        ; 计算结束?
       btfss              STATUS, Z
       goto        CRC_Data10

       return
```

```
;<>-------------------------------------------------------------
;              CRC date end
;<>-------------------------------------------------------------
```

说明： CRC 的计算原理如下（一个字节的简单例子）

```
    11011000 00000000 00000000  <- 一个字节数据, 左移 16b
   ^10001000 00010000 1         <- CRC-CCITT 多项式, 17b
    -------------------------
     1010000 00010000 10        <- 中间余数
    ^1000100 00001000 01
     -------------------------
       10100 00011000 1100
      ^10001 00000010 0001
       ----------------------
         101 00011010 110100
        ^100 01000000 100001
         --------------------
           1 01011010 01010100
          ^1 00010000 00100001
           ------------------
             01001010 01110101  <- 16b CRC
```

仿此，可推出两个字节数据计算如下：d 为数据，p 为项式，a 为余数

```
    dddddddd dddddddd 00000000 00000000 <- 数据 D ( D1, D0, 0, 0 )
   ^pppppppp pppppppp p                 <- 多项式 P
    ----------------------------------

    ...
             aaaaaaaa aaaaaaaa 0        <- 第一次的余数 A’ ( A’1, A’
0 )
            ^pppppppp pppppppp p
             -------------------------

             ...
                      aaaaaaaa aaaaaaaa <- 结果 A ( A1, A0 )
```

由此与一字节的情况比较，将两个字节分开计算如下：

先算高字节：

```
    dddddddd 00000000 00000000 00000000 <- D1, 0, 0, 0
   ^pppppppp pppppppp p                  <- P
    ---------------------------------
    ...
             aaaaaaaa aaaaaaaa           <- 高字节部分余数 PHA1, PHA0
```

此处的部分余数与前面两字节算法中的第一次余数有如下关系，即 A’1 = PHA1 ^ D0, A’0 = PHA0：

```
             aaaaaaaa aaaaaaaa           <- PHA1, PHA0
            ^dddddddd                    <- D0
             -----------------
             aaaaaaaa aaaaaaaa           <- A’1, A’0
```

低字节的计算：

```
             aaaaaaaa 00000000 00000000 <- A’1, 0, 0
            ^pppppppp pppppppp p        <- P
             ---------------------------
             ...
                      aaaaaaaa aaaaaaaa <- 低字节部分余数 PLA1, PLA0
                     ^aaaaaaaa          <- A’0 ，  即 PHA0
                      -----------------
                      aaaaaaaa aaaaaaaa <- 最后的 CRC ( A1, A0 )
```

总结以上内容可得规律如下：

设部分余数函数

```
    PA = f( d )
```

其中 d 为一个字节的数据（注意，除非 n = 0 ，否则就不是原始数据，见下文）

第 n 次的部分余数

```
    PA( n ) = ( PA( n - 1 ) << 8 ) ^ f( d )
```

其中的

```
    d = ( PA( n - 1 ) >> 8 ) ^ D( n )
```

其中的 D(n) 才是一个字节的原始数据。

公式如下：

```
    PA( n ) = ( PA( n - 1 ) << 8 ) ^ f( ( PA( n - 1 ) >> 8 ) ^ D( n ) )
```

可以注意到函数 f(d) 的参数 d 为一个字节，对一个确定的多项式 P， f(d) 的返回值 是与 d 一一对应的，总数为 256 项，将这些数据预先算出保存在表里，f(d)就转换为一 个查表的过程，速度也就可以大幅提高，这也就是查表法计算 CRC 的原理。

再来看 CRC 表是如何计算出来的，即函数 f(d) 的实现方法。分析前面一个字节数据的 计算过程可发现，d 对结果的影响只表现为对 P 的移位异或，看计算过程中的三个 8 位 的列中只低两个字节的最后结果是余数，而数据所在的高 8 位列最后都被消去了，因其 中的运算均为异或，不产生进位或借位，故每一位数据只影响本列的结果，即 d 并不直接 影响结果。再将前例变化一下重列如下：

```
    11011000
    -------------------------
    10001000 00010000 1         // P
  ^ 1000100 00001000 01         // P
  ^  000000 00000000 000        // 0
  ^   10001 00000010 0001       // P
  ^    0000 00000000 00000      // 0
  ^     100 01000000 100001     // P
  ^      00 00000000 0000000    // 0
  ^       1 00010000 00100001 // P
          -------------------
            01001010 01110101
```

现在的问题就是如何根据 d 来对 P 移位异或了，从上面的例子看，也可以理解为每步 移位，但根据 d 决定中间余数是否与 P 异或。从前面原来的例子可以看出，决定的条件是中间余数的最高位为 0，因为 P 的最高位一定为 1，即当中间余数与 d 相应位异或的最高位为 1 时，中间余数移位就要和 P 异或，否则只需移位即可。其方法如下例（上例的变形，注意其中空格的移动表现了 d 的影响如何被排除在结果之外）：

```
    d --------a--------
    1 00000000 00000000 <- HSB = 1
      0000000 000000000 <- a <<= 1
      0001000 000100001 <-不含最高位的 1
      -----------------
```

```
1 0001000 000100001
  001000 0001000010
  000100 0000100001
  ----------------
0 001100 0001100011 <- HSB = 0
  01100 00011000110
  ----------------
1 01100 00011000110 <- HSB = 1
  1100 000110001100
  0001 000000100001
  ----------------
1 1101 000110101101 <- HSB = 0
  101 0001101011010
  ----------------
0 101 0001101011010 <- HSB = 1
  01 00011010110100
  00 01000000100001
  ----------------
0 01 01011010010101 <- HSB = 0
  1 010110100101010
  ----------------
0 1 010110100101010 <- HSB = 1
   0101101001010100
   0001000000100001
  ----------------
   0100101001110101 <- CRC
```

结合这些，前面的程序就好理解了。

循环冗余校验码

CRC（Cyclic Redundancy Check)循环冗余校验码

是常用的校验码，在早期的通信中运用广泛，因为早期的通信技术不够可靠（不可靠性的来源是通信技术决定的，比如电磁波通信时受雷电等因素的影响），不可靠的通信就会带来‘确认信息’的困惑，书上提到红军和蓝军通信联合进攻山下的敌军的例子，第一天红军发了条信息要蓝军第二天一起进攻，蓝军收到之后，发一条确认信息，但是蓝军担心的是‘确认信息’如果也不可靠而没有成功到达红军那里，那自己不

是很危险？于是红军再发一条‘对确认的确认信息’，但同样的问题还是不能解决，红军仍然不敢贸然行动。

对通信的可靠性检查就需要‘校验’，校验是从数据本身进行检查，它依靠某种数学上约定的形式进行检查，校验的结果是可靠或不可靠，如果可靠就对数据进行处理，如果不可靠，就丢弃重发或者进行修复。

CRC 码是由两部分组成，前部分是信息码，就是需要校验的信息，后部分是校验码，如果 CRC 码共长 n 个 bit，信息码长 k 个 bit，就称为(n,k)码。 它的编码规则是：

1、首先将原信息码(kbit)左移 r 位(k+r=n)

2、运用一个生成多项式 g(x)（也可看成二进制数）用模 2 除上面的式子，得到的余数就是校验码。

非常简单，要说明的：模 2 除就是在除的过程中用模 2 加，模 2 加实际上就是我们熟悉的异或运算，就是加法不考虑进位，公式是：

0+0=1+1=0,1+0=0+1=1

即‘异’则真，‘非异’则假。

由此得到定理：a+b+b=a 也就是‘模 2 减’和‘模 2 加’直值表完全相同。

有了加减法就可以用来定义模 2 除法，于是就可以用生成多项式 g(x)生成 CRC 校验码。

例如： g(x)=x4+x3+x2+1,(7,3)码,信息码 110 产生的 CRC 码就是：

对于 g(x)=x4+x3+x2+1 的解释：（都是从右往左数）x4 就是第五位是 1，因为没有 x1 所以第 2 位就是 0。

11101 | 110,0000（设 a=11101 ， b=1100000）

取 b 的前 5 位 11000 跟 a 异或得到 101

101 加上 b 没有取到的 00 得到 10100

然后跟 a 异或得到 01001

也就是余数 1001

余数是 1001，所以 CRC 码是 110,1001

标准的 CRC 码是，CRC-CCITT 和 CRC-16，它们的生成多项式是：

CRC-CCITT=x^16+x^12+x^5+1

CRC-16=x^16+x^15+x^2+1

在通信系统的数据传输过程中，由于信道中各种复杂因素的影响，往往使传输的信号受到干扰，造成误码的出现。接收方为了检查所接收的数据是否有误码，可采用多种检测方法。差错控制编码是目前数据传输过程中普遍采用的一种提高数据通信可靠性的方法，而 CRC 是一种在实际通信中应用很广泛的差错控制编码，具有很强的检错能力。
CRC 校验将输入比特（ ）表示为下列多项式的系数：

式中，D 可以看成一个时延因子，表示相应的比特所处的位置。
校验比特（ ）可由下式求得：

式中， 表示取余数，是生成多项式。式中的除法与普通的多项式长除法相同，其差别是系数是二进制，其运算以模 2 为基础。常用的几个 L 阶 CRC 生成多项式为：

其中， 和 产生的校验比特为 16 比特， 产生的校验比特为 32 比特。
在实际数据传输过程中，发送方将 CRC 校验码附加在所传数据流的尾部一并传送；接收方用同样的生成多项式去除接收到的数据流，若余数为零，则可判断所接收到的数据是正确的。否则，可判断数据在传输过程中产生了误码。

2 CRC 校验的 C 语言实现
下面我们以 CRC-16 为例来说明任意长度数据流的 CRC 校验码生成过程。我们采用将数据流分成若干个 8bit 字符，并由低字节到高字节传送的并行方法来求 CRC 校验码。具体计算过程为：用一个 16bit 的寄存器来存放 CRC 校验值，且设定其初值为 0x0000；将数据流的第一个 8bit 与 16bit 的 CRC 寄存器的高字节相异或，并将结果存入 CRC 寄存器高字节；CRC 寄存器左移一位，最低 1bit 补零，同时检查移出的最高 1bit，若移出的最高 1bit 为 0，则继续按上述过程左移，若最高 1bit 为 1，则将 CRC 寄存器中的值与生成多项式码相异或，结果存入 CRC 寄存器值；继续左移并重复上述处理方法，直到将 8bit 数据处理完为止，则此时 CRC 寄存器中的值就是第一个 8bit 数据对应的 CRC 校验码；然后将此时 CRC 寄存器的值作为初值，用同样的处理方法重复上述步骤来处理下一个 8bit 数据流，直到将所有的 8bit 字符都处理完后，此刻 CRC 寄存器中的值即为整个数据流对应的 CRC 校验码。
下面示出了其计算过程的流程图：

在用 C 语言编写 CRC 校验码的实现程序时我们应该注意，生成多项式 对应的十六进制数为 0x18005，由于 CRC 寄存器左移过程中，移出的最高位为 1 时与 相异或，所以与 16bit 的 CRC 寄存器对应的生成多项式的十六进制数可用 0x8005 表示。下面给出并行处理 8bit 数据流的 C 源程序：

```
unsigned short crc_dsp(unsigned short reg, unsigned char data_crc)
//reg 为 crc 寄存器， data_crc 为将要处理的 8bit 数据流
{
unsigned short msb; //crc 寄存器将移出的最高 1bit
unsigned short data;
unsigned short gx = 0x8005, i = 0; //i 为左移次数， gx 为生成多项式

data = (unsigned short)data_crc;
data = data << 8;
reg = reg ^ data;
```

```
do
{
msb = reg & 0x8000;
reg = reg << 1;
if(msb == 0x8000)
{
reg = reg ^ gx;
}
i++;
}
while(i < 8);
return (reg);
}
```

以上为处理每一个 8bit 数据流的子程序，在计算整个数据流的 CRC 校验码时，我们只需将 CRC_reg 的初值置为 0x0000，求第一个 8bit 的 CRC 值，之后，即可将上次求得的 CRC 值和本次将要处理的 8bit 数据作为函数实参传递给上述子程序的形参进行处理即可，最终返回的 reg 值便是我们所想得到的整个数据流的 CRC 校验值。

3 结论

本文根据 CRC 校验的基本原理，提出了一种并行计算任意长度 CRC 校验值的算法，并给出了源程序，且该源程序可作为一子函数来直接被调用，可灵活的应用在不同的工程当中。此算法已应用与某系统当中，实践证明，该算法正确可靠。

其实 CRC 可以用查表的方法。

```
/*
CRC 校验算法实现  C 语言代码实现
文件  CRC16.h
*/
#ifndef _CRC16_h
#define _CRC16_h
static unsigned char auchCRCHi[] = {
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
```

```
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40
};
/* CRC 低位字节值表*/
static char auchCRCLo[] = {
0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06,
0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04, 0xCC, 0x0C, 0x0D, 0xCD,
0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A,
0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC, 0x14, 0xD4,
0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,
0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3,
0xF2, 0x32, 0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,
0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29,
0xEB, 0x2B, 0x2A, 0xEA, 0xEE, 0x2E, 0x2F, 0xEF, 0x2D, 0xED,
0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60,
0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62, 0x66, 0xA6, 0xA7, 0x67,
0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68,
0x78, 0xB8, 0xB9, 0x79, 0xBB, 0x7B, 0x7A, 0xBA, 0xBE, 0x7E,
0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71,
0x70, 0xB0, 0x50, 0x90, 0x91, 0x51, 0x93, 0x53, 0x52, 0x92,
0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
```

```
0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B,
0x99, 0x59, 0x58, 0x98, 0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B,
0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42,
0x43, 0x83, 0x41, 0x81, 0x80, 0x40
};
/* 要进行 CRC 校验的消息 */ /* 消息中字节数 */
unsigned short CRC16(unsigned char *puchMsg,unsigned short usDataLen )
{
unsigned char uchCRCHi = 0xFF ; /* 高 CRC 字节初始化 */
unsigned char uchCRCLo = 0xFF ; /* 低 CRC 字节初始化 */
unsigned uIndex ; /* CRC 循环中的索引 */
while (usDataLen--) /* 传输消息缓冲区 */
{
uIndex = uchCRCHi ^ *puchMsg++ ; /* 计算 CRC */
uchCRCHi = uchCRCLo ^ auchCRCHi[uIndex];
uchCRCLo = auchCRCLo[uIndex] ;
}
return (uchCRCHi < < 8 | uchCRCLo) ;
}
#endif //_CRC16_h

/*
CRC16.cpp
生成 CRC 码实现
*/
#include <stdio.h>
#include "CRC16.h"
void main()
{
unsigned char ppp[13] = {0x12,0x34};
    unsigned short result;
    // 计算 CRC
    result = CRC16(ppp, 2);
printf("%x\n",result);
}
```

有三四个月没有上 CSDN 了，从现在开始吧！