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。

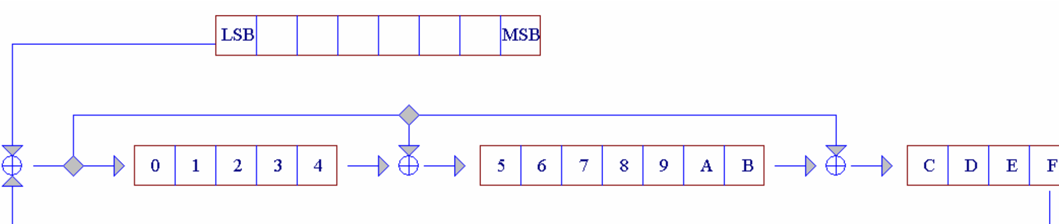

图 1　生成 CRC-CCITT 的移位寄存器的作用原理

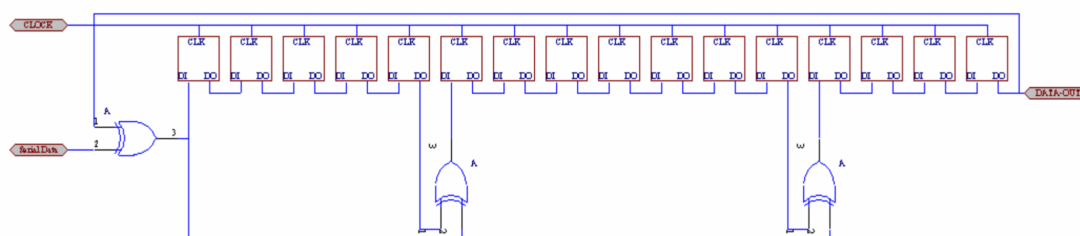

图 2　用于计算 CRC_CCITT 的移位寄存器的电路配置

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

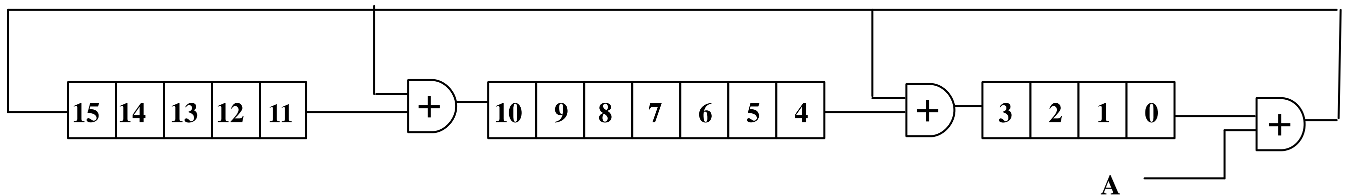

图 3 CRC-CCITT 反相运算

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[]=
    {0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0};
```

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;

反相运算的全字节查表法就很容易了，懒的写了。

图 1**，图** 2 **是用** protel DXP **画的，自己感觉太丑了，哪位哥们知道哪种工具画这种东西最方便，通知一声。嘿。**

freebirds
ouyangxianping@263.net