CRC校验一探究竟

曹嘉辉

2020.4.28

github page: <https://github.com/cjhonlyone/A-guide-of-CRC>

目录

[1. 原理 2](#_Toc39841730)

[1.1 差错控制编码 2](#_Toc39841731)

[1.2 线性分组码 2](#_Toc39841732)

[1.3 循环码 3](#_Toc39841733)

[1.4 CRC码 5](#_Toc39841734)

[2. 长除法c语言实现 7](#_Toc39841735)

[3. 线性移位寄存器电路实现 9](#_Toc39841736)

[3.1 长除法到寄存器电路的推导 9](#_Toc39841737)

[3.2 线性移位寄存器的c语言仿真 11](#_Toc39841738)

[3.3 线性移位寄存器的verilog仿真 12](#_Toc39841739)

[4. 初值问题 13](#_Toc39841740)

[4.1 为什么结果不同 13](#_Toc39841741)

[4.2 给定CRC求信息码 14](#_Toc39841742)

[4.3 测试初值与结果的关系 15](#_Toc39841743)

[4.4 总结一下 17](#_Toc39841744)

[5. CRC的种类 18](#_Toc39841745)

[CRC-16/CCITT 18](#_Toc39841746)

[CRC-16/IBM-3740 19](#_Toc39841747)

[参考文献 19](#_Toc39841748)

## 1. 原理

### 1.1 差错控制编码

按功能分，有**检错码**、纠错码和纠删码。检错码只能检测误码；纠错码只可以纠正误码；而纠删码既可以检测误码也可以纠正误码，如果无法纠正就发出一个错误指示或者简单删除误码。

按信息码元与附加监督码元之间的检验关系分，有**线性码**和非线性码。信息码元与监督码元之间的关系为线性关系，满足一组线性方程式，就称为线性码；反之是非线性码。

按信息码元与监督码元之间的约束方式不同可以分为**分组码**与卷积码。在分组码中，编码后的码元序列每n位为一组，里面包含k个信息码元，r个附加监督码元，n=k+r，监督码元只与本组的信息码有关，与其他组的信息码无关；卷积码的监督码元不仅与本组信息码有关，还与前面码组的信息码元有关。

按信息码元在编码后是否保持原来的形式不变可以分为**系统码**与非系统码。系统码中信息码元被编码后保持不变，而非系统码的信息码元被编码后则被改变。

### 1.2 线性分组码

线性分组码中的信息码元与监督码元由线性方程联系起来，主要性质如下：

* （1）任意两许用码组之和（模2和）仍为一许用码组，线性码具有封闭性。
* （2）码的最小距离等于非零码的最小重量（非零码元的数目为码的重量）。

#### 1.2.1 监督矩阵

这里是信息码元，是监督码元。

是监督矩阵，表面信息码元与监督码元之间的校验关系完全由决定，可以写为形式的监督矩阵称为典型监督矩阵。典型监督矩阵的各行一定是线性无关的。

#### 1.2.2 生成矩阵

是典型监督矩阵中的转置。

在的前面补上阶单位矩阵即为生成矩阵。

因为由它可以从信息码元生成整个码组，所以称它为生成矩阵。

### 1.3 循环码

循环码是**线性分组码**的一个重要子类，并且易于用带反馈的移位寄存器实现。

循环码的特点：

* （1）循环码中任一许用码组经过循环移位后仍为一许用码组。
* （2）线性分组码的封闭性。

可以用码多项式来表示一个码组：

可以表示为：

是一个实变量，它的幂次代表移位的次数。上述码组左移一位记作：

它的码多项式为：

由此可知，左移位的码组的码多项式为：

是除以的商式，是所得的余式。

#### 1.3.1 循环码的生成多项式

循环码的信息码组有个，因此循环码许用码组也有个。

类似于之前的生成矩阵有：

为不大于阶的多项式，是不大于阶的多项式，因此是一个阶的多项式。

是左移位所得，由上一个公式可知它是的倍式：

所以有：

因此一定是的因式。

#### 1.3.2 循环码的生成矩阵

取为：

当输入信息码元为时：

由此可知所有码多项式必为的倍式。

#### 1.3.3 系统循环码

这样生成的循环码并不是系统码，系统码要求码的左位为信息码元，随后是位监督码元，相当于码多项式为：

其中是监督吗多项式，对应监督码元。

把移到等号右边，移到等号左边，就变成了一个被除数等于商乘除数加余数的形式：

由此可知：

构造系统循环码时只需要将信息码元升阶，也就是乘以，然后以为模，所得余式即为监督码元。

系统码的生成矩阵必为典型形式，与单位矩阵每行对应的信息多项式为：

相应的监督多项式为：

得到生成矩阵中每行的码多项式为：

因此系统循环码的生成矩阵一般表示为：

### 1.4 CRC码

#### 1.4.1 实现原理

循环冗余校验（Cyclic redundancy check, CRC）码，是一种系统循环码。

系统循环码的构成为：

这里信息多项式是 ，生成多项式是 ，即为我们所需要的CRC校验码。表示在信息码后面加上个零。是系统循环码，高位为信息码，低位为监督码。

可以知道系统循环码较容易的实现方式是将信息码升次幂后除以生成多项式，然后将所得余式放在升幂后的信息多项式后。

这种实现过程主要使用的就是多项式除法。

#### 1.4.2 模2除法

* 例1

CRC有很多标准，我们随便选取一个示例。

我们发送1字节的信息‘A’，也就是：

先对升次幂，也就是在其后补16个0：

接下来就开始长除法求余数：

01000101-------------------->Q(x)  
 g(x) --------------------------  
10001000000100001 | 010000010000000000000000---->M(x)x^16  
 10001000000100001  
 --------------------------  
 000010100001000010000  
 10001000000100001  
 --------------------------  
 0010100100011000100  
 10001000000100001  
 --------------------------  
 00101100011100101---->R(x) 0x58e5

由上面的结果就可以得到

把放在后面就是循环系统码

* 例2

依然使用上面的生成多项式，唯一的区别是我们给信息码前面补8个0，相当于发送

0000000001000101-------------------->Q(x)  
 g(x) ----------------------------------  
10001000000100001 | 00000000010000010000000000000000---->M(x)x^16  
 10001000000100001  
 ----------------------------------  
 000010100001000010000  
 10001000000100001  
 ----------------------------------  
 0010100100011000100  
 10001000000100001  
 ----------------------------------  
 00101100011100101---->R(x) 0x58e5

可以很容易看到，在信息码前面加零是对最后的余式没有影响的，最后得到的依然是相同的

## 2. 长除法c语言实现

上面提到的长除法的实现流程为：

1. **把CRC寄存器（16位）初始化为0**；
2. 在信息码后面追加16个0；
3. 所有信息码与0都已经移入，转到4；否则继续，

* CRC寄存器左移一位，移入一位信息码到CRC寄存器的最低位，
* 如果CRC寄存器移出的一位是1，CRC寄存器与生成多项式异或；
* 如果CRC寄存器移出的一位是0，不做操作，继续回到3；

1. 此时CRC寄存器中的值就是我们求得的余式，即CRC监督码。

u16 crc16\_ccitt\_nondirect (u8 \*data, u8 length, u32 init\_value)  
{  
 u16 i, j;  
 u32 crc\_reg = init\_value;  
 u16 crc\_gx = 0x1021;  
 u16 crc\_num = 16;  
 u8 crc\_byte;  
 // 处理所有字节 先移入每个字节的最高位  
 for (j = 0; j<length; ++j, ++data)  
 {  
 // 处理每个bit  
 for (i = 0; i<8; ++i)  
 {  
 // 左移一位，下一位bit移入crc\_reg  
 crc\_reg = (crc\_reg << 1) | ((crc\_byte & (0x01 << (7-i))) >> (7-i));  
 if ((crc\_reg & (0x00000001 << crc\_num)) == (0x00000001 << crc\_num))  
 {  
 // crc\_reg 移出的是1  
 // crc\_reg 减去（异或）生成多项式gx  
 crc\_reg = crc\_reg ^ crc\_gx;  
 }  
 }  
 }  
 return crc\_reg & 0x0000ffff;   
}

**注意：实际实现的过程中就引出了CRC计算时寄存器的初值问题。当寄存器初值为0时，就相当于在信息码前面加零，对最后运算得出的余式没有影响；当寄存器初值不为0式，就会有不同的结果。**

一个大牛Ross N. Williams是这么说的：

Most CRC algorithms initialize their register to zero. However, some initialize it to a non-zero value. In theory (i.e. with no assumptions about the message), the initial value has no affect on the strength of the CRC algorithm, the initial value merely providing a fixed starting point from which the register value can progress. However, in practice, some messages are more likely than others, and it is wise to initialize the CRC algorithm register to a value that does not have "blind spots" that are likely to occur in practice. By "blind spot" is meant a sequence of message bytes that do not result in the register changing its value. In particular, any CRC algorithm that initializes its register to zero will have a blind spot of zero when it starts up and will be unable to "count" a leading run of zero bytes. As a leading run of zero bytes is quite common in real messages, it is wise to initialize the algorithm register to a non-zero value.

大多数CRC算法把他们的寄存器初始化为0。但是，有些把它初始化为非零值。理论上（不对信息进行任何假设），初始值的选择不影响CRC算法的有效性，初始值仅仅提供了一个寄存器值开始运算的固定起点。然而，在实践中，有些信息出现的可能性要更高，因此把CRC寄存器初始化为一个不太可能在实际中出现“盲点”的值是更明智的。“盲点”是指一系列不会导致CRC寄存器值变化的信息字节。特别是当CRC寄存器被初始化为0时就具有“零盲点”，当它运行的时候就不知道前面有多少个零字节。而零字节在实际应用中充当信息的前导字节是非常普遍的，所以把CRC寄存器初始化为非零值是一个明智的做法。

函数运算输出：

int main()  
{  
 u8 d[10];  
 u8 \*ptr = d;  
 u8 d\_len = 1;  
 d[0] = 'A'; // 0x41, 0b01000001  
 d[1] = 0x00;d[2] = 0x00; // 在信息码后面补零  
   
 // CRC寄存器初值为0x0000  
 printf("CRC寄存器初值为0x0000, CRC = 0x%04x\n",\  
 crc16\_ccitt\_nondirect(ptr, d\_len+2, 0x0000));  
 // CRC寄存器初值为0x0000, CRC = 0x58e5   
   
 // CRC寄存器初值为0xffff  
 printf("CRC寄存器初值为0xffff, CRC = 0x%04x\n",\  
 crc16\_ccitt\_nondirect(ptr, d\_len+2, 0xffff));  
 // CRC寄存器初值为0xffff, CRC = 0x9479  
}

## 3. 线性移位寄存器电路实现

### 3.1 长除法到寄存器电路的推导

多项式除法可以用带反馈的线性移位寄存器来实现。

以为例推导：

生成多项式为，

数据序列为，

CRC寄存器为，，为串行输入第个数据后寄存器第位的值。

如果在第个数后新输入一个数据为：

* 讨论：

如果，余式为；

如果，余式为

可知。

* 讨论：

如果，余式为；

如果，余式为；

可知。

由以上结果可知：

所以循环移位寄存器的递推关系为：

对于我们前面举的例子：

循环移位寄存器的关系为：

相应的RTL代码就是：

always @ (posedge clk) begin  
 if (reset) begin  
 c <= 16'h0000;  
 end else if (enable) begin  
 if (init) begin  
 c <= 16'h0000;  
 end else begin  
 c[0] <= c[15] ^ m;  
 c[1] <= c[0];  
 c[2] <= c[1];  
 c[3] <= c[2];  
 c[4] <= c[3];  
 c[5] <= c[4] ^ c[15] ^ m;  
 c[6] <= c[5];  
 c[7] <= c[6];  
 c[8] <= c[7];  
 c[9] <= c[8];  
 c[10] <= c[9];  
 c[11] <= c[10];  
 c[12] <= c[11] ^ c[15] ^ m;  
 c[13] <= c[12];  
 c[14] <= c[13];  
 c[15] <= c[14];  
 end  
 end  
end

### 3.2 线性移位寄存器的c语言仿真

用c语言再模拟一下这个过程，因为我们推导的时候已经考虑到了在信息码后面补零，现在使用线性移位寄存器的时候就不用在信息码后面再补零了。

u16 crc16\_ccitt\_direct(u8 \*data, u8 length, u32 init\_value)  
{  
 u16 i, j;  
 u32 crc\_reg = init\_value;  
 u16 crc\_gx = 0x1021;  
 u16 crc\_num = 16;  
 u32 crc\_bit16, crc\_in;  
 u8 crc\_byte;  
 // 处理所有字节  
 for (j = 0; j<length; ++j, ++data)  
 {  
 // 处理每个bit  
 for (i = 0; i<8; ++i)  
 {  
 // crc\_reg 移出的最高位  
 crc\_bit16 = (crc\_reg & (0x0001 << (crc\_num -1 ))) >> (crc\_num -1 );  
 // 下一位移入的bit  
 crc\_in = ((crc\_byte & (0x01 << (7-i))) >> (7-i));  
  
 switch(crc\_bit16 + crc\_in)  
 {  
 case 0x00000000:  
 case 0x00000002:  
 // 异或为0，只进行左移  
 crc\_reg = (crc\_reg << 1) ;  
 break;  
 case 0x00000001:  
 // 异或为1，左移后再与g(x)异或  
 crc\_reg = (crc\_reg << 1);  
 crc\_reg = crc\_reg ^ crc\_gx;  
 break;  
 default:  
 break;  
 }  
 }  
 }  
 return crc\_reg & 0x0000ffff;  
}

函数运算输出：

int main()  
{  
 u8 d[10];  
 u8 \*ptr = d;  
 u8 d\_len = 1;  
 d[0] = 'A'; // 0x41, 0b01000001  
 d[1] = 0x00;d[2] = 0x00; // 在信息码后面补零  
   
 // CRC寄存器初值为0x0000  
 printf("CRC寄存器初值为0x0000, CRC = 0x%04x\n",\  
 crc16\_ccitt\_direct(ptr, d\_len, 0x0000));  
 // CRC寄存器初值为0x0000, CRC = 0x58e5   
   
 // CRC寄存器初值为0xffff  
 printf("CRC寄存器初值为0xffff, CRC = 0x%04x\n",\  
 crc16\_ccitt\_direct(ptr, d\_len, 0xffff));  
 // CRC寄存器初值为0xffff, CRC = 0xb915  
}

### 3.3 线性移位寄存器的verilog仿真

写个testbench验证一下：

iverilog -s testbench -o rtl/tb\_serial\_crc.vvp rtl/tb\_serial\_crc.v rtl/serial\_crc.v  
vvp -N rtl/tb\_serial\_crc.vvp  
VCD info: dumpfile testbench.vcd opened for output.  
m = 0, i = 0, crc\_out\_0000 = 0000, crc\_out\_ffff = ffff  
m = 1, i = 1, crc\_out\_0000 = 0000, crc\_out\_ffff = efdf  
m = 0, i = 2, crc\_out\_0000 = 1021, crc\_out\_ffff = dfbe  
m = 0, i = 3, crc\_out\_0000 = 2042, crc\_out\_ffff = af5d  
m = 0, i = 4, crc\_out\_0000 = 4084, crc\_out\_ffff = 4e9b  
m = 0, i = 5, crc\_out\_0000 = 8108, crc\_out\_ffff = 9d36  
m = 0, i = 6, crc\_out\_0000 = 1231, crc\_out\_ffff = 2a4d  
m = 1, i = 7, crc\_out\_0000 = 2462, crc\_out\_ffff = 549a  
crc\_0000 = 58e5  
crc\_ffff = b915

可以看到跟上面c语言仿真的结果是相同的。

## 4. 初值问题

### 4.1 为什么结果不同

仔细看前面的仿真结果，问题来了。

当寄存器初值是0x0000的时候循环移位寄存器的实现方法与长除法的结果是相同的，都是0x58e5；

而当寄存器初值是0xffff的时候循环移位寄存器实现的结果是0xb915，长除法的结果是0x9479，这是为啥呢？

|  |  |  |
| --- | --- | --- |
| 寄存器初值 | 硬件实现/循环移位寄存器 | 软件实现/长除法 |
| 0x0000 | 0x58e5 | 0x58e5 |
| 0xffff | 0xb915 | 0x9479 |

回顾一下前面长除法到循环移位寄存器的推导过程。

基础的长除法，我们并没有寄存器初值这个概念，直接对后面补零的信息码进行长除求余式，就可以得到我们所需要的CRC。

但是在求余式的过程中，我们发现**对于长除法来说，信息码前面加多少零都是不影响最后所求的CRC的**。而在后面c语言实现长除法的时候由于编程需要，出现了一个CRC寄存器，信息码一个一个移入，最后得到的CRC就存在这个寄存器里，自然而然的就把它初始化为了0x0000，反正初始化为0对后面的CRC运算也没有影响。

**长除法的初值实际有啥影响呢，也就是可以看作在信息码的前面又人为添加了两个字节的信息，之后才是我们想要校验的信息码和补充的0**。

再来看我们硬件循环移位寄存器实现的推导，整个推导就没有提到与初值相关的事情。

我们的**假设是前k个bit已经计算出了一个CRC**。推导描述的内容是，**如果再增加一个bit，这个新的CRC跟旧的CRC是什么关系**。所以我们这个推导得到的是一个**递推关系**，有点像高中数学数列里的递推关系式，数列的确定还需要初始值的确定，也就是第一个CRC是啥。而第一个CRC就是我们CRC寄存器的初值。

这个CRC寄存器的初值本质上也是一个CRC，同样是由**一串数据**通过长除法计算得到，为了统一长除法和移位寄存器法，我们就得好奇，到底啥样的数据才能算出（长除法）和这个CRC初始值相同的CRC呢？

### 4.2 给定CRC求信息码

下面是长除法计算的公式：

现在已知生成多项式与，**第二个等式左边的部分低位都是0，我们的CRC也是位，所以利用这个关系就可以计算出相应的**。

我们的CRC标准还是用之前的，

CRC初值选择上面有争议的0xffff。

因为计算过程只与低16位相关，并且有16个方程，所以设

计算，只考虑低16位，即忽略:

我们的条件是：

则所对应的方程为：

根据这个方程我们就可以解出是0x8C1F。

解出的这个0x8C1F是商，再带入这个公式，计算模2乘法：

就可以计算出M(x)对应的信息码是0x84CF。也就是说用长除法计算0x84CF的CRC，最后结果就是0xFFFF。

### 4.3 测试初值与结果的关系

用c代码仿真一下。

#### 4.3.1 长除法的性质

//信息码字节数m\_k为0，长除法初值为0x84cf  
//crc16\_ccitt\_nondirect(ptr, m\_k+2, 0x84CF);  
长除 数据长度:0 初值:0x84cf  
信息码: | 0x00 0x00  
CRC = 0xffff  
  
//信息码字节数m\_k为2，长除法初值为0x0000  
//crc16\_ccitt\_nondirect(ptr, m\_k+2, 0x0000);  
长除 数据长度:2 初值:0x0000  
信息码: 0x84 0xcf | 0x00 0x00  
CRC = 0xffff

通过这个例子我们可以确认两点：

* 长除法计算出0x84cf的CRC值为0xffff（代码段1）
* 给长除法的CRC寄存器赋初值就相当于在信息码之前增加这个初值，然后再做初值为0的普通长除法（代码段2）

#### 4.3.2 长除法与线性移位寄存器电路的关系

接下来验证长除法与线性移位寄存器电路的关系。

//strcpy((char \*)ptr, "123456789");  
//信息码字节数m\_k为9  
//crc16\_ccitt\_direct(ptr, m\_k, 0xffff);  
串行 数据长度:9 初值:0xffff  
0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39  
CRC = 0x29b1  
  
//strcpy((char \*)ptr+2, "123456789");  
//\*ptr=0x84;\*(ptr+1)=0xcf;  
//信息码字节数m\_k为11  
//crc16\_ccitt\_nondirect(ptr, m\_k+2, 0x0000);  
长除 数据长度:11 初值:0x0000  
信息码: 0x84 0xcf 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 | 0x00 0x00  
CRC = 0x29b1  
  
//strcpy((char \*)ptr, "123456789");  
//信息码字节数m\_k为9  
//crc16\_ccitt\_nondirect(ptr, m\_k+2, 0x84CF);  
长除 数据长度:9 初值:0x84cf  
信息码: 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 | 0x00 0x00  
CRC = 0x29b1

通过这个例子我们可以确认两点：

* 线性移位寄存器初值为0xffff时计算出测试信息CRC为0x29b1（代码段1）
* 线性移位寄存器初值为0xffff相当于前面默认有一串信息通过长除法计算出的CRC为0xffff
* 因为0x84cf通过长除法计算出的CRC为0xffff，把它添加到测试信息前，对这一段信息使用长除法也可以计算出CRC为0x29b1（代码段2）
* 再信息码前添加一段信息就相当于改变长除法计算CRC时的初始寄存器
* 使用初值为0x84cf的长除法计算测试信息码得到的CRC也是0x29b1

#### 4.3.3 初值为0x0000时的情况

再来看初值为0x0000时的情况：

//strcpy((char \*)ptr, "123456789");  
//信息码字节数m\_k为9  
//crc16\_ccitt\_nondirect(ptr, m\_k+2, 0x0000);  
长除 数据长度:9 初值:0x0000  
信息码: 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 | 0x00 0x00  
CRC = 0x31c3  
  
//strcpy((char \*)ptr, "123456789");  
//信息码字节数m\_k为9  
//crc16\_ccitt\_direct(ptr, m\_k, 0x0000);  
串行 数据长度:9 初值:0x0000  
0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39  
CRC = 0x31c3

在这个测试中我们发现，长除法和线性移位寄存器的初值为0x0000时计算出的CRC竟然一样，不是说线性移位寄存器的初值是一个CRC，长除法的初值表示的是一段信息码吗？

原因在于对于零初值的长除法，输入信息码为0，输出的CRC也是0，也就是说0x0000的CRC是0x0000，线性移位寄存器的初值为0x0000相当于在信息码前面添加了一个0x0000再进行长除法计算。而长除法计算时前面无论加多少个零对结果都是没有影响的，所以就导致初值为0的线性移位寄存器与长除法计算出的CRC结果相同。

### 4.4 总结一下

* 由定义推导出的CRC计算方法是零初值的长除法，这相当于是一个通项公式。
* 线性移位寄存器的推导表示了新增一个信息码，已经计算出的CRC通过怎样的运算可以的得到新的CRC，这相当于是一个递推公式。
* 长除法计算时的初值是一段信息码，相当于把它添加在原有信息码的前面再进行零初值长除法计算。
* 线性移位寄存器的初值一个CRC，相当于把计算出这个CRC的信息码添加在原有信息码的前面再进行零初值长除法计算。
* 0x0000的CRC是0x0000
* 初值为0x0000的长除法与初值为0x0000的线性移位寄存器的计算结果相同
* 0x84cf的CRC是0xffff
* 初值为0x84cf的长除法与初值为0xffff的线性移位寄存器的计算结果相同

In theory (i.e. with no assumptions about the message), the initial value has no affect on the strength of the CRC algorithm, the initial value merely providing a fixed starting point from which the register value can progress.  
理论上初值对CRC算法的有效性没有影响，它只影响CRC算法的”盲点“。

所谓盲点，我个人的理解就是，不同的信息码可以计算出相同的CRC。

* 对于初值为0x0000的长除法，它的盲点就是0x0000，在信息码的前面加多少个零都不影响CRC的值。
* 对于其他初值的算法，这个盲点就比较复杂了，暂不讨论，更深的讨论应该放到CRC的检测误码性能中。

## 5. CRC的种类

根据我们上面的推导与实现的过程，可以得到确定一个CRC算法的特征：

* width CRC的长度
* poly 生成多项式
* init 初始值
* 上面这三个特征基本上就是核心内容了

经过多方考察，CRC算法的特征还有以下三点：

* refin 输入字节比特序是否反转
* refout 输出CRC比特序是否反正
* xorout 输出CRC是否需要异或一个值

前三点更重要的说明算法的理论部分，而后三点更多的是表明算法的实现部分。

### CRC-16/CCITT

我们全篇讨论最多的就**类似于**这个CRC16-CCITT算法，它的主要特征是：

* width=16
* poly=0x1021
* init=0xffff

但是网络上并没有一个清晰的、统一的、完整的文件对它进行描述。前两条没有什么争论，而第三条就有问题了，这个初值到底是长除法的初值还是串行实现的初值，也就是这个初值到底是一个信息码还是一个CRC？

### CRC-16/IBM-3740

这个CCITT是非常模糊的，它并不精确，所以我又找到一个网站，上面列出了已经经过证实的CRC16算法，可以看出“CRC-16/IBM-3740”更符合我们所形容的CRC，它也给出了测试数据的CRC计算结果。

“CRC-16/IBM-3740”的主要特征是

* width=16
* poly=0x1021
* init=0xffff
* refin=false
* refout=false
* xorout=0x0000

标准测试信息码”123456789“，CRC为0x29b1。

与我们前面仿真结果对比，可以确定这个网站里的初值意思是一个CRC，也就是这是一个线性移位寄存器实现运算的结果。

话又说回来，再一次重复，CRC算法计算的初值选择并不影响CRC校验的有效性。所以我们选择初值的时候在发送方与接收方之间保持一致就可以了，确定好你们使用的CRC的每个特征，再给出一个测试信息计算结果，保证校验的正确性就可以了。

## 参考文献

[1] 曹志刚, 钱亚生. 现代通信原理[M]. 清华大学出版社有限公司, 1992.

[2] CRC16-CCITT by Joe Geluso <http://srecord.sourceforge.net/crc16-ccitt.html#refs>

[3] CRC calculation, by Sven Reifegerste <http://www.zorc.breitbandkatze.de/crc.html>

[4] "A Painless Guide to CRC Error Detection Algorithms" by Ross N. Williams. <https://www.zlib.net/crc_v3.txt>

[5] Cyclic redundancy check <https://en.wikipedia.org/wiki/Cyclic_redundancy_check>

[6] CRC RevEng <http://reveng.sourceforge.net/crc-catalogue/16.htm#crc.cat.crc-16-kermit>