# CRC 校验一探究竟

曹嘉辉

Edit 2021.2.26

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

## 目录

## 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

$$
\begin{aligned}
H &= [P_{r\times k} \mid I_{r\times r}] \\
A &= [A_{1\times k} \mid A_{1\times r}] \\
0 &= [0_{1\times n}] \\
HA^T &= 0^T
\end{aligned}
\tag{1}
$$

这里 $A_{1\times k}$ 是信息码元，$A_{1\times r}$ 是监督码元。

$H$ 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 $H$ 决定，可以写为 $[P_{r\times k} \mid I_{r\times r}]$ 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

$$
\begin{aligned}
Q &= P_{r\times k}^{T} \\
G &= [I_{k\times k} \mid Q] \\
A &= A_{1\times k} * G
\end{aligned} \tag{2}
$$

$Q$ 是典型监督矩阵中 $P$ 的转置。

在 $Q$ 的前面补上 $k$ 阶单位矩阵 $I$ 即为生成矩阵 $G$。

因为由它可以从信息码元 $A_{1\times k}$ 生成整个码组 $A$，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：

$$
A = (a_{n-1}, a_{n-2}, \cdots, a_1, a_0) \tag{3}
$$

可以表示为：

$$
A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x^1 + a_0 \tag{4}
$$

$x$ 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：

$$
A^{(1)} = (a_{n-2}, a_{n-3}, \cdots, a_0, a_{n-1}) \tag{5}
$$

它的码多项式为：

$$
A^{(1)}(x) = a_{n-2}x^{n-1} + a_{n-3}x^{n-2} + \cdots + a_0x^1 + a_{n-1} \tag{6}
$$

由此可知，左移 $i$ 位的码组的码多项式为：

$$\begin{aligned} A(x)x^i &= Q(x)(x^n+1) + A^{(i)}(x) \\ A^{(i)}(x) &= A(x)x^i \ mod \ (x^n+1) \end{aligned} \tag{7}$$

$Q(x)$ 是 $x^iA(x)$ 除以 $x^n+1$ 的商式，$A^{(i)}(x)$ 是所得的余式。

**1.3.1 循环码的生成多项式** $(n,k)$ 循环码的信息码组 $M(x)$ 有 $2^k$ 个，因此循环码许用码组 $A(x)$ 也有 $2^k$ 个。

类似于之前的生成矩阵有：

$$A(x) = M(x)g(x) \tag{8}$$

$M(x)$ 为不大于 $k-1$ 阶的多项式，$A(x)$ 是不大于 $n-1$ 阶的多项式，因此 $g(x)$ 是一个 $n-k$ 阶的多项式。

$$x^kg(x) = Q(x)(x^n+1) + g^{(k)}(x) \tag{9}$$

$g^{(k)}(x)$ 是 $g(x)$ 左移 $k$ 位所得，由上一个公式可知它是 $g(x)$ 的倍式：

$$g^{(k)}(x) = M(x)g(x) \tag{10}$$

所以有：

$$\begin{aligned} x^kg(x) &= Q(x)(x^n+1) + M(x)g(x) \\ x^kg(x) + M(x)g(x) &= Q(x)(x^n+1) \\ (x^k + M(x))g(x) &= F(x)g(x) = Q(x)(x^n+1) \end{aligned} \tag{11}$$

因此 $g(x)$ 一定是 $x^n+1$ 的因式。

**1.3.2 循环码的生成矩阵** 取 $G(x)$ 为：

$$G(x) = \begin{bmatrix} x^{k-1}g(x) \\ x^{k-2}g(x) \\ \vdots \\ g(x) \end{bmatrix} \tag{12}$$

当输入信息码元为 $[m_{k-1}, m_{k-2}, \cdots, m_0]$ 时：

$$\begin{aligned} A(x) &= [m_{k-1}, m_{k-2}, \cdots, m_0]G(x) \\ &= [m_{k-1}x^{k-1}, m_{k-2}x^{k-2}, \cdots, m_0]g(x) \\ &= M(x)g(x) \end{aligned} \tag{13}$$

由此可知所有码多项式 $A(x)$ 必为 $g(x)$ 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 $k$ 位为信息码元，随后是 $n-k$ 位监督码元，相当于码多项式为：

$$\begin{aligned} A(x) &= M(x)x^{n-k} + R(x) \\ &= m_{k-1}x^{n-1} + m_{k-2}x^{n-2} + \cdots + m_0x^{n-k} + \\ &\quad r_{n-k-1}x^{n-k-1} + \cdots + r_1x + r_0 \end{aligned} \tag{14}$$

其中 $R(x) = r_{n-k-1}x^{n-k-1} + \cdots + r_1x + r_0$ 是监督吗多项式，对应监督码元 $(r_{n-k-1}, \cdots, r_0)$。

把 $R(x)$ 移到等号右边，$M(x)x^{n-k}$ 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：

$$\begin{aligned} M(x)x^{n-k} &= A(x) + R(x) \\ &= M(x)g(x) + R(x) \end{aligned} \tag{15}$$

由此可知：

$$R(x) = M(x)x^{n-k} \, mod \, g(x) \tag{16}$$

构造系统循环码时只需要将信息码元 $M(x)$ 升 $n-k$ 阶，也就是乘以 $D^{n-k}$，然后以 $g(x)$ 为模，所得余式 $R(x)$ 即为监督码元。

系统码的生成矩阵必为典型形式 $G=[I_k\,Q]$，与单位矩阵 $I_k$ 每行对应的信息多项式为：

$$m_i(x)=m_i x^{k-i}=x^{k-i},\ i=1,2,\cdots,k \tag{17}$$

相应的监督多项式为：

$$r_i(x)=x^{k-i}x^{n-k}=x^{n-i}\ mod\ g(x) \tag{18}$$

得到生成矩阵中每行的码多项式为：

$$C_i(x)=x^{n-i}+r_i(x) \tag{19}$$

因此系统循环码的生成矩阵一般表示为：

$$G(x)=\begin{bmatrix}C_1(x)\\C_2(x)\\\vdots\\C_k(x)\end{bmatrix}=\begin{bmatrix}x^{n-1}+r_1(x)\\x^{n-1}+r_1(x)\\\vdots\\x^{n-k}+r_k(x)\end{bmatrix} \tag{20}$$

### 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：

$$\begin{aligned}\frac{M(x)x^{n-k}}{g(x)}&=Q(x)+\frac{R(x)}{g(x)}\\A(x)&=M(x)x^{n-k}+R(x)\end{aligned} \tag{21}$$

这里信息多项式是 $M(x)$，生成多项式是 $g(x)$，$R(x)$ 即为我们所需要的 CRC 校验码。$M(x)x^{n-k}$ 表示在信息码后面加上 $n-k$ 个零。$A(x)$ 是系统循环码，高 $k$ 位为信息码，低 $n-k$ 位为监督码。

$$R(x) = M(x)x^{n-k} \, mod \, g(x) \tag{22}$$

可以知道系统循环码较容易的实现方式是将信息码升 $n-k$ 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 $g(x)$ 示例。

$$\begin{gathered} g(x) = x^{16} + x^{12} + x^5 + 1 \\ (10001000000100001) \end{gathered} \tag{23}$$

我们发送 1 字节的信息 'A'，也就是 $41H$：

$$\begin{gathered} M(x) = x^6 + 1 \\ (01000001) \end{gathered} \tag{24}$$

先对 $M(x)$ 升 $n-k=16$ 次幂，也就是在其后补 16 个 0：

$$\begin{gathered} M(x)x^{16} = (x^6 + 1)x^{16} \\ (01000001000000000000000) \end{gathered} \tag{25}$$

接下来就开始长除法求余数：

```
                    01000101--------------------->Q(x)
      g(x)          --------------------------
10001000000100001 | 01000001000000000000000----->M(x)x^16
                     10001000000100001
```

```
                        -------------------------
                          00001010000010000010000
                              10001000000100001
                        -------------------------
                               0010100100011000100
                                 10001000000100001
                        -------------------------
                                  00101100011100101----->R(x) 0x58e5
```

由上面的结果就可以得到 $R(x)$

$$R(x) = x^{14} + x^{12} + x^{11} + x^7 + x^6 + x^5 + x^2 + 1 \\ (0101100011100101) \tag{26}$$

把 $R(x)$ 放在 $M(x)$ 后面就是循环系统码 $A(x)$

$$\begin{aligned} A(x) &= M(x)x^{16} + R(x) \\ &= (x^6 + 1)x^{16} + x^{14} + x^{12} + x^{11} + x^7 + x^6 + x^5 + x^2 + 1 \\ &(01000001\ 0101100011100101) \end{aligned} \tag{27}$$

- 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 $00H, 00H, 41H$

```
                          0000000001000101--------------------->Q(x)
      g(x)                ---------------------------------
10001000000100001 | 00000000010000010000000000000000----->M(x)x^16
                              10001000000100001
                          ---------------------------------
                                00001010000010000010000
                                    10001000000100001
                          ---------------------------------
                                       0010100100011000100
                                         10001000000100001
```

```
---------------------------------
                  0010110001110010 1----->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 $R(x)$

## 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；
4. 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 $crc16$ 为例推导：

生成多项式为 $g(x)$，$g(x) = x^{16} + g_{15}x^{15} + g_{14}x^{14} + \cdots + g_1x + g_0$

数据序列为 $M(x)$，$M(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1x + m_0$

CRC 寄存器为 $C(x)$，$C(x) = c_{15}^i x^{15} + c_{14}^i x^{14} + \cdots + c_1^i x + c_0^i$，$C_{r-1}^i$ 为串行输入第 $i$ 个数据后寄存器第 $r-1$ 位的值。

$$\begin{aligned} &c_{15}^i x^{15} + c_{14}^i x^{14} + \cdots + c_1^i x + c_0^i \\ =&(m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1x + m_0)x^{16} \ mod \ g(x) \end{aligned} \tag{28}$$

如果在第 $i$ 个数后新输入一个数据为 $m$：

$$\begin{aligned} &c_{15}^{i+1} x^{15} + c_{14}^{i+1} x^{14} + \cdots + c_1^{i+1} x + c_0^{i+1} \\ =&[(m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1x + m_0)x^{17} + mx^{16}] \ mod \ g(x) \\ =&(m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1x + m_0)x^{17} \ mod \ g(x) + mx^{16} \ mod \ g(x) \end{aligned} \tag{29}$$

- 讨论 $mx^{16} \ mod \ g(x)$：

如果 $m = 1$，余式为 $g(x) + x^{16} = g_{15}x^{15} + g_{14}x^{14} + \cdots + g_1x + g_0$；

如果 $m = 0$，余式为 $0$

可知 $mx^{16} \ mod \ g(x) = mg_{15}x^{15} + mg_{14}x^{14} + \cdots + mg_1x + mg_0$。

- 讨论 $[(c_{15}^i x^{15} + c_{14}^i x^{14} + \cdots + c_1^i x + c_0^i)x] \ mod \ g(x)$：

如果 $c_{15}^i = 1$，余式为 $(c_{14}^i + g_{15})x^{15} + (c_{13}^i + g_{14})x^{14} + \cdots + (c_0^i + g_1)x + g_0$；

如果 $c_{15}^i = 0$，余式为 $c_{14}^i x^{15} + \cdots + c_1^i x^2 + c_0^i x$；

可知 $[(c_{15}^i x^{15} + c_{14}^i x^{14} + \cdots + c_1^i x + c_0^i)x] \, mod \, g(x) = \$c_{14}^i + c_{15}^i g_{15})x^{15} + (c_{13}^i + c_{15}^i g_{14})x^{14} + \cdots + (c_0^i + c_{15}^i g_1)x + c_{15}^i g_0$。

由以上结果可知：

$$
\begin{aligned}
&= (c_{14}^i + c_{15}^i g_{15})x^{15} + (c_{13}^i + c_{15}^i g_{14})x^{14} + \cdots + (c_0^i + c_{15}^i g_1)x + c_{15}^i g_0 + \\
&\quad m g_{15} x^{15} + m g_{14} x^{14} + \cdots + m g_1 x + m g_0 \\
&= (c_{14}^i + (c_{15}^i + m) g_{15})x^{15} + (c_{13}^i + (c_{15}^i + m) g_{14})x^{14} + \cdots + \\
&\quad (c_0^i + (c_{15}^i + m) g_1)x + (c_{15}^i + m) g_0
\end{aligned}
\tag{30}
$$

所以循环移位寄存器的递推关系为：

$$
\begin{aligned}
c_{15}^{i+1} &= c_{14}^i + (c_{15}^i + m) g_{15} \\
c_{14}^{i+1} &= c_{13}^i + (c_{15}^i + m) g_{14} \\
&\vdots \\
c_1^{i+1} &= c_0^i + (c_{15}^i + m) g_1 \\
c_0^{i+1} &= (c_{15}^i + m) g_0
\end{aligned}
\tag{31}
$$

对于我们前面举的例子：

$$
\begin{aligned}
g(x) &= x^{16} + x^{12} + x^5 + 1 \\
g_{12} &= g_5 = g_0 = 1 \\
g_{others} &= 0
\end{aligned}
\tag{32}
$$

循环移位寄存器的关系为：

$$
\begin{aligned}
c_{12}^{i+1} &= c_{11}^i + c_{15}^i + m \\
c_5^{i+1} &= c_4^i + c_{15}^i + m \\
c_0^{i+1} &= c_{15}^i + m \\
c_k^{i+1} &= c_{k-1}^i,\ k \neq 12, 5, 0
\end{aligned}
\tag{33}
$$

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。

$$CRC(x) = M(x)x^{n-k} \, mod \, g(x) \tag{34}$$

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**。

$$
\begin{aligned}
init\_value &= (i_{15}, \cdots, i_0)\ M(x) \\
CRC_1(x) &= M(x)x^{n-k}\ mod\ g(x) \\
init\_value &= (0_{15}, \cdots, 0_0)\ \&\&\ M_{new} = (i_{15}, \cdots, i_0, m_{k-1}, m_{k-2}, \cdots, m_0) \\
CRC_2(x) &= M_{new}(x)x^{n-k}\ mod\ g(x) \\
CRC_1(x) &\equiv CRC_2(x)
\end{aligned}
\tag{35}
$$

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。

$$
\begin{aligned}
CRC^i(x) -> CRC^{i+1}(x) \\
init\ value = CRC^0(x)
\end{aligned}
\tag{36}
$$

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：

$$
\begin{aligned}
CRC(x) = M(x)x^{n-k}\ mod\ g(x) \\
M(x)x^{n-k} = Q(x)g(x) + CRC(x)
\end{aligned}
\tag{37}
$$

现在已知生成多项式 $g(x)$ 与 $CRC(x)$，**第二个等式左边的部分低 $n-k$ 位都是 0，我们的 CRC 也是 $n-k$ 位，所以利用这个关系就可以计算出相应的** $M(x)$。

$$\begin{aligned}(0_{15},\cdots,0_0) &\equiv [Q(x)g(x)]lSB16bit + CRC(x)\\ [Q(x)g(x)]LSB16bit &\equiv ((0_{15},\cdots,0_0) + CRC(x)\end{aligned} \tag{38}$$

我们的 CRC 标准还是用之前的，

$$\begin{aligned}g(x) &= x^{16} + x^{12} + x^5 + 1\\ n - k &= 16\end{aligned} \tag{39}$$

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

$$Q(x) = q_{15}x^{15} + q_{14}x^{14} + \cdots + q_0 \tag{40}$$

计算 $Q(x)g(x)$，只考虑低 16 位，即忽略 $x^{16}$ ：

$$\begin{aligned}Q(x)g(x) &= (q_{15}x^{15} + q_{14}x^{14} + \cdots + q_0)(x^{16} + x^{12} + x^5 + 1)\\ &= (q_{15}x^{15} + q_{14}x^{14} + \cdots + q_0)(x^{12} + x^5 + 1)\\ &= (q_{15}x^{15} + q_{14}x^{14} + \cdots + q_0)x^{12} +\\ &\quad (q_{15}x^{15} + q_{14}x^{14} + \cdots + q_0)x^5 +\\ &\quad (q_{15}x^{15} + q_{14}x^{14} + \cdots + q_0)\\ &= q_3x^{15} + q_2x^{14} + q_1x^{13} + q_2x^{12} +\\ &\quad q_{10}x^{15} + q_9x^{14} + \cdots + q_1x^6 + q_0x^5 +\\ &\quad q_{15}x^{15} + q_{14}x^{14} + \cdots + q_0\end{aligned} \tag{41}$$

我们的条件是：

$$(Q(x)g(x))_{LSB16bit} \equiv (0_{15},\cdots,0_0) + CRC(x) = 0xffff \tag{42}$$

则所对应的方程为：

$$
\begin{aligned}
(q_3 + q_{10} + q_{15})x^{15} &= x^{15} \\
(q_3 + q_9 + q_{14})x^{14} &= x^{14} \\
(q_3 + q_8 + q_{13})x^{13} &= x^{13} \\
(q_3 + q_7 + q_{12})x^{12} &= x^{12} \\
(q_6 + q_{11})x^{11} &= x^{11} \\
(q_5 + q_{10})x^{10} &= x^{10} \\
(q_4 + q_9)x^9 &= x^9 \\
(q_3 + q_8)x^8 &= x^8 \\
(q_2 + q_7)x^7 &= x^7 \\
(q_1 + q_6)x^6 &= x^6 \\
(q_0 + q_5)x^5 &= x^5 \\
(q_4)x^4 &= x^4 \\
(q_3)x^3 &= x^3 \\
(q_2)x^2 &= x^2 \\
(q_1)x^1 &= x^1 \\
(q_0)x^0 &= x^0
\end{aligned}
\tag{43}
$$

根据这个方程我们就可以解出 $(q_{15}, q_{14}, \cdots, q_0)$ 是 0x8C1F。

解出的这个 0x8C1F 是商，再带入这个公式，计算模 2 乘法：

$$
M(x)x^{n-k} = Q(x)g(x) + CRC(x) \tag{44}
$$

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 $CRC(x) = M(x)x^{n-k} \, mod \, g(x)$，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 $M(x)$ 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 $M(x)$ 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. 并行处理

在 CRC 的实际应用中，输入给 CRC 算法的数据一般是以字节为单位，因此，探究 CRC 校验的并行处理方法也是很有必要的。

$$\begin{aligned} c_{12}^{i+1} &= c_{11}^{i} + c_{15}^{i} + m \\ c_{5}^{i+1} &= c_{4}^{i} + c_{15}^{i} + m \\ c_{0}^{i+1} &= c_{15}^{i} + m \\ c_{k}^{i+1} &= c_{k-1}^{i},\ k \neq 12, 5, 0 \end{aligned} \tag{45}$$

根据上面这个递推关系可以看出当前 CRC 寄存器的值只与上一时刻的 CRC 寄存器值与输入数据 m 相关，根据计算关系构造一个矩阵 F

$$F = \begin{bmatrix} g_{15} & 1 & 0 & 0 & \cdots & 0 & 0 \\ g_{14} & 0 & 1 & 0 & \cdots & 0 & 0 \\ g_{13} & 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ g_{2} & 0 & 0 & 0 & \cdots & 1 & 0 \\ g_{1} & 0 & 0 & 0 & \cdots & 0 & 1 \\ g_{0} & 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \tag{46}$$

同时我们把 CRC 寄存器写为一个列向量：

$$CRC^{i} = [c_{15}^{i}, c_{14}^{i}, \cdots, c_{1}^{i}, c_{0}^{i}]^{T} \tag{47}$$

那么上面的递推关系就可以用一个矩阵乘法来表示：

$$CRC^{i+1} = F \cdot (CRC^{i} + [m, \cdots, 0]^{T})\ mod\ 2 \tag{48}$$

如果连续输入一个字节的数据，$[m0, m1, m2, \cdots, m7]$，即：

$$\begin{aligned} CRC^{i+8} = &(F^{8} \cdot CRC^{i} + \\ &F^{8} \cdot [m0, \cdots, 0]^{T} + F^{7} \cdot [m1, \cdots, 0]^{T} + \cdots + \\ &F^{2} \cdot [m6, \cdots, 0]^{T} + F \cdot [m7, \cdots, 0]^{T})\ mod\ 2 \end{aligned} \tag{49}$$

计算出这几个 F 的幂次矩阵，再带入初始 CRC 和输入的信息码，就可以得到 CRC 并行计算的公式。

```
% first input is x(8)
clear
crc = sym('crc',[16,1]);
g = sym('g',[16,1]);
x = sym('x',[1,8]);
m = sym(zeros(16,8));
m(1,:) = flip(x);
F = sym(zeros(16,16));
F(1:15,2:16) = eye(15);
F(:,1) = g;
crcnext = F^8*flip(crc)+F^8*m(:,1)+F^7*m(:,2)+F^6*m
    (:,3)+F^5*m(:,4)+F^4*m(:,5)+F^3*m(:,6)+F^2*m(:,7)+F
    *m(:,8);
crcnext = subs(crcnext, g,
    [0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,1]')
```

运行这段程序：

```
crcnext =

            crc8 + crc12 + crc16 + x4 + x8
            crc7 + crc11 + crc15 + x3 + x7
            crc6 + crc10 + crc14 + x2 + x6
 crc5 + crc9 + crc13 + crc16 + x1 + x5 + x8
                         crc4 + crc15 + x7
                         crc3 + crc14 + x6
                         crc2 + crc13 + x5
            crc1 + crc12 + crc16 + x4 + x8
       crc11 + crc15 + crc16 + x3 + x7 + x8
       crc10 + crc14 + crc15 + x2 + x6 + x7
        crc9 + crc13 + crc14 + x1 + x5 + x6
                               crc13 + x5
```

```
crc12 + crc16 + x4 + x8
crc11 + crc15 + x3 + x7
crc10 + crc14 + x2 + x6
 crc9 + crc13 + x1 + x5
```

模 2 加就是异或运算，这里都是位操作，因为 Verilog 更容易描述一个比特的位操作，所以这里我就直接使用 Verilog 来展示 CRC 每一位的计算公式。

```
newcrc[16] = crc[8] ^ crc[12] ^ crc[16] ^ x[4] ^ x[8];
newcrc[15] = crc[7] ^ crc[11] ^ crc[15] ^ x[3] ^ x[7];
newcrc[14] = crc[6] ^ crc[10] ^ crc[14] ^ x[2] ^ x[6];
newcrc[13] = crc[5] ^ crc[9] ^ crc[13] ^ crc[16] ^ x[1] ^ x[5] ^ x[8];
newcrc[12] = crc[4] ^ crc[15] ^ x[7];
newcrc[11] = crc[3] ^ crc[14] ^ x[6];
newcrc[10] = crc[2] ^ crc[13] ^ x[5];
newcrc[9] = crc[1] ^ crc[12] ^ crc[16] ^ x[4] ^ x[8];
newcrc[8] = crc[11] ^ crc[15] ^ crc[16] ^ x[3] ^ x[7] ^ x[8];
newcrc[7] = crc[10] ^ crc[14] ^ crc[15] ^ x[2] ^ x[6] ^ x[7];
newcrc[6] = crc[9] ^ crc[13] ^ crc[14] ^ x[1] ^ x[5] ^ x[6];
newcrc[5] = crc[13] ^ x[5];
newcrc[4] = crc[12] ^ crc[16] ^ x[4] ^ x[8];
newcrc[3] = crc[11] ^ crc[15] ^ x[3] ^ x[7];
newcrc[2] = crc[10] ^ crc[14] ^ x[2] ^ x[6];
newcrc[1] = crc[9] ^ crc[13] ^ x[1] ^ x[5];
```

这里我们定义了 crc 寄存器是从 bit16 到 bit1 的，因为这样从 matlab 生成的公式比较容易转化到 verilog。

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 rtl
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
crc_0000_p = 58e5
crc_ffff_p = b915
```

相比之前的 testbench，这次多了两个并行的输出，可以看到跟我们串行计算的结果是一样的，但串行计算需要用 8 个时钟周期，并行计算只需要一个时钟周期就够了。验证了我们并行实现的结果也是正确的。

## 6. 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