一文讲透CRC校验码

最近工作用到CRC校验，顺便整理本篇文章和大家一起研究。

一、CRC概念

1. 什么是CRC？

CRC（Cyclic Redundancy Checksum）是一种纠错技术，代表循环冗余校验和。

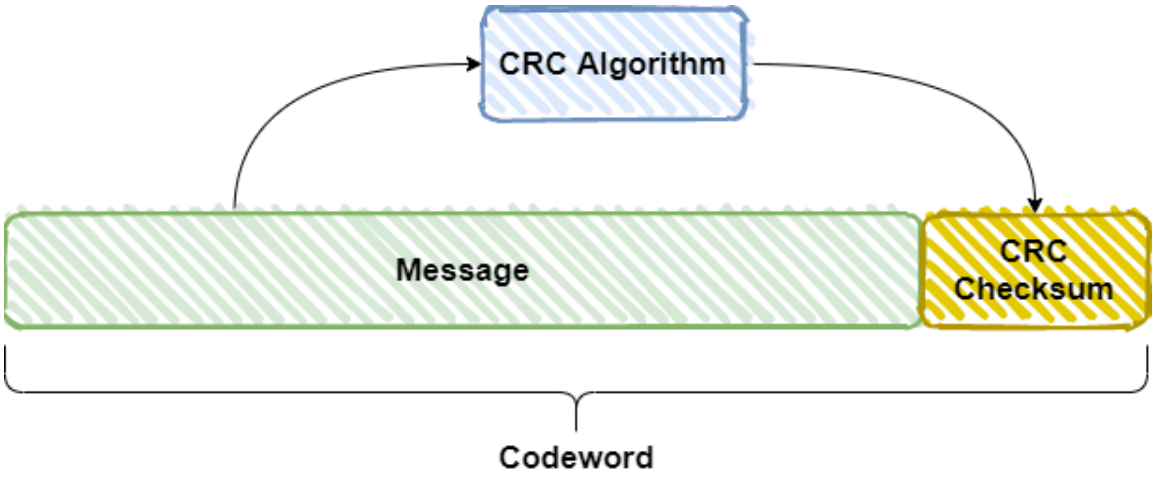
数据通信领域中最常用的一种差错校验码，其信息字段和校验字段长度可以任意指定，但要求通信双方定义的CRC标准一致。主要用来检测或校验数据传输或者保存后可能出现的错误。它的使用方式可以说明如下图所示：

在数据传输过程中，无论传输系统的设计再怎么完美，差错总会存在，这种差错可能会导致在链路上传输的一个或者多个帧被破坏(出现比特差错，0变为1，或者1变为0)，从而接受方接收到错误的数据。

为尽量提高接受方收到数据的正确率，在接收方接收数据之前需要对数据进行差错检测，当且仅当检测的结果为正确时接收方才真正收下数据。检测的方式有多种，常见的有**奇偶校验**、**因特网校验**和**循环冗余校验**等。

2. 使用方法概述

循环冗余校验是一种用于校验通信链路上数字传输准确性的计算方法（通过某种数学运算来建立数据位和校验位的约定关系的 ）。



发送方计算机使用某公式计算出被传送数据所含信息的一个值，并将此值 附在被传送数据后，接收方计算机则对同一数据进行 相同的计算，应该得到相同的结果。

如果这两个 CRC结果不一致，则说明发送中出现了差错，接收方计算机可要求发送方计算机重新发送该数据。

3. 应用广泛

在诸多检错手段中，CRC是最著名的一种。CRC的全称是循环冗余校验，其特点是:检错能力强，开销小，易于用编码器及检测电路实现。从其检错能力来看，它所不能发现的错误的几率仅为0.0047%以下。

从性能上和开销上考虑，均远远优于奇偶校验及算术和校验等方式。

因而，在数据存储和数据通讯领域，CRC无处不在：著名的通讯协议X.25的FCS(帧检错序列)采用的是CRC-CCITT，WinRAR、NERO、ARJ、LHA等压缩工具软件采用的是CRC32，磁盘驱动器的读写采用了CRC16，通用的图像存储格式GIF、TIFF等也都用CRC作为检错手段。

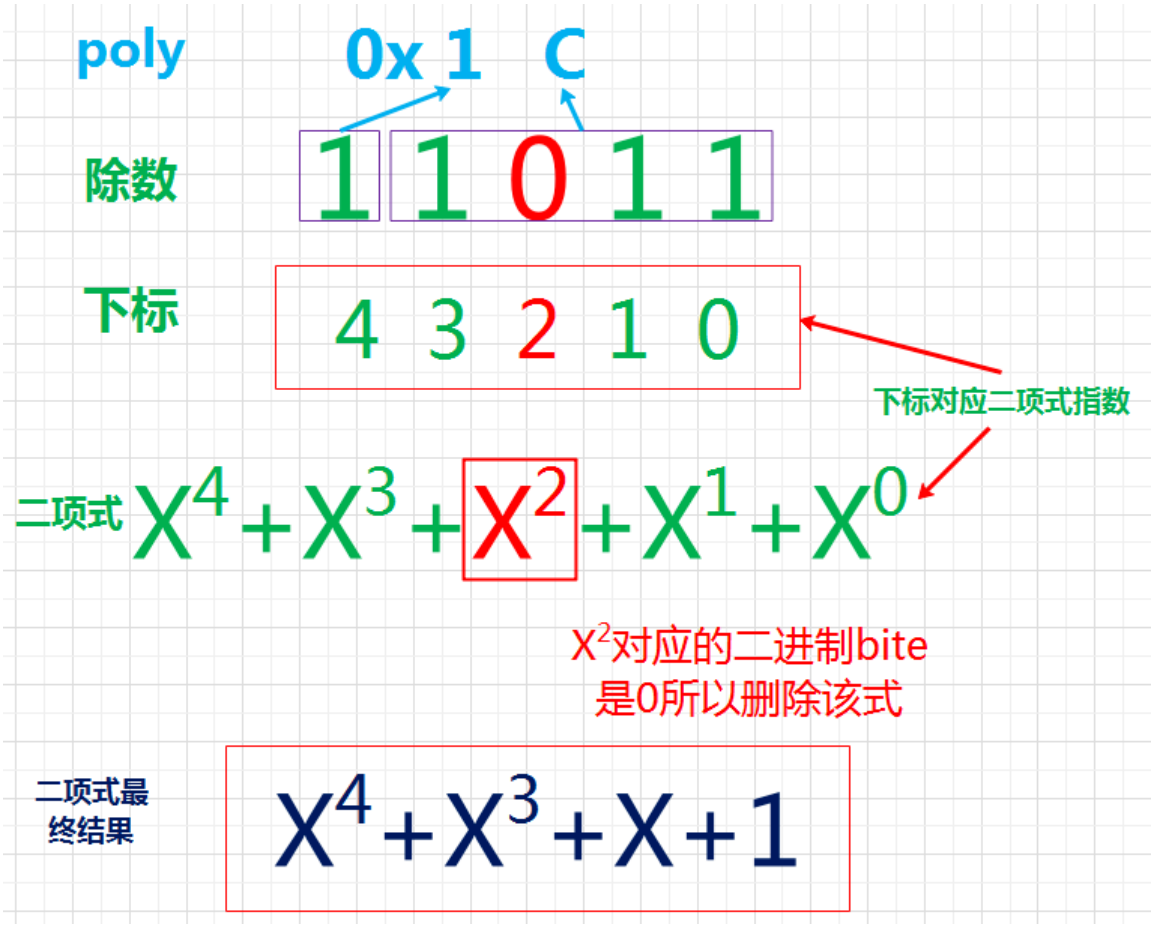
二、CRC名称的定义

这里需要知道几个组成部分或者说计算概念：多项式公式、多项式简记式、数据宽度、初始值、结果异或值、输入值反转、输出值反转、参数模型。

1、多项式公式

对于CRC标准除数，一般使用多项式（或二项式）公式表示，如下图中除数11011（poly值为0x1b）的二项式为$G(X)=X^4+X^3+X+1$，X的指数就代表了该bit位上的数据为1,（最低位为0）。

这里特别注意一下位数问题，除数的位数为二项式最高次幂+1（4+1=5），这个很重要。



2、多项式简记式

通过对CRC的基本了解我们知道，多项式的首尾必定为1，而这个1的位置在下一步计算一定为0，所以就把前面这个1给省略掉了，出现了一个叫简记式的东西，如上例中除数11011的简记式为1011，很多看过CRC高级语言源码的人会知道，对于CRC_16标准下$G(X)=X^{16}+X^{15}+X^2+1$（16#18005）的poly值实际上是8005，这里使用的就是简记式。后面会对这个用法做一个说明。

3、数据宽度

数据宽度指的就是CRC校验码的长度（二进制位数），知道了CRC的运算概念和多项式，就可以理解这个概念了，CRC长度始终要比除数位数少1，与简记式长度是一致的。

以上三个数据就是我们经常能够用到的基本数据

4、初始值与结果异或值

在一些标准中，规定了初始值，则数据在进行上述二项式运算之前，需要先将要计算的数据与初始值的最低字节进行异或，然后再与多项式进行计算。

而在结果异或值不为零的情况下，则需要将计算得到的CRC结果值再与结果异或值进行一次异或计算，得到的最终值才是我们需要的CRC校验码。

这里可以看出，初始值与结果值的位数要求与数据宽度一致。

5、输入值反转与输出值反转

输入值反转的意思是在计算之前先将二项式反转，然后再用得到的新值和数据进行计算。如对于$G(X)=X^{16}+X^{15}+X^2+1$（16#18005），其正向值为1 1000 0000 0000 0101，反转值则为1010 0000 0000 0001 1

输出值反转则是将最终得到的CRC结果反转。

通常，输入值反转后的结果值也会是反转的，所以这两个选项一般是同向的，我们只有在在线CRC计算器中会看到自由选择正反转的情况存在。

三、常见的CRC算法

虽然CRC可以任意定义二项式、数据长度等，但没有一个统一的标准的话，就会让整个计算变得非常的麻烦。但实际上，不同的厂家经常采用不同的标准算法，这里列出了一些国际常用的模型表：

名称	多项式	表示法	应用举例
CRC-8	X^8+X^2+X+1	0X107	

名称	多项式	表示法	应用举例
CRC-12	$X^{12}+X^{11}+X^3+X^2+X+1$	0X180F	telecom systems
CRC-16	$X^{16}+X^{15}+X^2+1$	0X18005	Bisync, Modbus, USB, ANSI X3.28, SIA DC-07, many others; also known as CRC-16 and CRC-16-ANSI
CRC-CCITT	$X^{16}+X^{12}+X^5+1$	0X11021	ISO HDLC, ITU X.25, V.34/V.41/V.42, PPP-FCS
CRC-32	$X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^5+X^4+X^2+X+1$	0x104C11DB7	ZIP, RAR, IEEE 802 LAN/FDDI, IEEE 1394, PPP-FCS
CRC-32C	$X^{32}+X^{28}+X^{27}+X^{26}+X^{25}+X^{23}+X^{22}+X^{20}+X^{19}+X^{18}+X^{14}+X^{13}+X^{11}+X^{10}+X^9+X^8+X^6+1$	0x11EDC6F41	iSCSI, SCTP, G.hn payload, SSE4.2, Btrfs, ext4, Ceph

四、CRC校验算法前置知识

在学习CRC校验算法之前，先复习一下CRC会涉及的主要几个主要的算法。

1. 异或

异或，就是不同为1，相同为0，运算符号是^。

```
0^0 = 0
0^1 = 1
1^1 = 0
1^0 = 1
```

异或运算存在如下几个规律，需要了解。

```
0^x = x 即0 异或任何数等于任何数
1^x = ~x 即1异或任何数等于任何数取反
x^x = 0 即任何数与自己异或，结果为0
a ^ b = b ^ a 交换律
a ^ (b ^ c) = (a ^ b) ^c 结合律
```

2. 模2加法

模2加法相对于普通的算术加法，主要的区别在模2加法，不做进位处理。具体结果如下。0+0 = 0 0+1 = 1 1+1 = 0 1+0 = 1 我们发现模2加法的计算结果，同异或运算结果一模一样。进一步推演，我们会发现，异或运算的5个规律，同样适合于模2加法。这里，就不在一一列举了。

3. 模2减法

模2减法相对于普通的算术减法，主要的区别在模2减法，不做借位处理。具体结果如下。0-0 = 0 0-1 = 1 1-1 = 0 1-0 = 1 我们发现模2减法的计算结果，同模2加法，以及异或的运算结果一模一样。进一步推演，我们会发现，异或运算的5个规律，同样适合于模2减法。这里，就不在一一列举了。

4. 模2除法

模2除法相对于普通的算术除法，主要的区别在模2除法，它既不向上位借位，也不比较除数和被除数的相同位数值的大小，只要以相同位数进行相除即可。

五、CRC原理

CRC原理：在K位信息码（目标发送数据）后再拼接R位校验码，使整个编码长度为N位，因此这种编码也叫（N,K）码。

通俗的说，就是在需要发送的信息后面附加一个数（即校验码），生成一个新的发送数据发送给接收端。这个数据要求能够使生成的新数据被一个特定的数整除。这里的整除需要引入模 2除法的概念。

那么，CRC校验的具体做法就是

（1）选定一个标准除数（K位二进制数据串）

（2）在要发送的数据（m位）后面加上K-1位0，然后将这个新数（M+K-1位）以模2除法的方式除以上面这个标准除数，所得到的余数也就是该数据的CRC校验码（注：余数必须比除数少且只少一位，不够就补0）

（3）将这个校验码附在原m位数据后面，构成新的M+K-1位数据，发送给接收端。

（4）接收端将接收到的数据除以标准除数，如果余数为0则认为数据正确。

注意：CRC校验中有两个关键点：

一是要预先确定一个发送端和接收端都用来作为除数的二进制比特串（或多项式）；

二是把原始帧与上面选定的除进行二进制除法运算，计算出FCS。

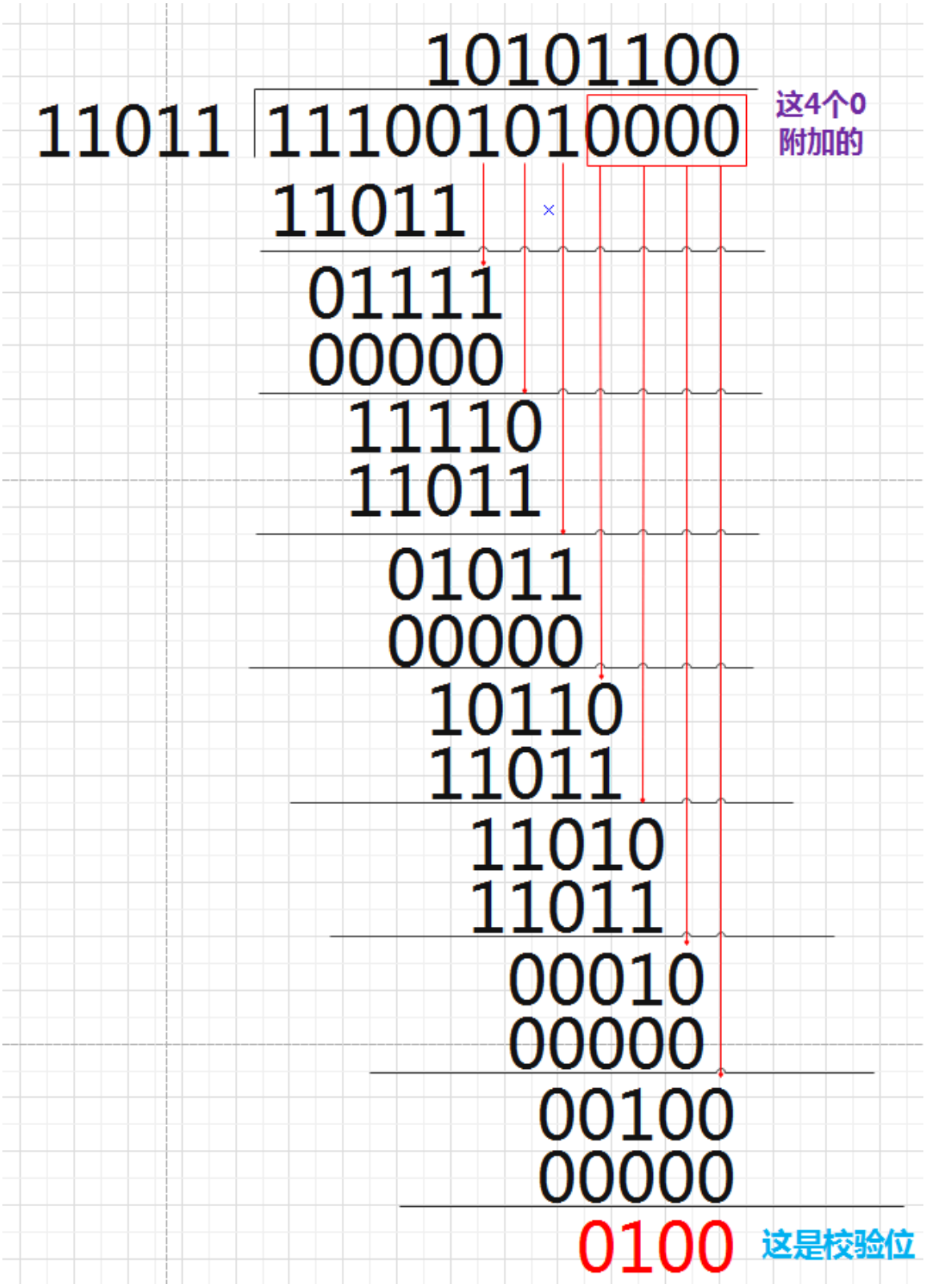
前者可以随机选择，也可按国际上通行的标准选择，但最高位和最低位必须均为“1”

六、循环冗余的计算

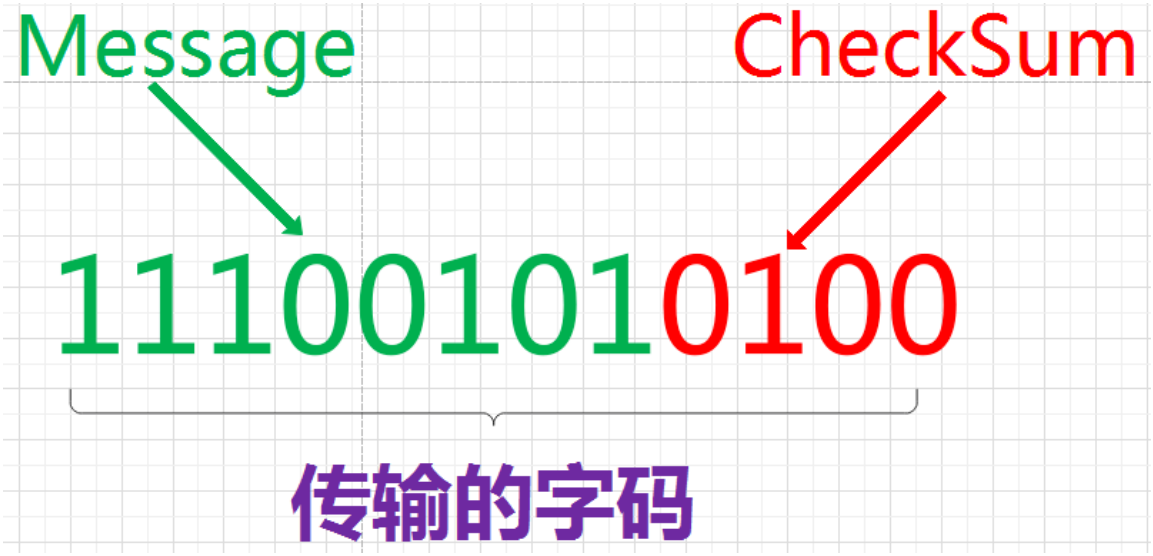
实例：

由于CRC-32、CRC-16、CCITT和CRC-4的编码过程基本一致，只有位数和生成多项式不一样，下面就举例，来说明CRC校验码生成过程。

对于数据1110 0101（16#E5），以指定除数11011求它的CRC校验码，其过程如下：



使用上面计算的校验和和消息数据，可以创建要传输的码字。



有时候，我们需要填充checksum到制定的位置，这就涉及到字节序问题，建议用memcpy()进行拷贝。

七、代码实现

实现算法参考网络相关代码，进行整理并验证，可直接使用。crc.c

```
/*
 *一口Linux
 *2021.6.21
 *version: 1.0.0
*/
#include "crc.h"
#include <stdio.h>

typedef enum {
 REF_4BIT = 4,
 REF_5BIT = 5,
 REF_6BIT = 6,
 REF_7BIT = 7,
 REF_8BIT = 8,
 REF_16BIT = 16,
 REF_32BIT = 32
}REFLECTED_MODE;

uint32_t ReflectedData(uint32_t data, REFLECTED_MODE mode)
{
 data = ((data & 0xffff0000) >> 16) | ((data & 0x0000ffff) << 16);
 data = ((data & 0xff00ff00) >> 8) | ((data & 0x00ff00ff) << 8);
 data = ((data & 0xf0f0f0f0) >> 4) | ((data & 0x0f0f0f0f) << 4);
 data = ((data & 0xcccccccc) >> 2) | ((data & 0x33333333) << 2);
 data = ((data & 0xaaaaaaaa) >> 1) | ((data & 0x55555555) << 1);

 switch (mode)
 {
 case REF_32BIT:
  return data;
 case REF_16BIT:
  return (data >> 16) & 0xffff;
 case REF_8BIT:
  return (data >> 24) & 0xff;
 case REF_7BIT:
  return (data >> 25) & 0x7f;
 case REF_6BIT:
```

```
  return (data >> 26) & 0x7f;
 case REF_5BIT:
  return (data >> 27) & 0x1f;
 case REF_4BIT:
  return (data >> 28) & 0x0f;
 }
 return 0;
}

uint8_t CheckCrc4(uint8_t poly, uint8_t init, bool refIn, bool refOut, u
int8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint8_t i;
 uint8_t crc;

 if (refIn == true)
 {
  crc = init;
  poly = ReflectedData(poly, REF_4BIT);

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x01)
    {
     crc >>= 1;
     crc ^= poly;
    }
    else
    {
     crc >>= 1;
    }
   }
  }

  return crc ^ xorOut;
 }
 else
 {
  crc = init << 4;
  poly <<= 4;
```

```
  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x80)
    {
     crc <<= 1;
     crc ^= poly;
    }
    else
    {
     crc <<= 1;
    }
   }
  }

  return (crc >> 4) ^ xorOut;
 }
}

uint8_t CheckCrc5(uint8_t poly, uint8_t init, bool refIn, bool refOut, u
int8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint8_t i;
 uint8_t crc;

 if (refIn == true)
 {
  crc = init;
  poly = ReflectedData(poly, REF_5BIT);

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x01)
    {
     crc >>= 1;
     crc ^= poly;
    }
    else
    {
     crc >>= 1;
    }
```

```
    }
   }

   return crc ^ xorOut;
  }
  else
  {
   crc = init << 3;
   poly <<= 3;

   while (length--)
   {
    crc ^= *buffer++;
    for (i = 0; i < 8; i++)
    {
     if (crc & 0x80)
     {
      crc <<= 1;
      crc ^= poly;
     }
     else
     {
      crc <<= 1;
     }
    }
   }

   return (crc >> 3) ^ xorOut;
  }
}

uint8_t CheckCrc6(uint8_t poly, uint8_t init, bool refIn, bool refOut, u
int8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint8_t i;
 uint8_t crc;

 if (refIn == true)
 {
  crc = init;
  poly = ReflectedData(poly, REF_6BIT);

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
```

```
   {
    if (crc & 0x01)
    {
     crc >>= 1;
     crc ^= poly;
    }
    else
    {
     crc >>= 1;
    }
   }
  }

  return crc ^ xorOut;
 }
 else
 {
  crc = init << 2;
  poly <<= 2;

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x80)
    {
     crc <<= 1;
     crc ^= poly;
    }
    else
    {
     crc <<= 1;
    }
   }
  }

  return (crc >> 2) ^ xorOut;
 }
}

uint8_t CheckCrc7(uint8_t poly, uint8_t init, bool refIn, bool refOut, u
int8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint8_t i;
```

```
uint8_t crc;

if (refIn == true)
{
 crc = init;
 poly = ReflectedData(poly, REF_7BIT);

 while (length--)
 {
  crc ^= *buffer++;
  for (i = 0; i < 8; i++)
  {
   if (crc & 0x01)
   {
    crc >>= 1;
    crc ^= poly;
   }
   else
   {
    crc >>= 1;
   }
  }
 }

 return crc ^ xorOut;
}
else
{
 crc = init << 1;
 poly <<= 1;

 while (length--)
 {
  crc ^= *buffer++;
  for (i = 0; i < 8; i++)
  {
   if (crc & 0x80)
   {
    crc <<= 1;
    crc ^= poly;
   }
   else
   {
    crc <<= 1;
   }
  }
 }
```

```
  return (crc >> 1) ^ xorOut;
 }
}

uint8_t CheckCrc8(uint8_t poly, uint8_t init, bool refIn, bool refOut, u
int8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint32_t i = 0;
 uint8_t crc = init;

 while (length--)
 {
  if (refIn == true)
  {
   crc ^= ReflectedData(*buffer++, REF_8BIT);
  }
  else
  {
   crc ^= *buffer++;
  }

  for (i = 0; i < 8; i++)
  {
   if (crc & 0x80)
   {
    crc <<= 1;
    crc ^= poly;
   }
   else
   {
    crc <<= 1;
   }
  }
 }

 if (refOut == true)
 {
  crc = ReflectedData(crc, REF_8BIT);
 }

 return crc ^ xorOut;
}

uint16_t CheckCrc16(uint16_t poly, uint16_t init, bool refIn, bool refOu
t, uint16_t xorOut,
```

```
 const uint8_t *buffer, uint32_t length)
{
 uint32_t i = 0;
 uint16_t crc = init;

 while (length--)
 {
  if (refIn == true)
  {
   crc ^= ReflectedData(*buffer++, REF_8BIT) << 8;
  }
  else
  {
   crc ^= (*buffer++) << 8;
  }

  for (i = 0; i < 8; i++)
  {
   if (crc & 0x8000)
   {
    crc <<= 1;
    crc ^= poly;
   }
   else
   {
    crc <<= 1;
   }
  }
 }

 if (refOut == true)
 {
  crc = ReflectedData(crc, REF_16BIT);
 }

 return crc ^ xorOut;
}

uint32_t CheckCrc32(uint32_t poly, uint32_t init, bool refIn, bool refOu
t, uint32_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint32_t i = 0;
 uint32_t crc = init;

 while (length--)
```

```
 {
  if (refIn == true)
  {
   crc ^= ReflectedData(*buffer++, REF_8BIT) << 24;
  }
  else
  {
   crc ^= (*buffer++) << 24;
  }

  for (i = 0; i < 8; i++)
  {
   if (crc & 0x80000000)
   {
    crc <<= 1;
    crc ^= poly;
   }
   else
   {
    crc <<= 1;
   }
  }
 }

 if (refOut == true)
 {
  crc = ReflectedData(crc, REF_32BIT);
 }

 return crc ^ xorOut;
}

uint32_t CrcCheck(CRC_Type crcType, const uint8_t *buffer, uint32_t leng
th)
{
 switch (crcType.width)
 {
 case 4:
  return CheckCrc4(crcType.poly, crcType.init, crcType.refIn, crcType.re
fOut,
   crcType.xorOut, buffer, length);
 case 5:
  return CheckCrc5(crcType.poly, crcType.init, crcType.refIn, crcType.re
fOut,
   crcType.xorOut, buffer, length);
 case 6:
```

```
  return CheckCrc6(crcType.poly, crcType.init, crcType.refIn, crcType.re
fOut,
   crcType.xorOut, buffer, length);
 case 7:
  return CheckCrc7(crcType.poly, crcType.init, crcType.refIn, crcType.re
fOut,
   crcType.xorOut, buffer, length);
 case 8:
  return CheckCrc8(crcType.poly, crcType.init, crcType.refIn, crcType.re
fOut,
   crcType.xorOut, buffer, length);
 case 16:
  return CheckCrc16(crcType.poly, crcType.init, crcType.refIn, crcType.r
efOut,
   crcType.xorOut, buffer, length);
 case 32:
  return CheckCrc32(crcType.poly, crcType.init, crcType.refIn, crcType.r
efOut,
   crcType.xorOut, buffer, length);
 }
 return 0;
}
```

crc.h

```
/*
 *一口Linux
 *2021.6.21
 *version: 1.0.0
*/
#ifndef __CRC_H__
#define __CRC_H__

#include <stdint.h>
#include <stdbool.h>

typedef struct {
 uint8_t width;
 uint32_t poly;
 uint32_t init;
 bool refIn;
 bool refOut;
 uint32_t xorOut;
}CRC_Type;

uint32_t CrcCheck(CRC_Type crcType, const uint8_t *buffer, uint32_t leng
th);

#endif
```

main.c

```
/*
 *—□Linux
 *2021.6.21
 *version: 1.0.0
*/
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "crc.h"

#define LENGTH 8
const uint8_t data[3][LENGTH] = {
 { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 },
 { 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 },
 { 0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0xbf, 0x7f }};

typedef struct {
 CRC_Type crcType;
 uint32_t result[3];
}CRC_Test;

CRC_Test crc4_ITU = { { 4, 0x03, 0x00, true, true, 0x00 }, { 0x0f, 0x0
a, 0x0e } };
CRC_Test crc5_EPC = { { 5, 0x09, 0x09, false, false, 0x00 }, { 0x00, 0x0
c, 0x17 } };
CRC_Test crc5_ITU = { { 5, 0x15, 0x00, true, true, 0x00 }, { 0x16, 0x0
a, 0x17 } };
CRC_Test crc5_USB = { { 5, 0x05, 0x1f, true, true, 0x1f }, { 0x10, 0x0
9, 0x17 } };
CRC_Test crc6_ITU = { { 6, 0x03, 0x00, true, true, 0x00 }, { 0x1d, 0x3
0, 0x00 } };
CRC_Test crc7_MMC = { { 7, 0x09, 0x00, false, false, 0x00 }, { 0x57, 0x3
0, 0x5b } };
CRC_Test crc8 = { { 8, 0x07, 0x00, false, false, 0x00 }, { 0x3e, 0xe1, 0
x36 } };
CRC_Test crc8_ITU = { { 8, 0x07, 0x00, false, false, 0x55 }, { 0x6b, 0xb
4, 0x63 } };
```

```
CRC_Test crc8_ROHC = { { 8, 0x07, 0xff, true, true, 0x00 }, { 0x6b, 0x7
8, 0x93 } };
CRC_Test crc8_MAXIM = { { 8, 0x31, 0x00, true, true, 0x00 }, { 0x83, 0x6
0, 0xa9 } };
CRC_Test crc16_IBM = { { 16, 0x8005, 0x0000, true, true, 0x0000 }, { 0xc
4f0, 0x2337, 0xa776 } };
CRC_Test crc16_MAXIM = { { 16, 0x8005, 0x0000, true, true, 0xffff }, { 0
x3b0f, 0xdcc8, 0x5889 } };
CRC_Test crc16_USB = { { 16, 0x8005, 0xffff, true, true, 0xffff }, { 0x3
04f, 0xd788, 0x53c9 } };
CRC_Test crc16_MODBUS = { { 16, 0x8005, 0xffff, true, true, 0x000
0 }, { 0xcfb0, 0x2877, 0xac36 } };
CRC_Test crc16_CCITT = { { 16, 0x1021, 0x0000, true, true, 0x0000 }, { 0
xeea7, 0xfe7c, 0x7919 } };
CRC_Test crc16_CCITT_FALSE = { { 16, 0x1021, 0xffff, false, false, 0x000
0 }, { 0x4792, 0x13a7, 0xb546 } };
CRC_Test crc16_X25 = { { 16, 0x1021, 0xffff, true, true, 0xffff }, { 0x6
dd5, 0x7d0f, 0xfa6a } };
CRC_Test crc16_XMODEM = { { 16, 0x1021, 0x0000, false, false, 0x000
0 }, { 0x76ac, 0x2299, 0x8478 } };
CRC_Test crc16_DNP = { { 16, 0x3D65, 0x0000, true, true, 0xffff }, { 0x7
bda, 0x0535, 0x08c4 } };
CRC_Test crc32 = { { 32, 0x04c11db7, 0xffffffff, true, true, 0xffffffff
f }, { 0x3fca88c5, 0xe0631a53, 0xa4051a26 } };
CRC_Test crc32_MPEG2 = { { 32, 0x4c11db7, 0xffffffff, false, false, 0x00
000000 }, { 0x14dbbdd8, 0x6509b4b6, 0xcb09d294 } };

void CrcTest(CRC_Test crcTest)
{
 uint32_t i;
 for (i = 0; i < 3; i++)
 {
  printf("%08x\t%08x\r\n", CrcCheck(crcTest.crcType, data[i], LENGTH), c
rcTest.result[i]);
 }
 printf("\r\n");
}

int main(void)
{
```

```
  CrcTest(crc4_ITU);
  CrcTest(crc5_EPC);
  CrcTest(crc5_ITU);
  CrcTest(crc5_USB);
  CrcTest(crc6_ITU);
  CrcTest(crc7_MMC);
  CrcTest(crc8);
  CrcTest(crc8_ITU);
  CrcTest(crc8_ROHC);
  CrcTest(crc8_MAXIM);
  CrcTest(crc16_IBM);
  CrcTest(crc16_MAXIM);
  CrcTest(crc16_USB);
  CrcTest(crc16_MODBUS);
  CrcTest(crc16_CCITT);
  CrcTest(crc16_CCITT_FALSE);
  CrcTest(crc16_X25);
  CrcTest(crc16_XMODEM);
  CrcTest(crc16_DNP);
  CrcTest(crc32);
  CrcTest(crc32_MPEG2);

  return 0;
 }
```

注意

不同的CRC算法，对ooH或FFH数据流的计算结果不一样，部分算法存在校验结果也为ooH或FFH的情况（也就意味着存储空间处于初始化状态时：全o或全1，CRC校验反而是正确的），在应用中需要注意避免。