循环冗余校验（CRC）算法入门引导

原创 2012年08月19日 12:42:34

标签：算法 (http://so.csdn.net/so/search/s.do?q=算法&t=blog) / 嵌入式 (http://so.csdn.net/so/search/s.do?q=嵌入式&t=blog) / 通讯 (http://so.csdn.net/so/search/s.do?q=通讯&t=blog) / byte (http://so.csdn.net/so/search/s.do?q=byte&t=blog) / c (http://so.csdn.net/so/search/s.do?q=c&t=blog)

148090

liyuanbhu (http://blog.c…

+关注

(http://blog.csdn.net/liyuanbhu)

码云

原创	粉丝	喜欢	11 (https://g utm_sour
400	1876	0	

他的最新文章

更多文章 (http://blog.csdn.net/liyuanbhu)

单球面折射的齐明点 (http://blog.csdn.net/liyuanbhu/article/details/78827655)

常用插接件2（DC 电源插头） (http://blog.csdn.net/liyuanbhu/article/details/78631408)

介绍几种电路上常用的塑料插接件（1） (http://blog.csdn.net/liyuanbhu/article/details/78510180)

QtChart 保存到图像文件 (http://blog.csdn.net/liyuanbhu/article/details/78452853)

陶哲轩实分析 3.4 (http://blog.csdn.net/liyuanbhu/article/details/78446914)

相关推荐

查表法的crc校验算法 (http://blog.csdn.net/jieffantfyan/article/details/52971062)

【转】CRC校验算法 (http://blog.csdn.net/youmu543/article/details/8074158)

CRC校验和CRC各种算法 (http://blog.csdn.net/chenlei_525/article/details/51513832)

CRC8算法 (http://blog.csdn.net/zjli321/article/details/52998468)

写给嵌入式程序员的循环冗余校验（CRC）算法入门引导

前言

CRC校验（循环冗余校验）是数据通讯中最常采用的校验方式。在嵌入式软件开发中，经常要用到CRC 算法对各种数据进行校验。因此，掌握基本的CRC算法应是嵌入式程序员的基本技能。可是，我认识的嵌入式程序员中能真正掌握CRC算法的人却很少，平常在项目中见到的CRC的代码多数都是那种效率非常低下的实现方式。

其实，在网上有一篇介绍CRC 算法的非常好的文章，作者是Ross Williams，题目叫：“A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS”。我常将这篇文章推荐给向我询问CRC算法的朋友，但不少朋友向我抱怨原文太长了，而且是英文的。希望我能写篇短点的文章，因此就有了本文。不过，我的水平比不了Ross Williams，我的文章肯定也没Ross Williams的写的好。因此，阅读英文没有障碍的朋友还是去读Ross Williams的原文吧。

本文的读者群设定为软件开发人员，尤其是从事嵌入式软件开发的程序员，而不是专业从事数学或通讯领域研究的学者（我也没有这个水平写的这么高深）。因此，本文的目标是介绍CRC算法的基本原理和实现方式，用到的数学尽量控制在高中生可以理解的深度。

另外，鉴于大多数嵌入式程序员都是半路出家转行过来的，不少人只会C语言。因此，文中的示例代码全部采用C语言来实现。作为一篇入门短文，文中给出的代码更注重于示范性，尽可能的保持易读性。因此，文中的代码并不追求最高效的实现，但对于一般的应用却也足够快速了。

从奇偶校验说起

所谓通讯过程的校验是指在通讯数据后加上一些附加信息，通过这些附加信息来判断接收到的数据是否和发送出的数据相同。比如说RS232串行通讯可以设置奇偶校验位，所谓奇偶校验就是在发送的每一个字节后都加上一位，使得每个字节中1的个数为奇数个或偶数个。比如我们要发送的字节是0x1a，二进制表示为0001 1010。

采用奇校验，则在数据后补上个0，数据变为0001 1010 0，数据中1的个数为奇数个（3个）

采用偶校验，则在数据后补上个1，数据变为0001 1010 1，数据中1的个数为偶数个（4个）

接收方通过计算数据中1个数是否满足奇偶性来确定数据是否有错。

奇偶校验的缺点也很明显，首先，它对错误的检测概率大约只有50%。也就是只有一半的错误它能够检测出来。另外，每传输一个字节都要附加一位校验位，对传输效率的影响很大。因此，在高速数据通讯中很少采用奇偶校验。奇偶校验优点也很明显，它很简单，因此可以用硬件来实现，这样可以减少软件的负担。因此，奇偶校验也被广泛的应用着。
奇偶校验就先介绍到这来，之所以从奇偶校验说起，是因为这种校验方式最简单，而且后面将会知道奇偶校验其实就是CRC 校验的一种(CRC-1)。

累加和校验

另一种常见的校验方式是累加和校验。所谓累加和校验实现方式有很多种，最常用的一种是在一次通讯数据包的最后加入一个字节的校验数据。这个字节内容为前面数据包中全部数据的忽略进位的按字节累加和。比如下面的例子：
我们要传输的信息为： 6、23、4
加上校验和后的数据包：6、23、4、33
这里 33 为前三个字节的校验和。接收方收到全部数据后对前三个数据进行同样的累加计算，如果累加和与最后一个字节相同的话就认为传输的数据没有错误。
累加和校验由于实现起来非常简单，也被广泛的采用。但是这种校验方式的检错能力也比较一般，对于单字节的校验和大概有1/256 的概率将原本是错误的通讯数据误判为正确数据。之所以这里介绍这种校验，是因为CRC校验在传输数据的形式上与累加和校验是相同的，都可以表示为：通讯数据 校验字节（也可能是多个字节）

初识 CRC 算法

CRC 算法的基本思想是将传输的数据当做一个位数很长的数。将这个数除以另一个数。得到的余数作为校验数据附加到原数据后面。还以上面例子中的数据为例：
6、23、4 可以看做一个2进制数： 0000011000010111 00000010
假如被除数选9，二进制表示为：1001
则除法运算可以表示为：

```
                1010, 1101, 0011, 1001
              ________________________________
1001 )0000, 0110, 0001, 0111, 0000, 0010
          100, 1
          -----
            1, 100
            1, 001
            -----
               111, 0
               100, 1
               ------
                10, 11
                10, 01
                ------
                  1011
                  1001
                  ------
                    10, 000
                     1, 001
                     ------
                       1110
                       1001
                      ------
                         101, 0
                         100, 1
                         -----
                             1010
                             1001
                             ----
                             0001
```

可以看到，最后的余数为1。如果我们将这个余数作为校验和的话，传输的数据则是：6、23、4、1
CRC 算法和这个过程有点类似，不过采用的不是上面例子中的通常的这种除法。在CRC算法中，将二进制数据流作为多项式的系数，然后进行的是多项式的乘除法。还是举个例子吧。
比如说我们有两个二进制数，分别为：1101 和1011。
1101 与如下的多项式相联系：$1x^3+1x^2+0x^1+1x^0=x^3+x^2+x^0$
1011与如下的多项式相联系：$1x^3+0x^2+1x^1+1x^0=x^3+x^1+x^0$
两个多项式的乘法：$(x^3+x^2+x^0)(x^3+x^1+x^0)=x^6+x^5+x^4+x^3+x^3+x^3+x^2+x^1+x^0$

得到结果后，合并同类项时采用模2运算。也就是说乘除法采用正常的多项式乘除法，而加减法都采用模2运算。所谓模2运算就是结果除以2后取余数。比如3 mod 2 = 1。因此，上面最终得到的多项式为：
$x^6+x^5+x^4+x^3+x^2+x^1+x^0$，对应的二进制数:111111
加减法采用模2运算后其实就成了一种运算了，就是我们通常所说的异或运算：

0+0=0	0-0=0
0+1=1	1-0=1
1+0=1	0-1=1
1+1=0	1-1=0

上面说了半天多项式，其实就算是不引入多项式乘除法的概念也可以说明这些运算的特殊之处。只不过几乎所有讲解 CRC 算法的文献中都会提到多项式，因此这里也简单的写了一点基本的概念。不过总用这种多项式表示也很罗嗦，下面的讲解中将尽量采用更简洁的写法。
除法运算与上面给出的乘法概念类似，还是遇到加减的地方都用异或运算来代替。下面是一个例子：
要传输的数据为：1101011011
除数设为：10011
在计算前先将原始数据后面填上4个0：11010110110000，之所以要补0，后面再做解释。

```
          1100001010
        ________________
10011 ) 11010110110000
        10011,,.,,....
        -----,,.,,....
         10011,.,,....
         10011,.,,....
         -----,.,,....
              10110...
              10011...
              -----...
                10100.
                10011.
                ------
                 1110 = Remainder
```

从这个例子可以看出，采用了模2的加减法后，不需要考虑借位的问题，所以除法变简单了。最后得到的余数就是CRC 校验字。为了进行CRC运算，也就是这种特殊的除法运算，必须要指定个被除数，在CRC算法中，这个被除数有一个专有名称叫做“生成多项式”。生成多项式的选取是个很有难度的问题，如果选的不好，那么检出错误的概率就会低很多。好在这个问题已经被专家们研究了很长一段时间了，对于我们这些使用者来说，只要把现成的成果拿来用就行了。
最常用的几种生成多项式如下：
CRC8=$X^8+X^5+X^4+X^0$
CRC-CCITT=$X^{16}+X^{12}+X^5+X^0$
CRC16=$X^{16}+X^{15}+X^2+X^0$
CRC12=$X^{12}+X^{11}+X^3+X^2+X^0$
CRC32=$X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^5+X^4+X^2+X^1+X^0$
有一点要特别注意，文献中提到的生成多项式经常会说到多项式的位宽（Width，简记为W），这个位宽不是多项式对应的二进制数的位数，而是位数减1。比如CRC8中用到的位宽为8的生成多项式，其实对应得二进制数有九位：100110001。另外一点，多项式表示和二进制表示都很繁琐，交流起来不方便，因此，文献中多用16进制简写法来表示，因为生成多项式的最高位肯定为1，最高位的位置由位宽可知，故在简记式中，将最高的1统一去掉了，如CRC32的生成多项式简记为04C11DB7实际上表示的是104C11DB7。当然，这样简记除了方便外，在编程计算时也有它的用处。
对于上面的例子，位宽为4（W=4），按照CRC算法的要求，计算前要在原始数据后填上W个0，也就是4个0。
位宽W=1的生成多项式(CRC1)有两种，分别是X^1和X^1+X^0，读者可以自己证明10 对应的就是奇偶校验中的奇校验，而11对应则是偶校验。因此，写到这里我们知道了奇偶校验其实就是CRC校验的一种特例，这也是我要以奇偶校验作为开篇介绍的原因了。

CRC算法的编程实现

说了这么多总算到了核心部分了。从前面的介绍我们知道CRC校验核心就是实现无借位的除法运算。下面还是通过一个例子来说明如何实现CRC校验。
假设我们的生成多项式为：100110001（简记为0x31），也就是CRC-8
则计算步骤如下：
（1） 将CRC寄存器（8-bits，比生成多项式少1bit）赋初值0

（2） 在待传输信息流后面加入8个0
（3） While (数据未处理完)
（4） Begin
（5） If (CRC寄存器首位是1)
（6） reg = reg XOR 0x31
（7） CRC寄存器左移一位，读入一个新的数据于CRC寄存器的0 bit的位置。
（8） End
（9） CRC寄存器就是我们所要求的余数。

63

实际上，真正的CRC 计算通常与上面描述的还有些出入。这是因为这种最基本的CRC除法有个很明显的缺陷，就是数据流的开头添加一些0并不影响最后校验字的结果。这个问题很让人恼火啊，因此真正应用的CRC算法基本都在原始的CRC算法的基础上做了些小的改动。
所谓的改动，也就是增加了两个概念，第一个是“余数初始值”，第二个是“结果异或值”。
所谓的“余数初始值”就是在计算CRC值的开始，给CRC寄存器一个初始值。“结果异或值”是在其余计算完成后将CRC寄存器的值在与这个值进行一下异或操作作为最后的校验值。
常见的三种CRC 标准用到个各个参数如下表。

	CCITT	CRC16	CRC32
校验和位宽W	16	16	32
生成多项式	x16+x12+x5+1	x16+x15+x2+1	x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x1+1
除数（多项式）	0x1021	0x8005	0x04C11DB7
余数初始值	0xFFFF	0x0000	0xFFFFFFFF
结果异或值	0x0000	0x0000	0xFFFFFFFF

加入这些变形后，常见的算法描述形式就成了这个样子了：
（1） 设置CRC寄存器，并给其赋值为“余数初始值”。
（2） 将数据的第一个8-bit字符与CRC寄存器进行异或，并把结果存入CRC寄存器。
（3） CRC寄存器向右移一位，MSB补零，移出并检查LSB。
（4） 如果LSB为0，重复第三步；若LSB为1，CRC寄存器与0x31相异或。
（5） 重复第3与第4步直到8次移位全部完成。此时一个8-bit数据处理完毕。
（6） 重复第2至第5步直到所有数据全部处理完成。
（7） 最终CRC寄存器的内容与“结果异或值”进行或非操作后即为CRC值。

示例性的C代码如下所示，因为效率很低，项目中如对计算时间有要求应该避免采用这样的代码。不过这个代码已经比网上常见的计算代码要好了，因为这个代码有一个crc的参数，可以将上次计算的crc结果传入函数中作为这次计算的初始值，这对大数据块的CRC计算是很有用的，不需要一次将所有数据读入内存，而是读一部分算一次，全读完后就计算完了。这对内存受限系统还是很有用的。

```cpp
#define POLY        0x1021
/**
 * Calculating CRC-16 in 'C'
 * @para addr, start of data
 * @para num, length of data
 * @para crc, incoming CRC
 */
uint16_t crc16(unsigned char *addr, int num, uint16_t crc)
{
    int i;
    for (; num > 0; num--)              /* Step through bytes in memory */
    {
        crc = crc ^ (*addr++ << 8);     /* Fetch byte from memory, XOR into CRC top byte*/
        for (i = 0; i < 8; i++)             /* Prepare to rotate 8 bits */
        {
            if (crc & 0x8000)            /* b15 is set... */
                crc = (crc << 1) ^ POLY;    /* rotate and XOR with polynomic */
            else                          /* b15 is clear... */
                crc <<= 1;                  /* just rotate */
        }                             /* Loop for 8 bits */
        crc &= 0xFFFF;                  /* Ensure CRC remains 16-bit value */
    }                               /* Loop until num=0 */
    return(crc);                    /* Return updated CRC */
}
```

上面的代码是我从http://mdfs.net/Info/Comp/Comms/CRC16.htm (http://mdfs.net/Info/Comp/Comms/CRC16.htm)找到的，不过原始代码有错误，我做了些小的修改。
下面对这个函数给出个例子片段代码：

```cpp
unsigned char data1[] = {'1', '2', '3', '4', '5', '6', '7', '8', '9'};
unsigned char data2[] = {'5', '6', '7', '8', '9'};
unsigned short c1, c2;
c1 = crc16(data1, 9, 0xffff);
c2 = crc16(data1, 4, 0xffff);
c2 = crc16(data2, 5, c2);
printf("%04x\n", c1);
printf("%04x\n", c2);
```

读者可以验算，c1、c2 的结果都为 29b1。上面代码中crc 的初始值之所以为0xffff，是因为CCITT标准要求的除数初始值就是0xffff。
上面的算法对数据流逐位进行计算，效率很低。实际上仔细分析CRC计算的数学性质后我们可以多位多位计算，最常用的是一种按字节查表的快速算法。该算法基于这样一个事实：计算本字节后的CRC码，等于上一字节余式CRC码的低8位左移8位，加上上一字节CRC右移 8位和本字节之和后所求得的CRC码。如果我们把8位二进制序列数的CRC(共256个)全部计算出来，放在一个表里，编码时只要从表中查找对应的值进行处理即可。
按照这个方法，可以有如下的代码（这个代码也不是我写的，是我在Micbael Barr的书“Programming Embedded Systems in C and C++” 中找到的，同样，我做了点小小的改动。）：

```cpp
/*
crc.h
*/

#ifndef CRC_H_INCLUDED
#define CRC_H_INCLUDED

/*
* The CRC parameters. Currently configured for CCITT.
* Simply modify these to switch to another CRC Standard.
*/
/*
#define POLYNOMIAL          0x8005
#define INITIAL_REMAINDER   0x0000
#define FINAL_XOR_VALUE     0x0000
*/
#define POLYNOMIAL          0x1021
#define INITIAL_REMAINDER   0xFFFF
#define FINAL_XOR_VALUE     0x0000

/*
#define POLYNOMIAL          0x1021
#define POLYNOMIAL          0xA001
#define INITIAL_REMAINDER   0xFFFF
#define FINAL_XOR_VALUE     0x0000
*/

/*
* The width of the CRC calculation and result.
* Modify the typedef for an 8 or 32-bit CRC standard.
*/
typedef unsigned short width_t;
#define WIDTH (8 * sizeof(width_t))
#define TOPBIT (1 << (WIDTH - 1))

/**
 * Initialize the CRC lookup table.
 * This table is used by crcCompute() to make CRC computation faster.
 */
void crcInit(void);

/**
 * Compute the CRC checksum of a binary message block.
 * @para message, 用来计算的数据
 * @para nBytes, 数据的长度
 * @note This function expects that crcInit() has been called
 *       first to initialize the CRC lookup table.
 */
width_t crcCompute(unsigned char * message, unsigned int nBytes);
```

```
50.
51. #endif // CRC_H_INCLUDED
```

```
[cpp]
1.  /*
2.   *crc.c
3.   */
4.
5.  #include "crc.h"
6.  /*
7.  * An array containing the pre-computed intermediate result for each
8.  * possible byte of input. This is used to speed up the computation.
9.  */
10. static width_t crcTable[256];
11.
12. /**
13.  * Initialize the CRC lookup table.
14.  * This table is used by crcCompute() to make CRC computation faster.
15.  */
16. void crcInit(void)
17. {
18.     width_t remainder;
19.     width_t dividend;
20.     int bit;
21.     /* Perform binary long division, a bit at a time. */
22.     for(dividend = 0; dividend < 256; dividend++)
23.     {
24.         /* Initialize the remainder.  */
25.         remainder = dividend << (WIDTH - 8);
26.         /* Shift and XOR with the polynomial.   */
27.         for(bit = 0; bit < 8; bit++)
28.         {
29.             /* Try to divide the current data bit.  */
30.             if(remainder & TOPBIT)
31.             {
32.                 remainder = (remainder << 1) ^ POLYNOMIAL;
33.             }
34.             else
35.             {
36.                 remainder = remainder << 1;
37.             }
38.         }
39.         /* Save the result in the table. */
40.         crcTable[dividend] = remainder;
41.     }
42. } /* crcInit() */
43.
44. /**
45.  * Compute the CRC checksum of a binary message block.
46.  * @para message, 用来计算的数据
47.  * @para nBytes, 数据的长度
48.  * @note This function expects that crcInit() has been called
49.  *       first to initialize the CRC lookup table.
50.  */
51. width_t crcCompute(unsigned char * message, unsigned int nBytes)
52. {
53.     unsigned int offset;
54.     unsigned char byte;
55.     width_t remainder = INITIAL_REMAINDER;
56.     /* Divide the message by the polynomial, a byte at a time. */
57.     for( offset = 0; offset < nBytes; offset++)
58.     {
59.         byte = (remainder >> (WIDTH - 8)) ^ message[offset];
60.         remainder = crcTable[byte] ^ (remainder << 8);
61.     }
62.     /* The final remainder is the CRC result. */
63.     return (remainder ^ FINAL_XOR_VALUE);
64. } /* crcCompute() */
```

上面代码中crcInit() 函数用来计算crcTable，因此在调用 crcCompute 前必须先调用 crcInit()。不过，对于嵌入式系统，RAM是很紧张的，最好将 crcTable 提前算好，作为常量数据存到程序存储区而不占用RAM空间。CRC 计算实际上还有很多内容可以介绍，不过对于一般的程序员来说，知道这些也就差不多了。余下的部分以后有时间了我再写篇文章来介绍吧。

最后，给出个 C++ 代码，实现了 CRC8、CRC16 和 CRC32 的计算。收集了常见的各种 CRC 系数。 代码可以从这里下载：https://code.csdn.net/liyuanbhu/crc_compute/tree/master

[cpp]

```cpp
#ifndef CRCCOMPUTE_H
#define CRCCOMPUTE_H

#include <stdint.h>

template <typename TYPE> class CRC
{
public:
    CRC();
    CRC(TYPE polynomial, TYPE init_remainder, TYPE final_xor_value);
    void build(TYPE polynomial, TYPE init_remainder, TYPE final_xor_value);
    /**
     * Compute the CRC checksum of a binary message block.
     * @para message, 用来计算的数据
     * @para nBytes, 数据的长度
     */
    TYPE crcCompute(char * message, unsigned int nBytes);
    TYPE crcCompute(char * message, unsigned int nBytes, bool reinit);
protected:
    TYPE m_polynomial;
    TYPE m_initial_remainder;
    TYPE m_final_xor_value;
    TYPE m_remainder;
    TYPE crcTable[256];
    int m_width;
    int m_topbit;
    /**
     * Initialize the CRC lookup table.
     * This table is used by crcCompute() to make CRC computation faster.
     */
    void crcInit(void);
};

template <typename TYPE>
CRC<TYPE>::CRC()
{
    m_width = 8 * sizeof(TYPE);
    m_topbit = 1 << (m_width - 1);
}

template <typename TYPE>
CRC<TYPE>::CRC(TYPE polynomial, TYPE init_remainder, TYPE final_xor_value)
{
    m_width = 8 * sizeof(TYPE);
    m_topbit = 1 << (m_width - 1);
    m_polynomial = polynomial;
    m_initial_remainder = init_remainder;
    m_final_xor_value = final_xor_value;

    crcInit();
}

template <typename TYPE>
void CRC<TYPE>::build(TYPE polynomial, TYPE init_remainder, TYPE final_xor_value)
{
    m_polynomial = polynomial;
    m_initial_remainder = init_remainder;
    m_final_xor_value = final_xor_value;

    crcInit();
}

template <typename TYPE>
TYPE CRC<TYPE>::crcCompute(char * message, unsigned int nBytes)
{
    unsigned int offset;
    unsigned char byte;
    TYPE remainder = m_initial_remainder;
    /* Divide the message by the polynomial, a byte at a time. */
    for( offset = 0; offset < nBytes; offset++)
    {
        byte = (remainder >> (m_width - 8)) ^ message[offset];
        remainder = crcTable[byte] ^ (remainder << 8);
    }
    /* The final remainder is the CRC result. */
```

```
    return (remainder ^ m_final_xor_value);
}

template <typename TYPE>
TYPE CRC<TYPE>::crcCompute(char * message, unsigned int nBytes, bool reinit)
{
    unsigned int offset;
    unsigned char byte;
    if(reinit)
    {
        m_remainder = m_initial_remainder;
    }
    /* Divide the message by the polynomial, a byte at a time. */
    for( offset = 0; offset < nBytes; offset++)
    {
        byte = (m_remainder >> (m_width - 8)) ^ message[offset];
        m_remainder = crcTable[byte] ^ (m_remainder << 8);
    }
    /* The final remainder is the CRC result. */
    return (m_remainder ^ m_final_xor_value);
}

class CRC8 : public CRC<uint8_t>
{
public:
    enum CRC8_TYPE {eCRC8, eAUTOSAR, eCDMA2000, eDARC, eDVB_S2, eEBU, eAES, eGSM_A, eGSM_B, eI_CODE,
                    eITU, eLTE, eMAXIM, eOPENSAFETY, eROHC, eSAE_J1850, eWCDMA};
    CRC8(CRC8_TYPE type);
    CRC8(uint8_t polynomial, uint8_t init_remainder, uint8_t final_xor_value)
        :CRC<uint8_t>(polynomial, init_remainder, final_xor_value){}
};

class CRC16 : public CRC<uint16_t>
{
public:
    enum CRC16_TYPE {eCCITT, eKERMIT, eCCITT_FALSE, eIBM, eARC, eLHA, eSPI_FUJITSU,
                     eBUYPASS, eVERIFONE, eUMTS, eCDMA2000, eCMS, eDDS_110, eDECT_R,
                     eDECT_X, eDNP, eEN_13757, eGENIBUS, eEPC, eDARC, eI_CODE, eGSM,
                     eLJ1200, eMAXIM, eMCRF4XX, eOPENSAFETY_A, eOPENSAFETY_B, ePROFIBUS,
                     eIEC_61158_2, eRIELLO, eT10_DIF, eTELEDISK, eTMS37157, eUSB,
                     eCRC_A, eMODBUS, eX_25, eCRC_B, eISO_HDLC, eIBM_SDLC, eXMODEM,
                     eZMODEM, eACORN, eLTE};
    CRC16(CRC16_TYPE type);
    CRC16(uint16_t polynomial, uint16_t init_remainder, uint16_t final_xor_value)
        :CRC<uint16_t>(polynomial, init_remainder, final_xor_value){}
};

class CRC32 : public CRC<uint32_t>
{
public:
    enum CRC32_TYPE {eADCCP, ePKZIP, eCRC32, eAAL5, eDECT_B, eB_CRC32, eBZIP2, eAUTOSAR,
                     eCRC32C, eCRC32D, eMPEG2, ePOSIX, eCKSUM, eCRC32Q, eJAMCRC, eXFER};
    CRC32(CRC32_TYPE type);
};


#endif // CRCCOMPUTE_H
```

[cpp]

```
#include "crcCompute.h"

template <typename TYPE>
void CRC<TYPE>::crcInit(void)
{
    TYPE remainder;
    TYPE dividend;
    int bit;
    /* Perform binary long division, a bit at a time. */
    for(dividend = 0; dividend < 256; dividend++)
    {
        /* Initialize the remainder.  */
        remainder = dividend << (m_width - 8);
        /* Shift and XOR with the polynomial.   */
        for(bit = 0; bit < 8; bit++)
        {
            /* Try to divide the current data bit.  */
            if(remainder & m_topbit)
```

```
            {
                remainder = (remainder << 1) ^ m_polynomial;
            }
            else
            {
                remainder = remainder << 1;
            }
        }
        /* Save the result in the table. */
        crcTable[dividend] = remainder;
    }
}

CRC8::CRC8(CRC8_TYPE type)
{
    switch (type)
    {
    case eCRC8:
        m_polynomial = 0x07; //http://reveng.sourceforge.net/crc-catalogue/all.htm
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    case eAUTOSAR:
        m_polynomial = 0x2f;
        m_initial_remainder = 0xff;
        m_final_xor_value = 0xff;
        break;
    case eCDMA2000:
        m_polynomial = 0x9b;
        m_initial_remainder = 0xFF;
        m_final_xor_value = 0x00;
        break;
    case eDARC:
        m_polynomial = 0x39;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    case eDVB_S2:
        m_polynomial = 0xd5;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    case eEBU:
    case eAES:
        m_polynomial = 0x1d;
        m_initial_remainder = 0xFF;
        m_final_xor_value = 0x00;
        break;
    case eGSM_A:
        m_polynomial = 0x1d;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    case eGSM_B:
        m_polynomial = 0x49;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0xFF;
        break;
    case eI_CODE:
        m_polynomial = 0x1d;
        m_initial_remainder = 0xFD;
        m_final_xor_value = 0x00;
        break;
    case eITU:
        m_polynomial = 0x07;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x55;
        break;
    case eLTE:
        m_polynomial = 0x9b;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    case eMAXIM:
        m_polynomial = 0x31;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    case eOPENSAFETY:
        m_polynomial = 0x2f;
```

```
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    case eROHC:
        m_polynomial = 0x07;
        m_initial_remainder = 0xff;
        m_final_xor_value = 0x00;
        break;
    case eSAE_J1850:
        m_polynomial = 0x1d;
        m_initial_remainder = 0xff;
        m_final_xor_value = 0xff;
        break;
    case eWCDMA:
        m_polynomial = 0x9b;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    default:
        m_polynomial = 0x07;
        m_initial_remainder = 0x00;
        m_final_xor_value = 0x00;
        break;
    }
    crcInit();

}

CRC16::CRC16(CRC16_TYPE type)
{
    switch (type)
    {
    case eCCITT_FALSE:
    case eMCRF4XX:
        m_polynomial = 0x1021;
        m_initial_remainder = 0xFFFF;
        m_final_xor_value = 0x0000;
        break;
    case eIBM:
    case eARC:
    case eLHA:
    case eBUYPASS:
    case eVERIFONE:
    case eUMTS:
        m_polynomial = 0x8005;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    case eSPI_FUJITSU:
        m_polynomial = 0x1021;
        m_initial_remainder = 0x1d0f;
        m_final_xor_value = 0x0000;
        break;
    case eCCITT:
    case eKERMIT:
    case eXMODEM:
    case eZMODEM:
    case eACORN:
    case eLTE:
        m_polynomial = 0x1021;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    case eCDMA2000:
        m_polynomial = 0xc867;
        m_initial_remainder = 0xffff;
        m_final_xor_value = 0x0000;
        break;
    case eCMS:
    case eMODBUS:
        m_polynomial = 0x8005;
        m_initial_remainder = 0xffff;
        m_final_xor_value = 0x0000;
        break;
    case eDDS_110:
        m_polynomial = 0x8005;
        m_initial_remainder = 0x800d;
        m_final_xor_value = 0x0000;
        break;
    case eDECT_R:
```

```
        m_polynomial = 0x0589;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0001;
        break;
    case eDECT_X:
        m_polynomial = 0x0589;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    case eDNP:
    case eEN_13757:
        m_polynomial = 0x3d65;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0xffff;
        break;
    case eGENIBUS:
    case eEPC:
    case eDARC:
    case eI_CODE:
    case eX_25:
    case eCRC_B:
    case eISO_HDLC:
    case eIBM_SDLC:
        m_polynomial = 0x1021;
        m_initial_remainder = 0xffff;
        m_final_xor_value = 0xffff;
        break;
    case eGSM:
        m_polynomial = 0x1021;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0xffff;
        break;
    case eLJ1200:
        m_polynomial = 0x6f63;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    case eMAXIM:
        m_polynomial = 0x8005;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0xffff;
        break;
    case eOPENSAFETY_A:
        m_polynomial = 0x5935;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    case eOPENSAFETY_B:
        m_polynomial = 0x755b;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    case ePROFIBUS:
    case eIEC_61158_2:
        m_polynomial = 0x1dcf;
        m_initial_remainder = 0xffff;
        m_final_xor_value = 0xffff;
        break;
    case eRIELLO:
        m_polynomial = 0x1021;
        m_initial_remainder = 0xb2aa;
        m_final_xor_value = 0x0000;
        break;
    case eT10_DIF:
        m_polynomial = 0x8bb7;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    case eTELEDISK:
        m_polynomial = 0xa097;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    case eTMS37157:
        m_polynomial = 0x1021;
        m_initial_remainder = 0x89ec;
        m_final_xor_value = 0x0000;
        break;
    case eUSB:
        m_polynomial = 0x8005;
```

```
        m_initial_remainder = 0xffff;
        m_final_xor_value = 0xffff;
        break;
    case eCRC_A:
        m_polynomial = 0x1021;
        m_initial_remainder = 0xc6c6;
        m_final_xor_value = 0x0000;
        break;
    default:
        m_polynomial = 0x8005;
        m_initial_remainder = 0x0000;
        m_final_xor_value = 0x0000;
        break;
    }
    crcInit();
}


CRC32::CRC32(CRC32_TYPE type)
{
    switch (type)
    {
    case eADCCP:
    case ePKZIP:
    case eCRC32:
    case eBZIP2:
    case eAAL5:
    case eDECT_B:
    case eB_CRC32:
        m_polynomial = 0x04c11db7;
        m_initial_remainder = 0xFFFFFFFF;
        m_final_xor_value = 0xFFFFFFFF;
        break;
    case eAUTOSAR:
        m_polynomial = 0xf4acfb13;
        m_initial_remainder = 0xFFFFFFFF;
        m_final_xor_value = 0xFFFFFFFF;
        break;
    case eCRC32C:
        m_polynomial = 0x1edc6f41;
        m_initial_remainder = 0xFFFFFFFF;
        m_final_xor_value = 0xFFFFFFFF;
        break;
    case eCRC32D:
        m_polynomial = 0xa833982b;
        m_initial_remainder = 0xFFFFFFFF;
        m_final_xor_value = 0xFFFFFFFF;
        break;
    case eMPEG2:
    case eJAMCRC:
        m_polynomial = 0x04c11db7;
        m_initial_remainder = 0xFFFFFFFF;
        m_final_xor_value = 0x00000000;
        break;
    case ePOSIX:
    case eCKSUM:
        m_polynomial = 0x04c11db7;
        m_initial_remainder = 0x00000000;
        m_final_xor_value = 0xFFFFFFFF;
        break;
    case eCRC32Q:
        m_polynomial = 0x814141ab;
        m_initial_remainder = 0x00000000;
        m_final_xor_value = 0x00000000;
        break;
    case eXFER:
        m_polynomial = 0x000000af;
        m_initial_remainder = 0x00000000;
        m_final_xor_value = 0x00000000;
        break;
    default:
        m_polynomial = 0x04C11DB7;
        m_initial_remainder = 0xFFFFFFFF;
        m_final_xor_value = 0xFFFFFFFF;
        break;
    }
    crcInit();
}
```

```cpp
#include <iostream>
#include <stdio.h>
#include "crcCompute.h"

using namespace std;

int main(int argc, char *argv[])
{

    CRC16 crc16(CRC16::eCCITT_FALSE);
    char data1[] = {'1', '2', '3', '4', '5', '6', '7', '8', '9'};
    char data2[] = {'5', '6', '7', '8', '9'};
    unsigned short c1, c2;
    c1 = crc16.crcCompute(data1, 9);
    c2 = crc16.crcCompute(data1, 4, true);
    c2 = crc16.crcCompute(data2, 5, false);


    printf("%04x\n", c1);
    printf("%04x\n", c2);

    return 0;
}
```

qq_38101444 (/qq_38101444) 3天前 10:45 40楼

(/qq_38101444) 有收获

回复

dutong0321 (/dutong0321) 2017-11-27 16:20 39楼

(/dutong0321) 6、2021可以看做一个2进制数： 0000011000010111 00000010

错了一位，正确的是 0000 0110 0001 0111 0000 0100

有点没看明白 再看看

回复

helongfu (/helongfu) 2017-09-20 08:27 38楼

(/helongfu) 最后的结果异或值操作呢？

回复

u013185754 (/u013185754) 2017-08-21 16:24 37楼

(/u013185754) 我想问问你的代码中

待测数据是否颠倒 RefIn：True； e)

计算值是否颠倒 RefOut：True

这两个没有体现对吧

回复

liyinhuicc (/liyinhuicc) 2017-07-24 15:23 36楼

(/liyinhuicc) 再点混乱

回复 1条回复

相关文章推荐

查表法的crc校验算法 (http://blog.csdn.net/jieffantfyan/article/details/52971062)

/* * Copy right : * File name : CRC16.c * Author : Roger.luo * version : V1....

jieffantfyan (http://blog.csdn.net/jieffantfyan) 2016年10月30日 10:35 1285

63

【转】CRC校验算法 (http://blog.csdn.net/youmu543/article/details/8074158)

CRC校验算法 CRC校验算法 CRC(Cyclic Redundancy Check)循环冗余校验是常用的数据校验方法，讲CRC算法的文章很多，之所以还要写这篇，是想换一个方...

youmu543 (http://blog.csdn.net/youmu543) 2012年10月15日 18:16 1087

CRC校验和CRC各种算法 (http://blog.csdn.net/chenlei_525/article/details/51513832)

1、简介 CRC即循环冗余校验码（Cyclic Redundancy Check）：是数据通信领域中最常用的一种查错校验码，其特征是信息字段和校验字段的长度可以任意选定。循环冗余检查（CRC）是...

chenlei_525 (http://blog.csdn.net/chenlei_525) 2016年05月27日 11:22 3818

CRC8算法 (http://blog.csdn.net/zjli321/article/details/52998468)

crc8校验算法、原理、查表法、计算法、c代码

zjli321 (http://blog.csdn.net/zjli321) 2016年11月01日 18:31 14360

CRC校验算法及实现 C (http://blog.csdn.net/u010913318/article/details/50523654)

标准CRC生成多项式如下表： 名称 生成多项式 简记式* 标准引用 CRC-4 x4+x+1 3 ...

u010913318 (http://blog.csdn.net/u010913318) 2016年01月15日 14:47 1019

CRC16 校验算法 (http://blog.csdn.net/chen249191508/article/details/52980936)

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...

chen249191508 (http://blog.csdn.net/chen249191508) 2016年10月31日 11:26 2179

CRC校验的快速算法的C语言实现 (http://blog.csdn.net/Sins_Cj/article/details/6723942)

CRC校验的快速算法的C语言实现 摘要：CRC循环冗余校验算法，是一种在数据存储和数据通讯领域中使用十分广泛的编码算法，具有强力的检错和纠错能力，并...

Sins_Cj (http://blog.csdn.net/Sins_Cj) 2011年08月27日 08:11 2817

CRC算法及原理 (http://blog.csdn.net/qinghecool/article/details/52816761)

CRC算法及原理 本文转自：http://www.cnblogs.com/FPGA_DSP/archive/2010/05/08/1730529.html CRC校验码的基本思想是利用线性编...

qinghecool (http://blog.csdn.net/qinghecool) 2016年10月14日 16:15 1809

63

CRC校验算法及C#程序实现 (http://blog.csdn.net/yangtang_newton/article/details/431…

CRC校验可以运用于传输数据过程中的验证，发送端发送有效数据时，先根据有效数据和生成多项式（比如CCITT标准的多项式是X16+X12+X5+1）计算出CRC校验码，把CRC校验码加到有效数据后面一起...

yangtang_newton (http://blog.csdn.net/yangtang_newton) 2009年07月01日 09:31 10123

CRC校验算法及C++程序实现 (http://blog.csdn.net/xinm1001/article/details/52821237)

CRC校验可以运用于传输数据过程中的验证，发送端发送有效数据时，先根据有效数据和生成多项式（比如CCITT标准的多项式是X16+X12+X5+1）计算出CRC校验码，把CRC校验码加到有效数据后面一起...

xinm1001 (http://blog.csdn.net/xinm1001) 2016年10月15日 09:24 3401

完整的16位CRC循环校验算法类（c#语言） (http://blog.csdn.net/deepwishly/article/det…

/// /// CRCTAB 表 /// private static ushort[] crctab = new ushort[256]{ ...

deepwishly (http://blog.csdn.net/deepwishly) 2010年04月23日 13:46 5937

CRC码计算及校验原理的最通俗诠释 (http://blog.csdn.net/lycb_gz/article/details/8201987)

在上一篇发布了我的最新著作《深入理解计算机网络》一书的原始目录（http://blog.csdn.net/lycb_gz/article/details/8199839），得到了许多读者朋友的高度关注...

lycb_gz (http://blog.csdn.net/lycb_gz) 2012年11月20日 08:28 33222

CRC的校验原理及硬件、软件算法实现 (http://blog.csdn.net/u012252959/article/details/…

转自：http://blog.163.com/yucheng_xiao/blog/static/76600192201393092918776/ 一、基本原理 CRC检验原理实际...

u012252959 (http://blog.csdn.net/u012252959) 2016年09月13日 10:58 1571

CRC校验算法的解析，暨对网上的CRC详解的补充 (http://blog.csdn.net/jim7424994/articl…

Calculation of the 16-bit CRC-CCITT for a one-byte message consisting of the letter “A”： ...

jim7424994 (http://blog.csdn.net/jim7424994) 2014年05月07日 10:53 1308

CRC校验 (http://blog.csdn.net/dengdaiforever/article/details/8040844)

在学TCP/IP中，关于Ethernet帧结构中的最后一部分帧校验字段FCS（4B），在编程通信程序时，我们需对数据链路层通信Ethernet帧进行校验，即对帧校验字段FCS进行校验。FCS采用32位...

dengdaiforever (http://blog.csdn.net/dengdaiforever) 2012年10月05日 09:22 4611

CRC校验 (http://blog.csdn.net/woshinia/article/details/22381037)

循环冗余校验（英语：Cyclic redundancy check，通称“CRC”）是一种根据网络数据数据包或计算机文件等数据产生简短固定位数校验码的一种散列函數，主要用来检测或校验数据传输或者保存后...

woshinia (http://blog.csdn.net/woshinia) 2014年03月28日 10:37 3330

java实现CRC校验码 (http://blog.csdn.net/qq355667166/article/details/6255243)

这两天项目中要使用到CRC校验功能，网上大量的例子是针对c、delphi的例子，前期没有做过，理论上也欠缺很多知识，在这里对java如何实现我们想要的crc校验功能做一下自己的总结，以下内容...

qq355667166 (http://blog.csdn.net/qq355667166) 2011年03月17日 08:30 13842

63

循环冗余检验CRC原理 (http://blog.csdn.net/wenqiang1208/article/details/71641414)

为什么引入CRC现实的通信链路都不会是理想的。这就是说，比特在传输的过程中可能会产生差错：1可能会变成0，0可能会变成1，这就叫做比特差错。在一段是时间内，传输错误的比特占所传输比特总数的比率成为误码...

wenqiang1208 (http://blog.csdn.net/wenqiang1208) 2017年05月11日 16:44 1540

【算法集中营】循环冗余校验 (http://blog.csdn.net/LG1259156776/article/details/51303…

CRC的全称为Cyclic Redundancy Check，中文名称为循环冗余校验。它是一类重要的线性分组码，编码和解码方法简单，检错和纠错能力强，在通信领域广泛地用于实现差错控制。实际上，除 数据...

LG1259156776 (http://blog.csdn.net/LG1259156776) 2016年05月03日 13:36 810

循环冗余校验码CRC原理和实例 (http://blog.csdn.net/France_man/article/details/41804…

今天同事问了一个CRC(循环冗余校验码)的问题，好奇心之下学习了一下。 首先说它的原理，百度百科上也有，我就简单说一下，它其实就是采用多项式编码的方法，对于要发送的信息码R，发送方和接收方约定好多项...

France_man (http://blog.csdn.net/France_man) 2014年12月08日 16:38 1285

CRC循环冗余校验码总结 (http://blog.csdn.net/u012993936/article/details/45337069)

一、CRC简介 先在此说明下什么是CRC：循环冗余码校验 英文名称为Cyclical Redundancy Check，简称CRC，它是利用除法及余数的原理来作错误侦测（Error Detectin...

u012993936 (http://blog.csdn.net/u012993936) 2015年04月28日 15:53 4637

差错检测和循环冗余检验crc (http://blog.csdn.net/liuweidagege/article/details/43635501)

差错检测 传输过程中可能会产生比特差错：1 可能会变成 0 而 0 也可能变成 1。 在一段时间内，传输错误的比特占所传输比特总数的比率称为误码率 BER(Bit Error Rate)。 ...

liuweidagege (http://blog.csdn.net/liuweidagege) 2015年02月08日 10:36 1627

CRC8 详解 (http://blog.csdn.net/haifengid/article/details/51753181)

CRC即循环冗余校验码（Cyclic Redundancy Check）：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。 CRC校验可以简单地描述为...

haifengid (http://blog.csdn.net/haifengid) 2016年06月24日 15:33 3876

CRC8 校验函数 (http://blog.csdn.net/bailang326/article/details/5717702)

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...

bailang326 (http://blog.csdn.net/bailang326) 2010年07月07日 10:07 11095

CRC8校验分析 (http://blog.csdn.net/husion01/article/details/17440333)

CRC即循环冗余校验码（Cyclic Redundancy Check）：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。 CRC校验可以简单地...

husion01 (http://blog.csdn.net/husion01) 2013年12月20日 14:12 12445

CRC8校验 (http://blog.csdn.net/doufuxian/article/details/40299267)

CRC按照位直接计算，比较灵活可以修改生成多项式 unsigned char CFrameInput::GetCheckSum(unsigned char *crcData, int crcDa...

doufuxian (http://blog.csdn.net/doufuxian) 2014年10月20日 13:11 1295

CRC8校验分析 (http://blog.csdn.net/tianshi_1988/article/details/50638010)

CRC即循环冗余校验码（Cyclic Redundancy Check）：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。 CRC校验可以简单地描...

tianshi_1988 (http://blog.csdn.net/tianshi_1988) 2016年02月05日 14:38 963

Delphi7高级应用开发随书源码 (http://download.csdn.net/download/ch…

ZIP

2003年04月30日 00:00 676KB 下载

循环冗余校验（CRC）算法 (http://blog.csdn.net/HandsomeHong/article/details/72935…

写给嵌入式程序员的循环冗余校验（CRC）算法入门引导 前言 CRC校验（循环冗余校验）是数据通讯中最常采用的校验方式。在嵌入式软件开发中，经常要用到CRC 算法对各种数据进行校验。因此，掌握基...

HandsomeHong (http://blog.csdn.net/HandsomeHong) 2017年06月08日 21:17 407

循环冗余校验（CRC）算法入门引导 (http://blog.csdn.net/weed_hz/article/details/2513…

写给嵌入式程序员的循环冗余校验（CRC）算法入门引导 前言 CRC校验（循环冗余校验）是数据通讯中最常采用的校验方式。在嵌入式软件开发中，经常要用到CRC 算法对各种数据进行校验。因此，掌握基...

weed_hz (http://blog.csdn.net/weed_hz) 2014年05月06日 15:22 862

几种CRC16算法 (http://blog.csdn.net/qsycn/article/details/5430886)

一. CRC16算法首先在源文件头文件加入表值: // ...

qsycn (http://blog.csdn.net/qsycn) 2010年03月30日 09:46 32832

CRC16校验码生成原理 (http://blog.csdn.net/u010784959/article/details/77505393)

CRC16-Modbus 生成多项式为CRC-16 : X16 + X15 + X2 + 1 对应 0x8005 移位寄存器初始化值为0xFFFF...

u010784959 (http://blog.csdn.net/u010784959) 2017年08月23日 14:21 241

CRC16校验原理及实现 (http://blog.csdn.net/lzy20ls/article/details/75309015)

CRC码由发送端计算，放置于发送信息报文的尾部。接收信息的设备再重新计算接收到信息报文的CRC，比较计算得到的CRC是否与接收到的相符，如果两者不相符，则表明出错。 校验码的计算多项式为(X16 +...

lzy20ls (http://blog.csdn.net/lzy20ls) 2017年07月18日 16:31 422

Modbus CRC16校验算法--查表法（已经过本人测试，工作良好） (http://blog.csdn.net/zgr…

代码如下： uchar auchCRCHi[]= { 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41...

zgrjkflmkyc (http://blog.csdn.net/zgrjkflmkyc) 2014年11月13日 08:54 13861

CRC16算法学习笔记 (http://blog.csdn.net/brother_mao/article/details/51916740)

一.CRC原理 CRC(Cyclic Redundancy Check）即循环冗余校检码，和奇偶校检码一样，是用来检验数据传输的错误的，其原理用到的是多项式除法。 首先一串比特流，可以表示为一个F...

brother_mao (http://blog.csdn.net/brother_mao) 2016年07月15日 12:48 5425

63

CRC16常见几个标准的算法及C语言实现 (http://blog.csdn.net/leumber/article/details/54…

CRC16常见的标准有以下几种，被用在各个规范中，其算法原理基本一致，就是在数据的输入和输出有所差异，下边把这些标准的差异列出，并给出C语言的算法实现。 CRC16_CCITT： 多项式x16+x12...

leumber (http://blog.csdn.net/leumber) 2017年01月10日 10:56 10491

CRC16算法函数 (http://blog.csdn.net/bakw/article/details/3957572)

//Delphi版function CRC16(P:PChar;Count:Cardinal):WORD; var I,CRC:WORD; CH:Byte; begin ...

bakw (http://blog.csdn.net/bakw) 2009年03月04日 20:10 3790

c++、java CRC16算法 (http://blog.csdn.net/txk15619567977/article/details/21710243)

c++代码 int get_crc16 (unsigned char *bufData, unsigned int buflen, unsigned char *pcrc) { int ret =...

txk15619567977 (http://blog.csdn.net/txk15619567977) 2014年03月21日 13:12 6565

【算法集中营】CRC16 三种算法及c实现 (http://blog.csdn.net/LG1259156776/article/det…

标准CRC生成多项式如下表： 名称 生成多项式 简记式* 标准引用 CRC-4 x4+x+1 3 ...

LG1259156776 (http://blog.csdn.net/LG1259156776) 2016年04月18日 09:21 1382

ZIP

Delphi7高级应用开发随书源码 (http://download.csdn.net/download/ch…

2003年04月30日 00:00 676KB 下载