CheckSum 算法

介绍

本文介绍 CheckSum 的生成，并讨论如何权衡 CRC 实现的大小和速度。如果你不熟悉 CRC，可以阅读 Ross N. Williams 的"A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS"。该文本对 CRC 提供了非常全面的介绍，他从一个简单的字节总和开始(byte1 + byte2 + byte3 +...)，并逐渐对其进行改进。最终，成为了表驱动的多项式版本，现在已经是非常标准了。

该文本已经有 20 多年的历史了，但是 CRC 从那时起就没有改变过，该文被认为是 CRC 领域的经典。

文本使用 CRC16 多项式(0x1021)和 2 字节的校验和。将其推广到更大的多项式和校验和大小非常简单，附录 A 中提供了 CRC32()的示例代码。

基础版本

在最基本的形式中，C 语言中的 CRC 实现可能看起来像这样：

```
typedef unsigned char * ptr;
typedef unsigned char  uint8_t;
typedef unsigned short  uint16_t;
typedef unsigned long  uint32_t;

uint16_t
crc_impl(uint16_t sum, ptr p, uint32_t len)
{
    while (len--)
  {
      uint8_t byte = *p++;

      for (int i = 0; i < 8; ++i)
      {
          uint16_t osum = sum;

          sum <<= 1;

          if (byte & 0x80)
              sum |= 1 ;

          if (osum & 0x8000)
              sum ^= 0x1021; // the polynomial

          byte <<= 1;
      }
```

```
27   }
28   return sum;
29 }
```

如果要将 crc_impl 返回的校验和与 CRC 工具(如 ielftool)计算出的校验和进行比较，则需要在处理完字节序列之后再向其中输入 2 个零。为什么会这样呢？Williams 在上面的文本中有解释。简而言之，答案正等待揭晓。

```
1 static const uint8_t zeroes[] = { 0, 0 };
2
3 uint16_t
4 crc(uint16_t sum, ptr p, uint32_t len)
5 {
6   return crc_impl(crc_impl(sum, p, len), (ptr)zeroes, 2);
7 }
```

在 Cortex-M3 上，以上代码编译为 72 字节(high balanced 优化)，每个校验和字节大约需要 125 个周期。

关于周期计数的说明

本文中所有的周期计数均是从模拟器中获得的。模拟器周期的精度不高，某些指令的时序在实际的处理器上可能会有所不同，实际的计数周期可能根据情况而更高或更低。

检查 ielftool 生成的校验和

- 设置链接器以填充并生成校验和。
- 使用填充字符串 0xFF(填充字符串的确切选择无关紧要，但是 0xFF 是最常见的填充值)。从 0x0 填充到 0x1FFF。同样，确切的值并不重要，但通常应填充并校验整个 ROM。
- 生成一个 2 字节的 CRC-16 校验和，初始值 0x0;
- 无补码，MSB 优先(无镜像)，8 位校验和单元大小。

现在是时候编写一个验证校验和的程序了。校验和本身必须从校验和计算过程中排除，这样做的原因是_checksum 的生产者不能使用校验和的字节来生成校验和。

第一种方法是确保校验和本身不计算校验和。从链接器定义的标签_checksum_begin 到校验和之前的最后一个字节，然后从校验和之后的第一个字节到链接器定义的标签_checksum_end 的所有字节进行校验。

```
1 extern uint16_t __checksum;        // linker defined
2 extern int      __checksum_begin;  // linker defined
3 extern int      __checksum_end;    // linker defined
4
5 int main()
6 {
7   // where checksumming starts
8   ptr p0 = (ptr)&__checksum_begin;
9
10   // the address of the checksum
```

```
11    ptr p1 = (ptr)&__checksum;
12
13    // number of bytes between the start address and the checksum
14    uint32_t len1 = (p1-p0),
15
16    // number of bytes after the checksum
17    len2 = ((ptr)&__checksum_end-p1)-1;
18
19    // the initial value of the checksum
20    uint16_t sum = 0;
21
22    // if they exist, checksum the bytes between the checksum start
23    // and the checksum
24
25    if (len1)
26     sum = crc(sum, p0, len1);
27
28    // ignore the bytes of the checksum itself
29    // if they exist, checksum the bytes between the first byte after
30    // the checksum and the end checksum end
31
32    if (len2)
33     sum = crc(sum, p1+2, len2);
34
35    // compare the computed checksum to the precomputed one
36    if (sum == __checksum)
37     return 1;
38
39    return 0;
40 }
```

该方法不需要修改.icf 文件，但是处理拆分范围很繁琐。另一种方法是将校验和移动到一个已知位置。使用一个自定义的.icf 文件，并对其进行修改，增加下面这一行：

```
place at address mem:0x40 { section .checksum };
```

0x40 可以替换成任何合适的(可用的)地址。通常，校验和应该放在要校验和的范围的开头或末尾的位置。这样做只会产生一个校验和范围，并且在校验和代码中更容易处理。中断向量(0x0-0x3F)通常被 ARM 设备占用，因此将校验和放在内存的末尾可能更好。填充 0x0-0x1FFD，并将校验和放在 0x1FFE 上。

```
1 int main()
2 {
3   // where checksumming starts
4   ptr p0 = (ptr)&__checksum_begin;
```

```
5
6   // number of bytes to checksum
7   uint32_t len1 = ((ptr)&__checksum_end-p0) + 1;
8
9   // the initial value of the checksum
10  uint16_t sum = 0;
11
12   // checksum the range
13   sum = crc(sum, p0, len1);
14
15   // compare the computed checksum to the precomputed one
16   if (sum == __checksum)
17    return 1;
18
19   return 0;
20 }
```

代码比第一种方法的示例更清晰、更小，更易于阅读。缺点就是需要修改.icf 文件。

ROM 字节表版本

现在许多 CRC 实现都是面向字节表驱动，ielftool 和 XLINK 中的实现是表驱动的。

```
1 static const uint16_t t[256] = {
2  0x0000,0x1021,0x2042,0x3063,0x4084,0x50a5,0x60c6,0x70e7,0x8108,0x9129,0xa14a,
3  0xb16b,0xc18c,0xd1ad,0xe1ce,0xf1ef,0x1231,0x0210,0x3273,0x2252,0x52b5,0x4294,
4  0x72f7,0x62d6,0x9339,0x8318,0xb37b,0xa35a,0xd3bd,0xc39c,0xf3ff,0xe3de,0x2462,
5  0x3443,0x0420,0x1401,0x64e6,0x74c7,0x44a4,0x5485,0xa56a,0xb54b,0x8528,0x9509,
6  0xe5ee,0xf5cf,0xc5ac,0xd58d,0x3653,0x2672,0x1611,0x0630,0x76d7,0x66f6,0x5695,
7  0x46b4,0xb75b,0xa77a,0x9719,0x8738,0xf7df,0xe7fe,0xd79d,0xc7bc,0x48c4,0x58e5,
8  0x6886,0x78a7,0x0840,0x1861,0x2802,0x3823,0xc9cc,0xd9ed,0xe98e,0xf9af,0x8948,
9  0x9969,0xa90a,0xb92b,0x5af5,0x4ad4,0x7ab7,0x6a96,0x1a71,0x0a50,0x3a33,0x2a12,
10  0xdbfd,0xcbdc,0xfbbf,0xeb9e,0x9b79,0x8b58,0xbb3b,0xab1a,0x6ca6,0x7c87,0x4ce4,
11  0x5cc5,0x2c22,0x3c03,0x0c60,0x1c41,0xedae,0xfd8f,0xcdec,0xddcd,0xad2a,0xbd0b,
12  0x8d68,0x9d49,0x7e97,0x6eb6,0x5ed5,0x4ef4,0x3e13,0x2e32,0x1e51,0x0e70,0xff9f,
13  0xefbe,0xdfdd,0xcffc,0xbf1b,0xaf3a,0x9f59,0x8f78,0x9188,0x81a9,0xb1ca,0xa1eb,
14  0xd10c,0xc12d,0xf14e,0xe16f,0x1080,0x00a1,0x30c2,0x20e3,0x5004,0x4025,0x7046,
15  0x6067,0x83b9,0x9398,0xa3fb,0xb3da,0xc33d,0xd31c,0xe37f,0xf35e,0x02b1,0x1290,
16  0x22f3,0x32d2,0x4235,0x5214,0x6277,0x7256,0xb5ea,0xa5cb,0x95a8,0x8589,0xf56e,
17  0xe54f,0xd52c,0xc50d,0x34e2,0x24c3,0x14a0,0x0481,0x7466,0x6447,0x5424,0x4405,
18  0xa7db,0xb7fa,0x8799,0x97b8,0xe75f,0xf77e,0xc71d,0xd73c,0x26d3,0x36f2,0x0691,
19  0x16b0,0x6657,0x7676,0x4615,0x5634,0xd94c,0xc96d,0xf90e,0xe92f,0x99c8,0x89e9,
20  0xb98a,0xa9ab,0x5844,0x4865,0x7806,0x6827,0x18c0,0x08e1,0x3882,0x28a3,0xcb7d,
21  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
};

uint16_t crc(uint16_t sum, ptr p, uint32_t len)
{
 while (len--)
  sum = t[(sum >> 8) ^ *p++] ^ (sum << 8);
 return sum;
}
```

Williams 对如何将基本的 CRC 算法转换为表驱动算法进行了相当详尽的讨论。对于那些有兴趣了解 CRC 如何工作的人来说，中间有几个步骤可能是他们感兴趣的。这个表驱动版本计算的 CRC 与上面的代码完全相同(在处理了两个 0 之后)。

在 Cortex-M3 上，表驱动版本编译为 552 字节(high balanced 优化)，CRC 表本身消耗其中的 512 字节。每个字节校验和大约需要 15 个周期，因此表驱动的版本要比基本版本快得多，但代价是占用的 ROM 空间要大得多。

RAM 字节表版本

也可以在程序中生成字节表，这将会占用较少的 ROM 空间(因为构建表的代码比表本身要小得多)。但会占用更多的 RAM 空间(表需要在运行时编写)。

```
void construct_crc_table(uint16_t poly, void * space)
{
  uint16_t * crc_table = (uint16_t *)space;

  for (uint16_t i = 0; i < 256 ; ++i)
  {
    uint16_t r = i << 8;

   for (int j = 0 ; j < 8 ; ++j)
    r = (r & 0x8000) ? (r << 1) ^ poly : (r << 1);

   crc_table[i] = r;
  }
}

uint16_t crc(void * table, uint16_t sum, ptr p, uint32_t len)
{
  uint16_t * t = (uint16_t *)table;

  while (len--)
```

```
    sum = t[(sum >> 8) ^ *p++] ^ (sum << 8);

  return sum;
}
```

在 Cortex-M3 上，基于 RAM 的版本编译为 74 字节(high balanced 优化)。表在 RAM 中构建，需要 512 个连续的 RAM 字节，不一定在所有系统上都可用。可以在堆栈或堆上构建表，也可以使用全局变量，选择取决于具体情况。

当 RAM 不足时，一种改进是在系统启动时，在初始化变量被初始化之前构建表，使用其数据结构(通常是初始化变量)的地址。校验和例程可以执行其工作，然后变量的初始化也将正常进行，从而破坏了表。如果使用这种方法，则必须格外小心，确保在执行初始化之后，表不会被构造或使用。

RAM 版本可以将不同的表用于不同的校验和计算。虽然这种情况不太可能发生，但是在同一个程序中可以使用不同的校验和多项式。

构造该表每个条目大约需 96 个周期。计算每个字节校验和大约需要 14 个周期，速度大致与 ROM 表版本相同。

ROM 半字节表

在这里可以观察到，该表 256 个条目是一个大的问题。如果校验和条目的大小为 2 字节，不可避免地会使用 512 字节的表空间。如果条目的数量可以减少，那么总体空间需求就可以减少，但是表中需要更多的查找操作。

较小单位的自然候选是 4 位，即半字节。面向半字节的表只有 16 个条目，因此如果校验和条目是 2 个字节，则需要 32 个字节。使用半字节来代替一个字节的代价是，你需要查找两个半字节来处理一个字节。

```
static const uint16_t t[] = {
 0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
 0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
};

uint16_t crc_nibble_rom(uint16_t sum, ptr p, uint32_t len)
{
    while (len--) {
   // hi nibble
  sum = t[(sum>>12)^(*p >> 4)]^(sum<<4);

    // lo nibble
    sum = t[(sum>>12)^(*p++ & 0xF)]^(sum<<4);
   }

  return sum;
}
```

RAM 半字节表

相应的，也有面向 RAM 版本的半字节表。

```
1 void construct_nibble_table(uint16_t poly, void * space)
2 {
3    uint16_t * crc_table = (uint16_t *)space;
4
5    for (uint16_t i = 0; i < 16 ; ++i) {
6      uint16_t r = i << 12;
7
8    for (int j = 0 ; j < 4 ; ++j)
9      r = (r & 0x8000) ? (r << 1) ^ poly : (r << 1);
10
11    crc_table[i] = r;
12    }
13 }
14
15 uint16_t crc_nibble_ram(void * table, uint16_t sum, ptr p, uint32_t len)
16 {
17    uint16_t * t = (uint16_t *)table;
18
19    while (len--) {
20    // hi nibble
21    sum = t[(sum>>12)^(*p >> 4)]^(sum<<4);
22
23    // lo nibble
24    sum = t[(sum>>12)^(*p++ & 0xF)]^(sum<<4);
25 }
26
27    return sum;
28 }
```

在 Cortex-M3 上，面向 RAM 版本的半字节表编译为 96 字节(high balanced)。表占用节连续的 32 个字空间。与面向字节的 RAM 版本所需的 256 字节相比，明显减少。构建 RAM 半字节表需要大约 52 个周期/条目，大约是面向字节版本所需的一半，因为每个条目处理的比特数是面向字节版本的一半。大约需要 19 个周期来校验每个字节。即使基于 ROM 半字节表也如此，但它明显落后于面向字节的版本。

ROM 使用(代码大小和常量表)

查表所需要的 RAM 空间(不包括堆栈的使用)。

CpB　Cycles per Byte，每个字节 CRC 校验所需的周期数

CpTE　Cycles per Table Entry，构造表的一个条目所需要的周期数

	ROM use	RAM use	CpB	CpTE	64k-cycles	64k-time@74MHz
basic crc	70	-	125	-	8.19M	110 ms
byte oriented ROM	552	-	15	-	983k	13.3 ms
byte oriented RAM	74	512	14	96	918k*	12.4 ms*
nibble oriented ROM	88	-	19	-	1.25M	16.8 ms
nibble oriented RAM	96	32	19	52	1.25M*	16.8 ms*

*不包括构造表

与 ROM 版本相比，RAM 版本的缺点就是必须构造查找表。

Table type	Entries	Cycles/entry	Time@74MHz
Nibble	16	52	0.01 ms
Byte	256	96	0.33 ms

如果该表可以用于多次校验和计算(假设在两次使用之间没有释放内存)，那么构建该表的成本可以分摊到多次使用中。如果释放内存，或者校验和只执行一次，那么构建表的成本应该添加到校验和成本中。

RAM 版本也有一个优势，你可以拥有多个具有不同多项式的表(创建另一个表或覆盖现有表)。在同一个程序中使用多个多项式的情况很少见，但也不是不可能。

较大的索引类型

可以使用比半字节或字节索引表更大的表。使用更大的表可以减少周期需求，因为查找表的次数更少，但需要更大的资源(16 位索引的 CRC16 表需要 128k 内存)，并且需要处理字节顺序。这种方法在特定的情况下可能会有意义，但是不在本文讨论的范围之内。

选择一个实现

特定项目的“最佳”解决方案将取决于您想要校验和的内容，以及校验和程序运行的频率。普遍的情况情况是在启动时或 flash 更新时对 ROM 进行校验。选择 CRC 实现时，有两点很重要：即实现的大小以及它处理字节的速度。一个实现必须足够小且足够快。一旦实现足够快，则应尽可能小。一旦实现适合可用的空间，则应尽可能的快。

基本的 CRC 算法在大约 110 ms (74 MHz)内校验 64k，它需要 70 字节的 ROM，不占用 RAM(不考虑堆栈的几个字节)。如果这已经满足要求，就没有理由再去考虑其他选择了，因为没有比它更小的实现可以选择。

另一方面，面向半字节的 ROM 版本算法比基础的 CRC 算法只大了 18 个字节，没有其他额外的资源需求，并且基本上快 6 倍。减少 85%的时间是否值得增加 18 个字节的 ROM？在多数情况下是值得的。

同样，面向字节的 ROM 版本比面向半字节的 ROM 版本大 464 字节。它将每个字节的周期需求降低了约 20%。节省的时间是否值得增加 464 字节的 ROM？多数情况下可能并非如此。

如果你想要字节导向的版本，但是不能够给它提供 552 字节 ROM，那么你可以选择只要 74 字节 ROM 占用的 RAM 版本，当然前提是你有可用的 512 个相邻的 RAM 字节。节省的时间是否值得(可能是暂时的)使用 512 字节 RAM？在很多情况下，可能不是。多数系统的 RAM 资源更少。

对很多人来说，面向半字节的 ROM 版本可能最佳的折衷方案。它相当小，而且相当快。

附录 A

正文中的 CRC16 示例的 CRC32 版本：

CRC32 的值为 0x4C11DB7 (实际上为 0x14C11DB7，但最高有效位通常会被丢弃，因为它不适合 32 位值)。

基础版本

```
uint32_t crc_impl(uint32_t sum, ptr p, uint32_t len)
{
    while (len--) {
        uint8_t byte = *p++;

        for (int i = 0; i < 8; ++i) {
            uint32_t osum = sum;

            sum <<= 1;

            if (byte & 0x80)
                sum |= 1 ;

            if (osum & 0x80000000)
                sum ^= 0x4C11DB7; // the polynominal

            byte <<= 1;
        }
    }
    return sum;
}

static const uint8_t zeroes[] = { 0, 0, 0, 0 };

uint32_t
crc(uint32_t sum, ptr p, uint32_t len)
{
    return crc_impl(crc_impl(sum, p, len), (ptr)zeroes, 4);
}
```

ROM 字节表版本

```
static const uint32_t t[256] = {
    0x00000000, 0x04c11db7, 0x09823b6e, 0x0d4326d9, 0x130476dc, 0x17c56b6b,
    0x1a864db2, 0x1e475005, 0x2608edb8, 0x22c9f00f, 0x2f8ad6d6, 0x2b4bcb61,
    0x350c9b64, 0x31cd86d3, 0x3c8ea00a, 0x384fbdbd, 0x4c11db70, 0x48d0c6c7,
    0x4593e01e, 0x4152fda9, 0x5f15adac, 0x5bd4b01b, 0x569796c2, 0x52568b75,
```

```
6     0x6a1936c8, 0x6ed82b7f, 0x639b0da6, 0x675a1011, 0x791d4014, 0x7ddc5da3,
7     0x709f7b7a, 0x745e66cd, 0x9823b6e0, 0x9ce2ab57, 0x91a18d8e, 0x95609039,
8     0x8b27c03c, 0x8fe6dd8b, 0x82a5fb52, 0x8664e6e5, 0xbe2b5b58, 0xbaea46ef,
9     0xb7a96036, 0xb3687d81, 0xad2f2d84, 0xa9ee3033, 0xa4ad16ea, 0xa06c0b5d,
10     0xd4326d90, 0xd0f37027, 0xddb056fe, 0xd9714b49, 0xc7361b4c, 0xc3f706fb,
11     0xceb42022, 0xca753d95, 0xf23a8028, 0xf6fb9d9f, 0xfbb8bb46, 0xff79a6f1,
12     0xe13ef6f4, 0xe5ffeb43, 0xe8bccd9a, 0xec7dd02d, 0x34867077, 0x30476dc0,
13     0x3d044b19, 0x39c556ae, 0x278206ab, 0x23431b1c, 0x2e003dc5, 0x2ac12072,
14     0x128e9dcf, 0x164f8078, 0x1b0ca6a1, 0x1fcdbb16, 0x018aeb13, 0x054bf6a4,
15     0x0808d07d, 0x0cc9cdca, 0x7897ab07, 0x7c56b6b0, 0x71159069, 0x75d48dde,
16     0x6b93dddb, 0x6f52c06c, 0x6211e6b5, 0x66d0fb02, 0x5e9f46bf, 0x5a5e5b08,
17     0x571d7dd1, 0x53dc6066, 0x4d9b3063, 0x495a2dd4, 0x44190b0d, 0x40d816ba,
18     0xaca5c697, 0xa864db20, 0xa527fdf9, 0xa1e6e04e, 0xbfa1b04b, 0xbb60adfc,
19     0xb6238b25, 0xb2e29692, 0x8aad2b2f, 0x8e6c3698, 0x832f1041, 0x87ee0df6,
20     0x99a95df3, 0x9d684044, 0x902b669d, 0x94ea7b2a, 0xe0b41de7, 0xe4750050,
21     0xe9362689, 0xedf73b3e, 0xf3b06b3b, 0xf771768c, 0xfa325055, 0xfef34de2,
22     0xc6bcf05f, 0xc27dede8, 0xcf3ecb31, 0xcbffd686, 0xd5b88683, 0xd1799b34,
23     0xdc3abded, 0xd8fba05a, 0x690ce0ee, 0x6dcdfd59, 0x608edb80, 0x644fc637,
24     0x7a089632, 0x7ec98b85, 0x738aad5c, 0x774bb0eb, 0x4f040d56, 0x4bc510e1,
25     0x46863638, 0x42472b8f, 0x5c007b8a, 0x58c1663d, 0x558240e4, 0x51435d53,
26     0x251d3b9e, 0x21dc2629, 0x2c9f00f0, 0x285e1d47, 0x36194d42, 0x32d850f5,
27     0x3f9b762c, 0x3b5a6b9b, 0x0315d626, 0x07d4cb91, 0x0a97ed48, 0x0e56f0ff,
28     0x1011a0fa, 0x14d0bd4d, 0x19939b94, 0x1d528623, 0xf12f560e, 0xf5ee4bb9,
29     0xf8ad6d60, 0xfc6c70d7, 0xe22b20d2, 0xe6ea3d65, 0xeba91bbc, 0xef68060b,
30     0xd727bbb6, 0xd3e6a601, 0xdea580d8, 0xda649d6f, 0xc423cd6a, 0xc0e2d0dd,
31     0xcda1f604, 0xc960ebb3, 0xbd3e8d7e, 0xb9ff90c9, 0xb4bcb610, 0xb07daba7,
32     0xae3afba2, 0xaafbe615, 0xa7b8c0cc, 0xa379dd7b, 0x9b3660c6, 0x9ff77d71,
33     0x92b45ba8, 0x9675461f, 0x8832161a, 0x8cf30bad, 0x81b02d74, 0x857130c3,
34     0x5d8a9099, 0x594b8d2e, 0x5408abf7, 0x50c9b640, 0x4e8ee645, 0x4a4ffbf2,
35     0x470cdd2b, 0x43cdc09c, 0x7b827d21, 0x7f436096, 0x7200464f, 0x76c15bf8,
36     0x68860bfd, 0x6c47164a, 0x61043093, 0x65c52d24, 0x119b4be9, 0x155a565e,
37     0x18197087, 0x1cd86d30, 0x029f3d35, 0x065e2082, 0x0b1d065b, 0x0fdc1bec,
38     0x3793a651, 0x3352bbe6, 0x3e119d3f, 0x3ad08088, 0x2497d08d, 0x2056cd3a,
39     0x2d15ebe3, 0x29d4f654, 0xc5a92679, 0xc1683bce, 0xcc2b1d17, 0xc8ea00a0,
40     0xd6ad50a5, 0xd26c4d12, 0xdf2f6bcb, 0xdbee767c, 0xe3a1cbc1, 0xe760d676,
41     0xea23f0af, 0xeee2ed18, 0xf0a5bd1d, 0xf464a0aa, 0xf9278673, 0xfde69bc4,
42     0x89b8fd09, 0x8d79e0be, 0x803ac667, 0x84fbdbd0, 0x9abc8bd5, 0x9e7d9662,
43     0x933eb0bb, 0x97ffad0c, 0xafb010b1, 0xab710d06, 0xa6322bdf, 0xa2f33668,
44     0xbcb4666d, 0xb8757bda, 0xb5365d03, 0xb1f740b4,
45 };
46
47 uint32_t crc(uint32_t sum, ptr p, uint32_t len)
48 {
```

```
49      while (len--) {
50          uint8_t i = (sum >> 24) ^ *p++;
51          sum = t[i] ^ (sum << 8);
52      }
53
54      return sum;
55  }
```

RAM 字节表版本

```
1  void construct_crc_table(uint32_t poly, void * space)
2  {
3      uint32_t * crc_table = (uint32_t *)space;
4
5      for (uint32_t i = 0; i < 256 ; ++i) {
6          uint32_t r = i << 24;
7
8          for (int j = 0 ; j < 8 ; ++j)
9              r = (r & 0x80000000) ? (r << 1) ^ poly : (r << 1);
10
11         crc_table[i] = r;
12     }
13 }
14
15 uint32_t crc_ram_table(void * table, uint32_t sum, ptr p, uint32_t len)
16 {
17     uint32_t * t = (uint32_t *)table;
18
19     while (len--) {
20         uint8_t i = (sum >> 24) ^ *p++;
21         sum = t[i] ^ (sum << 8);
22     }
23
24     return sum;
25 }
```

ROM 半字节表版本

```
1  static const uint32_t t[16] = {
2      0x00000000, 0x04c11db7, 0x09823b6e, 0x0d4326d9,
3      0x130476dc, 0x17c56b6b, 0x1a864db2, 0x1e475005,
4      0x2608edb8, 0x22c9f00f, 0x2f8ad6d6, 0x2b4bcb61,
5      0x350c9b64, 0x31cd86d3, 0x3c8ea00a, 0x384fbdbd,
```

```
};

uint32_t crc(uint32_t sum, ptr p, uint32_t len)
{
    while (len--) {
        // hi nibble
        sum = t[(sum>>28)^(*p >> 4)]^(sum<<4);
        // lo nibble
        sum = t[(sum>>28)^(*p++ & 0xF)]^(sum<<4);
    }
    return sum;
}
```

RAM 半字节版本

```
void construct_nibble_table(uint32_t poly, void * space)
{
   uint32_t * crc_table = (uint32_t *)space;
   for (uint32_t i = 0; i < 16 ; ++i) {
      uint32_t r = i << 28;
      for (int j = 0 ; j < 4 ; ++j)
         r = (r & 0x80000000) ? (r << 1) ^ poly : (r << 1);
      crc_table[i] = r;
   }
}

uint32_t crc(void * table, uint32_t sum, ptr p, uint32_t len)
{
    uint32_t * t = (uint32_t *)table;
    while (len--) {
        // hi nibble
        sum = t[(sum>>28)^(*p >> 4)]^(sum<<4);
        // lo nibble
        sum = t[(sum>>28)^(*p++ & 0xF)]^(sum<<4);
    }
    return sum;
}
```

附录 B

反转/镜像代码

有时需要反转字节来匹配 CRC(可能是因为 CRC 是在使用反转/镜像 CRC 硬件生成的，现在必须用软件来验证)。

```
static uint8_t nibb[] = {
    0, // 0  = 0000 = 0000 = 0
    8, // 1  = 0001 = 1000 = 8
    4, // 2  = 0010 = 0100 = 4
    12, // 3  = 0011 = 1100 = C
    2, // 4  = 0100 = 0010 = 2
    10, // 5  = 0101 = 1010 = A
    6, // 6  = 0110 = 0110 = 6
    14, // 7  = 0111 = 1110 = E
     1, // 8  = 1000 = 0001 = 1
     9, // 9  = 1001 = 1001 = 9
     5, // 10 = 1010 = 0101 = 5
     13, // 11 = 1011 = 1101 = D
     3, // 12 = 1100 = 0011 = 3
     11, // 13 = 1101 = 1011 = E
     7, // 14 = 1110 = 0111 = 7
     15  // 15 = 1111 = 1111 = F
};

uint8_t mirror8(uint8_t byte)
{
    return (nibb[byte & 0xF] << 4) | (nibb[byte >> 4]);
}

uint16_t mirror16(uint16_t hw)
{
    return (mirror8(hw & 0xFF) << 8) | mirror8(hw >> 8);
}

void generate_mirror_byte_table(void * space)
{
    uint8_t * mirror = (uint8_t *)space;

    for (uint16_t i = 0 ; i < 256 ; ++i)
        mirror[i] = (nib[i & 0xF] << 4) | nib[i >> 4];
}
```

如果可以为镜像表提供省 256 个字节 ROM，请使用该表。否则使用 mirror8，不需要 RAM，代价是花费大约 6 个周期/镜像字节。

附录 C

初始值代码

一些 CRC 算法使用非零初值。表驱动算法和基本算法使用初值的方式不同，因此无法为所选 CRC 程序的调用提供相同的初值。

```
// this is for indirect / prefixed values
// call this with the initial value and use
// the returned value instead of the initial value
// when calling a table driven (but not the basic!)
// crc routine for the first time
uint16_t fixup_table(uint16_t sum)
{
    uint16_t crc = crc_table[(sum >> 8) & 0xFF];
    crc = crc_table[(crc >> 8) ^ (sum & 0xFF)] ^ (crc << 8);
     return crc;
}

// this is for direct / non prefixed values
// call this with the initial value and use
// the returned value instead of the initial value
// when calling the basic (but not the table driven!)
// crc routine for the first time
uint16_t fixup_basic(uint16_t init)
{
    for (uint16_t i = 0 ; i < 16 ; ++i) {
        uint16_t bit = init & 1;

        if (bit)
            init ^= 0x1021; // poly, change as needed or make it a parameter

        init >>= 1;

        if (bit)
            init |= 0x8000;
    }
    return init;
}
```

附录 D

用于输出 CRC 表的代码。

此代码不适用于使用 CRC 的程序。它的目的是运行生成输出，即表的声明，然后使用 CRC 在程序的源代码中输入。

该程序可以输出字节或半字节表，并且可以使用 16 位或 32 位多项式。添加新的 CRC 算法相当简单。

- 将名称添加到 CrcType 枚举中
- 将枚举添加到设置大小和多项式的开关中

该代码只测试了 16 位和 32 位多项式，以及 16 和 256 条目的表。其他的配置可能也是可以的，但可能需要对代码做一些扩展。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef unsigned char * ptr;
5  typedef unsigned char uint8_t;
6  typedef unsigned short uint16_t;
7  typedef unsigned long uint32_t;
8
9  typedef struct crc_config {
10     uint32_t mCrcMask;
11     uint32_t mValueMask;
12     uint32_t mPoly;
13     uint16_t mElements;
14     uint8_t mSize;
15     uint8_t mInitialShiftCount;
16     uint8_t mBitLimit;
17     uint8_t mIndent;
18 } crc_config;
19
20 typedef enum { kByteTable, kNibbleTable } TableType;
21 typedef enum { kCrc16, kCrc32 } CrcType;
22
23 uint32_t entry(uint32_t i, crc_config const * cc)
24 {
25     uint32_t r = i << cc->mInitialShiftCount;
26     uint32_t m = cc->mCrcMask;
27     uint32_t p = cc->mPoly;
28     uint8_t l = cc->mBitLimit;
29
30     for (int j = 0 ; j < l ; ++j)
31         r = (r & m) ? ((r << 1) ^ p) : (r << 1);
32
33     r &= cc->mValueMask;
34
35     return r;
36 }
37
38 void fill_in_crc_object(TableType table, CrcType crc, crc_config * cc)
```

```
39 {
40 // setup table type related values
41     switch (table) {
42     case kByteTable:
43         cc->mBitLimit = 8;
44         break;
45     case kNibbleTable:
46         cc->mBitLimit = 4;
47         break;
48     default:
49         printf("'%d' is not a valid table type\n", table);
50         abort();
51     }
52
53     cc->mElements = (1u << cc->mBitLimit);
54
55 // setup algorithm type related values
56     uint8_t size;
57     uint32_t poly;
58
59     switch (crc) {
60     case kCrc16:
61         size = 2;
62         poly = 0x1021;
63         break;
64
65     case kCrc32:
66         size = 4;
67         poly = 0x4C11DB7;
68         break;
69
70     /* add additional crc types here */
71     default:
72         printf("'%d' is not a valid polynomial type\n", crc);
73         abort();
74     }
75
76     cc->mSize = size;
77     cc->mPoly = poly;
78     uint8_t bits = size*8;
79
80 // setup derived values
81 // most significant bit set
```

```
82      cc->mCrcMask          = (1u << (bits-1));
83
84  // all bits set
85      cc->mValueMask = (1u << bits)-1;
86
87  // initial shift count
88      cc->mInitialShiftCount = bits - cc->mBitLimit;
89
90  // formatting values
91      cc->mIndent = 2;
92  }
93
94  void print_crc_table(crc_config * cc)
95  {
96      printf("static const uint%d_t t[%d] = {\n",
97            cc->mSize*8, cc->mElements);
98
99      char buf[30];
100
101 // create the formatting string for table entries
102     sprintf(buf, "0x%%0%d%s", cc->mSize*2, "lx, ");
103
104     uint8_t isNew = 1, count = 0;
105
106 // max number of crc values per row
107     uint8_t limit = (80 - cc->mIndent) / (cc->mSize*2 + 4);
108
109     for (uint16_t i = 0 ; i < cc->mElements ; ++i) {
110         // fix indentation for first row entry
111         if (isNew) isNew = !printf("%*s", cc->mIndent, "");
112
113         printf(buf, entry(i, cc));
114
115         // possibly break line
116         if (++count >= limit) count = !(isNew = printf("\n"));
117     }
118
119     printf("\n};\n");
120 }
121
122 main()
123 {
124     crc_config conf, *cc = &conf;
```

```
125     fill_in_crc_object(kNibbleTable, kCrc16, cc);
126     print_crc_table(cc);
127     return 0;
128 }
```