Ross N. Williams

Элементарное руководство по CRC-алгоритмам обнаружения ошибок

Все, что Вы хотели бы знать о CRC-алгоритмах, но боялись спросить, опасаясь, что ошибки Ваших знаний могут быть обнаружены

A painless guide to CRC error detection algorithms[†]

Ross N. Williams

Версия	: 3.0
Дата	: 19 августа 1993 г.
Автор	: Ross N. Williams.
E-mail	: ross@guest.adelaide.edu.au.
FTP	: ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt[†]
Компания	: Rocksoft™ Pty Ltd.
Адрес	: 16 Lerwick Avenue, Hazelwood Park 5066, Australia.
Факс	: +61 8 373-4911 (c/- Internode Systems Pty Ltd).
Телефон	: +61 8 379-9217 (с 10 утра до 10 вечера по "Adelaide Australia time").
Замечание	: "Rocksoft" is a trademark of Rocksoft Pty Ltd, Australia.
Статус	: Copyright (C) Ross Williams, 1993. Однако, разрешается делать и распространять копии этого документа при условии сохранения этого информационного блока и копирайта. Программы на языке C, включенные в текст, являются общим достоянием.
Благодарности	: Выражаю огромную благодарность Jean-loup Gailly (jloup@chorus.fr) и Mark Adler (me@quest.jpl.nasa.gov), которые взяли на себя труд вычитать этот текст и выловить как мелких блох, так и больших жирных жуков (bugs).

[†] Оригинал статьи может быть найден по адресу:
ftp://www.internode.net.au/clients/rocksoft/papers/crc_v3.txt (Здесь и далее прим. перев.)

Оглавление

Предисловие

Данная статья посвящена полному и точному описанию CRC (Cyclic Redundancy Codes — Циклические Избыточные Коды) и реализации табличных алгоритмов их вычисления. Большая часть литературы, касающаяся CRC вообще, и их табличным разновидностям особенно, достаточно сложна и запутанна (по крайней мере, мне так показалось). Статья была написана с целью дать простое и вместе с тем точное описание CRC, вникающее во все тонкости реализации его высокоскоростных вариантов. Предложена параметрическая модель CRC-алгоритма, названная "Rocksoft™ Model CRC Algorithm", которая может быть настроена таким образом, чтобы работать подобно большинству стандартных реализаций алгоритмов расчета CRC, и которая, одновременно, является хорошим примером для демонстрации особенностей некоторых из них. Кроме того, приведен неоптимизированный пример на языке C, а также 2 варианта высокоскоростной табличной реализации, и программа генерации таблицы поиска для расчета CRC.

1. Введение: обнаружение ошибок

Методы обнаружения ошибок предназначены для выявления повреждений сообщений при их передаче по зашумленным каналам (вносящих эти ошибки). Для этого передающее устройство создает некоторое число, называемое контрольной суммой и являющееся функцией сообщения, и добавляет его к этому сообщению. Приемное устройство, используя тот же самый алгоритм, рассчитывает контрольную сумму принятого сообщения и сравнивает ее с переданным значением. Например, если мы для расчета контрольной суммы используем простое сложение байтов сообщения по модулю 256, то может возникнуть примерно следующая ситуация. (Все числа примера десятичные.)

```
Сообщение                       :  6 23  4
Сообщение с контрольной суммой  :  6 23  4 33
Сообщение после передачи        :  6 27  4 33
```

Как Вы видите, второй байт сообщения при передаче оказался измененным с 23 на 27. Приемник может обнаружить ошибку, сравнивая переданную контрольную сумму (33) с рассчитанной им самим: 6 + 27 + 4 = 37. Если при правильной передаче сообщения окажется поврежденной сама контрольная сумма, то такое сообщение будет неверно интерпретировано, как искаженное. Однако, это не самая плохая ситуация. Более опасно, одновременное повреждение сообщения и контрольной суммы таким образом, что все сообщение можно посчитать достоверным. К сожалению, исключить такую ситуацию невозможно, и лучшее, чего можно добиться, это снизить вероятность ее появления, увеличивая количество информации в контрольной сумме (например, расширив ее с одного до 2 байт).

Ошибки иного рода возникают при сложных преобразованиях сообщения для удаления из него избыточной информации. Однако, данная статья посвящена только расчетам CRC, которые относятся к классу алгоритмов, не затрагивающих самого сообщения и лишь добавляющих в его конце контрольную сумму:

<исходное неизмененное сообщение> <контрольная сумма>

2. Требования сложности

В предыдущем разделе мы показали, что повреждение сообщения может быть обнаружено, используя в качестве алгоритма контроля простое суммирование байтов сообщения по модулю 256:

```
Сообщение                       :  6 23  4
Сообщение с контрольной суммой  :  6 23  4 33
Сообщение после передачи        :  6 27  4 33
```

Недостаток этого алгоритма в том, что он слишком прост. Если произойдет несколько искажений, то в 1 случае из 256 мы не сможем их обнаружить. Например:

```
Сообщение                       :  6 23  4
Сообщение с контрольной суммой  :  6 23  4 33
Сообщение после передачи        :  8 20  5 33
```

Для повышения надежности мы могли бы изменить размер регистра с 8-битного на 16-битный (то есть суммировать по модулю 65536 вместо модуля 256), что скорее

всего снизит вероятность ошибки с 1/256 до 1/65536. Хотя это и неплохая идея, однако, она имеет тот недостаток, что применяемая формула расчета не "случайна" в должной степени — каждый суммируемый байт оказывает влияние лишь на один байт суммирующего регистра, при этом ширина самого регистра не имеет никакого значения. Например, во втором случае суммирующий регистр мог бы иметь ширину хоть мегабайт, однако ошибка все равно не была бы обнаружена. Проблема может быть решена лишь заменой простого суммирования более сложной функцией, чтобы каждый новый байт оказывал влияние на весь регистр контрольной суммы.

Таким образом, мы сформулировали 2 требования для формирования надежной контрольной суммы:

Ширина: Размер регистра для вычислений должен обеспечивать изначальную низкую вероятность ошибки (например, 32-байтный регистр обеспечивает вероятность ошибки $1/2^{32}$).

Случайность: Необходим такой алгоритм расчета, когда каждый новый байт может оказать влияние на любые биты регистра.

Обратите внимание, что хотя термин "контрольная сумма" изначально описывал достаточно простые суммирующие алгоритмы, однако, в настоящее время он используется в более широком смысле для обозначения сложных алгоритмов расчета, таких как CRC. Алгоритмы CRC, которые мы и будем рассматривать в дальнейшем, очень хорошо удовлетворяют второму условию и, кроме того, могут быть адаптированы для работы с различной шириной контрольной суммы.

3. Основная идея, заложенная в алгоритме CRC

С чего нам следует начать в наших поисках алгоритмов более сложных, чем простое суммирование? На ум приходят самые экзотичные схемы. Мы могли бы составить таблицы с использованием цифр числа π, или хешировать каждый поступающий байт по всем байтам содержимого регистра. Мы могли бы даже держать в памяти огромный телефонный справочник и использовать каждый поступающий байт, скомбинированный с содержимым регистра, в качестве индекса нового телефонного номера, который затем загружается в регистр. Возможности безграничны.

Однако, нам нет необходимости заходить столь далеко — надо всего лишь сменить арифметическое действие. Если сложение, очевидно, не пригодно для формирования эффективной контрольной суммы, то таким действием вполне может оказаться деление при условии, что делитель имеет ширину регистра контрольной суммы.

Основная идея алгоритма CRC состоит в представлении сообщения виде огромного двоичного числа, делении его на другое фиксированное двоичное число и использовании остатка этого деления в качестве контрольной суммы. Получив сообщение, приемник может выполнить аналогичное действие и сравнить полученный остаток с "контрольной суммой" (переданным остатком).

Приведу пример. Предположим, что сообщение состоит из 2 байт (6, 23), как в предыдущем примере. Их можно рассматривать, как шестнадцатеричное число 0167h, или как двоичное число 0000-0110-0001-0111. Предположим, что ширина регистра контрольной суммы составляет 1 байт, а в качестве делителя используется 1001, тогда сама контрольная сумма будет равна остатку от деления 0000-0110-0001-0111 на 1001. Хотя в данной ситуации деление может быть выполнено с использованием стандартных 32-битных регистров, в общем случае это не верно. Поэтому воспользуемся делением "в столбик", которому нас учили в школе (вспомнили?). Только на этот раз, оно будет выполняться в двоичной системе счисления:

```
           ...0000010101101 = 00AD =  173 = Частное
          ----_---_---_---_
9= 1001 ) 0000011000010111 = 0617 = 1559 = Делимое
Делитель  0000.,,,,,,,,,,,
          ----.,,,,,,,,,,,
           0000,,,,,,,,,,,
           0000,,,,,,,,,,,
           ----,,,,,,,,,,,
            0001,,,,,,,,,,
            0000,,,,,,,,,,
            ----,,,,,,,,,,
             0011.,,,,,,,,,
             0000.,,,,,,,,,
             ----.,,,,,,,,,
              0110..,,,,,,,
              0000..,,,,,,,
              ----..,,,,,,,
               1100..,,,,,,
               1001..,,,,,,
               ====..,,,,,,
                0110.,,,,,,
                0000.,,,,,,
                ----.,,,,,,
                 1100,,,,,,
                 1001,,,,,,
                 ====,,,,,,
                  0111.,,,,
                  0000.,,,,
                  ----.,,,,
                   1110,,,,
                   1001,,,,
                   ====,,,,
                    1011,,,
                    1001,,,
                    ====,,,
                     0101,,
                     0000,,
                     ----
                      1011
                      1001
                      ====
                      0010 = 02 = 2 = Остаток
```

В десятичное виде это будет звучать так: "частное от деления 1559 на 9 равно 173 и 2 в остатке".

Хотя влияние каждого бита исходного сообщения на частное не столь существенно, однако 4-битный остаток во время вычислений может радикально измениться, и чем больше байтов имеется в исходной сообщении (в делимом), тем сильнее меняется каждый раз величина остатка. Вот почему деление оказывается применимым там, где обычное сложение работать отказывается.

В нашем случае передача сообщения вместе с 4-битной контрольной суммой выглядела бы (в шестнадцатеричном виде) следующим образом: 06172, где 0617 — это само сообщение, а 2 — контрольная сумма. Приемник, получив сообщение, мог бы выполнить аналогичное деление и проверить, равен ли остаток переданному значению (2).

4. Полиномиальная арифметика

Хотя арифметическое деление, описанное в предыдущем разделе, очень похоже на схему расчета контрольной суммы, называемой CRC, сам же алгоритм CRC несколько более сложен, и, чтобы понять его, нам необходимо окунуться в теорию целых чисел.

Ключевым словом, которое Вы будете постоянно слышать при работе с CRC, является слово "полином". Все CRC-алгоритмы основаны на полиномиальных вычислениях, и для любого алгоритма CRC можно указать, какой полином он использует. Что это значит?

Вместо представления делителя, делимого (сообщения), частного и остатка в виде положительных целых чисел (как это было сделано в предыдущем разделе), можно представить их в виде полиномов с двоичными коэффициентами или в виде строки бит, каждый из которых является коэффициентом полинома. Например, десятичное число 23 в шестнадцатеричной системе счисления имеет вид 17, а в двоичном — 10111, что совпадает с полиномом:

```
1*x^4 + 0*x^3 + 1*x^2 + 1*x^1 + 1*x^0
```

или, упрощенно:

```
x^4 + x^2 + x^1 + x^0
```

И сообщение, и делитель могут быть представлены в виде полиномов, с которыми как и раньше можно выполнять любые арифметические действия; только теперь нам придется не забывать о иксах. Предположим, что мы хотим перемножить, например, 1101 и 1011. Это можно выполнить, как умножение полиномов:

```
   (x^3 + x^2 + x^0)(x^3 + x^1 + x^0)
= (x^6 + x^4 + x^3
 + x^5 + x^3 + x^2
 + x^3 + x^1 + x^0) = x^6 + x^5 + x^4 + 3*x^3 + x^2 + x^1 + x^0
```

Теперь для получения правильного ответа нам необходимо указать, что X равен 2, и выполнить перенос бита от члена `3*x^3`. В результате получим:

```
x^7 + x^3 + x^2 + x^1 + x^0
```

Все это очень похоже на обычную арифметику, с той лишь разницей, что основание у нас лишь предполагается, а не строго задано. Что же из этого следует?

А то, что **если** мы считаем, "X" нам **не известен**, то мы **не можем** выполнить перенос. Нам не известно, что `3*x^3` — это то же самое, что и `x^4 + x^3`, так как мы не знаем, что `X = 2`. В полиномиальной арифметике связи между коэффициентами не установлены, и, поэтому, коэффициенты при каждом члене полинома становятся строго типизированными — коэффициент при `x^2` имеет иной тип, чем при `x^3`.

Если коэффициенты каждого члена полинома совершенно изолированы друг от друга, то можно работать с любыми видами полиномиальной арифметики, просто меняя правила, по которым коэффициенты работают. Одна из таких схем для нам чрезвычайно интересна, а именно, когда коэффициенты складываются по модулю 2 без переноса — то есть коэффициенты могут иметь значения лишь 0 или 1, перенос не учитывается. Это называется "полиномиальная арифметика по модулю 2". Возвращаясь к предыдущему примеру:

```
   (x^3 + x^2 + x^0)(x^3 + x^1 + x^0)
= (x^6 + x^4 + x^3
 + x^5 + x^3 + x^2
 + x^3 + x^1 + x^0)
=  x^6 + x^5 + x^4 + 3*x^3 + x^2 + x^1 + x^0
```

По правилам обычной арифметики, коэффициент члена `3*x^3` распределяется по другим членам полинома, используя механизм переноса и предполагая, что `X = 2`. В "полиномиальной арифметике по модулю 2" нам не известно, чему равен "X", переносов здесь не существует, а все коэффициенты рассчитываются по модулю 2. В результате получаем:

```
= x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0
```

Как пишет Д. Кнут [Knuth81, стр.400] : "Читатель может подметить сходство между полиномиальной арифметикой и арифметикой высокой точности (multiple-precision arithmetic) (раздел 4.3.1.), когда основание b заменяется на x. Основное различие со-

стоит в том, что коэффициент u_k члена x^k в полиномиальной арифметике считается малой величиной, или величиной, не связанной с ближайшими коэффициентами x^{k-1} (и x^{k+1}), поэтому перенос от одного члена к другому в ней отсутствует. Фактически, полиномиальная арифметика по модулю b во многом аналогична арифметике высокой точности по основанию b за исключением подавления всех переносов."

Таким образом, полиномиальная арифметика по модулю 2 — это фактически двоичная арифметика по модулю 2 без учета переносов. Хотя полиномы имеют удобные математические средства для анализа CRC-алгоритмов и алгоритмов коррекции ошибок, для упрощения дальнейшего обсуждения они будут заменены на непосредственные арифметические действия в системе, с которой они изоморфны, а именно в системе двоичной арифметики без переносов.

5. Двоичная арифметика без учета переносов

Оставив полиномы вне поля нашего внимания, мы можем сфокусировать его на обычной арифметике, так как действия, выполняемые во время вычисления CRC, являются арифметическими операциями без учета переносов. Часто это называется полиномиальной арифметикой, но, как я уже говорил, о ней здесь больше уже не будет сказано ни слова, а вместо этого мы будет обсуждать арифметику CRC. А так как эта арифметическая система является ключевой часть расчетов CRC, нам лучше заняться ею. Итак, начнем.

Сложение двух чисел в CRC-арифметике полностью аналогично обычному арифметическому действия за исключением отсутствия переносов из разряда в разряд. Это означает, что каждая пара битов определяет результат своего разряда вне зависимости от результатов других пар. Например:

```
 10011011
+11001010
 --------
 01010001
```

Для каждой пары битов возможны 4 варианта:

```
0+0=0
0+1=1
1+0=1
1+1=0   (перенос отсутствует)
```

То же самое справедливо и для вычитания:

```
 10011011
-11001010
 --------
 01010001
```

когда имеются также 4 возможные комбинации:

```
0-0=0
0-1=1   (зацикливание)
1-0=1
1-1=0
```

Фактически, как операция сложения, так и операция вычитания в CRC-арифметике идентичны операции "Исключающее ИЛИ" (eXclusive OR — XOR), что позволяет заменить 2 операции первого уровня (сложение и вычитание) одним действием, которое, одновременно, оказывается инверсным самому себе. Очень удобное свойство такой арифметики.

Сгруппировав сложение и вычитание в одно единое действие, CRC-арифметика исключает из поля своего внимания все величины, лежащие за пределами самого старшего своего бита. Хотя совершенно ясно, что значение 1010 больше, чем 10, это оказывается не так, когда говорят, что 1010 должно быть больше, чем 1001.

Чтобы понять почему, попытайтесь получить из 1010 значение 1001, отняв или прибавив к нему одну и ту же величину:

```
1001 = 1010 + 0011
1001 = 1010 - 0011
```

Это сводит на нет всякое представление о порядке.

Определив действие сложения, перейдем к умножению и делению. Умножение, как и в обычной арифметике, считается суммой значений первого сомножителя, сдвинутых в соответствии со значением второго сомножителя.

```
      1101
   x  1011
      ----
      1101
     1101.
    0000..
   1101...
   -------
   1111111  Обратите внимание: при суммировании используется CRC-сложение
```

Деление несколько сложнее, так как нам требуется знать, когда "одно число превращается в другое ". Чтобы сделать это, нам потребуется ввести слабое понятие размерности: число X больше или равно Y, если позиция самого старшего единичного бита числа X более значима (или соответствует) позиции самого старшего единичного бита числа Y. Ниже приведен пример деления (пример заимствован из [Tanenbaum81]).

```
              1100001010
            ---------------
10011 )  11010110110000
         10011,,.,,,,....
         -----,,.,,,,....
          10011,.,,,....
          10011,.,,,....
          -----,.,,....
           00001.,,....
           00000.,,....
           -----.,,....
            00010,,....
            00000,,....
            -----,,....
             00101,....
             00000,....
             -----,....
              01011....
              00000....
              -----....
               10110...
               10011...
               ------...
                01010..
                00000..
                ------..
                 10100.
                 10011.
                 ------.
                  01110
                  00000
                  -----
                   1110 = Остаток
```

Это так и есть. Однако, прежде, чем идти дальше, стоит еще немного задержаться на этих действиях.

Мы уже обратили внимание, что действия сложения и вычитания в нашей арифметике — это одно и то же. В таком случае, `A+0=A` и `A-0=A`. Это очевидное свойство нам очень пригодится в дальнейшем.

Чтобы уметь обращаться с CRC-умножением и делением, необходимо хорошо почувствовать их сущность.

Если число A получено умножением числа B, то в CRC-арифметике это означает, что существует возможность сконструировать число A из нуля, применяя операцию XOR к число B, сдвинутому на различное количество позиций. Например, если A равно 0111010110, а B — 11, то мы может сконструировать A из B следующим способом:

```
  0111010110
= .......11.
+ ....11....
+ ...11.....
  .11.......
```

Однако, если бы A было бы равно 0111010111, то нам бы не удалось составить его с помощью различных сдвигов числа 11 (почему? — читайте дальше!), поэтому говорят, что в CRC-арифметике оно не делится на B.

Таким образом, мы видим, что CRC-арифметика сводится главным образом к операции "Исключающее ИЛИ" некоторого значения при различных величинах сдвига.

6. Полностью рабочий пример

Определив все правила CRC-арифметики, мы можем теперь охарактеризовать вычисление CRC как простое деление, чем оно фактически и является. В данном разделе приводятся дополнительные детали и пример.

Чтобы выполнить вычисление CRC, нам необходимо выбрать делитель. Говоря математическим языком, делитель называется генераторным полиномом (generator polinomial), или просто полиномом, и это ключевое слово любого CRC-алгоритма. Вероятно, было более приятнее назвать делитель как-нибудь иначе, однако тема полиномов настолько глубоко въелась в данную область вычислений, что было бы неразумно избегать ее терминологии. Для простоты мы будет называть CRC-полином просто полиномом.

Вы можете выбрать и использовать в CRC любой полином. Однако, некоторые справляются со своей задачей лучше остальных, поэтому имеет смысл их протестировать, чему и будет посвящен один из следующих разделов.

Степень полинома W (Width — ширина) (позиция самого старшего единичного бита) чрезвычайно важна, так как от нее зависят все остальные расчеты. Обычно выбирается степень 16 ил 32, так как это облегчает реализацию алгоритма на современных компьютерах. Степень полинома — это действительная позиция старшего бита, например, степень полинома 10011 равна 4, а не 5. В качестве примера в дальнейшем мы и будет использовать этот полином (полином 4й степени "10011").

Выбрав полином приступим к расчетам. Это будет простое деление (в терминах CRC-арифметики) сообщения на наш полином. Единственное, что надо будет сделать до начала работы, так это дополнить сообщение W нулевыми битами. Итак, начнем.

Исходное сообщение	: 1101011011
Полином	: 10011
Сообщение, дополненное W битами	: 11010110110000

Теперь просто поделим сообщение на полином, используя правила CRC-арифметики. Ранее мы уже рассматривали это действие.

```
                1100001010 = Частное (оно никого не интересует)
           ---------------
10011 )    11010110110000 = выровненное сообщение (1101011011 + 0000)
=Полином   ,,.,.,,.,,....
           10011,,.,,....
           -----,,.,,....
            10011,.,,....
            10011,.,,....
            -----,.,,....
             00001.,,....
             00000.,,....
             -----.,,....
              00010,,....
              00000,,....
              -----,,....
               00101,....
               00000,....
               -----,....
                01011....
                00000....
                -----....
                 10110...
                 10011...
                 -----...
                  01010..
                  00000..
                  ------..
                   10100.
                   10011.
                   -----.
                    01110
                    00000
                    -----
                     1110 = Остаток = Контрольная сумма!
```

В результате получаем частное, которое нас не интересует и, поэтому, отбрасывается за ненадобностью, и остаток, который и используется в качестве контрольной суммы. Расчет закончен.

Как правило, контрольная сумма добавляется к сообщению и вместе с ним передается по каналам связи. В нашем случае будет передано следующее сообщение: 11010110111110.

На другом конце канала приемник может сделать одно из равноценных действий:

1. Выделить текст собственно сообщения, вычислить для него контрольную сумму (не забыв при этом дополнить сообщение W битами), и сравнить ее с переданной.
2. Вычислить контрольную сумму для всего переданного сообщения (без добавления нулей), и посмотреть, получится ли в результате нулевой остаток.

Оба эти варианта совершенно равноправны. Однако, в следующем разделе мы будет работать со вторым вариантом, которое является математически более правильным.

Таким образом, при вычислении CRC необходимо выполнить следующие действия:

1. Выбрать степень полинома W и полином G (степени W).
2. Добавить к сообщению W нулевых битов. Назовем полученную строку M'.
3. Поделим M' на G с использованием правил CRC-арифметики. Полученный остаток и будет контрольной суммой.

Вот и все!

7. Выбор полинома

Выбор полинома чем-то сродни черной магии, и читатель отсылается к работе [Tanenbaum81] (стр.130-132), где этот вопрос обсуждается очень подробно. Цель же данного раздела состоит лишь в том, чтобы запугать до смерти любого, кто захочет задаться идеей создания собственного полинома. Если же Вас не интересует, почему один полином лучше другого, и Вы лишь хотите составить высокоскоростную реализацию алгоритма, то выберите себе один из перечисленных в конце статьи полиномов и просто перейдите к следующему разделу.

Во-первых, надо отметить, что переданное сообщение T является произведением полинома. Чтобы понять это, обратите внимание, что 1) последние W бит сообщения — это остаток от деления дополненного нулями исходного сообщения (помните?) на выбранный полином, и 2) сложение равносильно вычитанию, поэтому прибавление остатка дополняет значение сообщения до следующего полного произведения. Теперь смотрите, если сообщение при передаче было повреждено, то мы получим сообщение T⊕E, где E — это вектор ошибки, а '⊕' — это CRC-сложение (или операция XOR). Получив сообщение, приемник делит T⊕E на G. Так как T mod G = 0, (T⊕E) mod G = E mod G. Следовательно, качество полинома, который мы выбираем для перехвата некоторых определенных видов ошибок, будет определяться набором произведений G, так как в случае, когда E также является произведением G, такая ошибка выявлена не будет. Следовательно, наша задача состоит в том, чтобы найти такие классы G, произведения которых будут как можно меньше похожи на шумы в канале передачи (которые и вызывают повреждение сообщения). Давайте рассмотрим, какие типы шумов в канале передачи мы можем ожидать.

Однобитовые ошибки. Ошибка такого рода означает, что E=1000...0000. Мы можем гарантировать, что ошибки этого класса всегда будет распознаны при условии, что в G по крайней мере 2 бита установлены в "1". Любое произведение G может быть сконструировано операциями сдвига и сложения, и, в тоже время, невозможно получить значение с 1 единичным битом сдвигая и складывая величину, имеющую более 1 единичного бита, так как в результате всегда будет присутствовать по крайней мере 2 бита.

Двухбитовые ошибки. Для обнаружения любых ошибок вида 100...000100...000 (то есть когда E содержит по крайней мере 2 единичных бита) необходимо выбрать такое G, которые бы не имело множителей 11, 101, 1001, 10001, и так далее. Мне не совсем ясно, как этого можно достигнуть (у меня нет за плечами чисто математического образования), однако, Tanenbaum уверяет, что такие полиномы должны существовать, и приводит в качестве примера полином с единичными битами в позициях 15, 14 и 1, который не может быть делителем ни одно числа меньше 1...1, где "..." 32767 нулей.

Ошибки с нечетным количество бит. Мы может перехватить любые повреждения, когда E имеет нечетное число бит, выбрав полином G таким, чтобы он имел четное количество бит. Чтобы понять это, обратите внимание на то, что 1) CRC-умножение является простой операцией XOR постоянного регистрового значения с различными смещениями; 2) XOR — это всего навсего операция переключения битов; и 3) если Вы применяете в регистре операцию XOR к величине с четным числом битов, четность количества единичные битов в регистре останется неизменной. Например, начнем с E=111 и попытаемся сбросить все 3 бита в "0" последовательным выполнением операции XOR с величиной 11 и одним из 2 вариантов сдвигов (то есть, "E=E XOR 011" и "E=E XOR 110"). Это аналогично задаче о перевертывании стаканов, когда за одно действие можно перевернуть одновременно любые два стакана. Большинство популярных CRC-полиномов содержат четное количество единичных битов. (Замечание: Tanenbaum также утверждает, что все ошибки с нечетным количеством битов могут быть выявлены, если в качестве G выбрать произведение 11.)

Пакетные ошибки. Пакетная ошибка выглядит как E=000...000111...11110000...00, то есть E состоит из нулей за исключением группы единиц где-то в середине. Эту величину можно преобразовать в E=(10000...00)(1111111...111), где имеется z нулей в левой части и n единиц в правой. Для выявления этих ошибок нам необходимо установить младший бит G в 1. При этом необходимо, чтобы левая часть не была множителем G. При этом всегда, пока G шире правой части, ошибка всегда будет распознана. Более четкое объяснение этого явления ищите у Tanenbaum, я же затрудняюсь это сделать. Кроме того, Tanenbaum уверяет, что вероятность пакетной ошибки с шириной, большей чем W, равна $(0.5)^W$.

На этом мы завершим раздел, посвященный искусству выбора полиномов.

Приведу несколько популярных полиномов:

16-битные:	(16,12,5,0)	[стандарт "X25"]
	(16,15,2,0)	["CRC-16"]
32-битные:	(32,26,23,22,16,12,11,10,8,7,5,4,2,1,0)	[Ethernet]

8. Прямая реализация CRC

На этом теория заканчивается; теперь мы займемся реализацией алгоритма. Для начала рассмотрим абсолютно примитивную малоэффективную реализацию алгоритма "в лоб", не пользуясь для ускорения вычислений никакими трюками. Затем постепенно мы будет преобразовывать нашу программу до тех пор, пока не получим тот компактный табличный код, который нам всем известен, и который многие из нас хотели бы понять.

Но сначала нам необходимо создать программу, выполняющую CRC-деление. Существуют по крайней мере 2 причины, по которым мы не можем использовать обычные машинные инструкции деления, имеющиеся на наших компьютерах. Первая, мы должны выполнять операцию по правилам CRC-арифметики. Вторая же состоит в том, что делимое может иметь размер в десятки мегабайтов, а современные процессоры не имеют таких регистров.

Итак, при выполнения операции CRC-деления мы будет "скармливать" наше сообщение регистру. На этом этапе разработки мы постараемся обращаться с нашими данными абсолютно точно. Во всех последующих примерах в качестве сообщения рассматривается поток байтов по 8 битов каждый, из которых 7й бит будет считаться самым значащим битом. Поток битов, полученный из этих байтов, будет передаваться, начиная с 7 бита, и заканчивая 0 битом первого байта, затем второго и так далее.

Помня об этих соглашениях, мы теперь может кратко обрисовать сам алгоритм CRC-деления. Для примера возьмем полином 4й степени (W=4), и пусть этим полиномом будет 10111. В таком случае для выполнения деления нам потребуется 4-битный регистр.

```
                  3   2   1   0   Биты
                +---+---+---+---+
Выдвигаемый <-- |   |   |   |   | <----- Выровненное сообщение
бит!            +---+---+---+---+

              1   0   1   1   1   = Полином
```

(Напомню: выровненное сообщение - это наша исходная последовательность, дополненная W нулевыми битами.)

Деление выполняется следующим способом:

```
Загрузим регистр нулевыми битами
Дополним хвостовую часть сообщения W нулевыми битами
While (пока еще есть необработанные биты)
  Begin
    Сдвинем регистр на 1 бит влево и поместим очередной
               еще не обработанный бит из сообщения в 0 позицию регистра.
    If (из регистра был выдвинут бит со значением "1")
       Регистр = Регистр XOR Полином
  End
Теперь в регистре содержится остаток
```

(Замечание: На практике условие IF может быть протестировано по содержанию старшего бита регистра до выполнения операции сдвига.)

Назовем такую реализацию "простой".

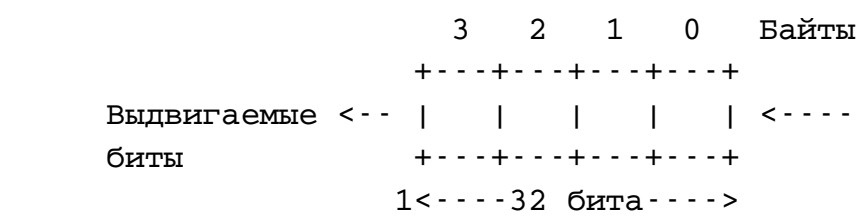
Это может показаться немного странным, но фактически все, что мы выполняли, было операциями вычитания полинома в различных степенях (что достигалось операциями сдвига) из исходного сообщения до тех пор, пока в результате у нас не осталось ничего, кроме остатка. Если Вам это не совсем ясно, то выполните эту операцию "в ручную" с числами.

Кроме того, Вы должны четко уяснить себе, что этот алгоритм может работать с полиномом любой степени W.

9. Реализация табличного алгоритма

Рассмотренный выше "простой" алгоритм является хорошей отправной точкой, так как он непосредственно связан с теорией, и потому, что он действительно прост. Однако, вследствие того, что он работает на уровне битов, его достаточно трудно закодировать (даже в C), а эффективность его работы низка, весь цикл выполняется по каждому биту. Для увеличения скорости работы нам необходимо найти способ заставить алгоритм работать с блоками, большими чем бит. Такими блоками могут быть полубайты (4 бита), байты, (8 бит), слова (16 бит) и длинные слова (32 бита), и даже более длинные фрагменты, если мы только сможем с ними справиться. С полубайтами лучше не связываться, так как они не выровнены на границу байта. Для увеличения скорости работы программы нам необходимо, чтобы информационные блоки были выровнены на границу байта, и, фактически, большинство табличных алгоритмов оперируют за один цикл именно с байтом.

Для простоты обсуждения давайте перейдем от 4-битного полинома к 32-битному. Наш регистр будет выглядеть точно также, как и ранее, с той лишь разницей, что ячейки теперь представляют не биты, а байты, а полином будет состоять из 33 битов (один подразумеваемый единичный бит и 32 "активных" бита) (W = 32).



Мы все еще можем воспользоваться "простым" алгоритмом. Давайте посмотрим, что он делает. Предположим, что алгоритм полностью переделан на работу с байтами, а старшие 8 бит 32-битового регистра (3 байт) имеют значение:

```
t7 t6 t5 t4 t3 t2 t1 t0
```

Во время очередного цикла "простого" алгоритма значение "t7" определяет, будет ли выполнена операция "Исключающее ИЛИ" (XOR) полинома во всем содержимым регистра. Если окажется, что t7=1, то операция выполняется, в противном случае она будет пропущена. Предположим также, что старшие 8 бит полинома имеют значения g7 g6…g0, тогда к следующему циклу старший байт будет иметь следующее содержимое:

```
         t6 t5 t4 t3 t2 t1 t0 ??
⊕                                        [Напомню: ⊕ означает XOR]
  t7 * (g7 g6 g5 g4 g3 g2 g1 g0)
```

"Новый" старший бит (который будет определять, что произойдет в следующем цикле) теперь имеет значение t6 ⊕ t7*g7. Обратите внимание, что с информационной точки зрения все, что требуется для вычисления значения "нового" старшего бита, содержится в 2 старших битах исходного старшего байта. Точно также, "следующий" старший бит может быть заранее вычислен, исходя их 3 старших битов t7, t6 и t5. В общем случае величину старшего бита регистра на k-цикле можно вычислить из k старших битов регистра. Давайте задержимся немного на этом этапе.

допустим, что мы используем старшие 8 бит регистра для расчета значения старшего бита регистра на 8м цикле. Предположим, что следующие 8 циклов мы выполняем, используя предвычисленные значения (которые можно записать в однобайтовый регистр и выдвигать из него последовательно по 1 биту). Тогда мы может заметить 3 вещи:

1. Величина старшего байта регистра теперь не будет иметь для нас никакого значения. Не будет иметь значения, сколько раз и с каким сдвигом полинома будет выполнена операция XOR, так как все биты так или иначе будут выдвинуты из правой части регистра за последующие 8 циклов.
2. Все оставшиеся биты регистра окажутся сдвинуты влево на 1 позицию, а самый правый байт регистра будет передвинут в позицию следующего байта левее его.
3. И, наконец, в процессе всего этого регистр окажется подвергнут серии операций XOR в соответствии с значениями битов предвычисленного управляющего байта.

Теперь рассмотрим влияние на регистр операций XOR с постоянной величиной при разных сдвигах. Например:

```
0100010  регистр
...0110  XOR
..0110.  XOR
0110...  XOR
-------
0011000
```

Из этого примера можно сделать следующий вывод: при выполнении операции XOR регистра с постоянной величиной при различных ее сдвигах всегда будет существовать некоторое значение, которое при применении операции "исключающее ИЛИ" с исходной величиной регистра даст нам тот же самый результат.

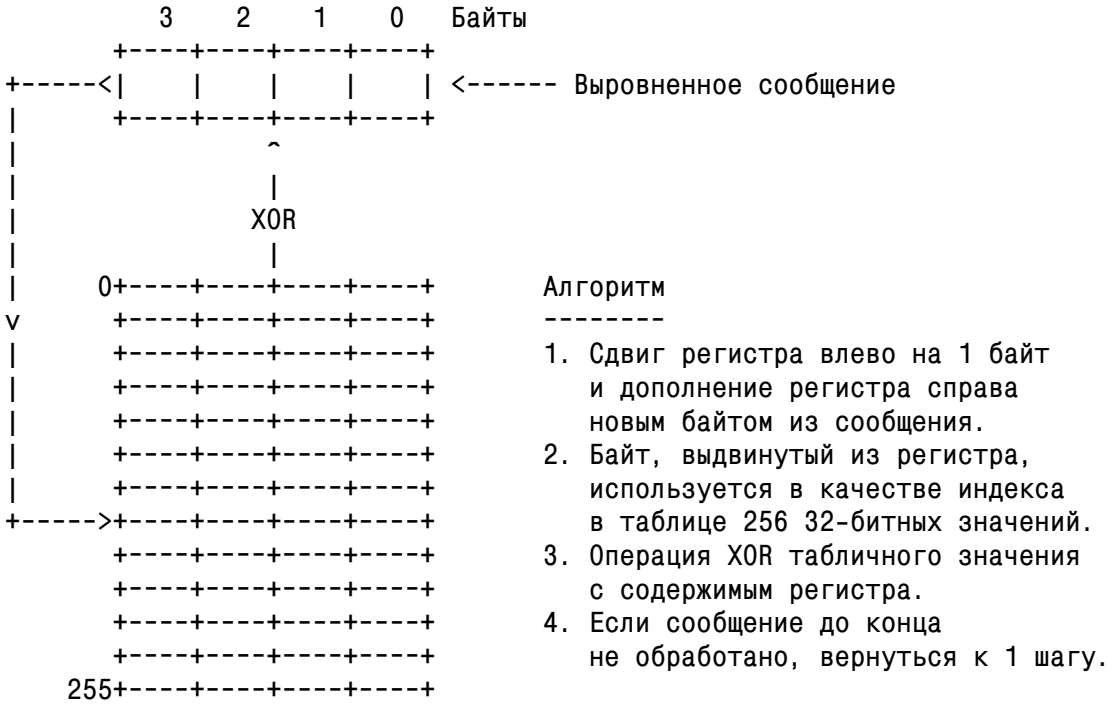
Возможно, у Вас уже появилась идея, как ускорить этот алгоритм. Сложив все вместе, получаем:

```
While (сообщение полностью не обработано)
  Begin
    Проверим старший байт регистра
    Рассчитаем на его основе контрольный байт
    Основываясь на значении контрольного байта,
              выполним операцию XOR нашего полинома с различными смещениями
    Сдвинем регистр на один байт влево, добавив справа новый байт из сообщения
    Выполним операцию XOR содержимого регистра с суммированным полиномом
  End
```

Пока у нас получилось не лучше, чем в случае "простого" алгоритма. Оказывается, однако, что большую часть вычислений можно провести заранее, а результаты собрать в таблицу. Тогда получим:

```
While (сообщение полностью не обработано)
  Begin
    Top = Старший_байт(Register);
    Register = (Register << 24) | Новый_байт_сообщения;
    Register = Register XOR Рассчитанная_таблица[Top];
  End
```

Вот так! Если Вам это понятно, то Вы поняли и основную идет табличного алгоритма расчета CRC. Это очень эффективный алгоритм, требующий только операций сдвига, OR, XOR, а также операции поиска в таблице. Графически он выглядит так.



На языке C это выглядит следующим образом:

```
r=0;
while (len--)
  {
   byte t = (r >> 24) & 0xFF;
   r = (r << 8) | *p++;
   r^=table[t];
  }
```

где `len` - это длина выровненного сообщения в байтах, `p` - указатель на это сообщение, `r` - регистр, `t` - временная переменная, а `table` - это заранее рассчитанная таблица значений. Этот код можно записать короче, правда он станет менее понятен:

```
r=0; while (len--) r = ((r << 8) | *p++) ^ table[(r >> 24) & 0xFF];
```

Это очень четкий и эффективный цикл, хотя случайному зрителю он кажется не вписывающимся в теорию CRC. Условимся называть его "Табличным" алгоритмом.

10. Слегка преобразованный табличный алгоритм

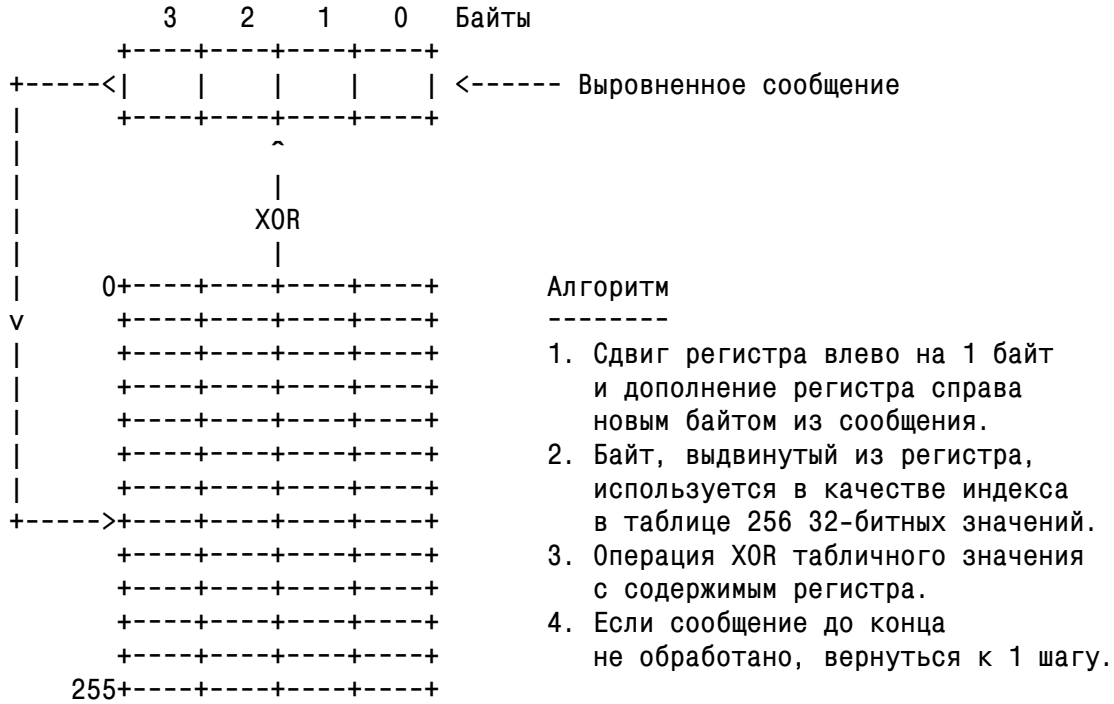
Несмотря на лаконичность строки

```
r=0; while (len--) r = ((r << 8) | *p++) ^ table[(r >> 24) & 0xFF];
```

любители оптимизаций не оставят ее в покое. Дело в том, что этот цикл работает с **выровненным** сообщением, то есть для его использования нам необходимо дополнить сообщение W/8 нулевыми байтами прежде, чем передавать указатель алгоритму. В зависимости от обстановки это может и не составить большой проблемы, однако, если обрабатываемый блок данных передается нам другой программой, такое дополнение может оказаться проблематичным. Одним из вариантов является простое добавление после основного цикла кода, обрабатывающего нулевые байты:

```
for (i=0; i<W/8; i++) r = (r << 8) ^ table[(r >> 24) & 0xFF];
```

Лично мне это решение довольно разумным. Однако, пожертвовав для ясности простотой алгоритма (которая, Вы должны это признать, всегда присуща нашим рассуждениям), теперь мы может преобразовать все таким образом, чтобы избежать и необходимости дополнения сообщения нулевыми байтами, и обработки каждого дополнительного нулевого байта в отдельности. Для пояснения этой возможности вернемся к уже рассмотренной нами диаграмме.

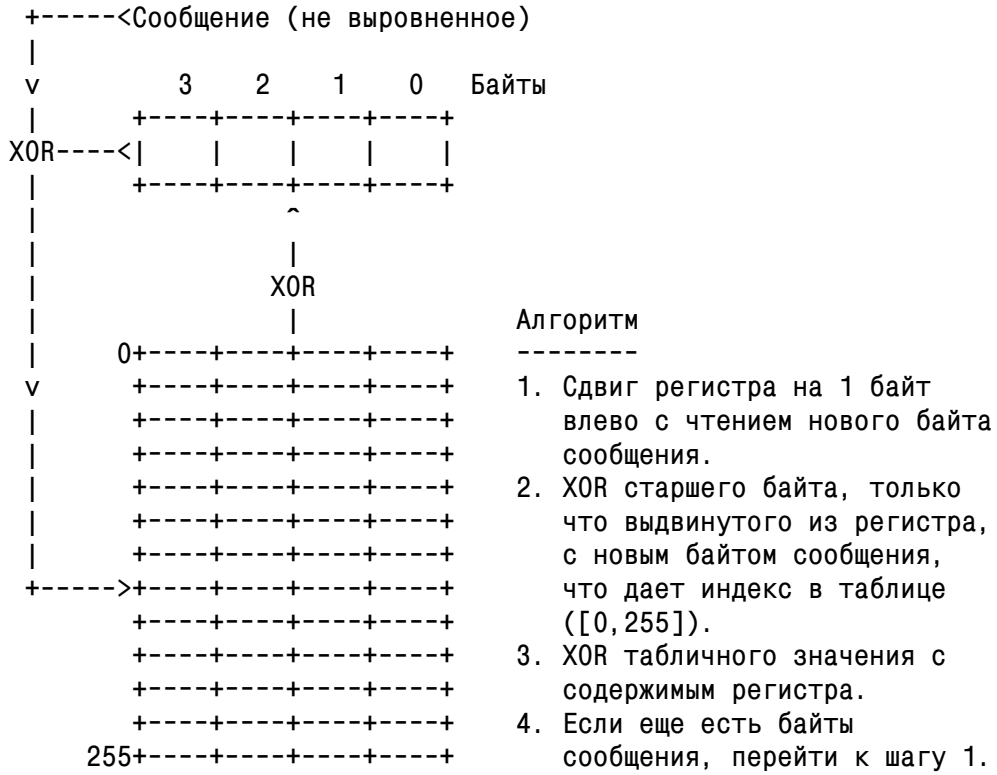


Обсудим сначала некоторые составные части обрабатываемого сообщения.

Хвост: W/8 дополнительных нулевых байта, которые появляются на последнем этапе работы алгоритма, чтобы быть вставленными в регистр по мере завершения обработки самого сообщения. Однако, они (нули!) не оказывают какого-либо влияния на регистр, так как 1) операция XOR с нулевым содержимым не меняет значения второго операнда; 2) эти 4 нулевых байта никогда не выдвигаются из левой части регистра, где их содержимое могло бы оказать хоть какое-то воздействие. Следовательно, единственное назначение этих нулевых байтов состоит в том, чтобы продлить работу алгоритма на W/8 циклов и позволить конечным байтам обрабатываемого сообщения пройти через весь регистр.

Заголовок: Если исходное содержимое регистра равно нулю, то начальные 4 цикла всего лишь загрузит первые 4 байта сообщения в регистр, сдвигая их справа налево. Это происходит потому, что первые 32 контрольных бита будут равны нулю, следовательно в результате операции XOR содержимое регистра меняться не будет. В случае же, когда в регистре изначально содержится ненулевое значение, в процессе первых 4 циклов все также будет происходить заполнение регистра 4 байтами сообщения с последующей операцией "Исключающее ИЛИ" с некоторым постоянным значением, которое целиком зависит от начального содержимого регистра.

Эти факты в совокупности с ассоциативным свойством операции "Исключающее ИЛИ" - `(A xor B) xor C = A xor (B xor C)` - означают, что байтам обрабатываемого сообщения нет никакой необходимости продираться через W/8 байтов регистра. Вместо этого они могут быть непосредственно скомбинированы операции XOR со старшим байтом регистра с использованием результата в качестве табличного индекса. Мы же получаем следующую модификацию алгоритма.



Замечание: Начальное значение регистра в данном случае должно быть тем же самым, что и начальное значение регистра предыдущего алгоритма, пропущенное 4 раза через таблицу. Таблица же такова, что если предыдущий алгоритм использовал "0", то и новый алгоритм также будет использовать это же значение.

Эти алгоритмы идентичны, и они дают идентичные результаты. Код на языке C выглядит следующим образом:

```
r=0; while (len--) r = (r<<8) ^ table[(r >> 24) ^ *p++];
```

и именно этот код Вы с наибольшей вероятностью найдете в современных реализациях табличного алгоритма расчета CRC. В некоторых случаях для большей переносимости кода используются маски "FF" и операции "AND", однако, идея цикла сохраняется. Такой алгоритм мы будет называть "Прямым табличным алгоритмом".

Во время изучения всего этого материала я пытался составить простой алгоритм, а затем получить из него табличную версию. Однако, когда я сравнил мой код с реальными программами вычисления CRC, то был совершенно сбит с толку тем, что байты для операции XOR извлекались с противоположного конца регистра. Прошло некоторое время, прежде чем я понял, что мы использовали один и тот же алгоритм. Одной из причин того, почему я взялся за эту статью, было то, что, хотя взаимосвязь между операцией деления и моим предыдущим табличным алгоритмом хотя и смутно, но все же прослеживается, она совершенно теряется, когда Вы начинаете вытаскивать байты с "неправильной" стороны регистра. Все это выглядит просто абсурдным!

Если Вы зашли уже столь далеко, что поняли теорию, практическую реализацию и ее оптимизированный вариант, то поймете и реальный код, к которому мы скоро перейдем. Могут ли здесь таиться какие-либо сложности! Однако, могут.

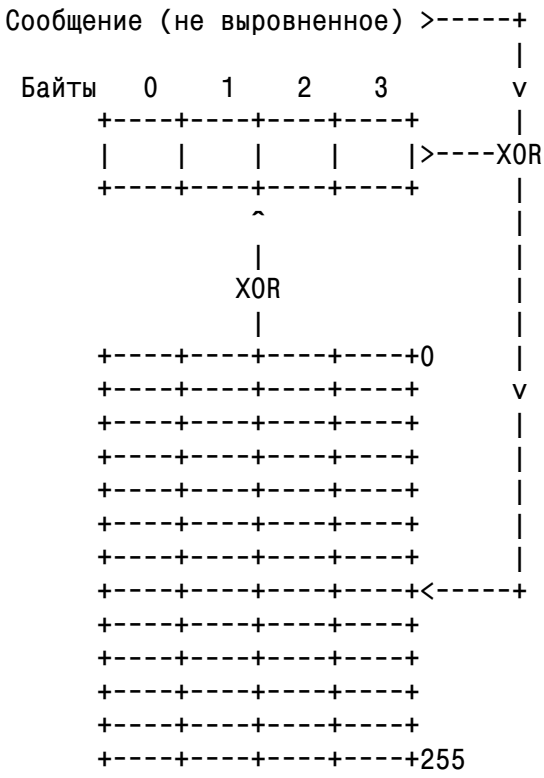
11. "Зеркальный" табличный алгоритм

Несмотря на то, что приведенный выше код оптимален настолько, насколько это только возможно, тем не менее всегда найдется некто, кто захочет усложнить себе (и другим) жизнь. Чтобы понять, почему это может произойти, нам придется обратится к миру аппаратного обеспечения.

Определение. Значение (или регистр) является отраженным (или обращенным), если все его биты поменялись местами относительно центра. Например: 0101 является 4-битным отражением величины 1010, а 0011 — это отражение 1100. Ну а величина 0111-0101-1010-1111-0010-0101-1011-1100 является отражением значения 0011-1101-1010-0100-1111-0101-1010-1110.

Получилось так, что UART (эти аккуратные маленькие микросхемы, которые выполняют операции последовательного ввода/вывода) имеют привычку передавать каждый байт, начиная с наименее значащего бита (бита 0), и заканчивая наиболее значащим (битом 7), т.е. в обратной — в зеркальной последовательности. В результате специалистам по аппаратному обеспечению, занимающимся аппаратным расчетом CRC на уровне битов, пришлось выбрать такие алгоритмы, которые работают с зеркально отраженными байтами. Сами байты обрабатываются в своем естественном порядке, однако их биты отражены относительно центра байта: бит 0 теперь становится битом 7, бит 1 — битом 6 и так далее. Не следует считать, что работа таких алгоритмов возможна только на аппаратном уровне. На каком-то этапе они, по-видимому, вышли на уровень программ, и кому-то пришлось написать код, который мог бы взаимодействовать с аппаратурой должным образом.

В такой ситуации любой нормальный программист обычно просто обменивает биты каждого байты перед его обработкой. Однако, такое естественное действие кажется совершенно неестественным в том перевернутом мире, поэтому вместо обмена битов лучше оставить их в покое и перевернуть сам мир вокруг них, что приводит нас к "зеркальному алгоритму", который идентичен только что рассмотренному нами за исключением того, что в нем все перевернуто кроме обрабатываемых байтов.



Замечания:

- Используется та же самая таблица, правда, содержимое каждой ее позиции зеркально обращено относительно центра позиции.
- Используется то же самое исходное значение регистра, только так же отраженное.
- Байты сообщения обрабатываются в том же порядке, что и раньше, то есть само сообщение не является зеркальным.
- Сами байты сообщения **не нужно** зеркально отображать, так как все остальное уже было отражено!

По окончании вычислений регистр будет содержать зеркальное отражение конечного значения CRC (остаток). На самом деле я затрудняюсь точно сказать, что нам с ним дальше делать, так как, по-видимому, аппаратные реализации вычисления CRC используют именно это зеркальное значение, следовательно его использование является справедливым. Должен заметить, что перевертывание всего мира, вероятно, является красивым инженерным решением, хотя и достаточно экстравагантным.

Мы будем называть этот алгоритм "зеркальным".

Независимо от того, имеет ли это смысл или не имеет, существование "зеркального" алгоритма в мире FTP сайтов приводит к тому, что половина реализаций расчета CRC использует его, а другая половина - нет. А вот это уже совершенно сбивает с толку. Особенно, мне кажется, что случайный пользователь, который пытается разобраться с "зеркальным" табличным алгоритмом с байтами, обрабатываемыми с "неправильного" конца не будет иметь никаких шансов, чтобы увязать код программы с теорией двоичного деления по модулю 2.

Могу ли я еще больше сбить Вас с толку? О, да, конечно, могу!

12. "Зеркальные" полиномы

Для тех, кто считает, что "зеркальной" реализации алгоритма не достаточно, существует иная идея, которая может совсем выбить почву из под ног. Это идея "зеркальных" полиномов.

Казалось бы, что зеркальное отражение хорошего полинома также дает в результате хороший полином! То есть, если значение полинома G=11101 является хорошим с точки зрения расчета CRC, то полином 10111 также будет удовлетворять нашим условиям. Как следствие, кажется вероятным, что всякий раз, когда организация (например, такая, как CCITT) стандартизирует некий хороший полином, остальной мир не может оставить без внимания его зеркальное отражение. Его **должны** использовать. В результате набор стандартных полиномов имеет соответствующий им набор зеркальных отражений, который также активно используется. Во избежание недоразумений будем называть такие полиномы "зеркальными".

X25	стандартный:	`1-0001-0000-0010-0001`
X25	"зеркальный":	`1-0000-1000-0001-0001`
CRC16	стандартный:	`1-1000-0000-0000-0101`
CRC16	"зеркальный":	`1-0100-0000-0000-0011`

Обратите внимание, что здесь отражается/обращается **весь полином целиком**, а не только хвостовые W битов. Это важное различие. В описанном в предыдущем разделе "зеркальном" алгоритме использовался полином, который фактически был идентичен полиному обычного " не-зеркального" алгоритма; их различие состояло лишь в том, что зеркально отражалось содержимое **байтов**. Точно также должны были быть отражены все 16/32-битные величины алгоритма. Напротив, **полный** полином включает в себя еще один старший подразумеваемый единичный бит, поэтому зеркальное отражение полинома — это не то же самое, что и отражение его младших 16 или 32 бит.

Из этого следует, что "зеркальный" алгоритм не эквивалентен исходному алгоритму с использованием "зеркального" полинома. Хотя, на самом деле, это более понятно, чем если бы они соответствовали один другому.

Если Вам что-то все еще не ясно, то не расстраивайтесь, так как мы очень скоро дойдем до реальных примеров. Остался последний раздел.

13. Начальные и конечные значения

В довершение ко всему нами виденному, CRC-алгоритмы различаются еще по 2 моментам:

- по начальному значению регистра;
- по значению, которое комбинируется по XOR с окончательным содержимым регистра.

Например, алгоритм "CRC-32" инициализирует регистр значением `0xFFFFFFFF` и выполняют операцию XOR окончательного значения также с величиной `0xFFFFFFFF`.

Большинство CRC-алгоритмов инициализируют регистр нулевым значением, однако некоторые предпочитают ненулевое. Теоретически (когда не делается никаких предположений относительно содержания сообщения) начальной значение не влияет на стойкость CRC, оно лишь является точкой отсчета, с которой алгоритм начинает работать. Однако, на практике некоторые сообщения более вероятны, чем другие, поэтому разумнее будет выбрать такое начальное значение регистра, которое бы не имело "слепых пятен". Под "слепым пятном" мы подразумеваем та-

кую последовательность байтов сообщения, которые не приводят к изменению содержимого регистра. В частности, любые CRC-алгоритмы, которые инициализируют свой регистр нулевым значением, будут иметь "слепое пятно" в отношении нулевых байтов в начале сообщения и не смогут оценить их количество. А так как в начале сообщения нулевые байты встречаются достаточно часто, разумнее инициализировать регистр значением, отличным от нуля.

14. Полное определение алгоритма

К настоящему моменту мы закончили рассмотрение всех аспектов табличных CRC-алгоритмов. В данном разделе мы попытаемся их классифицировать.

Итак, CRC-алгоритмы различаются по:

- степени полинома;
- значению полинома;
- начальному содержимому регистра;
- обращаются ли биты каждого байта перед их обработкой;
- требуется ли "проталкивать" байты сообщения через регистр, или они комбинируются по XOR с содержимым регистра, а затем делается поиск в таблице;
- нужно ли обращать конечный результат (как в "зеркальной" версии);
- значению, которое комбинируется по XOR с окончательным содержимым регистра.

Для того, чтобы в дальнейшем говорить об особенностях различных CRC-алгоритмов, нам необходимо определить все эти пункты более точно. По этой причине в следующем разделе будет сделана попытка составить параметрическую модель CRC-алгоритма. В дальнейшем при рассмотрении частных реализаций мы сможем просто определить ее в терминах этой модели.

15. Параметрическая модель CRC-алгоритма

Сейчас мы займемся составлением точной параметрической модели CRC-алгоритма, которую, за неимением лучшего, мы будем называть моделью Rocksoft™ (а, собственно, почему бы и нет?).

Данная модель будет рассмотрена исключительно с точки зрения функционирования, игнорируя при этом все детали реализаций. Основной целью будет создание способа точного описания различных частных алгоритмов CRC, вне зависимости от того, насколько сложно они реализованы. С этой точки зрения модель должна быть настолько простой и точной, насколько это возможно, и при этом как можно более понятной.

Модель CRC-алгоритма Rocksoft™ базируется на рассмотренном ранее прямом табличном алгоритме. Однако, этот алгоритм должен быть параметризован так, чтобы отражать поведение различных реальных алгоритмов.

Для функционирования алгоритма в качестве "зеркального" добавим две логические переменные: одну, определяющую обращение поступающих байтов, а другую, указывающую, нужно ли обращать окончательное значение контрольной суммы. Ограничивая "зеркальность" алгоритма только трансформацией на входе и выходе, мы избегаем ненужного усложнения при рассмотрении "зеркальных" и "не-зеркального" алгоритмов.

Дополнительный параметр позволяет инициализировать регистр начальным значением. Еще один параметр указывает на необходимость комбинации содержимого регистра с некоторым значением перед выдачей конечного результата.

Опишем теперь все параметры модели:

Name: Это имя, присвоенное данному алгоритму. Строковое значение.

Width: Степень алгоритма, выраженная в битах. Она всегда на единицу меньше длины полинома, но равна его степени.

Poly: Собственно полином. Это битовая величина, которая для удобства может быть представлена шестнадцатеричным числом. Старший бит при этом опускается. Например, если используется полином 10110, то он обозначается числом "06h". Важной особенностью данного параметра является то, что он всегда представляет собой необращенный полином, младшая часть этого параметра во время вычислений всегда является наименее значащими битами делителя вне зависимости от того, какой — "зеркальный" или прямой алгоритм моделируется.

Init: Этот параметр определяет исходное содержимое регистра на момент запуска вычислений. Именно это значение должно быть занесено в регистр в прямой табличном алгоритме. В принципе, в табличных алгоритмах мы всегда может считать, что регистр инициализируется нулевым значением, а начальное значение комбинируется по XOR с содержимым регистра после N цикла. Данный параметр указывается шестнадцатеричным числом.

RefIn: Логический параметр. Если он имеет значение "False" ("Ложь"), байты сообщения обрабатываются, начиная с 7 бита, который считается наиболее значащим, а наименее значащим считается бит 0. Если параметр имеет значение "True" ("Истина"), то каждый байт перед обработкой обращается.

RefOut: Логический параметр. Если он имеет значение "False" ("Ложь"), то конечное содержимое регистра сразу передается на стадию XorOut, в противном случае, когда параметр имеет значение "True" ("Истина"), содержимое регистра обращается перед передачей на следующую стадию вычислений.

XorOut: W-битное значение, обозначаемое шестнадцатеричным числом. Оно комбинируется с конечным содержимым регистра (после стадии RefOut), прежде чем будет получено окончательное значение контрольной суммы.

Check: Это поле, собственно, не является частью определения алгоритма, и, в случае противоречия с ним предшествующих параметров, именно эти предыдущие параметры имеют наибольший приоритет. Данное поле служит контрольным значением, которое может быть использовано для слабой проверки правильности реализации алгоритма. Поле содержит контрольную сумму, рассчитанную для ASCII строки "123456789" (шестнадцатеричные значение "313233...").

После определения всех этих параметров, наша модель может быть использована для точного описания особенностей каждого CRC-алгоритма. Ниже приведен пример спецификации популярного варианта алгоритма "CRC-16".

```
Name   : "CRC-16"
Width  : 16
Poly   : 8005
Init   : 0000
RefIn  : True
RefOut : True
XorOut : 0000
Check  : BB3D
```

16. Каталог параметров стандартных реализаций CRC-алгоритма

Здесь мне хотелось бы привести списки параметров наиболее часто используемых CRC-алгоритмов. Однако, большинство алгоритмов, с которыми мне до настоящего момента приходилось иметь дело, документированы столь скупо, что полностью это не представляется возможным. Все, что мне удалось, это составить список полиномов некоторых стандартных реализация CRC.

X25 standard	: 1021	[CRC-CCITT, ADCCP, SDLC/HDLC]
X25 reversed	: 0811	
CRC16 standard	: 8005	
CRC16 reversed	: 4003	[LHA]
CRC32	: 04C11DB7	[PKZIP, AUTODIN II, Ethernet, FDDI]

Я был бы чрезвычайно признателен всякому, кто смог бы дополнить списки параметров для параметрических моделей этих стандартов.

Однако, некоторые конкретные программы, по-видимому, следуют приведенным ниже спецификациям. Сможет ли кто-нибудь подтвердить или опровергнуть их (или получить контрольные суммы, которые я не смог закодировать и рассчитать)?

Name : "CRC-16/CITT"
Width : 16
Poly : 1021
Init : FFFF
RefIn : False
RefOut : False
XorOut : 0000
Check : ?

Name : "XMODEM"
Width : 16
Poly : 8408
Init : 0000
RefIn : True
RefOut : True
XorOut : 0000
Check : ?

Name : "ARC"
Width : 16
Poly : 8005
Init : 0000
RefIn : True
RefOut : True
XorOut : 0000
Check : ?

Ниже приведена спецификация для алгоритма CRC-32, который, как считается, используется в PKZip, AUTODIN II, Ethernet и FDDI.

Name : "CRC-32"
Width : 32
Poly : 04C11DB7
Init : FFFFFFFF
RefIn : True
RefOut : True
XorOut : FFFFFFFF
Check : CBF43926

17. Реализация модельного алгоритма

В настоящем разделе приведена реализация модельного алгоритма на языке C, состоящая их 2 файлов — заголовочного (.h) и собственно программы (.c). Если Вы просто читаете эту статью, то можете пропустит данный раздел, перейдя непосредственно к следующему.

Для получения работоспособного кода сконфигурируйте его в соответствии с приведенными выше параметрами CRC-16 или CRC-32 и проверьте, вычислив контрольную сумма для тестовой строки "123456789" (смотрите предыдущий раздел).

```
/*****************************************************************************/
/*                           Начало файла crcmodel.h                         */
/*****************************************************************************/
/*                                                                           */
/* Автор  : Ross Williams (ross@guest.adelaide.edu.au.).                     */
/* Дата   : 3 июня 1993.                                                     */
/* Статус : Public domain.                                                   */
/*                                                                           */
/* Описание: Это заголовочный файл (.h) реализации модельного CRC-алгоритма  */
/* Rocksoft^tm.  Подробная информация, касающаяся модели Rocksoft^tm,        */
/* смотрите в статье Ross Williams "A Painless Guide to CRC Error Detection  */
/* Algorithms" (ross@guest.adelaide.edu.au.).  Данная статья может быть      */
/* найдена по адресу "ftp.adelaide.edu.au/pub/rocksoft"                      */
/* Замечание: "Rocksoft" является зарегистрированной торговой маркой         */
/*            Rocksoft Pty Ltd, Adelaide, Australia.                         */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/* Использование данной программы                                            */
/* ------------------------------                                            */
/* Шаг 1: Объявить переменную типа cm_t и переменную (например, p_cm) типа   */
/*        p_cm_t, и инициализировать ее как указатель на первую переменную   */
/*        т.е.  p_cm_t p_cm = &cm_t.                                         */
/*                                                                           */
/* Шаг 2: Присвоить значения параметров модели членам структуры              */
/*        Если Вы не знаете, какие значения следует присваивать,             */
/*        то просмотрите раздел 16 документа                                 */
/*        Например:                                                          */
/*            p_cm->cm_width = 16;                                           */
/*            p_cm->cm_poly  = 0x8005L;                                      */
/*            p_cm->cm_init  = 0L;                                           */
/*            p_cm->cm_refin = TRUE;                                         */
/*            p_cm->cm_refot = TRUE;                                         */
/*            p_cm->cm_xorot = 0L;                                           */
```

```
/*        Замечание: полином указывается без старшего значащего бита          */
/*                   т.е. 18005 превращается в 8005                            */
/*        Замечание: Величина Width указывается на один бит меньше            */
/*                   длины полинома                                            */
/*                                                                             */
/* Шаг 3: Инициализируйте модель вызовом cm_ini(p_cm)                          */
/*                                                                             */
/* Шаг 4: Обработайте все сообщения, создав необходимое количество            */
/*        вызовов процедуры cm_nxt.  Например, cm_nxt(p_cm,ch)                 */
/*                                                                             */
/* Шаг 5: Извлеките полученную величину CRC с помощью crc = cm_crc(p_cm);      */
/*        Если CRC является 16-битной величиной, то она размещается            */
/*        в 16 младших битах результата.                                       */
/*                                                                             */
/*******************************************************************************/
/*                                                                             */
/* Замечания по реализации                                                     */
/* -----------------------                                                     */
/* ПЕРЕНОСИМОСТЬ: Этот пакет был закодирован таким образом, чтобы он мог      */
/* выполняться на как можно большем числе систем.  Например, имена всех       */
/* внешних идентификаторов ограничены 6 символами, а все внутренние - 8.      */
/* Префикс cm (означающий CRC Model) использован во избежание совпадения      */
/* имен переменных.  Данный пакет независим от каких-либо внешних программ.   */
/*                                                                             */
/* ЭФФЕКТИВНОСТЬ: Данный пакет (и его реализация) не предназначены             */
/* для скоростных вычислений.  Основное назначение - это моделирование         */
/* спецификаций CRC алгоритмов.  Если Вам необходима высокая скорость, то Вы  */
/* можете составить собственную табличную реализацию, основываясь на знаниях, */
/* полученных из данного материала.  Пакет также предназначен для проверки    */
/* правильности Ваших решений: если Вы обнаружите, или составите собственный  */
/* CRC-алгоритм, и захотите описать его в терминах параметрической модели     */
/* Rocksoft^tm, то он должен будет (при указанных значениях) давать тот же    */
/* результат, что и данная модель.                                             */
/*                                                                             */
/*******************************************************************************/

/* Следующий #ifndef охватывает весь заголовочный файл целиком, */
/* исключая его повторное использование.                        */
#ifndef CM_DONE
#define CM_DONE

/*******************************************************************************/
/* Следующие определения касаются моего собственного стиля программирования.  */
/* Идентификатор DONE_STYLE - это флаг, исключающий повторное использование   */
/* этих определений.                                                           */

#ifndef DONE_STYLE

typedef unsigned long   ulong;
typedef unsigned        bool;
typedef unsigned char * p_ubyte_;
```

```
#ifndef TRUE
#define FALSE 0
#define TRUE  1
#endif

/* Если у Вас отсутствует прототип, то используйте второе определение */
#define P_(A) A
/* #define P_(A) () */

/* Раскомментируйте следующее определение, если у Вас нет прототипа для  void */
/* typedef int void; */

#endif

/*****************************************************************************/

/* Абстрактный тип CRC модели */
/* -------------------------- */
/* Переменные следующего типа содержат контекст экземпляра вычислений     */
/* конкретной модели.  Большинство полей являются параметрами модели,     */
/* которые необходимо задать перед первым вызовом cm_ini.                 */
typedef struct
  {
   int   cm_width;   /* Параметр: Степень полинома в битах [8,32].        */
   ulong cm_poly;    /* Параметр: Полином алгоритма                       */
   ulong cm_init;    /* Параметр: Начальное значение регистра             */
   bool  cm_refin;   /* Параметр: Обращать байты на входе?                */
   bool  cm_refot;   /* Параметр: Обращать конечное значение CRC?         */
   ulong cm_xorot;   /* Параметр: Величина для комбинации по XOR          */
                     /*           с конечным значением CRC.               */
   ulong cm_reg;     /* Контекст: Контекст времени исполнения.            */
  } cm_t;
typedef cm_t *p_cm_t;

/*****************************************************************************/

/* Функции реализации модели */
/* ------------------------- */
/* Следующие функции "оживляют" абстрактный тип cm_t. */

void cm_ini P_((p_cm_t p_cm));
/* Инициализация конкретного экземпляра модели CRC.                  */
/* Все поля параметра должны быть заданы перед вызовом этой функции */

void cm_nxt P_((p_cm_t p_cm,int ch));
/* Обработка единичного байта сообщения [0,255]. */

void cm_blk P_((p_cm_t p_cm,p_ubyte_ blk_adr,ulong blk_len));
/* Обработка блока байтов сообщения. */

ulong cm_crc P_((p_cm_t p_cm));
/* Вывод величины CRC, вычисленной к настоящему моменту. */
```

```
/*****************************************************************************/

/* Функции для расчета таблицы */
/* --------------------------- */
/* Следующая функция может быть использована для расчета таблицы CRC         */
/* как во время исполнения, так и для создания статических таблиц.           */

ulong cm_tab P_((p_cm_t p_cm,int index));
/* Возвращает содержимое i-позиции таблицы просмотра для заданного алгоритма. */
/* Функция проверяет поля cm_width, cm_poly, cm_refin и индекс в таблице      */
/* (значение в диапазоне [0,255] и возвращает младшие cm_width байтов         */
/* содержимого таблицы                                                        */

/*****************************************************************************/

/* Конец заголовочного файла (закрытие "скобки" #ifndef) */
#endif

/*****************************************************************************/
/*                            Конец файла crcmodel.h                         */
/*****************************************************************************/

/*****************************************************************************/
/*                            Начало файла crcmodel.c                        */
/*****************************************************************************/
/*                                                                           */
/* Автор  : Ross Williams (ross@guest.adelaide.edu.au.).                     */
/* Дата   : 3 июня 1993.                                                     */
/* Статус : Public domain.                                                   */
/*                                                                           */
/* Описание: Это программный файл (.c) реализации модельного CRC-алгоритма   */
/* Rocksoft^tm.  Подробная информация, касающаяся модели Rocksoft^tm,        */
/* смотрите в статье Ross Williams "A Painless Guide to CRC Error Detection  */
/* Algorithms" (ross@guest.adelaide.edu.au.).  Данная статья может быть      */
/* найдена по адресу "ftp.adelaide.edu.au/pub/rocksoft"                      */
/* Замечание: "Rocksoft" является зарегистрированной торговой маркой         */
/*            Rocksoft Pty Ltd, Adelaide, Australia.                         */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/* Замечания по реализации программы                                         */
/* ---------------------------------                                         */
/* Для исключения дублирования информации и возникновения случайных          */
/* расхождений спецификации используемых функций здесь не представлены.     */
/* Их описание дано в заголовочном файле.  Эта программа предназначена       */
/* для проверки, поэтому мне хотелось сделать ее как можно короче и проще    */
/* (было бы чрезвычайно сложно представить все мнои достижения в написании   */
/* программ на языке C (например, реализацию ввода данных)).                 */
/*                                                                           */
/*****************************************************************************/
```

```
#include "crcmodel.h"

/**************************************************************************/

/* Следующие определения сделаны для улучшения читаемости кода. */

#define BITMASK(X) (1L << (X))
#define MASK32 0xFFFFFFFFL
#define LOCAL static

/**************************************************************************/

LOCAL ulong reflect P_((ulong v,int b));
LOCAL ulong reflect (v,b)
/* Функция возвращает значение v с обращенными b [0,32] младшими байтами. */
/* Например: reflect(0x3e23L,3) == 0x3e26                                  */
ulong v;
int   b;
{
 int   i;
 ulong t = v;
 for (i=0; i<b; i++)
   {
    if (t & 1L)
       v|=  BITMASK((b-1)-i);
    else
       v&= ~BITMASK((b-1)-i);
    t>>=1;
   }
 return v;
}

/**************************************************************************/

LOCAL ulong widmask P_((p_cm_t));
LOCAL ulong widmask (p_cm)
/* Возвращает длинное слово, чье значение равно (2^p_cm->cm_width)-1.      .*/
/* Этот сделано для улучшения переносимости пакета (чтобы обойтись без <<32). */
p_cm_t p_cm;
{
 return (((1L<<(p_cm->cm_width-1))-1L)<<1)|1L;
}

/**************************************************************************/

void cm_ini (p_cm)
p_cm_t p_cm;
{
 p_cm->cm_reg = p_cm->cm_init;
}

/**************************************************************************/
```

```
void cm_nxt (p_cm,ch)
p_cm_t p_cm;
int    ch;
{
 int   i;
 ulong uch  = (ulong) ch;
 ulong topbit = BITMASK(p_cm->cm_width-1);

 if (p_cm->cm_refin) uch = reflect(uch,8);
 p_cm->cm_reg ^= (uch << (p_cm->cm_width-8));
 for (i=0; i<8; i++)
   {
    if (p_cm->cm_reg & topbit)
       p_cm->cm_reg = (p_cm->cm_reg << 1) ^ p_cm->cm_poly;
    else
       p_cm->cm_reg <<= 1;
    p_cm->cm_reg &= widmask(p_cm);
   }
}

/*****************************************************************************/

void cm_blk (p_cm,blk_adr,blk_len)
p_cm_t   p_cm;
p_ubyte_ blk_adr;
ulong    blk_len;
{
 while (blk_len--) cm_nxt(p_cm,*blk_adr++);
}

/*****************************************************************************/

ulong cm_crc (p_cm)
p_cm_t p_cm;
{
 if (p_cm->cm_refot)
    return p_cm->cm_xorot ^ reflect(p_cm->cm_reg,p_cm->cm_width);
 else
    return p_cm->cm_xorot ^ p_cm->cm_reg;
}

/*****************************************************************************/

ulong cm_tab (p_cm,index)
p_cm_t p_cm;
int    index;
{
 int   i;
 ulong r;
 ulong topbit = BITMASK(p_cm->cm_width-1);
 ulong inbyte = (ulong) index;
```

```
 if (p_cm->cm_refin) inbyte = reflect(inbyte,8);
 r = inbyte << (p_cm->cm_width-8);
 for (i=0; i<8; i++)
    if (r & topbit)
       r = (r << 1) ^ p_cm->cm_poly;
    else
       r<<=1;
 if (p_cm->cm_refin) r = reflect(r,p_cm->cm_width);
 return r & widmask(p_cm);
}

/*****************************************************************************/
/*                          Конец файла crcmodel.c                           */
/*****************************************************************************/
```

18. Создай собственную табличную реализацию

Вопреки всему мною сказанному о понимании и определении CRC-алгоритмов, механизм их высокоскоростной реализации достаточно тривиален. Фактически существует лишь 2 вида: нормальный и "зеркальный". Обычный алгоритм выполняет сдвиг влево и относится к параметрическим моделям с RefIn=FALSE и RefOt=FALSE. "Зеркальные" алгоритмы выполняют сдвиг вправо, а оба упомянутых выше параметра имеют значение TRUE. (Если у Вас возникнет мысль установить один из них в TRUE, а другой — в FALSE, то с последствиями этого Вы должны будете разбираться сами!) Другие параметры (Init и XorOt) могут быть закодированы в виде макросов. Ниже приведена реализация нормального 32-битного варианта (16-битный вариант выполняется аналогично).

```
unsigned long crc_normal ();
unsigned long crc_normal (blk_adr,blk_len)
unsigned char *blk_adr;
unsigned long  blk_len;
{
 unsigned long crc = INIT;
 while (blk_len--)
    crc = crctable[((crc>>24) ^ *blk_adr++) & 0xFFL] ^ (crc << 8);
 return crc ^ XOROT;
}
```

Ниже приведен "зеркальный" вариант:

```
unsigned long crc_reflected ();
unsigned long crc_reflected (blk_adr,blk_len)
unsigned char *blk_adr;
unsigned long  blk_len;
{
 unsigned long crc = INIT_REFLECTED;
 while (blk_len--)
    crc = crctable[(crc ^ *blk_adr++) & 0xFFL] ^ (crc >> 8));
 return crc ^ XOROT;
}
```

Замечание. Я тщательно проверил оба приведенных фрагмента, однако я не компилировал и не тестировал их. В принципе, для Вас это не должно играть никакой роли вне зависимости от того, **что** Вы кодируете, всегда можно заранее утверждать, что код будет нормально работать, если Вы следовали всем ранее обсужденным правилам. А это было именно так — рассмотренная модель давала все необходимые наставления.

Замечание. Если скорость расчетов Вас не очень сильно заботит, то лучше использовать данную (проверенную) реализацию модели.

19. Генерация таблицы просмотра

Единственным компонентом, который выпал в предыдущем разделе из нашего поля зрения, была таблица просмотра. Данная таблица может быть рассчитана во время исполнения программы с помощью процедуры cm_tab, или заранее составлена и встроена в программу на C. В любом случае, необходимо отметить, что ее содержимое зависит только от полинома и параметров RefIn и RefOt, причем главная роль здесь принадлежит полиному.

Приведенная программа создает любую 16- или 32-битную таблицу просмотра. Если данная программа Вас не интересует, то Вы может перейти непосредственно к следующему разделу.

```
/**************************************************************************/
/*                            Начало файла crctable.c                     */
/**************************************************************************/
/*                                                                        */
/* Автор  : Ross Williams (ross@guest.adelaide.edu.au.).                  */
/* Дата   : 3 июня 1993.                                                  */
/* Версия : 1.0.                                                          */
/* Статус : Public domain.                                                */
/*                                                                        */
/* Описание: Данная программа генерирует таблицы просмотра CRC-алгоритма  */
/* и выводит ее в файл (который затем может быть вставлен в программу на C).  */
/* Программа может быть сконфигурирована для создания таблицы любого вида,    */
/* который относится к CRC модели Rocksoft^tm.                            */
/* Подробную информацию, касающуюся модели Rocksoft^tm,                   */
/* смотрите в статье Ross Williams "A Painless Guide to CRC Error Detection   */
/* Algorithms" (ross@guest.adelaide.edu.au.).  Данная статья может быть       */
/* найдена по адресу "ftp.adelaide.edu.au/pub/rocksoft"                   */
/*                                                                        */
/* Замечание: "Rocksoft" является зарегистрированной торговой маркой      */
/*            Rocksoft Pty Ltd, Adelaide, Australia.                      */
/*                                                                        */
/**************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "crcmodel.h"

/**************************************************************************/

/* Параметры таблицы                                                      */
/* =================                                                      */
```

```
/* Следующие параметры полностью определяют режим генерации таблицы.  Перед   */
/* запуском программы необходимо модифицировать лишь определения.              */
/*                                                                             */
/*    TB_FILE  - имя файла для вывода сгенерированной таблицы                  */
/*    TB_WIDTH - ширина таблицы в байтах (2 или 4 байта)                       */
/*    TB_POLY  - полином степени TB_WIDTH байт                                 */
/*    TB_REVER - флаг обращения таблицы                                        */
/*                                                                             */
/* Пример:                                                                     */
/*    #define TB_FILE   "crctable.out"                                         */
/*    #define TB_WIDTH  2                                                      */
/*    #define TB_POLY   0x8005L                                                */
/*    #define TB_REVER  TRUE                                                   */

#define TB_FILE   "crctable.out"
#define TB_WIDTH  4
#define TB_POLY   0x04C11DB7L
#define TB_REVER  TRUE

/*****************************************************************************/

/* Различные определения */

#define LOCAL static
FILE *outfile;
#define WR(X) fprintf(outfile,(X))
#define WP(X,Y) fprintf(outfile,(X),(Y))

/*****************************************************************************/

LOCAL void chk_err P_((char *));
LOCAL void chk_err (mess)
/* Если mess - не пустая строка, то она выдается, а программа останавливает   */
/* работу.  В противном случае производится проверка наличия ошибки вывода    */
/* в файл, и, если ошибка была, программа останавливает работу.               */
char *mess;
{
 if (mess[0] != 0   ) {printf("%s\n",mess); exit(EXIT_FAILURE);}
 if (ferror(outfile)) {perror("chk_err");   exit(EXIT_FAILURE);}
}

/*****************************************************************************/

LOCAL void chkparam P_((void));
LOCAL void chkparam ()
{
 if ((TB_WIDTH != 2) && (TB_WIDTH != 4))
    chk_err("chkparam: Width parameter is illegal.");
/*  chk_err("Ошибка: Недопустимое значение параметра Width.");                */
 if ((TB_WIDTH == 2) && (TB_POLY & 0xFFFF0000L))
    chk_err("chkparam: Poly parameter is too wide.");
/*  chk_err("Ошибка: Полином имеет слишком большой размер.");                 */
```

```
 if ((TB_REVER != FALSE) && (TB_REVER != TRUE))
    chk_err("chkparam: Reverse parameter is not boolean.");
/*  chk_err("Ошибка: Параметр обращения должен иметь логическое значение.");  */
}

/*****************************************************************************/

LOCAL void gentable P_((void));
LOCAL void gentable ()
{
 WR("/*****************************************************************/\n");
 WR("/*                                                               */\n");
 WR("/* CRC LOOKUP TABLE                                              */\n");
 WR("/* ================                                              */\n");
 WR("/* The following CRC lookup table was generated automagically    */\n");
 WR("/* by the Rocksoft^tm Model CRC Algorithm Table Generation       */\n");
 WR("/* Program V1.0 using the following model parameters:            */\n");
 WR("/*                                                               */\n");
 WP("/*    Width   : %1lu bytes.                                         */\n",
    (ulong) TB_WIDTH);
 if (TB_WIDTH == 2)
 WP("/*    Poly    : 0x%04lX                                           */\n",
    (ulong) TB_POLY);
 else
 WP("/*    Poly    : 0x%08lXL                                       */\n",
    (ulong) TB_POLY);
 if (TB_REVER)
 WR("/*    Reverse : TRUE.                                            */\n");
 else
 WR("/*    Reverse : FALSE.                                           */\n");
 WR("/*                                                               */\n");
 WR("/* For more information on the Rocksoft^tm Model CRC Algorithm,  */\n");
 WR("/* see the document titled \"A Painless Guide to CRC Error        */\n");
 WR("/* Detection Algorithms\" by Ross Williams                        */\n");
 WR("/* (ross@guest.adelaide.edu.au.). This document is likely to be  */\n");
 WR("/* in the FTP archive \"ftp.adelaide.edu.au/pub/rocksoft\".        */\n");
 WR("/*                                                               */\n");
 WR("/*****************************************************************/\n");
 WR("\n");
 switch (TB_WIDTH)
   {
    case 2: WR("unsigned short crctable[256] =\n{\n"); break;
    case 4: WR("unsigned long  crctable[256] =\n{\n"); break;
    default: chk_err("gentable: TB_WIDTH is invalid.");
   }
 chk_err("");

 {
  int i;
  cm_t cm;
  char *form    = (TB_WIDTH==2) ? "0x%04lX" : "0x%08lXL";
  int   perline = (TB_WIDTH==2) ? 8 : 4;
```

```
  cm.cm_width = TB_WIDTH*8;
  cm.cm_poly  = TB_POLY;
  cm.cm_refin = TB_REVER;

  for (i=0; i<256; i++)
    {
     WR(" ");
     WP(form,(ulong) cm_tab(&cm,i));
     if (i != 255) WR(",");
     if (((i+1) % perline) == 0) WR("\n");
     chk_err("");
    }

 WR("};\n");
 WR("\n");
 WR("/*********************************************************************/\n");
 WR("/*                    End of CRC Lookup Table                        */\n");
 WR("/*********************************************************************/\n");
 WR("");
 chk_err("");
}
}

/*****************************************************************************/

main ()
{
 printf("\n");
 printf("Rocksoft^tm Model CRC Algorithm Table Generation Program V1.0\n");
 printf("------------------------------------------------------------\n");
 printf("Output file is \"%s\".\n",TB_FILE);
 chkparam();
 outfile = fopen(TB_FILE,"w"); chk_err("");
 gentable();
 if (fclose(outfile) != 0)
    chk_err("main: Couldn't close output file.");
 printf("\nSUCCESS: The table has been successfully written.\n");
}
/*****************************************************************************/
/*                           Конец файла crctable.c                          */
/*****************************************************************************/
```

20. Резюме

Данная статья дает детальное описание CRC-алгоритмов и их теоретических основ, постепенно раскрывая сложности реализации, начиная с простых сдвиговых и кончая табличными реализациями. Описывается параметрическая модель, которая может быть использована для точного определения частных случаев CRC-алгоритмов, а также ее программная реализация. В заключении дается программа генерации таблиц просмотра для CRC-алгоритмов.

21. Поправки

Если Вы считаете, что какая-либо часть данного документа не ясно написана или в чем-то не верна, или Вы имеете некую дополнительную информацию или предложения, как этот документ можно улучшить, пожалуйста, свяжитесь с автором. Особенно мне хотелось бы иметь описания стандартных реализаций CRC-алгоритмов с помощью параметрической модели Rocksoft™.

A. Словарь

Контрольная сумма (Checksum) — Число, которое является функцией некоторого сообщения. Буквальная интерпретация данного слова указывает на то, что выполняется простое суммирование байтов сообщения, что, по-видимому, и делалось в ранних реализациях расчетов. Однако, на сегодняшний момент, несмотря на использование более сложных схем, данный термин все имеет широкое применение.

CRC —Сокращение словосочетания "Cyclic Redundancy Code" (Циклический Избыточный Код). Тогда как термин "Контрольная сумма", по-видимому, используется для обозначения любого механизма не криптографического контроля информации, термин "CRC" зарезервирован только для алгоритмов, основанных на идее полиномиального деления.

G — Этот символ используется в тексте статьи для обозначения полинома.

Сообщение (Message) — Данные, для которых рассчитывается контрольная сумма. Обычно представлена в виде последовательности байтов. То, какой из битой в байте — первый или последний, считается наиболее значащим, определяется алгоритмом расчета CRC.

Полином (Polynomial) — Полином является делителем CRC-алгоритма.

Обращение (Reflect) — Двоичное число обращается (отражается) обменом всех битов байта относительно его центра. Например, 1101 является обращением величины 1011.

Модель CRC-алгоритма Rocksoft™ — Параметрический алгоритм, целью которого является создание основы для описания CRC-алгоритма. Как правило, CRC-алгоритм может быть описан указанием используемого полинома. Однако, для создания точной реализации необходимо также знать начальные и конечные параметры.

Степень (Width) — Степень (или ширина) CRC-алгоритма соответствует степени используемого полинома (или длине полинома минус единица). Например, если используется полином 11010, то степень алгоритма равна 4. Обычно используется степень, кратная 8.

B. Ссылки

[Griffiths87] Griffiths, G., Carlyle Stones, G., "The Tea-Leaf Reader Algorithm: An Efficient Implementation of CRC-16 and CRC-32", Communications of the ACM, 30(7), pp.617-620.

В данной статье описывается высокоскоростная табличная реализация CRC-алгоритмов. Использованный способ слегка странен и может быть заменен алгоритмом Sarwate.

[Knuth81] Knuth, D.E., "The Art of Computer Programming", Volume 2: Seminumerical Algorithms, Section 4.6. (Кнут Д.Е. "Искусство программирования для ЭВМ", т. 2 "Получисленные алгоритмы" — М., "Мир", 1977).

[Nelson 91] Nelson, M., "The Data Compression Book", M&T Books, (501 Galveston Drive, Redwood City, CA 94063), 1991, ISBN: 1-55851-214-4.

Если Вас интересует, как на самом деле реализовано вычисление 32-битной контрольной суммы, смотрите страницы 440 и 446-448 данного издания.

[Sarwate88] Sarwate, D.V., "Computation of Cyclic Redundancy Checks via Table Look-Up", Communications of the ACM, 31(8), pp.1008-1013.

В этой работе описывается высокоскоростная табличная реализация CRC-алгоритма, которая является развитием алгоритма "чайных листьев" (tea-leaf algorithm). В приложении (которое само по себе является довольно ценным источником) описываются методы, используемые большинством современных программ расчета CRC. Однако, данная работа достаточна сложна для изучения.

[Tanenbaum81] Tanenbaum, A.S., "Computer Networks", Prentice Hall, 1981, ISBN: 0-13-164699-0.

Раздел 5.3.5 на страницах 128-132 предоставляет очень ясное описание кодирования CRC. Однако, табличный метод расчета здесь не описан.

C. Другие, обнаруженные мной, но не просмотренные ссылки

Boudreau, Steen, "Cyclic Redundancy Checking by Program," AFIPS Proceedings, Vol. 39, 1971.

Davies, Barber, "Computer Networks and Their Protocols," J. Wiley & Sons, 1979.

Higginson, Kirstein, "On the Computation of Cyclic Redundancy Checks by Program," The Computer Journal (British), Vol. 16, No. 1, Feb 1973.

McNamara, J. E., "Technical Aspects of Data Communication," 2nd Edition, Digital Press, Bedford, Massachusetts, 1982.

Marton and Frambs, "A Cyclic Redundancy Checking (CRC) Algorithm," Honeywell Computer Journal, Vol. 5, No. 3, 1971.

Nelson M., "File verification using CRC", Dr Dobbs Journal, May 1992, pp.64-67.

Ramabadran T.V., Gaitonde S.S., "A tutorial on CRC computations", IEEE Micro, Aug 1988.

Schwaderer W.D., "CRC Calculation", April 85 PC Tech Journal, pp.118-133.

Ward R.K, Tabandeh M., "Error Correction and Detection, A Geometric Approach" The Computer Journal, Vol. 27, No. 3, 1984, pp.246-253.

Wecker, S., "A Table-Lookup Algorithm for Software Computation of Cyclic Redundancy Check (CRC)," Digital Equipment Corporation memorandum, 1974.

Перевод выполнен на сайте:
http://dore.on.ru

Перевод и оформление:
Sergey R.