Anarchriz/DREAD

CRC, и как его восстановить

CRC and how to Reverse it[†]
Anarchriz/DREAD

Вступление

В этой статье даны краткое описание CRC и способов его восстановление. Многие программисты и исследователи не знают деталей работы CRC, и практически никто не знает способов его восстановления, несмотря на то, что такие знания могли бы им очень пригодиться. В первой части статьи приведено общее описание способов вычисления CRC, что может быть использовано для защиты данных или программ. Вторая часть рассказывает, каким образом можно восстановить CRC-32, что может быть использовано для взлома некоторых защит. По-видимому, эти методы используются теми, кто "исправляет" для Вас CRC, хотя я сомневаюсь, чтобы они рассказывали Вам, как это делается.

Сразу же хочу оговориться, что в статье используется довольно много математических выкладок. Я считаю, что они понятны любому программисту среднего уровня. Зачем это нужно? Ну что ж, если Вы не знаете, зачем при обсуждении CRC необходима математика, то я предлагаю Вам поскорее воспользоваться кнопкой "X" в правом верхнем углу экрана. Теперь, полагаю, все оставшиеся читатели имеют достаточные познания в области двоичной арифметики.

Часть 1. Путеводитель по CRC: что это такое, и как он вычисляется

Cyclic Redundancy Code (CRC) – циклический избыточный код

Все мы слышали о CRC. Если Вы забыли об этом, то Вам напомнят о нем неприятные сообщения от RAR, ZIP и других архиваторов, когда файл окажется поврежден в результате некачественного соединения или повреждения флоппи-диска. CRC — это значение, которое вычисляется для некоторого блока данных, например, для каждого файла во время архивации. При развертывании файла архиватор сравнивает это значение со вновь вычисленным CRC распакованного файла. Если они совпадают, то существует очень большая вероятность того, что этот новый файл получился идентичным исходному. При использовании CRC-32 вероятность пропустить изменение данных составляет всего $1/2^{32}$.

[†] Оригинал статьи можно найти по адресу:
http://huizen.dds.nl/~noway66/programming/crc.htm (Здесь и далее прим. перев.)

Некоторые считают, что CRC означает Cyclic Redundancy Check (Циклическая избыточная проверка). Если бы это было так, то тогда в большинстве случаев этот термин использовался бы некорректно, ведь нельзя говорить, что "CRC равно 12345678" — как это проверка может чему-то равняться?. Кроме того, многие говорят, что программа имеет проверку по CRC — как это так, Циклическая избыточная проверка может иметь проверку? Напрашивается вывод: CRC является сокращением выражения "Циклический избыточный код" и никак не "Циклическая избыточная проверка".

Как делаются эти вычисления? Основная идея состоит в том, чтобы представить файл, как одну огромную строку бит, и поделить ее на некоторое число; оставшийся в результате остаток и есть CRC! У Вас всегда будет оставаться остаток (правда, иногда он может оказаться равным нулю), который частенько лишь на один бит меньше делителя — 9/3=3, остаток =0; (9+2)/3=3, остаток =2.

С битами деление выполняется иначе — оно представляет собой последовательное вычитание (X раз) из делимого некоторого числа (делителя), что в результате всегда оставит Вам некоторый остаток. Если Вы хотите восстановить исходное значение, то Вам необходимо умножить делитель на "X" (или сложить его само с собой "X" раз) и в конце добавить остаток.

Вычисление CRC использует особый вид вычитания и сложения, своего рода "новую арифметику". Компьютер "забывает" делать перенос при вычислении каждого бита.

Давайте взглянем на примеры: 1й — это обычное вычитание, а 2 и 3 — специальное:

```
      -+
(1) 1101    (2) 1010  1010    (3) 0+0=0  0-0=0
    1010-       1111+ 1111-       0+1=1 *0-1=1
    ----        ----  ----        1+0=1  1-0=1
    0011        0101  0101       *1+1=0  1-1=0
```

В 1м примере во втором столбце справа производится вычисление `0-1=-1`, следовательно, необходимо "занять" один бит из следующего разряда: `(10+0)-1=1`. (Это аналогично обычному десятичному вычитанию "на бумаге".) Особый случай — 1+1 в нормальных вычислениях дает в результате 10, и единица "переносится" в старший разряд. В специальных действиях (примеры 2 и 3) такой перенос "забывается". Аналогично, 0-1 должно бы равняться -1, что требует заему бита из старшего разряда (смотрите пример 1). Этот заем при вычислении также "забывают" сделать. Если Вы знакомы с программированием, то это должно Вам напомнить операцию "Исключающее ИЛИ" (eXclusive OR, или более привычно — XOR), чем оно фактически и является.

Посмотрим теперь на деление:

Вот как оно делается в нормальной арифметике:

```
1001/1111000\1101 13              9/120\13
     1001    -                      09  -|
     ----                           --   |
      1100                           30  |
      1001    -                      27  -
      ----                           --
       0110                           3 -> остаток
       0000    -
       ----
        1100
        1001    -
        ----
         011 -> 3, остаток
```

В CRC-арифметике все немного по-другому (пример 3):

```
1001/1111000\1110      9/120\14 остаток 6 (120/9=14 и остаток 6)
     1001    -
     ----
      1100
      1001    -
      ----
       1010
       1001    -
       ----
        0110
        0000    -
        ----
         110 -> остаток
```

Частное от деления совершенно не важно, и его нет необходимости запоминать, так как оно лишь на пару бит меньше той последовательности, для которой Вы хотите рассчитать CRC. Что действительно важно, так это остаток! Именно он может сообщить нечто достаточно важное обо всем файле. Фактически, это и есть CRC.

Обзор реального вычисления CRC

Для вычисления CRC нам необходимо выбрать делитель, который с этого момента мы будет называть полиномом. Степень полинома (W — Width) — это номер позиции его старшего бита, следовательно, полином 1001 будет иметь степень "3", а не "4". Обратите внимание, что старший бит всегда должен быть равен 1, следовательно, после того, как Вы выбрали степень полинома, Вам необходимо подобрать лишь значения младших его битов.

Если Вы хотите вычислять CRC для последовательности бит, Вам следует убедиться, что обработаны все биты. Поэтому в конце последовательности необходимо добавить W нулевых бит. Так для примера 3 мы может утверждать, что последовательность битов равнялась 1111. Взгляните на более сложный пример (пример 4):

```
Полином                              = 10011, степень W=4
Последовательность бит + W нулей = 110101101 + 0000

10011/1101011010000\110000101 (о частном мы не заботимся)
      10011|||||||| -
      -----||||||||
       10011|||||||
       10011|||||||   -
       -----|||||||
        00001||||||
        00000||||||    -
        -----||||||
         00010|||||
         00000|||||     -
         -----|||||
          00101||||
          00000||||      -
          -----||||
           01010|||
           00000|||       -
           -----|||
            10100||
            10011||        -
            -----||
             01110|
             00000|         -
             -----|
              11100
              10011          -
              -----
               1111 -> остаток -> то есть CRC!
```

Здесь необходимо упомянуть 2 важных момента:

1. Операция XOR выполняется только в том случае, когда старший бит последовательности равен 1, в противном случае мы просто сдвигаем ее на один бит влево.
2. Операция XOR выполняется только с младшими W битами, так как старший бит всегда в результате дает 0.

Обзор табличного алгоритма

Понятно, что алгоритм, основанный на битовых вычислениях очень медленен и неэффективен. Было бы намного лучше проводить вычисления с целыми байтами. Но в этом случае мы сможем иметь дело лишь с полиномами, имеющими степень, кратную 8 (то есть величине байта). Давайте представим себе такие вычисления на основе полинома 32 степени (W=32).

```
                3   2   1   0    байт
              +---+---+---+---+
Выдвинутый <--|   |   |   |   |<-- последовательность, дополненная
бит           +---+---+---+---+    W нулевыми битами (здесь 32 бита)

             1<--- 32 бит  --->    полином степени 4*8 бит
```

(Схема 1)

Здесь показан регистр, который используется для сохранения промежуточного результата вычисления CRC, и который я будут называть регистром CRC или просто регистром. Когда бит, выдвинутый из левой части регистра, равен 1, то выполняется операция "Исключающее ИЛИ" (XOR) содержимого регистра с младшими W битами полинома (их в данном случае 32). Фактически мы выполняет описанную выше операцию деления.

А что, если (как я уже предложил) сдвигать одновременно целые группы бит. Рассмотрим пример вычисления CRC длиной 8 бит со сдвигом 4 бит одновременно.

До сдвига регистр равен: `10110100`

Затем 4 бита выдвигаются с левой стороны, тогда как справа вставляются новые 4 бита. В данном примере выдвигается группа `1011`, а вдвигается — `1101`.

Таким образом мы имеем:

текущее содержимое 8 битного регистра (CRC)	: `01001101`
4 старших бита, только что выдвинутые слева	: `1011`
используемый полином (степень W=8)	: `101011100`

Теперь вычислим новое значение регистра:

```
Старш. Регистр
биты
---- --------
1011 01001101          Старшие биты и регистр
1010 11100    ⊕ (*1)   Полином складывается по XOR, начиная с 3 бита
                       старшей группы (естественно, он равен 1)
-------------
0001 10101101          Результат операции XOR
```

В старшей группе в позиции 0 у нас остался еще один бит, равный 1

```
0001 10101101          Предыдущий результат
   1 01011100 ⊕ (*2)   Полином складывается по XOR, начиная с 0 бита
                       старшей группы (естественно, он тоже равен 1)
-------------
0000 11110001          Результат второй операции XOR
^^^^
```

Теперь все 4 бита старшей группы содержат нули, поэтому нам больше нет необходимости выполнять в ней операцию XOR.

То же самое значение регистра могло быть получено, если операнд (*1) сложить по XOR с операндом (*2). Это возможно в результате ассоциативного свойства этой операции — (a XOR b) XOR c = a XOR (b XOR c).

```
1010 11100             Полином, прикладываемый к 3 биту старшей группы
   1 01011100 ⊕ (*2)   Полином, прикладываемый 0 биту старшей группы
-------------
1011 10111100   (*3)   Результат операции XOR
```

Если теперь применить к исходному содержимому регистра операцию XOR со значением (*3), то получим:

```
1011 10111100
1011 01001101 ⊕        Старшие биты и регистр
-------------
0000 11110001
```

Видите? Тот же самый результат! Значение (*3) для нас достаточно важно, так как в случае, когда старшая группа бит равна 1011, младшие W=8 бит всегда будут равны 10111100 (естественно, для данного примера). Это означает, что мы можем заранее рассчитать величины XOR-слагаемого для каждой комбинации старшей группы бит. Обратите внимание, что эта старшая группа в результате всегда превратится в 0.

Вернемся снова к схеме 1. Для каждой комбинации битов старшей группы (8 битов), выдвигаемых из регистра, мы можем рассчитать слагаемое. Получим таблицу из 256 (или 2^8) двойных слов (32 бита) (такая таблица для CRC-32 приведена в приложении).

На мета-языке этот алгоритм можно записать следующим образом:

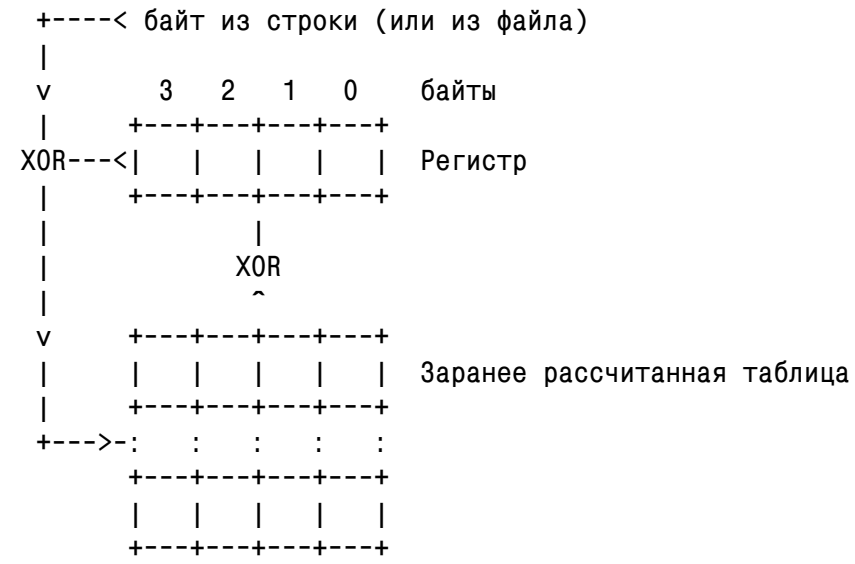
```
While (в строке имеются необработанные биты)
  Begin
    Top = top_byte of register ;
    Register = (сдвиг Register на 8 бит влево) OR (новые биты из строки);
    Register = Register XOR табличное_значение[Top];
  End
```

Прямой табличный алгоритм

Описанный выше алгоритм можно оптимизировать. Нет никакой необходимости постоянно "проталкивать" байты строки через регистр. Мы можем непосредственно применять операцию XOR байтов, выдвинутых из регистра, с байтами обрабатываемой строки. Результат будет указывать на позицию в таблице, значение из которой и будет в следующем шаге складываться в регистром.

Я не могу точно объяснить, почему такая последовательность действий дает тот же самый результат (он не следует из свойств операции "Исключающее ИЛИ"), однако, его большим преимуществом является отсутствие необходимости дополнять строку нулевыми байтами/битами (если Вы сможете найти этому объяснение, пожалуйста, сообщите его мне :)).

Давайте взглянем на схему алгоритма (схема 2):

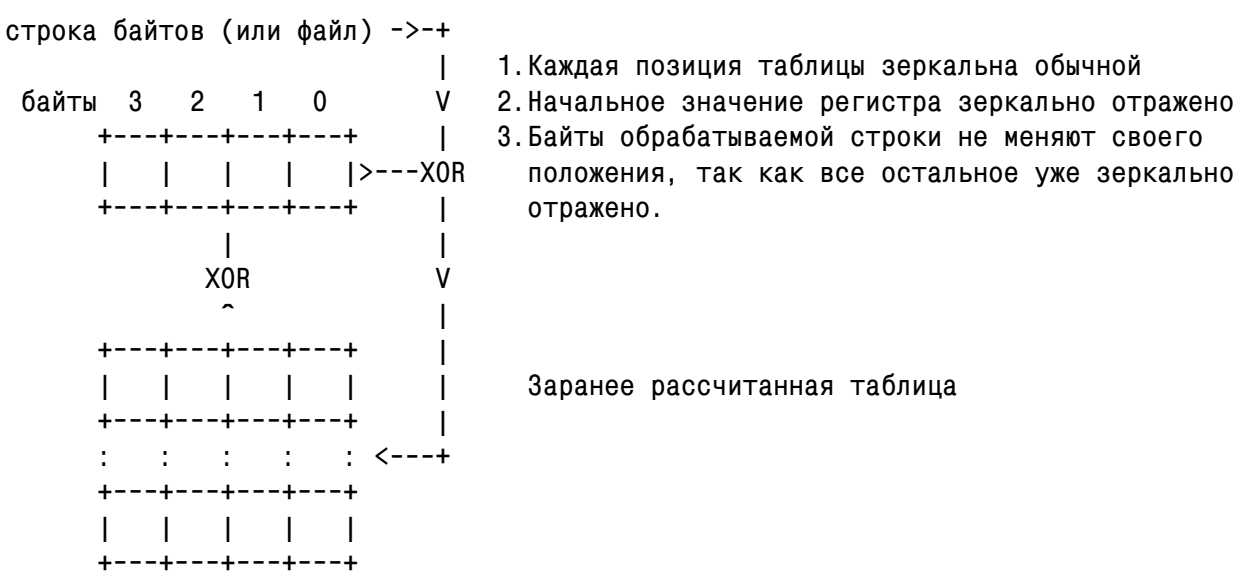


"Зеркальный" прямой табличный алгоритм

Чтобы жизнь не казалась медом, была создана еще и так называемая "зеркальная" версия этого алгоритма. В "зеркальном" регистре все биты отражены относительно центра. Например, '0111011001' есть "отражение" значения '1001101110'.

Эта версия обязана своим возникновением существованию UART (микросхема последовательного ввода/вывода), которая посылает биты, начиная с наименее значимого (бит 0) и заканчивая самым старшим (бит 7), то есть в обратном порядке.

Вместо того, чтобы менять местами биты перед их обработкой, можно зеркально отразить все остальные значения, участвующие в вычислениях, что в результате дает очень компактную реализацию программы. Так, при расчетах биты сдвигаются вправо, полином зеркально отражается относительно его центра, и, естественно, используется зеркальная таблица предвычисленных значений.



(Схема 3)

В этом случае наш алгоритм станет следующим:
1. Сдвигаем регистр на 1 байт вправо.
2. Выполняем операцию XOR только что выдвинутого вправо из регистра байта с байтом обрабатываемой строки для получения номера позиции в таблице ([0,255]).
3. Табличное значение складываем по XOR с содержимым регистра.
4. Если байты еще остались, то повторяем шаг 1.

Некоторые реализации алгоритма на ассемблере

Для начала приведен полный стандарт алгоритма CRC-32:

Название	: "CRC-32"
Степень	: 32
Полином	: 04C11DB7
Начальное значение	: FFFFFFFF
Отражение	: Да
Последнее XOR с	: FFFFFFFF

Для любознательных (в качестве приза) стандарт алгоритма CRC-16:

Название	: "CRC-16"
Степень	: 16
Полином	: 8005
Начальное значение	: 0000
Отражение	: Да
Последнее XOR с	: 0000

Величина, обозначенная "Последнее XOR с", представляет собой то значение, которое складывается по XOR с содержимым регистра перед получением окончательного результата CRC.

Кроме того, существуют "зеркальные" CRC-полиномы, однако, они выходят за рамки данной статьи. Если Вы хотите узнать о них по подробнее, обращайтесь с дополнительной литературе.

При написании примера на ассемблере я использовал 32-битый код в 16-битном режиме DOS (поэтому в примерах Вы обнаружите 16- и 32-битную смесь), что, однако, не сложно перевести в настоящий 32-битный код. Обратите внимание, что примеры на ассемблере полностью протестированы, примеры на языках Java и C являются их производными.

Итак. Вот реализация вычисления таблицы CRC-32:

```
        xor     ebx, ebx        ;ebx=0, будет использоваться как указатель
InitTableLoop:
        xor     eax, eax        ;eax=0 для получения новой позиции
        mov     al, bl          ;младшие 8 бит ebx копируются в младшие 8 бит eax
                ;генерируется номер позиции
        xor     cx, cx
entryLoop:
        test    eax, 1
        jz      no_topbit
        shr     eax, 1
        xor     eax, poly
        jmp     entrygoon
```

```
no_topbit:
        shr     eax, 1
entrygoon:
        inc     cx
        test    cx, 8
        jz      entryLoop
        mov     dword ptr[ebx*4 + crctable], eax
        inc     bx
        test    bx, 256
        jz      InitTableLoop
```

Замечания
- crctable - это массив из 256 значений, рассчитываемая нами таблица CRC;
- eax сдвигается вправо, так как в данном случает используется "зеркальный" алгоритм;
- по той же причине обрабатываются младшие 8 бит…

А вот реализация того же алгоритма на языке Java или C (int — 32-битное значение):

```
for (int bx=0; bx<256; bx++){
  int eax=0;
  eax=eax&0xFFFFFF00+bx&0xFF;     // инструкция 'mov al,bl'
  for (int cx=0; cx<8; cx++){
    if (eax&&0x1) {
      eax>>=1;
      eax^=poly;
    }
    else eax>>=1;
  }
  crctable[bx]=eax;
}
```

Реализация вычислений CRC-32 с помощью составленной таблицы:

```
computeLoop:
        xor     ebx, ebx
        xor     al, [si]
        mov     bl, al
        shr     eax, 8
        xor     eax, dword ptr[4*ebx+crctable]
        inc     si
        loop   computeLoop
        xor     eax, 0FFFFFFFFh
```

Замечания
- 'ds:si' указатель буфера, где находится обрабатываемая строка;
- cx количество обрабатываемых байтов (длина строки);
- eax содержит текущее значение CRC;
- таблица заранее вычисленных значений получена приведенной выше программой;
- начальное значение CRC в случае CRC-32 — FFFFFFFF;
- после окончания вычислений значение CRC складывается по XOR с величиной FFFFFFFF (аналог операции NOT).

На языках Java или C это будет выглядеть следующим образом:

```
for (int cx=0; cx>=8;†
   eax^=crcTable[ebx];
 }
 eax^=0xFFFFFFFF;
```

Таким образом, мы завершили первое знакомство с CRC. Если Вы хотите узнать о них немного больше, то прочтите другую мою статью, ссылку на которую можно найти в конце документа.

Хорошо! Теперь приступим с более интересной части: восстановлению CRC!

Часть 2. Восстановление CRC

Когда я размышлял об о способах восстановления CRCs Эта проблема возникала у меня неоднократно. Я пытался "обмануть" CRC, изобретая такую последовательность байтов, чтобы оказалось безразличным, какие символы и сколько их будет стоять перед ней. Я не добился успехаs Затем я понял, что таким способом успеха добиться невозможно, так как алгоритм CRC построен так, чтобы вне зависимости от того, какой **бит** Вы меняете, результат вычисления **всегда** (хорошо, почтиs всегда) преображается совершенно неузнаваемо. Попробуйте сами поэкспериментировать с какой-либо простейшей CRC программой... :)

Я понял, что могу "корректировать" CRC лишь **после** изменения необходимых мне байтов. Я смог составить такую последовательность, которая преобразовывала CRC так, как мне это было необходимо!

Рассмотрим конкретный пример.

Имеется последовательность байтов:

`01234567890123456789012345678901234567890123456789012`

Вам необходимо изменить байты, выделенные подчеркиванием (с 9 по 26). Кроме того, Вам понадобятся еще 4 байта (до 30й позиции) для построения такой последовательности, которая восстановит исходное значение CRC.

Когда Вы начинаете вычислять CRC-32 все идет прекрасно до байта в 9 позиции; в измененной же последовательности, начиная с этой точки, CRC меняется кардинально. Даже после 26 позиции, байты после которой изменений не претерпели, получить исходного значения CRC оказывается невозможно.

Однако, прочитав статью до конца, Вы поймете, что следует предпринять в такой ситуации. В двух словах это можно описать следующим образом:

1. Необходимо вычислить значение CRC вплоть до 9 позиции и запомнить его.
2. Продолжить расчет CRC до 30 позиции включительно (байты, которые Вы собираетесь менять, плюс 4 байта дополнительно) и также запомнить значение.
3. Рассчитать значение CRC для "новых" байтов, включая 4 дополнительных (то есть всего для 27-9+4=22 байтов), используя в расчетах величину, полученную в шаге 1, снова его запомнить.
4. Теперь мы имеем "новое" значение CRC. Однако, нам необходимо, чтобы CRC имело "старое" значение. Для расчета 4 дополнительных байтов необходимо "обратить" алгоритм.

До настоящего момента мы могли выполнить только этапы с первого по третий. Теперь поучимся делать четвертый.

---

† Фрагмент кода в оригинале пропущен.

Восстановление CRC-16

Для начала мы займемся восстановлением 16-разрядного CRC. Итак, мы сделали в последовательности все необходимые изменения, и теперь нам необходимо вернуть прежнее значение CRC. Мы знаем старое значение CRC (рассчитанное до изменения последовательности), а также его новое значение. Нам необходимо составить такую 2-байтную последовательность, которая позволит изменить текущее значение CRC в его прежнее состояние.

Сначала мы рассчитаем CRC с учетом 2 неизвестных байт (назовем их "X" и "Y"). Предположим, что регистр имеет значения a1 и a0, а вдвигаемый байт равен 00. Взглянем еще раз на схему 3 и приступим.

Возьмем последовательность "X Y". Байты обрабатываются слева-направо. Предположим, регистр равен "a1 a0". Будем обозначать операцию XOR символом "⊕".

Обработаем 1 байт ("X"):

a0⊕X	"старший" байт (1)
b1 b0	табличная последовательность для этого старшего байта
00 a1	регистр, сдвинутый вправо
00⊕b1 a1⊕b0	результат операции XOR двух предыдущих строк

Теперь регистр содержит: (b1) (a1⊕b0).

Обработаем 2 байт ("Y"):

(a1⊕b0)⊕Y	"старший" байт (2)
c1 c0	табличная последовательность для этого старшего байта
00 b1	регистр, сдвинутый вправо
00⊕c1 b1⊕c0	результат операции XOR двух предыдущих строк

В результате регистр будет иметь значение (c1) (b1⊕c0).

Запишем эти вычисления в несколько ином виде:

a0 ⊕ X	=(1)	указатель табличного значения b1 b0
a1 ⊕ b0 ⊕ Y	=(2)	указатель табличного значения c1 c0
b1 ⊕ c0	=d0	новый младший байт регистра
c1	=d1	новый старший байт регистра
(1) (2)		

Теперь давайте переварим полученные результаты :). (Не отчаивайтесь, пример с реальными числами не за горами).

Подумайте! Мы хотим получить в регистре значение d1 d0 (исходное CRC), зная содержимое регистра обработки измененной последовательности символов (значение a1 a0). Вопрос — какие 2 байта, или, другими словами, какие значения "X" и "Y" нам необходимо использовать при вычислении CRC?

Начнем рассуждать с конца. d0 должно быть равно b1⊕c0, а d1 = c1. Однако, я так и слышу Ваш возглас: "Как я могу узнать значения b1 и c0?!!!". Но нужно ли мне напоминать Вам о Таблице? А значение c0 можно получить из слова c0 c1, ведь мы знаем значение c1. Следовательно нам потребуется процедура поиска. А, если Вы нашли это слово, то уж постарайтесь запомнить и его индекс, с помощью которого мы сможем найти старшие группы байт, то есть величины (1) и (2).

Но как же нам получить b1 b0, несмотря на то, что мы нашли c1 c0? Если b1⊕c0=d0, то b1=d0⊕c0! Ну, а затем, найдя b1, применим процедуру поиска для получения слова b1 b0. Теперь у нас есть все для вычисления значений "X" и "Y"! Превосходно, не правда ли?!!!

a1⊕b0⊕Y=(2), поэтому Y=a1⊕b0⊕(2)

a0⊕X=(1), следовательно X=a0⊕(1)

Численный пример восстановления CRC-16

Давайте теперь рассмотрим реальный числовой пример:

- исходное значение регистра: (a1=)DEh (a0=)ADh
- нужно получить: (d1=)12h (d0=)34h

Найдем в таблице для CRC-16 (см. Приложение) строку, начинающуюся с 12h. Это позиция 38h, которая содержит значение 12C0h. Это единственное такое значение, ведь, вспомните, мы рассчитывали каждую позицию в таблице для всех возможных величин старшей группы, всего их было 256.

Теперь мы знаем, что (2)=38, c1=12, а c0=C0, следовательно b1=C0⊕34=F4, и следующим шагом необходимо найти позицию, значение в которой начинается с F4h. Это оказалась позиция 4Fh содержащая F441h. Получаем, (1)=4F, b1=F4, b0=41. Теперь у нас есть все, что рассчитать "X" и "Y":

```
Y=a1⊕b0⊕(2) =DE⊕41⊕38=A7
X=a0⊕(1)    =AD+4F    =E2
```

Следовательно, чтобы изменить значение регистра CRC-16 со значения DEAD на значение 1234 необходимы байты E2 A7 (именно в таком порядке).

Как Вы видите, чтобы восстановить CRC, Вы должны "просчитать" его в обратном порядке и запомнить все полученные значения. При реализации программы просмотра таблицы обратите внимание, что для процессоров фирмы Intel характерно запоминание байтов в обратном порядке (сначала младший, а затем старший байты).

Теперь, когда Вы, надеюсь, поняли, как восстанавливать CRC-16, займемся CRC-32.

Восстановление CRC-32

CRC-32 восстанавливается столь же просто, как и CRC-16. Правда, нам теперь придется иметь дело с 4 байтами вместо 2. Итак, смотрите и сравнивайте.

Допустим, что мы имеем 4 байтовую строку "X Y Z W", байты берутся с **левой** стороны.

Предположим, что в регистре содержится значением "a3 a2 a1 a0", при этом "a3" является старшим, а "a0" — младшим (наименее значащим) байтом.

Обработаем первый байт ("X"):

```
a0⊕X                                  рассчитанная старшая группа (1)
b3     b2     b1         b0    найденная для нее табличная последовательность
00     a3     a2         a1            сдвиг регистра вправо
00⊕b3 a3⊕b2 a2⊕b1      a1⊕b0         сложение двух предыдущих строк по XOR
```

Новое содержимое регистра: (b3) (a3⊕b2) (a2⊕b1) (a1⊕b0)

Обработаем второй байт ("Y"):

```
(a1⊕b0)⊕Y                             рассчитанная старшая группа (2)
c3     c2     c1         c0    найденная для нее табличная последовательность
00     b3     a3⊕b2      a2⊕b1         сдвиг регистра вправо
00⊕c3 b3⊕c2 a3⊕b2⊕c1  a2⊕b1⊕c0      сложение двух предыдущих строк по XOR
```

Новое содержимое регистра: (c3) (b3⊕c2) (a3⊕b2⊕c1) (a2⊕b1⊕c0)

Обработаем третий байт ("Z"):

```
(a2⊕b1⊕c0)⊕Z                          рассчитанная старшая группа (3)
d3     d2     d1         d0    найденная для нее табличная последовательность
00     c3     b3⊕c2      a3⊕b2⊕c1      сдвиг регистра вправо
00⊕d3 c3⊕d2 b3⊕c2⊕d1  a3⊕b2⊕c1⊕d0 сложение двух предыдущих строк по XOR
```

Новое содержимое регистра: (d3) (c3⊕d2) (b3⊕c2⊕d1) (a3⊕b2⊕c1⊕d0)

Обработаем четвертый байт ("W"):

```
(a3⊕b2⊕c1⊕d0)⊕W                       рассчитанная старшая группа (4)
e3     e2     e1         e0    найденная для нее табличная последовательность
00     d3     c3⊕d2      b3⊕c2⊕d1      сдвиг регистра вправо
00⊕e3 d3⊕e2 c3⊕d2⊕e1  b3⊕c2⊕d1⊕e0 сложение двух предыдущих строк по XOR
```

Новое содержимое регистра: (e3) (d3⊕e2) (c3⊕d2⊕e1) (b3⊕c2⊕d1⊕e0)

Теперь я перепишу это в несколько ином виде:

```
a0 ⊕ X                          =(1)   позиция  b3 b2 b1 b0  в таблице
a1 ⊕ b0 ⊕ Y                     =(2)   позиция  c3 c2 c1 c0  в таблице
a2 ⊕ b1 ⊕ c0 ⊕ Z                =(3)   позиция  d3 d2 d1 d0  в таблице
a3 ⊕ b2 ⊕ c1 ⊕ d0 ⊕ W           =(4)   позиция  e4 e3 e2 e1  в таблице
     b3 ⊕ c2 ⊕ d1 ⊕ e0          =f0
          c3 ⊕ d2 ⊕ e1          =f1
               d3 ⊕ e2          =f2
                    e3          =f3
     (1)   (2)   (3)   (4)
```

(схема 4)

Короче, вся последовательность действий аналогична 16-битной версии.

Теперь я приведу пример с реальными значениями. Таблица для CRC-32 дана в Приложении.

Предположим, что начальное значение CRC-регистра (a3 a2 a1 a0) равно "AB CD EF 66".

Нам необходимо получить величину (f3 f2 f1 f0) "56 33 14 78".

Итак:

```
первый байт позиции                          ее номер   содержимое позиции
e3=f3                              =56 -> 35h=(4) 56B3C423 (e3 e2 e1 e0)
d3=f2⊕e2          =33⊕B3           =E6 -> 4Fh=(3) E6635C01 (d3 d2 d1 d0)
c3=f1⊕e1⊕d2       =14⊕C4⊕63        =B3 -> F8h=(2) B3667A2E (c3 c2 c1 c0)
b3=f0⊕e0⊕d1⊕c2 =78⊕23⊕5C⊕66 =61 -> DEh=(1) 616BFFD3 (b3 b2 b1 b0)
```

Теперь, когда мы имеем все необходимые значения, вычисляем:

```
X=(1)⊕             a0 =            DE⊕66 = B8
Y=(2)⊕         b0⊕a1 =         F8⊕D3⊕EF = C4
Z=(3)⊕     c0⊕b1⊕a2 =     4F⊕2E⊕FF⊕CD = 53
W=(4)⊕ d0⊕c1⊕b2⊕a3 = 35⊕01⊕7A⊕6B⊕AB = 8E
```

Вывод: чтобы изменить значение CRC-32 с "ABCDEF66" на "56331478" необходима последовательность байтов "B8 C4 53 8E".

Алгоритм восстановления CRC-32

Если взглянуть на ручные расчеты необходимой последовательности байтов для изменения значения CRC с "a3 a2 a1 a0" на "f3 f2 f1 f0", то их перевод в простой и компактный алгоритм покажется достаточно сложным.

Рассмотрим еще раз обобщенную версию окончательных расчетов:

```
                                                Позиция
X  = (1) ⊕                              a0      0
Y  = (2) ⊕                      b0 ⊕    a1      1
Z  = (3) ⊕              c0 ⊕    b1 ⊕    a2      2
W  = (4) ⊕ d0 ⊕         c1 ⊕    b2 ⊕    a3      3
f0 =  e0 ⊕ d1 ⊕         c2 ⊕    b3              4
f1 =  e1 ⊕ d2 ⊕         c3                      5
f2 =  e2 ⊕ d3                                   6
f3 =  e3                                        7
```

(Схема 5)

Это очень похоже на схему 4, добавлены лишь некоторые значения, что, однако, поможет нам составить понятный алгоритм. Возьмем буфер длиной 8 байт, по одному байту на каждую строку схемы 5. Заполним байты с 0 по 3 значениями a0-a3, а байты с 4 по 7 — f0-f3. Как и прежде, возьмем значение e3, которое равно f3, и поищем полную последовательность в таблице CRC, а затем выполним сложение этого значения по XOR с регистром, начиная с позиции 4 (как на схеме 5). Автоматически получаем значение d3 — мы выполнили "f3 f2 f1 f0" XOR "e3 e2 e1 e0", а f2⊕e2=d3. Так как мы знаем, чему равно значение (4) (это номер позиции для e3 e2 e1 e0), то сложим его по XOR со значением байта 3. Зная значение d3, поищем полное значение d3 d2 d1 d0, и повторим сложение по XOR, но на одну позицию правее, то есть с позиции 3 (следите по схеме!). Номер позиции для d3 d2 d1 d0 (значение (3)) складывается со значением в позиции 2. В позиции 5 получаем величину c3, так как "f1⊕e1⊕d2=c3".

Продолжим работу до операции XOR последовательности b3 b2 b1 b0 в позиции 1. Все! Теперь байты 0-3 буфера содержат искомые значение X-W!

Обобщим только что описанный алгоритм:

1. В 8-байтном буфере заполним позиции 0-3 значениями a0-a3 (начальные значения байтов CRC), а позиции 4-7 значениями f0-f3 (байты CRC, которые необходимо получить в результате).
2. Возьмем значение из позиции 7 и используем его для поиска в таблице полного значения.
3. Сложим его (dword) по XOR с содержимым буфера, начиная с позиции 4.
4. Номер позиции в таблице, в которой содержалось обнаруженное значение, сложим по XOR с содержимым позиции 3.
5. Повторим шаги 2-4, каждый раз уменьшая номер позиции в буфере на 1.

Реализация алгоритма восстановления

Теперь самое время обсудить реализацию алгоритма. Ниже приведен код алгоритма восстановления CRC-32 на языке ассемблера (несложно модифицировать его для других языков программирования или иных стандартов расчета CRC). Обратите внимание, что для процессоров фирмы Intel запись и чтение двойных слов происходит в обратном порядке.

```
crcBefore       dd (?)
wantedCrc       dd (?)
buffer          db 8 dup (?)

        mov     eax, dword ptr[crcBefore] ;/*
        mov     dword ptr[buffer], eax
        mov     eax, dword ptr[wantedCrc] ; шаг 1
        mov     dword ptr[buffer+4], eax  ;*/

        mov     di, 4
 computeReverseLoop:
        mov     al, byte ptr[buffer+di+3] ;/*
        call   GetTableEntry              ; шаг 2 */
        xor     dword ptr[buffer+di], eax ; шаг 3
        xor     byte ptr[buffer+di-1], bl ; шаг 4
        dec     di                        ;/*
        jnz    computeReverseLoop         ; шаг 5 */
```

Замечание: Используются регистры eax, di и bx.

Реализация процедуры GetTableEntry

```
crctable        dd 256 dup (?)       ;должна быть глобальной и, естественно,
                                     ; изначально инициализированной
        mov     bx, offset crctable-1
 getTableEntryLoop:
        add     bx, 4                ;указатель (crctable-1)+k*4 (k:1..256)
        cmp     [bx], al             ;значение обязательно будет найдено
        jne     getTableEntryLoop
        sub     bx, 3
        mov     eax, [bx]
        sub     bx, offset crctable
        shr     bx, 2
        ret
```

На выходе из процедуры регистр eax будет иметь полное содержимое из позиции в таблице, а регистр bx — номер этой позиции.

Заключение

Ну вот мы и добрались до конца этой статьи. Если Вам кажется, что теперь все эти программы, защищенные с помощью CRC должны "поднять лапки к верху", то должен Вас разочаровать. Очень легко сделать анти-анти-CRC код. Для правильного восстановления CRC требуется точно знать, какая часть кода проверяется по CRC, и какой алгоритм при этом используется. Простейшей мерой защиты будет

использование 2 различных методов расчета, или комбинирование его с другой системой защиты.

Тем не менеес Я надеюсь, что чтение этого бреда было для Вас интересным, и что Вы получили от этого такое же удовольствие, каким для меня было удовольствие его написать.

Выражаю большую благодарность бета-тестерам Douby/DREAD и Knotty Dread за полезные комментарии, которые позволили улучшить эту статью.

Пример программы коррекции CRC-32 можно найти на моем сайте http://surf.to/anarchriz -> Programming -> Projects (она, правда, все еще находится в состоянии бета-версии, однако, общие идеи в ней почерпнуть Вы все-таки сможете).

Дополнительную информацию по группе DREAD можно найти на сайте http://dread99.cjb.net.

Если у Вас все еще останутся вопросы, то Вы можете связаться со мной по электронной почте anarchriz@hotmail.com, или на каналах EFnet (IRC) #DreaD, #Win32asm, #C.I.A и #Cracking4Newbies (проверяйте их именно в этой последовательности).

CYA ALL! - Anarchriz

"The system makes its morons, then despises them for their ineptitude, and rewards its 'gifted few' for their rarity." - Colin Ward

("Система всегда плодит неугодных себе, презирает их за их безрассудство, и вознаграждает одареннейших из них за их исключительность." — Колин Уорд.)

Приложение

CRC-16 Table

```
00h   0000 C0C1 C181 0140 C301 03C0 0280 C241
08h   C601 06C0 0780 C741 0500 C5C1 C481 0440
10h   CC01 0CC0 0D80 CD41 0F00 CFC1 CE81 0E40
18h   0A00 CAC1 CB81 0B40 C901 09C0 0880 C841

20h   D801 18C0 1980 D941 1B00 DBC1 DA81 1A40
28h   1E00 DEC1 DF81 1F40 DD01 1DC0 1C80 DC41
30h   1400 D4C1 D581 1540 D701 17C0 1680 D641
38h   D201 12C0 1380 D341 1100 D1C1 D081 1040

40h   F001 30C0 3180 F141 3300 F3C1 F281 3240
48h   3600 F6C1 F781 3740 F501 35C0 3480 F441
50h   3C00 FCC1 FD81 3D40 FF01 3FC0 3E80 FE41
58h   FA01 3AC0 3B80 FB41 3900 F9C1 F881 3840

60h   2800 E8C1 E981 2940 EB01 2BC0 2A80 EA41
68h   EE01 2EC0 2F80 EF41 2D00 EDC1 EC81 2C40
70h   E401 24C0 2580 E541 2700 E7C1 E681 2640
78h   2200 E2C1 E381 2340 E101 21C0 2080 E041

80h   A001 60C0 6180 A141 6300 A3C1 A281 6240
88h   6600 A6C1 A781 6740 A501 65C0 6480 A441
90h   6C00 ACC1 AD81 6D40 AF01 6FC0 6E80 AE41
98h   AA01 6AC0 6B80 AB41 6900 A9C1 A881 6840

A0h   7800 B8C1 B981 7940 BB01 7BC0 7A80 BA41
A8h   BE01 7EC0 7F80 BF41 7D00 BDC1 BC81 7C40
B0h   B401 74C0 7580 B541 7700 B7C1 B681 7640
B8h   7200 B2C1 B381 7340 B101 71C0 7080 B041
```

```
C0h   5000 90C1 9181 5140 9301 53C0 5280 9241
C8h   9601 56C0 5780 9741 5500 95C1 9481 5440
D0h   9C01 5CC0 5D80 9D41 5F00 9FC1 9E81 5E40
D8h   5A00 9AC1 9B81 5B40 9901 59C0 5880 9841

E0h   8801 48C0 4980 8941 4B00 8BC1 8A81 4A40
E8h   4E00 8EC1 8F81 4F40 8D01 4DC0 4C80 8C41
F0h   4400 84C1 8581 4540 8701 47C0 4680 8641
F8h   8201 42C0 4380 8341 4100 81C1 8081 4040
```

CRC-32 Table

```
00h   00000000 77073096 EE0E612C 990951BA
04h   076DC419 706AF48F E963A535 9E6495A3
08h   0EDB8832 79DCB8A4 E0D5E91E 97D2D988
0Ch   09B64C2B 7EB17CBD E7B82D07 90BF1D91

10h   1DB71064 6AB020F2 F3B97148 84BE41DE
14h   1ADAD47D 6DDDE4EB F4D4B551 83D385C7
18h   136C9856 646BA8C0 FD62F97A 8A65C9EC
1Ch   14015C4F 63066CD9 FA0F3D63 8D080DF5

20h   3B6E20C8 4C69105E D56041E4 A2677172
24h   3C03E4D1 4B04D447 D20D85FD A50AB56B
28h   35B5A8FA 42B2986C DBBBC9D6 ACBCF940
2Ch   32D86CE3 45DF5C75 DCD60DCF ABD13D59

30h   26D930AC 51DE003A C8D75180 BFD06116
34h   21B4F4B5 56B3C423 CFBA9599 B8BDA50F
38h   2802B89E 5F058808 C60CD9B2 B10BE924
3Ch   2F6F7C87 58684C11 C1611DAB B6662D3D

40h   76DC4190 01DB7106 98D220BC EFD5102A
44h   71B18589 06B6B51F 9FBFE4A5 E8B8D433
48h   7807C9A2 0F00F934 9609A88E E10E9818
4Ch   7F6A0DBB 086D3D2D 91646C97 E6635C01

50h   6B6B51F4 1C6C6162 856530D8 F262004E
54h   6C0695ED 1B01A57B 8208F4C1 F50FC457
58h   65B0D9C6 12B7E950 8BBEB8EA FCB9887C
5Ch   62DD1DDF 15DA2D49 8CD37CF3 FBD44C65

60h   4DB26158 3AB551CE A3BC0074 D4BB30E2
64h   4ADFA541 3DD895D7 A4D1C46D D3D6F4FB
68h   4369E96A 346ED9FC AD678846 DA60B8D0
6Ch   44042D73 33031DE5 AA0A4C5F DD0D7CC9

70h   5005713C 270241AA BE0B1010 C90C2086
74h   5768B525 206F85B3 B966D409 CE61E49F
78h   5EDEF90E 29D9C998 B0D09822 C7D7A8B4
7Ch   59B33D17 2EB40D81 B7BD5C3B C0BA6CAD

80h   EDB88320 9ABFB3B6 03B6E20C 74B1D29A
84h   EAD54739 9DD277AF 04DB2615 73DC1683
88h   E3630B12 94643B84 0D6D6A3E 7A6A5AA8
8Ch   E40ECF0B 9309FF9D 0A00AE27 7D079EB1

90h   F00F9344 8708A3D2 1E01F268 6906C2FE
94h   F762575D 806567CB 196C3671 6E6B06E7
98h   FED41B76 89D32BE0 10DA7A5A 67DD4ACC
9Ch   F9B9DF6F 8EBEEFF9 17B7BE43 60B08ED5
```

```
A0h   D6D6A3E8 A1D1937E 38D8C2C4 4FDFF252
A4h   D1BB67F1 A6BC5767 3FB506DD 48B2364B
A8h   D80D2BDA AF0A1B4C 36034AF6 41047A60
ACh   DF60EFC3 A867DF55 316E8EEF 4669BE79

B0h   CB61B38C BC66831A 256FD2A0 5268E236
B4h   CC0C7795 BB0B4703 220216B9 5505262F
B8h   C5BA3BBE B2BD0B28 2BB45A92 5CB36A04
BCh   C2D7FFA7 B5D0CF31 2CD99E8B 5BDEAE1D

C0h   9B64C2B0 EC63F226 756AA39C 026D930A
C4h   9C0906A9 EB0E363F 72076785 05005713
C8h   95BF4A82 E2B87A14 7BB12BAE 0CB61B38
CCh   92D28E9B E5D5BE0D 7CDCEFB7 0BDBDF21

D0h   86D3D2D4 F1D4E242 68DDB3F8 1FDA836E
D4h   81BE16CD F6B9265B 6FB077E1 18B74777
D8h   88085AE6 FF0F6A70 66063BCA 11010B5C
DCh   8F659EFF F862AE69 616BFFD3 166CCF45

E0h   A00AE278 D70DD2EE 4E048354 3903B3C2
E4h   A7672661 D06016F7 4969474D 3E6E77DB
E8h   AED16A4A D9D65ADC 40DF0B66 37D83BF0
ECh   A9BCAE53 DEBB9EC5 47B2CF7F 30B5FFE9

F0h   BDBDF21C CABAC28A 53B39330 24B4A3A6
F4h   BAD03605 CDD70693 54DE5729 23D967BF
F8h   B3667A2E C4614AB8 5D681B02 2A6F2B94
FCh   B40BBE37 C30C8EA1 5A05DF1B 2D02EF8D
```

Ссылки

"A painless guide to CRC error detection algorithm"
ftp://ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt†

(Ручаюсь, что это "безболезненное руководство" окажется значительно более чувствительным, чем мое "краткое" описание ;)

Кроме того, я пользовался различными случайными источниками, чтобы лучше понять алгоритм расчета CRC-32.

Ссылки на программы расчета CRC??? Попробуйте поискать файлы "CRC.ZIP" или "CRC.EXE", воспользовавшись чем-либо вроде ftpsearch (http://ftpsearch.lycos.com?form=advanced)

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

Перевод и оформление:
Sergey R.

† К сожалению данная ссылка не работает. Статья была найдена по адресу: ftp://www.internode.net.au/clients/rocksoft/papers/crc_v3.txt