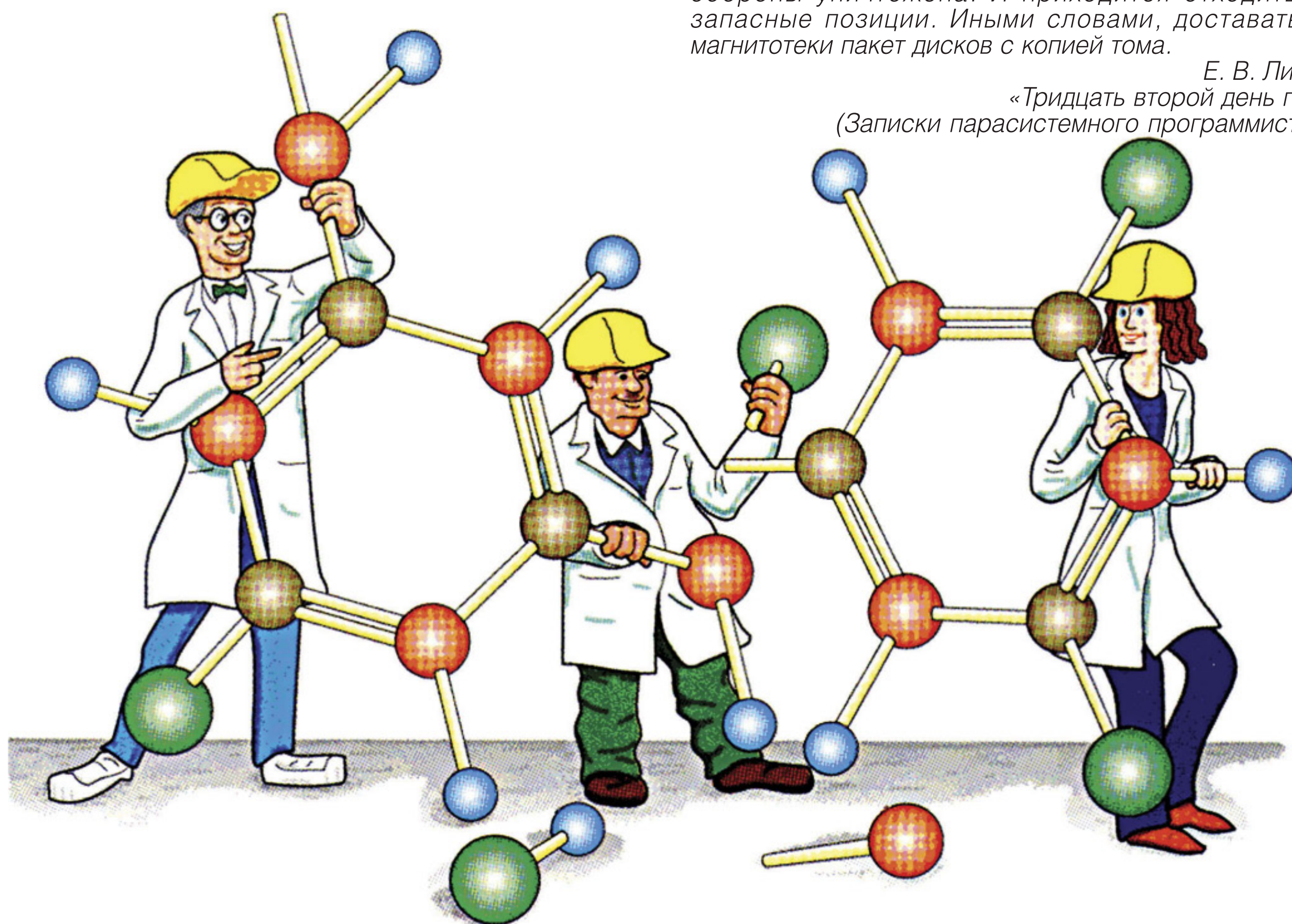
МОГУЩЕСТВО КОДОВ РИДА-СОЛОМОНА

Энтропия слепа, но терпелива. Рано или поздно, обстреливая наши позиции по квадратам, она нанесет удар по штабу, по центру связи. И тогда первая линия обороны уничтожена. И приходится отходить на запасные позиции. Иными словами, доставать из магнитотеки пакет дисков с копией тома.

Е. В. Лишак
«Тридцать второй день года.
(Записки парасистемного программиста).»



ИЛИ ИНФОРМАЦИЯ, ВОСКРЕСШАЯ ИЗ ПЕПЛА

Все вы наверняка слышали о существовании помехозащитных кодов Рида-Соломона, широко использующихся в устройствах передачи и хранения данных для обнаружения и исправления как одиночных, так и групповых (!) ошибок. Область их применения необычайно широка – кодеры/декодеры Рида-Соломона можно найти и в ленточных запоминающих устройствах, и в контроллерах оперативной памяти, и в модемах, и в жестких дисках, и в CD-ROM/DVD-приводах и т. д. Благодаря им некоторые продвинутые архиваторы безболезненно переносят порчу нескольких секторов носителя, содержащего архив, а подчас и полное разрушение целого тома многотомного архива. Еще коды Рида-Соломона позволяют защитному механизму автоматически восстанавливать байтики, хакнутые взломщиком и/или искаженные в результате сбоя программного/аппаратного обеспечения. Короче говоря, если владение техникой помехозащитного кодирования не превращает вас в Бога, то, по крайней мере, поднимает на Олимп, где среди бесшумных вентиляторов и безглючных операционных систем снуют великие компьютерные гуру.

КРИС КАСПЕРСКИ

В то же время лишь немногие программисты могут похвастаться собственной реализацией алгоритмов Рида-Соломона. Да и зачем? Готовых библиотек море: от прагматичных коммерческих пакетов до бесплатных исходников, распространяемых по лицензии GNU. Как говорится, бери – не хочу[1]. Что ж, в использовании библиотек есть вполне определенный практический смысл, но никакой хакер не доверит управление программе до тех пор, пока не поймет, как именно она работает (а эта публикация именно для хакеров и предназначена, естественно, «хакеров» в хорошем значении этого слова).

С другой стороны, при анализе программного обеспечения, распространяемого без исходных кодов, вы не сможете идентифицировать алгоритм Рида-Соломона, если только заранее не разберетесь во всех его тонкостях. Допустим, вам встретилась защита, хитрым образом манипулирующая с EDC/ECC-полями ключевых секторов, считанных ею с лазерного диска, и каждый такой сектор содержит две умышленно внесенные ошибки (плюс еще ошибки, естественным путем возникающие при небрежном обращении с CD), причем одна из этих ошибок ложная и исправлять ее не нужно. При штатном копировании защищенного диска микропроцессорная начинка CD-ROM автоматически исправляет все ошибки, которые она только может исправить, в результате чего происходит искажение ключевых меток и, как следствие, защищенная программа перестанет работать. Можно, конечно, скопировать диск в «сыром» режиме, т.е. без исправления ошибок, но тогда копия будет содержать как непредумышленные, так и предумышленные ошибки, в результате чего даже при незначительном повреждении оригинала корректирующих возможностей кодов Рида-Соломона уже окажется недостаточно, и диск просто перестанет читаться (а как вы хотели? копирование дисков в сыром режиме ведет к накоплению ошибок и потому крайне непрактично с любой точки зрения).

Владение базовыми принципами помехозащитного кодирования позволит вам разобраться с логикой работы защитного механизма и понять, какие конкретные ошибки следует исправлять, а какие нет.

К сожалению, подавляющее большинство публикаций на тему кодов Рида-Соломона написаны на языке высшей математики, для постижения которой и университетских знаний подчас оказывается недостаточно (да и все ли хакеры знают математику?), в результате чего все эти сильно теоретизированные руководства забрасываются на полку.

Программная реализация корректирующих кодов Рида-Соломона действительно очень сложна и действительно требует определенной математической подготовки, изложение основ которой может показаться скучным и неинтересным для «системщиков» и «железячников», но иного пути, видимо, нет. В конце концов никто не обещал вам, что быть программистом – легко, а хорошим программистом быть еще труднее. Так что не говорите потом, что я вас не предупреждал! Шутка! Расслабьтесь и разгоните свой страх перед высшей математикой прочь. По ходу описания вам встретится пара формул (ну куда же в математике без формул?), но во всех остальных случаях я буду говорить на интернациональном программистском языке – языке Си, понятным любому системщику. В общем, пристегивайте ремни и поднимайте свои головы с клавиатуры – мы поехали! Конечной целью нашего путешествия станет построение отказоустойчивого RAID-массива 5 уровня на базе… нескольких обыкновенных IDE/SCSI-контроллеров жестких дисков. 4/5 объема такого массива будут отданы непосредственно под полезные данные, а 1/5 – под избыточную информацию, позволяющую восстановить содержимое любого жесткого диска из пяти данных, даже если он будет полностью уничтожен!

Корректирующие коды и помехоустойчивое кодирование (азы)

Персональные компьютеры с их битами и байтами настолько прочно вошли в нашу жизнь, что программисты вообще перестали задумываться о теории кодирования информации, принимая ее как должное. Между тем, здесь все не так просто, как может показаться на первый взгляд.

Фактически кодирование есть ни что иное, как преобразование сообщения в последовательность кодовых символов, так же называемых кодовыми словами. Любое дискретное сообщение состоит из конечного числа элементов: в частности, текст состоит из букв, изображение состоит из пикселей, машинная программа состоит из команд и т. д., – все они образуют алфавит источника сообщения. При кодировании происходит преобразование элементов сообщения в соответствующие им числа – кодовые символы, причем каждому элементу сообщения присваивается уникальная совокупность кодовых символов, называемая кодовой комбинацией. Совокупность кодовых комбинаций, образующих сообщение, и есть код. Множество возможных кодовых символов называется кодовым алфавитом, а их количество (далее по тексту обозначаемое малой латинской m) – основанием кода.

Впрочем, все это вы уже наверняка знаете (а если не знаете, то без труда найдете исчерпывающее объяснение основ кодирования в любом учебнике по информатике), но знаете ли вы, что такое расстояние Хемминга? Это минимальное количество различий между двумя различными допустимыми кодовыми словами и в теории помехоустойчивого кодирования расстояние Хемминга играет основополагающую роль. Рассмотрим, например, следующий 4-битный код:

Общее представление

Коды Рида-Соломона – недвоичные совершенные систематические линейные блочные коды, относящиеся к классу циклических кодов с числовым полем, отличным от GF(2), и являющиеся подмножеством кодов Боуза-Чоудхури-Хоквингема. Корректирующие способности кодов Рида-Соломона напрямую зависят от количества контрольных байт. Добавление r контрольных байт позволяет обнаруживать r произвольным образом искаженных байт, гарантированно восстанавливая r/2 байт из них.

```
Листинг 1. Пример простейшего 4-битного кода с расстоянием
Хемминга, равным единице. Такой код широко используется в
вычислительной технике, несмотря на его невозможность обна-
ружить ошибки.

0 → 0000; 4 → 0100;    8 → 1000;    12 → 1100;
1 → 0001; 5 → 0101;    9 → 1001;    13 → 1101;
2 → 0010; 6 → 0110;    10 → 1010;   14 → 1110;
3 → 0011; 7 → 0111;    11 → 1011;   15 → 1111;
```

Это обыкновенный двоичный код, который можно встретить в некоторых однокристаллках, вмещающий в свои 4 бита 16 символов (т.е. с его помощью можно закодировать 16 букв алфавита). Как нетрудно убедиться, два любых символа отличаются по меньшей мере на один бит, следовательно, расстояние Хемминга для такого кода равно единице (что условно обозначает как d = 1).

А вот другой 4-битный код:

```
Листинг 2. Пример 4-битного кода с расстоянием Хемминга,
равным двум, способного обнаруживать одиночные ошибки.

0 → 0000; 4 → 1001;
1 → 0011; 5 → 1010;
2 → 0101; 6 → 1100;
3 → 0110; 7 → 1111;
```

На этот раз два произвольных символа отличаются как минимум в двух позициях, за счет чего информационная емкость такого кода сократилась с 16 до 8 символов. «Постойте-постойте! – воскликнет иной читатель. – Что это за бред? Куда делась комбинация 0001 или 0010, например?». Нет, это не бред, и указанных комбинаций бит в данном коде действительно нет, точнее, они есть, но объявлены запрещенными. Благодаря этому обстоятельству наш подопечный код способен обнаруживать любые одиночные ошибки. Возьмем, например, символ «1010» и исказим в нем произвольный бит (но только один!). Пусть это будет второй слева бит, тогда искаженный символ станет выглядеть так: «1**1**10». Поскольку комбинация «1110» является запрещенной, декодер может засвидетельствовать наличие ошибки. Увы, только засвидетельствовать, но не исправить, т.к. для исправления даже одного-единственного сбойного байта требуется увеличить расстояние Хемминга как минимум до трех. Поскольку 4-битный код с d = 3 способен вмещать в себя лишь два различных символа, то он крайне ненагляден и потому нам лучше выбрать код с большей разрядностью. Хорошо, пусть это будет 10-битный код с d = 5.

```
Листинг 3. Пример 10-битного кода, с расстоянием Хемминга,
равным пяти, способного обнаруживать 4-битные ошибки,
а исправлять 2-битные.

0000000000 0000011111    1111100000    1111111111
```

Возьмем, к примеру, символ «0000011111» и искорежим два любых бита, получив в итоге что-то наподобие: «0**1**0011**0**111». Поскольку такая комбинация является запрещенной, декодер понимает, что произошла ошибка. Достаточно очевидно, что если количество сбойных бит меньше расстояния Хемминга хотя бы наполовину, то декодер может гарантированно восстановить исходный символ. Действительно, если между двумя любыми разрешенными символами существует не менее пяти различий, то искажение двух бит всякого такого символа приведет к образованию нового символа (обозначим его k), причем расстояние Хемминга между k и оригинальным символом равно числу непосредственно искаженных бит (т.е. в нашем случае двум), а расстояние до ближайшего соседнего символа равно: d – k (т.е. в нашем случае трем). Другими словами, пока d – k > k декодер может гарантированно восстановить искаженный символ. В тех случаях, когда d > k > d – k, успешное восстановление уже не гарантируется, но при удачном стечении обстоятельств все-таки оказывается в принципе возможным.

Возвращаясь к нашему символу «0000011111», давайте на этот раз исказим не два бита, а четыре: «0**1**00**11**0**1**0**1**» и попробуем его восстановить. Изобразим процесс восстановления графически:

```
Листинг 4. Восстановление 4-битной ошибки.

0000000000 0000011111    1111100000    1111111111
0100110101 0100110101    0100110101    0100110101
---------- ----------    ----------    ----------
5 отличий   4 отличия     6 отличий     5 отличий
```

Грубо говоря, обнаружив ошибку, декодер последовательно сличает искаженный символ со всеми разрешенными символами алфавита, стремясь найти символ наиболее «похожий» на искаженный. Точнее, символ с наименьшим числом различий, а еще точнее, символ, отличающийся от искаженного не более чем в (d – 1) позициях. Легко видеть, что в данном случае нам повезло, и восстановленный символ совпал с истинным. Однако если бы четыре искаженных бита распределились бы так: «0**1111**11111», то декодер принял бы этот символ за «1111111111» и восстановление оказалось бы неверным.

Таким образом, исправляющая способность кода определяется по следующей формуле: для обнаружения r ошибок расстояние Хемминга должно быть больше или равно r, а для коррекции r ошибок расстояние Хемминга должно быть по крайней мере на единицу больше удвоенного количества r.

```
Листинг 5. Корректирующие способности простого кода Хемминга.

обнаружение ошибок:        d ≥ r
исправление ошибок:        d > 2r
информационная емкость:    2^(n/d)
```

Теоретически количество обнаруживаемых ошибок неограничено, практически же информационная емкость кодовых слов стремительно тает с ростом d. Допустим, у нас есть 24 байта данных, и мы хотели бы исправлять до двух ошибок на каждый такой блок. Тогда нам придется добавить к этому блоку еще 49 байт, в результате чего реальная информационная емкость блока сократится всего… до 30%! Хорошенькая перспектива, не так ли? Столь плачевный результат объясняется тем, что биты кодового слова изолированы друг от друга и изменение одного из них никак не сказывается на окружающих. А что если…

Пусть все биты, номера которых есть степень двойки, станут играть роль контрольных битов, а оставшиеся и будут обычными («информационными») битами сообщения. Каждый контрольный бит должен отвечать за четность суммы[2] некоторой принадлежащей ему группы битов, причем один и тот же информационный бит может относиться к различным группам. Тогда один информационный бит сможет влиять на несколько контрольных и потому информационная емкость слова значительно (можно даже сказать,

чудовищно) возрастет. Остается только выбрать наиболее оптимальное разделение сфер влияния.

Согласно методу помехозащитного кодирования, предложенного Хеммингом, для того чтобы определить, какие контрольные биты контролируют информационный бит, стоящий в позиции k, мы должны разложить k по степеням двойки:

Таблица 1. Разделение бит на контрольные и информационные.

позиция	какими битами контролируется	
1 (A)	$2^0 = 1$	это контрольный бит, никто его не контролирует
2 (B)	$2^1 = 2$	это контрольный бит, никто его не контролирует
3	$2^0+2^1 = 1+2 = 3$	контролируется 1 и 2 контрольными битами
4 (C)	$2^2 = 4$	это контрольный бит, никто его не контролирует
5	$2^0+2^2 = 1+4 = 5$	контролируется 1 и 4 контрольными битами
6	$2^1+2^2 = 2+4 = 6$	контролируется 2 и 4 контрольными битами
7	$2^0+2^1+2^2 = 1+2+4 = 7$	контролируется 1, 2 и 4 контрольными битами
8 (D)	$2^3 = 8$	это контрольный бит, никто его не контролирует

Давайте в порядке закрепления материала попробуем пощупать коды Хемминга «вживую» и вручную рассчитаем контрольную сумму 4-битного символа «0101». После резервирования «квартир» для контрольных битов (выделенных в тексте жирным шрифтом) наш символ будет выглядеть так: **AB**0**C**101**D**. Теперь остается только рассчитать значения битов A, B, C и D:

- Бит A, контролирующий биты 3, 5 и 7 равен нулю, т.к. их сумма (0 + 1 + 1) четна.
- Бит B, контролирующий биты 3, 6 и 7 равен одному, т.к. их сумма (0 + 0 + 1) нечетна.
- Бит C, контролирующий биты 5, 6 и 7 равен нулю, т.к. их сумма (1 + 0 + 1) четна.
- Бит D никакой роли не играет, т.к. он контролирует лишь те биты, которые расположены справа от него, а никаких битов справа от него уже и нет. И приведен он лишь затем, чтобы получить 8 бит – общепринятый байт.

Таким образом, «новоиспеченное» кодовое слово будет выглядеть так: «**01**0**0**101», где жирным шрифтом выделены контрольные биты.

```
Листинг 6. Кодовое слово вместе с информационными битами.

AB0C101D
12345678
```

Допустим, при передаче наше слово было искажено в одной позиции и стало выглядеть так: **0100**111. Сможем ли мы обнаружить такую ошибку? А вот сейчас и проверим! Так, бит A должен быть равен: (0 + 1 + 1) % 2 = 0, что соответствует истине. Бит B должен быть равен (0 + 1 + 1) % 2 = 0, а в нашем слове он равен единице. Запомним номер «неправильного» контрольного бита и продолжим. Бит C должен быть равен (1 + 1 + 1) % 2 = 1, а он равен нулю! Ага, значит, контрольные биты в позициях 2 (бит B) и 4 (бит C) обнаруживают расхождение с действительностью. Их сумма (2 + 4 = 6) и дает позицию сбойного бита. Действительно, в данном случае номер искаженного бита равен 6, инвертируем его, тем самым восстанавливая наше кодовое слово в исходный вид.

А что если искажение затронет не информационный, а контрольный бит? Проверка показывает, что позиция ошибки успешно обнаруживается и в этом случае, и контрольный бит при желании может быть легко восстановлен по методике, уже описанной выше (только есть ли в этом смысл? ведь контрольные биты все равно «выкусываются» в процессе декодирования кодового слова).

На первый взгляд кажется, что коды Хемминга жутко неэффективны, ведь на 4 информационных бита у нас приходится 3 контрольных, однако поскольку номера контрольных бит представляют собой степень двойки, то с ростом разрядности кодового слова они начинают располагаться все реже и реже. Так, ближайший к биту C контрольный бит D находится в позиции 8 (т.е. в трех шагах), зато контрольный бит E отделен от бита D уже на (2^4 - 2^3 - 1) = 7 «шагов», а контрольный бит F и вовсе на (2^5 - 2^4 - 1) = 15 «шагов».

Таким образом, с увеличением разрядности обрабатываемого блока, эффективность кодов Хемминга стремительно нарастает, что и показывает следующая программа:

```
Листинг 7. Расчет эффективной информационной емкости кодов
Хемминга для слов различной длины.

main()
{
    int a;
    int _pow = 1;
    int old_pow = 1;
    int N, old_N = 1;

    printf( "* * * hamming code efficiency test * * * ↵
            by Kris Kaspersky\n"\
            " BLOCK_SIZE    FUEL UP    EFFICIENCY\n"\
            "-----------------------------------\n");
    for (a = 0; a < MAX_POW; a++)
    {
            N = _pow - old_pow - 1 + old_N;

            printf("%8d    %8d    %8.1f%%\n",_pow, N, (float) ↵
                    N/_pow*100);

            // NEXT
            old_pow = _pow; _pow = _pow * 2; old_N = N;

    } printf("-----------------------------------\n");
}
```

```
Листинг 8. Результат расчета эффективной информационной
емкости кодов Хемминга для слов различной длины.

BLOCK_SIZE     FUEL UP    EFFICIENCY
-----------------------------------
         1           0        0.0%
         2           0        0.0%
         4           1       25.0%
         8           4       50.0%
        16          11       68.8%
        32          26       81.3%
        64          57       89.1%
       128         120       93.8%
       256         247       96.5%
       512         502       98.0%
      1024        1013       98.9%
      2048        2036       99.4%
      4096        4083       99.7%
      8192        8178       99.8%
     16384       16369       99.9%
     32768       32752      100.0%
     65536       65519      100.0%
    131072      131054      100.0%
    262144      262125      100.0%
    524288      524268      100.0%
-----------------------------------
```

Из приведенной выше распечатки видно, что при обработке блоков, дотягивающихся хотя бы до 1024 бит, накладными расходами на контрольные биты можно полностью пренебречь.

К сожалению, коды Хемминга способны исправлять лишь одиночные ошибки, т.е. допускают искажение всего

лишь одного сбойного бита на весь обрабатываемый блок. Естественно, с ростом размеров обрабатываемых блоков увеличивается и вероятность ошибок. Поэтому выбор оптимальной длины кодового слова является весьма нетривиальной задачей, как минимум требующей знания характера и частоты возникновения ошибок используемых каналов передачи информации. В частности, для ленточных накопителей, лазерных дисков, винчестеров и тому подобных устройств коды Хемминга оказываются чрезвычайно неэффективными. Зачем же тогда мы их рассматривали? А затем, что понять прогрессивные системы кодирования (к которым в том числе относятся и коды Рида-Соломона), ринувшись атаковать их «с нуля», практически невозможно, ибо они завязаны на сложной, действительно высшей математике, но ведь не Боги горшки обжигают, верно?

Идея кодов Рида-Соломона

Если говорить упрощенно, то основная идея помехозащитного кодирования Рида-Соломона заключается в умножении информационного слова, представленного в виде полинома D, на неприводимый полином G^3, известный обеим сторонам, в результате чего получается кодовое слово C, опять-таки представленное в виде полинома.

Декодирование осуществляется с точностью до наоборот: если при делении кодового слова C на полином G декодер внезапно получает остаток, то он может рапортовать наверх об ошибке. Соответственно, если кодовое слово разделилось нацело, его передача завершилась успешно.

Если степень полинома G (называемого так же порождающим полиномом) превосходит степень кодового слова по меньшей мере на две степени, то декодер может не только обнаруживать, но и исправлять одиночные ошибки. Если же превосходство степени порождающего полинома над кодовым словом равно четырем, то восстановлению поддаются и двойные ошибки. Короче говоря, степень полинома k связана с максимальным количеством исправляемых ошибок t следующим образом: $k = 2*t$. Следовательно, кодовое слово должно содержать два дополнительных символа на одну исправляемую ошибку. В то же время максимальное количество распознаваемых ошибок равно t, т.е. избыточность составляет один символ на каждую распознаваемую ошибку.

В отличие от кодов Хемминга, коды Рида-Соломона могут исправлять любое разумное количество ошибок при вполне приемлемом уровне избыточности. Спрашиваете, за счет чего это достигается? Смотрите, в кодах Хемминга контрольные биты контролировали лишь те информационные биты, что находятся по правую сторону от них и игнорировали всех «левосторонних» товарищей. Обратимся к таблице 1: добавление восьмого контрольного бита D ничуть не улучшило помехозащищенность кодирования, поскольку контрольному биту D было некого контролировать. В кодах же Рида-Соломона контрольные биты распространяют свое влияние на все информационные биты и потому с увеличением количества контрольных бит увеличивается и количество распознаваемых/устраняемых ошибок. Именно благодаря последнему обстоятельству, собственно, и вызвана ошеломляющая популярность корректирующих кодов Рида-Соломона.

Теперь о грустном. Для работы с кодами Рида-Соломона обычная арифметика, увы, не подходит и вот почему. Кодирование предполагает вычисления по правилам действия над многочленами, с коэффициентами которых надо выполнять операции сложения, вычитания, умножения и деления, причем все эти действия не должны сопровождаться каким-либо округлением промежуточных результатов (даже при делении!), чтобы не вносить неопределенность. Причем и промежуточные, и конечные результаты не имеют права выходить за пределы установленной разрядной сетки… «Постойте! – воскликнет внимательный читатель. – Да ведь это невозможно! Чтобы при умножении и не происходило «раздувания» результатов, кто же в этот бред поверит?!»

Впрочем, если как следует подумать головой, частично призвав на помощь и другие части тела, можно сообразить, что умножать информационное слово на порождающий полином вовсе не обязательно, можно поступить гораздо хитрее:

- Добавляем к исходному информационному слову D справа k нулей, в результате чего у нас получается слово длины $n = m + r$ и полином X^r*D, где m – длина информационного слова.
- Делим полученный полином X^r*D на порождающий полином G и вычисляем остаток от деления R, такой что: $X^r*D = G*Q + R$, где Q – частное, которое мы благополучно игнорируем за ненадобностью, – сейчас нас интересует только остаток.
- Добавляем остаток R к информационному слову D, в результате чего получаем симпатичное кодовое слово C, информационные биты которых хранятся отдельно от контрольных бит. Собственно, тот остаток, который мы получили в результате деления, и есть корректирующие коды Рида-Соломона. Между нами говоря, способ кодирования, при котором информационные и контрольные символы хранятся раздельно, называется систематическим кодированием и такое кодирование весьма удобно с точки зрения аппаратной реализации.
- Мысленно прокручиваем предыдущие пункты, пытаясь обнаружить, на какой же стадии вычислений происходит выход за разрядную сетку и… такой стадии нет! Остается лишь отметить, что информационное слово + корректирующие коды можно записать как: $T = X^r*D + R = G*Q$.

Декодирование полученного слова T осуществляется точно так же, как уже и было описано ранее. Если при делении T (которое в действительности является произведением G на Q) на порождающий полином G образуются остаток, то слово T искажено и, соответственно, наоборот.

Теперь вопрос на засыпку. Как вы собираетесь осуществлять деление полиномов в рамках общепринятой алгебры? В целочисленной арифметике деление определено не для всех пар чисел (вот, в частности, 2 нельзя разделить на 3, а 9 нельзя разделить на 4, без потери значимости, естественно). Что же касается «плавучки», то ее точность еще та (в смысле точность катастрофически недостаточная для эффективного использования кодов Рида-Соломона), к тому же она достаточно сложна в аппаратной реализации. Ладно, в IBM PC с процессором Pentium быстродействующий математический сопроцессор всем нам дан

по дефолту, но что делать разработчикам ленточных накопителей, винчестеров, CD-приводов, наконец? Пихать в них четвертый Пень?! Нет уж, увольте, лучше воспользоваться специальной арифметикой – арифметикой конечных групп, называемых полями Галуа. Достоинство этой арифметики в том, что операции сложения, вычитания, умножения и деления определены для всех членов поля (естественно, исключая ситуацию деления на ноль), причем число, полученное в результате любой из этих операций, обязательно присутствует в группе! Т.е. при делении любого целого числа A, принадлежащего множеству 0…255, на любое целое число B из того же множества (естественно, B не должно быть равно нулю), мы получим число C, входящее в данное множество. А потому потерь значимости не происходит, и никакой неопределенности не возникает!

Таким образом, корректирующие коды Рида-Соломона основаны на полиномиальных операциях в полях Галуа и требуют от программиста владения сразу несколькими аспектами высшей математики из раздела теории чисел. Как и все «высшее», придуманное математиками, поля Галуа есть абстракция, которую невозможно ни наглядно представить, ни «пощупать» руками. Ее просто надо принять как набор аксиом, не пытаясь вникнуть в его смыл, достаточно всего лишь знать, что она работает, вот и все. А еще есть полиномы немеряных степеней и матрицы в пол-Европы, от которых нормальный системщик не придет в восторг (увы, программист-математик скорее исключение, чем правило).

Поэтому, прежде чем ринуться в непроходимые джунгли математического леса абстракций, давайте сконструируем макет кодера/декодера Рида-Соломона, работающий по правилам обычной целочисленной алгебры. Естественно, за счет неизбежного в этом случае расширения разрядной сетки такому кодеру/декодеру будет очень трудно найти практическое применение, но… зато он нагляден и позволяет не только понять, но и почувствовать принцип работы корректирующих кодов Рида-Соломона.

Мы будем исходить из того, что если $g = 2^n + 1$, то для любого a из диапазона $0…2^n$, произведение $a*g = c$ (где c – кодовое слово), будет представлять, по сути, полную мешанину битов обоих исходных чисел.

Допустим $n = 2$, тогда $g = 3$. Легко видеть: на что бы мы не умножали g – хоть на 0, хоть на 1, хоть на 2, хоть на – 3, полученный результат делится нацело на g в том и только том случае, если никакой из его бит не инвертирован (то есть, попросту говоря, одиночные ошибки отсутствуют).

Остаток от деления однозначно указывает на позицию ошибки (при условии, что ошибка одиночная, групповые же ошибки данный алгоритм исправлять не способен). Точнее, если ошибка произошла в позиции x, то остаток от деления k будет равен $k = 2^x$. Для быстрого определения x по k можно воспользоваться тривиальным табличным алгоритмом. Впрочем, для восстановления сбойного бита знать его позицию совершенно необязательно, достаточно сделать R = e ^ k, где e – искаженное кодовое слово, ^ – операция XOR, а R – восстановленное кодовое слово.

В общем, законченная реализация кодера/декодера может выглядеть так:

Листинг 9. Простейший пример реализации кодера/декодера Рида-Соломона, работающего по обычной арифметике (т.е. с неоправданным расширением разрядной сетки), и исправляющим любые одиночные ошибки в одном 8-битном информационном слове (впрочем, программу легко адаптировать и под 16-байтовые информационные слова). Обратите внимание, что кодер реализуется чуть ли не на порядок проще декодера. В настоящем декодере Рида-Соломона, способном исправлять групповые ошибки, этот разрыв еще значительнее.

```
// ВНИМАНИЕ! данный кодер/декодер построен на основе
// обычной арифметики, не арифметики полей Галуа,
// в результате чего его практические возможности более
// чем ограничены, тем не менее он нагляден и удобен для
// изучения
#include <stdio.h>

// ширина входного информационного символа (бит)
#define SYM_WIDE  8

// входные данные (один байт)
#define DATAIN    0x69

// номер бита, который будет разрушен сбоем
#define ERR_POS   3

// неприводимый полином
#define MAG (1<<(SYM_WIDE*1) + 1<<(SYM_WIDE*0))

// -------------------------------------------------------
// определение позиции ошибки x по остатку k от деления
// кодового слова на полином k = 2^x, где "^" - возведение
// в степень; функция принимает k и возвращает x
// -------------------------------------------------------
int pow_table[9] = {1,2,4,8,16,32,64,128,256};
lockup(int x) {int a;for(a=0;a<9;a++) ↲
    if(pow_table[a]==x)return a; return -1;}

main()
{
    int i; int g; int c; int e; int k;

    fprintf(stderr,"simplest Reed-Solomon endoder/decoder ↲
          by Kris Kaspersky\n\n");
    // входные данные (информационное слово)
    i = DATAIN;
    // неприводимый полином
    g = MAG;
    printf("i = %08x      DATAIN)\ng = %08x     ↲
          (POLYNOM)\n", i, g);

    // КОДЕР РИДА-СОЛОМОНА (простейший, но все-таки кое-как
    // работающий).
    // Вычисляем кодовое слово, предназначенное для передачи
    c = i * g;    printf("c = %08x     (CODEWORD)\n", c);
    // конец КОДЕРА

    // передаем с искажениями
    e = c ^ (1<<ERR_POS); printf("e = %08x     ↲
          (RAW RECIVED DATA+ERR)\n\n", e);
    /* ^^^^ искажаем один бит, имитируя ошибку передачи */

    // ДЕКОДЕР РИДА-СОЛОМОНА
    // проверяем на наличие ошибок передачи
    // (фактически это простейший декодер Рида-Соломона)
    if (e % g)
    {
          // ошибки обнаружены, пытаемся исправить
          printf("RS decoder says: (%x) ↲
                error detected\n{\n", e % g);
          // k = 2^x, где x - позиция сбойного бита
          k = (e % g);
          printf("\t0 to 1 err  position: %x\n", lockup(k));
          printf("\trestored codeword is: %x\n}\n", (e ^= k));
    }
    printf("RECEIVED DATA IS: %x\n", e / g);
    // КОНЕЦ ДЕКОДЕРА
}
```

Листинг 10. Результат работы простейшего кодера/декодера Рида-Соломона. Обратите внимание: искаженный бит удалось успешно исправить, однако для этого к исходному информационному слову пришлось добавить не два, а целых три бита (если вы возьмете в качестве входного слова максимально допустимое 8-битное значение 0xFF, то кодовое слово будет равно 0x1FE00, а так как 2^{10} = 10000, то свободных разрядов уже не хватает и приходится увеличивать разрядную сетку до 2^{11}, в то время как младшие биты кодового слова фактически остаются незадействованными и "пра-

```
вильный" кодер должен их "закольцевать", грубо говоря замкнув
обрабатываемые разряды на манер кольца.

   i = 00000069     (DATAIN)
   g = 00000200     (POLYNOM)
   c = 0000d200     (CODEWORD)
   e = 0000d208     (RAW RECIVED DATA+ERR)

   RS decoder says: (8) error detected
   {
       0 to 1 err  position: 3
       restored codeword is: d200
   }
   RECEIVED DATA IS: 69
```

Заключение

Вот мы и познакомились с азами кодирования информации и разобрались с основными идеями, лежащими в основе построения помехозащитных кодов. Следующая статья этого цикла подробно расскажет о превратностях полиномиальной арифметики и (о ужас!) полях Галуа. Затем, на фундаменте данного математического аппарата, мы сможем возвести дворец отказоустойчивых RAID-систем, да и не только их…

[1] «…Из-за ошибок в реализации данный код вместо исправления ошибок добавляет новые. Поэтому данный код больше недоступен» – комментарий к исходным текстам GNU кодера/декодера Reed-Solomon. Вот и верь после этого в надежность Linux в целом и в GNU'тый библиотечный код в частности.

[2] Если сумма проверяемых бит четна, то контрольный бит равен нулю и, соответственно, наоборот.

[3] Полином, который не разлагается в произведение полиномов меньшей степени.

Что читать

Несмотря на то что данный цикл статей является вполне самодостаточным и весь минимально необходимый математический аппарат излагает самостоятельно без отсылок к сторонней литературе, желание углубить свои знания вполне естественно и его можно только приветствовать. А потому будет лучше, если вы не ограничитесь одной этой статьей, а перевернете целые горы специализированной литературы, с каждым разом все больше и больше ужасаясь глубине той пропасти, что отделяет ваши поверхностные представления от действительно настоящих знаний. Теория помехоустойчивого кодирования столь обширна, что для ее изучения потребуется как минимум целая жизнь.

1. Blahut Richard. Theory and Practice of Error Control Codes. Mass.: Addison-Wesley, 1983. – Очень хорошая книжка из категории «must have»; по слухам, есть в электронном виде в сети, однако, к сожалению, самой книжки я так и не нашел, но тучи ссылок на нее убедительно свидетельствуют о высоком качестве последней. Также имеется ее русскоязычный перевод, выпущенный издательством «Мир» (см. ниже).
2. Блейхут Р. Теория и практика кодов, контролирующих ошибки. М.: Мир, 1986. – 576с. – Технически грамотный и добротный перевод уже упомянутой выше книги Блейхута (ах, какие в издательстве Мир были переводчики!), электронной копии в сети, к сожалению, нет.
3. James Plank. A tutorial on Reed-Solomon Coding for fault-tolerance in RAID-like systems. – Неплохое руководство по использованию кодов Рида-Соломона для построения отказоустойчивых RAID-подобных систем, ориентированное на математически неподготовленных системных программистов и доходчиво объясняющее суть помехоустойчивого кодирования с примерами исходных текстов на Си. Электронная копия доступна по адресу: http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.pdf. Настоятельно рекомендую прочитать, даже если вы и не собираетесь заниматься сборкой RAID.
4. Joel Sylvester. Reed Solomon Codes. – Предельно краткое описание принципов работы кодов Рида-Соломона с блок-схемами вместо исходных текстов. На практическое руководство не тянет, но общую картину все-таки дает, почитайте: http://www.elektrobit.co.uk/pdf/reedsolomon.pdf.
5. Tom Moore. REED-SOLOMON PACKAGE. (old tutorial). – Роскошный сборник разнообразных руководств по кодам Рида-Соломона, наверное, лучший из всех, что я видел. Включает в себя кратное описание основ теории полей Галуа, базовые принципы построения кодеров/декодеров Рида-Соломона и законченные примеры реализации самих кодеров/декодеров на языке Си (правда, недостаточно добросовестно прокомментированные). Сей stuff неоднократно промелькивал в ФИДО и последний раз постился 28 декабря 1994 года в конференции comp.compression. Его легко найти в «Google» по ключевым словам «Reed-Solomon+main+ECC». Настоятельно рекомендую.
6. Ross N.Williams. A painless guide to CRC error detection algorithms. – Подробное руководство по CRC, полезное достаточно внятным и доступным описанием полиномиальной арифметики, без которой работа с кодами Рида-Соломона просто немыслима. Доступно в электронной форме по следующему адресу: ftp://www.internode.net.au/clients/rocksoft/papers/crc_v3.txt. Также имеется его неплохой перевод на русский язык, легко отыскивающийся в сети по запросу «Элементарное руководство по CRC-алгоритмам обнаружения ошибок». Настоятельно рекомендую.
7. ftape (драйвер ленточного накопителя из дистрибутива Linux). – Ну какая же запись на магнитную ленту обходится без корректирующих кодов? Представить себе такое, довольно затруднительно. Поэтому анализ исходных текстов драйверов ленточных накопителей дает довольно-таки богатую пищу для размышлений (при условии, конечно, что исследуемый драйвер действительно использует коды Рида-Соломона, а не что-нибудь другое). Драйвер ftape как раз и является тем драйвером, что вам нужен, а непосредственно сам код, ответственный за кодирование/декодирование кодов Рида-Соломона вынесен в файл ftape-ECC.c/ftape-ECC.h. Это достаточно аккуратный, хорошо структурированный и даже местами слегка комментируемый код, также рекомендую.
8. James S. Plank GFLIB. C Procedures for Galois Field Arithmetic and Reed-Solomon Coding. – Библиотечка для работы с кодами Рида-Соломона. Содержит в себе полные исходные тексты всех необходимых функций и распространяется по лицензии GPL. Найти ее можно на любом GNU-сайте, например, здесь: http://www.cs.utk.edu/~plank/plank/gflib/gflib.tar.