ELEKTROBIT

Reed Solomon Codes

Written : January 2001
Author : Joel Sylvester

Reed Solomon forward error correcting codes have become commonplace in modern digital communications. Although invented in 1960 by Irving Reed and Gustave Solomon, then working at MIT Lincoln Labs, it was many years before technology caught up and was able to provide efficient hardware implementations.

In turn Reed and Solomons work was based on an area of mathematics invented by French mathematician Evariste Galois in the 1830's. It pains some mathematicians to find that the field of number theory, one of the more esoteric areas of mathematics which Galois helped found, has proved so useful.

Versions of Reed Solomon codes are now used in error correction systems found just about everywhere, including...

- Storage devices (hard disks, compact disks, DVD, barcodes)
- Wireless communications (mobile phones, microwave links)
- Digital television
- Satellite communications (including deep space missions like Voyager)
- Broadband modems (ADSL, xDSL etc)

Reed Solomon codes work by adding extra information (redundancy) to the original data. The encoded data can then be stored or transmitted. When the encoded data is recovered it may have errors introduced, for instance by scratches on the CD, imperfections on a hard disk surface or radio frequency interference with mobile phone reception. The added redundancy allows a decoder (with certain restrictions) to detect which parts of the received data are corrupted, and correct them. The number of errors the code can correct depends on the amount of redundancy added.

Properties of Reed-Solomon Codes

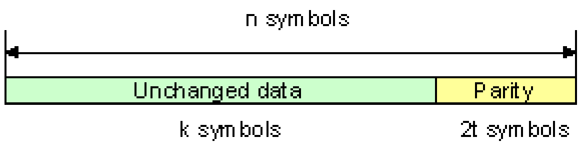
The topic of error correcting codes is extensive, and most texts treat all codes equally whether they are easily implemented or not. Reed-Solomon codes have certain properties which make them useful in the real world.

RS codes are a *systematic linear block code*. Its a block code because the code is put together by splitting the original message in to fixed length blocks. Each block is further sub divided into m-bit symbols. Each symbol is a fixed width, usually 3 to 8 bits wide.

The linear nature of the codes ensures that in practice every possible m-bit word is a valid symbol. For instance with an 8-bit code all possible 8 bit words are valid for encoding, and you don't have to worry about what data (whether it's binary, ASCII etc) you are transmitting. Systematic means that the encoded data consists of the original data with the extra 'parity' symbols appended to it.

An RS code is partially specified as an RS(n,k) with m-bit symbols. For instance the DVB code is RS(204,188) using 8-bit symbols. The n refers to the number of encoded symbols in a block, whilst k refers to the number of original message symbols. The difference n-k (usually called 2t) is the number of parity symbols have been appended to make the encoded block.

An RS decoder can correct up to (n-k)/2 or t symbols, ie any t symbols can be corrupted in any way, and the original symbols can be recovered. Thus the DVB code splits the message into blocks 188 symbols long. The 16 parity symbols (2t = 204-188 = 16) are then appended to produce the full 204 symbol long code. Up to 8 (t = 16/2) symbol errors can then be corrected.



RS(n,k) code

The power of Reed Solomon codes lies in being able to just as easily correct a corrupted symbol with a single bit error as it can a symbol with all its bits in error. This makes RS codes particularly suitable for correcting burst errors. Usually the encoded data is transmitted or stored as a sequence of bits. With the DVB code, a sequence of up to 56 consecutive bits could be corrupted affecting at most 8 symbols, and the original message could still be recovered. However it does mean that RS codes are relatively sensitive to evenly spaced errors. In the DVB code if 9 symbols have a single bit error then no corrections can be made. Other codes such as convolutional codes are better at correcting randomly occurring errors. Often the RS encoded block is further encoded in a convolutional code to try and cope with both burst and random errors.

Galois Field Arithmetic and Reed Solomon Codes

Reed Solomon codes are based on finite fields, often called Galois fields.

Rather than look at individual numbers and equations, the approach of modern mathematicians is to look at all the numbers that can be obtained from some given initial collection by using operators such as addition, subtraction, multiplication and division. The resulting collection is called a field. Some fields, like the set of integers, are infinite. This causes problems when you try to represent them on a computer with a fixed length word.

Galois fields have the useful property that any operation on an element of the field will always result in another element of the field. The field is also finite, so it can be fully represented by a fixed length binary word. An arithmetic operation that, in traditional mathematics, result in a value outwith the field gets mapped back in to the field - it's a form of modulo arithmetic.

For instance, in the Galois field used for the DVB RS codec, only the numbers 0 to 255 exist. The operation 20 times 20 does not result in 400, but gets wrapped around to 13. Don't try and make sense of that result, it is not simple modulo 256.

This means that algorithms using Galois arithmetic, in contrast to traditional binary arithmetic, do not have to cope with over or under flow exceptions. Galois arithmetic has very little to do with counting things, 2+2 is not necessarily 4. For ease of handling the Galois field elements are often called by their binary equivalent, but this can be misleading. The binary value 20 (00010100) maps to the 52nd element in the DVB Galois field, not the 20th. There are many Galois fields, and part of the RS specification is to define which field is used.

Galois arithmetic is ideally suited to hardware implementation. Addition and subtraction consists of simply xor'ing two symbols together. Multiplication is a little more difficult, (as always) but can be done using purely combinational logic. Alternative architectures can do multiplications using shift registers. Trade off's can be made between speed and the hardware resources used. This makes Galois arithmetic ideal for implementation using FPGAs or ASICs.

Galois arithmetic is less well suited to DSPs or microprocessors. Here the single cycle binary multipliers are of no use, and multiplication may take many clock cycles. There exist library routines for Matlab and Mathematica for high level programming, and 'C' routines for lower level programming. Having said that, the TMS320C6400 DSP has Galois arithmetic operators in it's instruction set.

An RS code with 8 bit symbols will use a Galois field GF(2^8), consisting of 256 symbols. Thus every possible 8 bit value is in the field. The order in which the symbols appear depends on the generator polynomial. This polynomial is used in a simple iterative algorithm to generate each element of the field. Different polynomials will generate different fields. For instance, the generator polynomial for DVB is $p(x) = 1 + x^2 + x^3 + x^4 + x^8$. This can be given the shorthand 285, from the binary value of the coefficients 100011101. From this the nth element of the field can be constructed by raising element 0 to the power n.

As an example, take the Galois field GF(2^4) - using 4 bit symbols, and generator polynomial 19. The polynomial is $p(x) = 1 + x + x^4$. The Galois field consists of 16 symbols as follows.

Power representation	Polynomial Representation	4-tuple or binary representation
0	0	0 0 0 0
1	1	1 0 0 0
α	α	0 1 0 0
α^2	α^2	0 0 1 0
α^3	α^3	0 0 0 1
α^4	$1 + \alpha$	1 1 0 0
α^5	$\alpha + \alpha^2$	0 1 1 0
α^6	$\alpha^2 + \alpha^3$	0 0 1 1
α^7	$1 + \alpha + \alpha^3$	1 1 0 1
α^8	$1 + \alpha^2$	1 0 1 0
α^9	$\alpha + \alpha^3$	0 1 0 1
α^{10}	$1 + \alpha + \alpha^2$	1 1 1 0
α^{11}	$\alpha + \alpha^2 + \alpha^3$	0 1 1 1
α^{12}	$1 + \alpha + \alpha^2 + \alpha^3$	1 1 1 1
α^{13}	$1 + \alpha^2 + \alpha^3$	1 0 1 1
α^{14}	$1 + \alpha^3$	1 0 0 1

Note that each element is the previous multiplied by α. By setting $p(\alpha) = 0$, then $\alpha^4 = 1 + \alpha$ (substituting α into the polynomial p(x)).

Thus in the above table α^4 is substituted with $1+ \alpha$, then

$\alpha^5 = (1 + \alpha)\alpha$,
$= \alpha + \alpha^2$.

$\alpha^6 = \alpha.\alpha^5$
$= \alpha^2 + \alpha^3$

$\alpha^7 = \alpha.\alpha^6$
$= \alpha^3 + \alpha^4$
$= 1 + \alpha + \alpha^3$

and so on.

The last parameter needed is the starting point for the RS generator polynomial (B0). For DVB this is zero. The RS generator is touched on later. Given n, k, the symbol width m, the Galois field polynomial p and starting root B0, the Reed-Solomon code is fully specified.

Reed Solomon Encoder

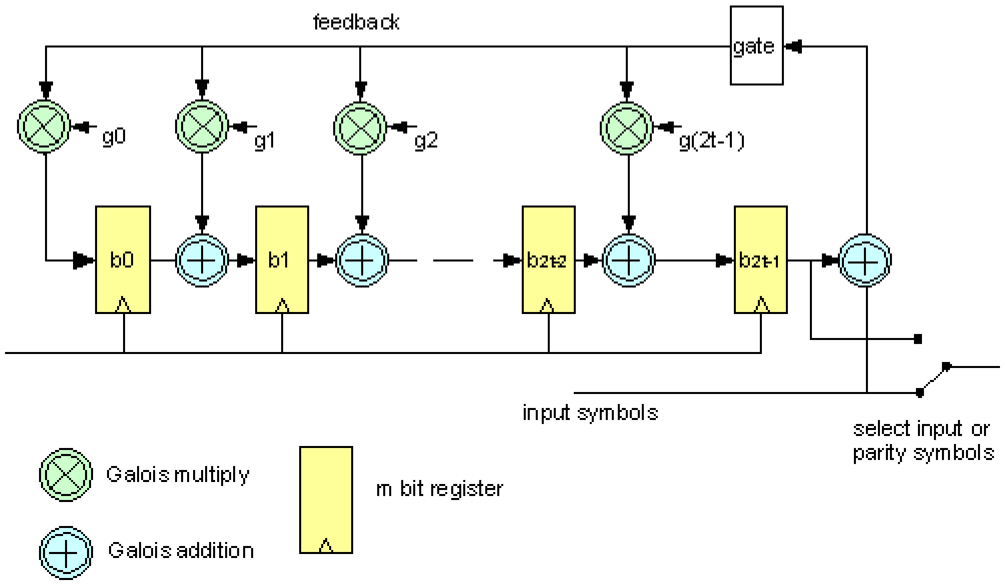
The encoder is the easy bit. Since the code is systematic, the whole of the block can be read into the encoder, and then output the other side without alteration. Once the kth data symbol has been read in, the parity symbol calculation is finished, and the parity symbols can be output to give the full n symbols.

Gross simplification coming up. The idea of the parity words is to create a long polynomial (n coefficients long – it contains the message and the parity) which can be divided exactly by the RS generator polynomial. That way, at the decoder the received message block can be divided by the RS generator polynomial. If the remainder of the division is zero, then no errors are detected. If there is a remainder, then there are errors. Dividing a polynomial by another is not conceptually easy, but if you follow the maths in some of the references its not too hard to understand.

The encoder acts to divide the polynomial represented by the k message symbols d(x) by the RS generator polynomial g(x). This generator polynomial is not the same as the Galois Field generator polynomial, but is derived from it.

$$x(n-k).d(x)/g(x) = q(x) + r(x)/g(x)$$

The term x(n-k) is a constant power of x, which is simply a shift upwards n-k places of all the polynomial coefficients in d(x). It happens as part of the shifting process in the architecture below. The remainder after the division r(x) becomes the parity. By concatenating the parity symbols on to the end of the k message symbols, an n coefficient polynomial is created which is exactly divisible by g(x).

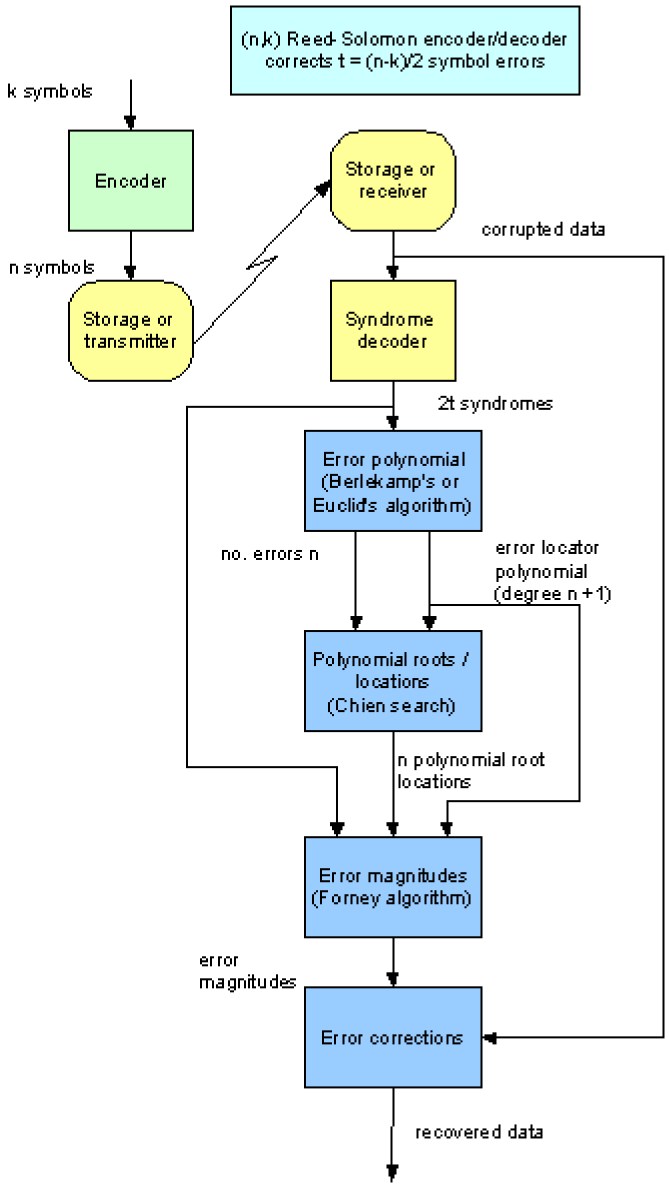


The encoder is a 2t tap shift register, where each register is m bits wide. The multiplier coefficients g0 to g(2t-1) are coefficients of the RS generator polynomial. The coefficients are fixed, which can be used to simplify the multipliers if required. The only hard bit is working out the coefficients, and for hardware implementations the values can often be hard coded.

At the beginning of a block all the registers are set to zero. From then on, at each clock cycle the symbol in each register is added to the product of the feedback symbol and the fixed coefficient for that tap, and passed on to the next register. The symbol in the last register becomes the feedback value on the next cycle. When all n input symbols have been read in, the parity symbols are sitting in the register, and it just remains to shift them out one by one.

Reed Solomon Decoder

Decoding is a far harder task than encoding. Typically about ten times more resources (be it logic, memory or processor cycles) are required to decode and correct the corrupted data.

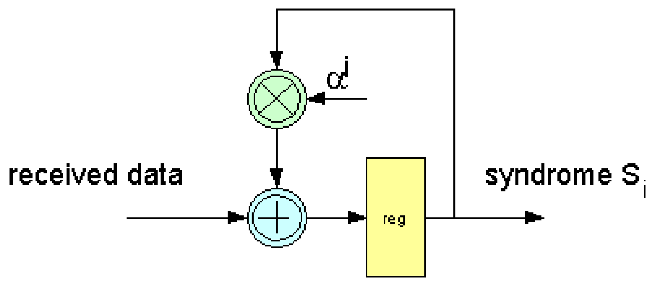

(n,k) Reed-Solomon encoder/decoder corrects t = (n-k)/2 symbol errors
k symbols
Encoder
n symbols
Storage or transmitter
Storage or receiver
corrupted data
Syndrome decoder
2t syndromes
Error polynomial (Berlekamp's or Euclid's algorithm)
no. errors n
error locator polynomial (degree n+1)
Polynomial roots / locations (Chien search)
n polynomial root locations
Error magnitudes (Forney algorithm)
error magnitudes
Error corrections
recovered data

The decode operation takes several stages. There are plenty of sources available for software implementations of the various algorithms required. Hardware implementations (FPGA or ASIC) are a little harder to come by, especially those with parameterised specifications. Texas Instruments give their software decoder away for free, whilst by comparison FPGA manufacturers such as Xilinx or Altera can charge many thousands of dollars for their hardware implementations.

Syndrome decoder

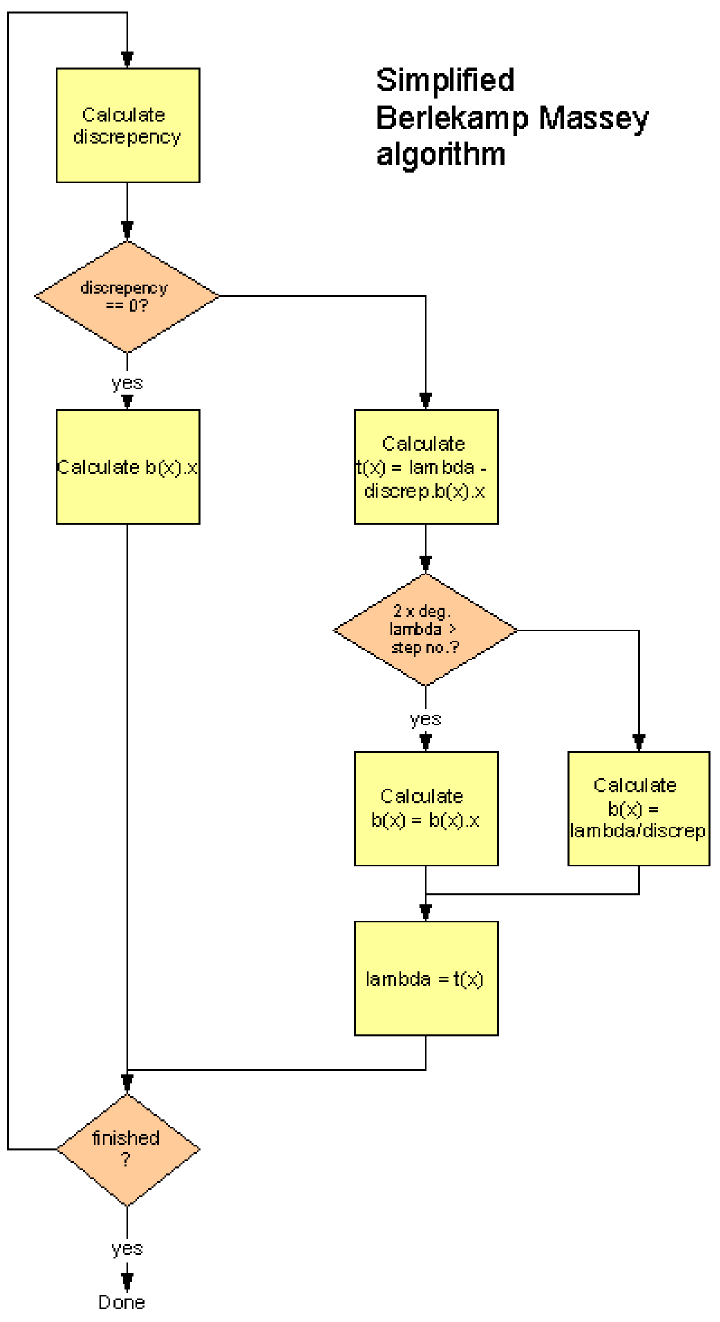
The first step in decoding the received symbol is to determine the data syndrome. Here the input received symbols are divided by the generator polynomial. The result should be zero (the parity was placed there to ensure that code is exactly divisible by the generator polynomial). If there there is a remainder, then there are errors. The remainder is called the syndrome.

Syndrome calculation can be done by an iterative process, such that the answer (2t syndrome symbols) is available as soon as the last parity symbol has been read in. The circuit below will generate the i'th syndrome, 2t of these will be needed for the full syndrome decoder. The syndromes depend only on the errors, not on the underlying encoded data.



ELEKTROBIT

Error polynomial lambda - Berlekamp-Massey and Euclids algorithm


Simplified Berlekamp Massey algorithm
Calculate discrepency
discrepency == 0?
yes
Calculate b(x).x
Calculate t(x) = lambda - discrep.b(x).x
2 x deg. lambda > step no.?
yes
Calculate b(x) = b(x).x
Calculate b(x) = lambda/discrep
lambda = t(x)
finished ?
yes
Done

The second step is to find the error polynomial lambda. This requires solving 2t simultaneous equations, one for each syndrome.

The 2t syndromes form a simultaneous equation with t unknowns. The unknowns are the locations of the errors. In general there are many possible solutions to the set of equations, but we assume that the one with the least number of errors is the correct one. This assumption is the reason that more than t errors can actually cause the decoder to corrupt the received signal further (if allowed to). If more than t errors occur, then there will exist a possible solution to the equations with less than t errors. Unfortunately this solution is unlikely to correct the right symbols.

The process of solving the simultaneous equations is usually split into two stages. First, an error location polynomial is found. This polynomial has roots which give the error locations. Then the roots of the error polynomial are found.

There are several methods of finding the error polynomial lambda, the two most popular are Euclid's Algorithm (easier to implement) and the Berlekamp-Massey Algorithm (more efficient use of hardware resources).

The algorithm iteratively solves the error locator polynomial by solving one equation after another and updating the error locator polynomial. If it turns out that it cannot solve the equation at some step, then it computes the error and weights it, increases the size of the error polynomial, and does another iteration. A maximum of 2t iterations are required. For n symbol errors, the algorithm gives a polynomial with n coefficients. At this point the decoder fails if there are more than t errors, and no corrections can be made. Doing so might actually introduce more errors than there were originally.

ELEKTROBIT

Finding the error polynomial roots - Chien search

Once the error polynomial lambda is known, its roots define where the errors are in the received symbol block. The most commonly used algorithm for this is the Chien search. This is a brute force and ignorance method, or more politely, an exhaustive search. All 2^m possible symbols are substituted into the error polynomial, one by one, and the polynomial evaluated. If the result comes to zero, you have a root.

Calculate the error magnitudes - Forney algorithm

You now know where the errors are, but not what they are. The next step is to use the syndromes and the error polynomial roots to derive the error values. This is usually done using the Forney algorithm. The algorithm is an efficient way of performing a matrix inversion. The algorithm works in two stages. First the error evaluator polynomial omega is calculated. This is done by convolving the syndromes with the error polynomial lambda (from the Berlekamp-Massey result). Omega is then calculated at each zero location, and divided by the derivative of lambda. Each calculation gives the error symbol at the corresponding location. If a bit is set in the error symbol, then the corresponding bit in the received symbol is in error, and must be inverted.

All that remains is to correct the received symbols. The symbols are read again from an intermediate store, and at each error location the received symbols xor'ed with the error symbol. Usually the parity symbols are stripped off.

Implementation

That's the basics - all that remains is simply a matter of implementation.

There are a number of of-the-shelf hardware implementations, for instance Advanced Hardware Architectures do a series of VLSI Reed Solomon codecs.

Alternatively, you can buy intellectual property in hardware description languages such as VHDL or Verilog. These tend to be far more flexible (any reasonable n,k,m) and are targeted at implementation in an FPGA or ASIC. The advantage here is that other tasks can be integrated in to the same device, for instance the Reed Solomon codec could be joined with a convolutional codec (for random error correction), and controlling microprocessor, memory and a transceiver for a complete system-on-a-chip.

Software implementations are usually slower and more costly than a dedicated hardware approach, although some DSP targets will come close.

Want to know more ?

Try the following if you want the maths...

Lin and Costello, "Error Control Coding: Fundamentals and Applications", Prentice Hall 1983, ISBN 013283796

If you want an optimised VHDL implementation, talk to us at Elektrobit. www.elektrobit.co.uk

If you want free 'C' code, try Robert Morelos-Zaragoza's page at
http://imailab-www.iis.u-tokyo.ac.jp/~robert/codes.html