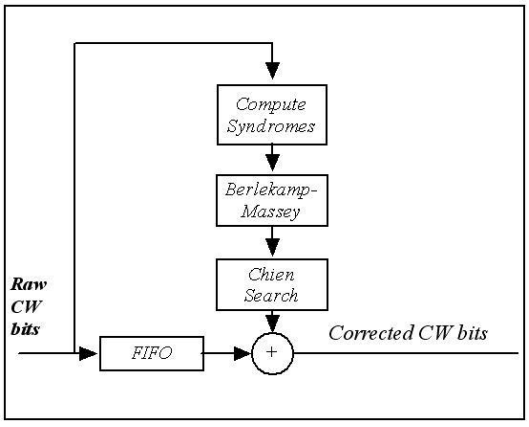
Fast Software BCH Encoder and Decoder
-FastBchEnDecR300-



A ChannelScience White Paper

Written by

Neal Glover

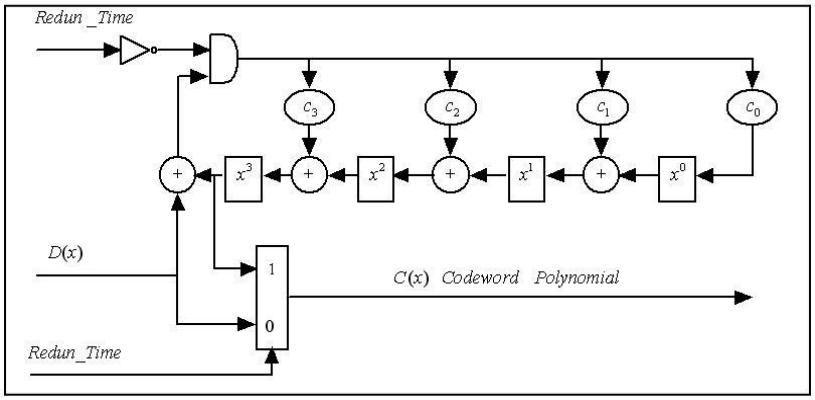
October 1, 2010

ChannelScience
Detecting the Future of Data Storage SM

7300 Cody Court
Plano TX 75024-3837 USA
connect@ChannelScience.com

(972) 814-3441 Voice
(972) 208-9095 FAX

Table of Contents



1. Executive Summary

Software encoding and decoding of binary BCH codes is possible for a range of throughput requirements. By carefully selecting the strategy and algorithms for encoding and decoding, this range can be extended. Such an implementation has been posted on the ChannelScience web site, www.ChannelScience.com, as a Visual Studio "C" project (FastBchEnDecR300). The implementation of this software and its capabilities are discussed herein.

2. Introduction

The purpose of this document is to give an overview of the fast encoder-decoder software "FastBchEnDecR300". And also to convey at a high level some insight into the binary BCH codes and the classical algorithms for encoding and decoding them. For detailed information on encoding and decoding algorithms monitor the www.ChannelScience.com web site for new postings.

Flexible Encoder and Decoder

"FastBchEnDecR300" is a flexible binary BCH code encoder and decoder. It supports most binary BCH codes of practical interest. It supports "m" of GF(2^m) from six to sixteen and it support "t", the maximum number of errors correctable by the code, from one to fourteen. The "C" project can be found at www.ChannelScience.com.

The program can be launched by double clicking the "exe" file that is part of the "C" project. All parameters and options are entered at the keyboard. The program prompts for all inputs and provides information to help with the entry of parameters and the selection of options. It should be possible to double click the "exe" file and use the program without having to refer to the source code.

The source code listing should have enough comments to enable an experienced programmer to use the functions as a design reference.

3. Binary BCH Codes

Binary BCH codes are random bit error correcting codes. The coefficients of the codeword polynomial are from the finite field of two elements GF(2), but the computation used by the error correction algorithms is performed over a larger finite field, GF(2^m) . The code generator polynomial for a binary BCH code is the least common multiple of a set of minimum polynomials. Two of the parameters for a binary BCH code are "m" and "t". The "m" parameter is "m" of GF(2^m) and it is determined by the number of data bits that must be supported by the code. The "t" parameter is the maximum number of bit errors that are to be correctable by the code. Binary BCH codes are generally preferred over Reed-Solomon codes when the errors seen by the code are random bit errors.

Binary BCH Preferred if Bit Errors

The classic encoding of a binary BCH code is accomplished with a bit-serial shift register similar to the simple shift register shown in Figure 1.

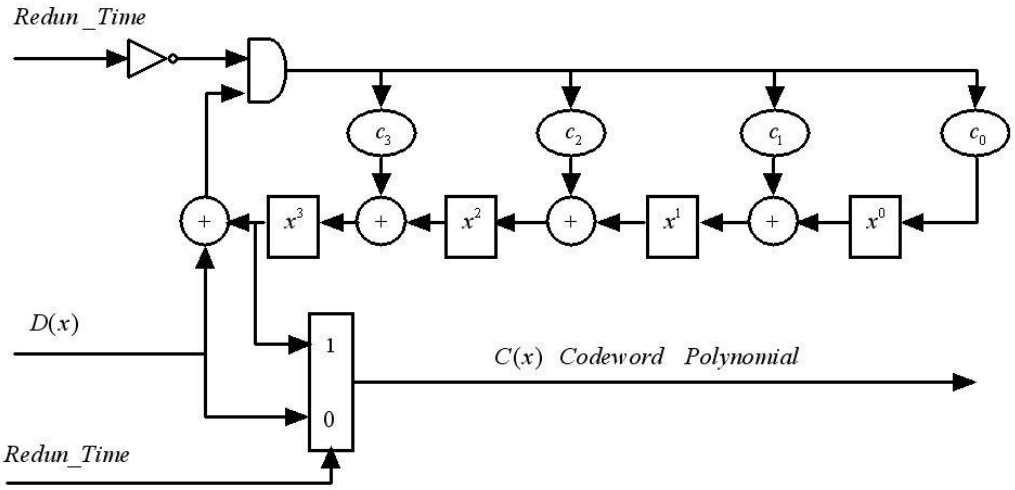


Figure 1

In the encoder circuit of Figure 1 the c_i (0 or 1) are coefficients of the code generator polynomial. On encode the *Redun_Time* signal is initially low and the circuit pre-multiplies the data polynomial by x^{n-k} and divides by *g(x)*, the code generator polynomial. When the *Redun_Time* signal goes high the remainder (redundancy) is output behind data.

Classical decoding of binary BCH codes is illustrated in Figure 2.

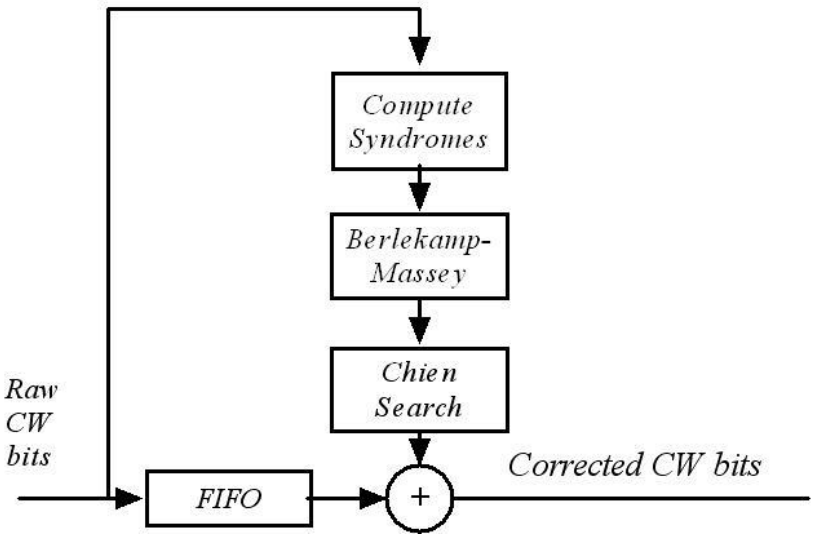


Figure 2

Three Steps to Decode

Three steps are required. The first step is syndrome computation. The second step is the computation of an error locator polynomial. This step is accomplished with an algorithm such as the Berlekamp-Massey algorithm or Euclids algorithm. The third step finds the roots of the error locator polynomial. The classical algorithm for the root finding step is the Chien Search, but other root finding algorithms exist.

Equation (1) is the equation for syndrome computation. There are several methods for implementing this equation. Each is a trade off between resources and speed.

$$S_i = C'(x) \bmod g_i(x) = C'(x)\big|_{r_i} \qquad (1)$$

where,

S_i = the ith syndrome

$C'(x)$ = the received codeword (possibly in error)

$g_i(x) = (x - r_i) =$ the ith factor of $G(x)$

$r_i =$ the ith root of $G(x)$

Algorithm 1 shows pseudo code for one version of the classical Berlekamp-Massey algorithm for binary BCH codes.

$a(x)=1, \quad b(x)=1, \quad dk=1, \quad Ln=0, \quad Lk=0, \quad m=-1, \quad t=(d-1)/2$

FOR $n=0$ TO $d-2$ STEP 2

$m=m+2$

$dn=\sum_{j=0}^{Ln} a_j S_{n-j} = S_n + \sum_{j=1}^{Ln} a_j S_{n-j}$

IF $dn \neq 0$

IF $Ln >= Lk+m$

$a(x)=a(x)-\frac{dn}{dk}x^m b(x)$

ELSE

$t(x)=a(x)$

$a(x)=a(x)-\frac{dn}{dk}x^m b(x)$

$Lt=Ln, \quad Ln=Lk+m, \quad Lk=Lt$

$b(x)=t(x)$

$dk=dn, \quad m=0$

END

END

END

IF $Ln>t$

Post "Uncorrectable Error"

END

$\sigma(x)=a(x)$

Algorithm 1

Algorithm 2 shows one version of the classical Chien search algorithm.

$k = 0$
FOR loc = 0 TO codeword length – 1
$z = \sigma(x)\big|_{\alpha^0} = \sum_{j=0}^{Ln} \sigma_j$
IF $z = 0$
$X_k = \alpha^{loc}$ // Save root
$k = k + 1$
END
$\sigma(x) = \sigma\left(\alpha^{-1} x\right)$
END
IF incorrect number of roots found post uncorrectable error

Algorithm 2

4. Design

Faster Encoding and Decoding

The FastBchEnDecR300 software incorporates several features that increase throughput by increasing the speed of encoding and decoding. Encoding speed is increased by building the binary redundancy vector in the bits of a set of 32 bit words to minimize the number of operations. Encoding speed is further increased by processing eight data bits at a time. The software that accomplishes this performs a function equivalent to that performed by a parallel hardware shift register. A large encode table facilitates this process. Shifting the current estimate of the binary redundancy vector by eight bits and XORing with it one vector (one set of 32-bit words) from the encode table accomplishes the equivalent of eight shifts of a bit-serial hardware shift register or one shift of an eight-bit parallel hardware shift register.

For Speed Compute a Remainder First

Decoding speed is increased by first computing a remainder and then if the remainder is nonzero, computing syndromes from the remainder. The remainder is computed in the same way that the

redundancy is computed for encoding. Eight data bits are processed at a time and the same large encode table is used to emulate the action of a parallel hardware shift register. The fast computation of a remainder is essential for achieving significant throughput for the error free case.

On decode, if the remainder is nonzero, syndromes are computed, then an error locator polynomial is computed and its roots are found. One of the techniques used by the software to speed up syndrome computation is computing even syndromes from odd ones This is not possible for all codes but it is possible for a binary BCH code.

Divide Down for Speed

The Chien Search is used for root finding and its speed is increased by working with the logs of coefficients of the error locator polynomial, by avoiding "mod" operations, and by dividing down the error locator polynomial each time a root is found. Speed is also increased by exiting the Chien search when there are only a few roots left to find. The exit occurs when the degree of the error locator polynomial reaches four for even "m" and when the degree reaches two for odd "m". After the exit, special algorithms are used to find the remaining roots.

Unroll Chien Search Loop for Speed

Speed is further increased by unrolling (straight line coding) the tightest loop. When a loop is unrolled the contents of the loop are replicated to eliminate loop overhead. If the number of times the body of the loop is executed changes dynamically, the code branches down into the replications to achieve the equivalent of looping the loop body the correct number of times.

Consider the example of unrolled code in Algorithm 3. It evaluates $\sigma(X)$ at α^k using Horner's method.

$y = 0$
SWITCH (degree of $\sigma(X)$)
CASE 5: $y = y\alpha^k + \sigma_5$
CASE 4: $y = y\alpha^k + \sigma_4$
CASE 3: $y = y\alpha^k + \sigma_3$
CASE 2: $y = y\alpha^k + \sigma_2$
CASE 1: $y = y\alpha^k + \sigma_1$
CASE 0: $y = y\alpha^k + \sigma_0$
END

Algorithm 3

At the end of the unrolled loop, *y* contains $\sigma(x)$ evaluated at α^k.

One additional Chien search speedup is used. If the error locator polynomial has no zero coefficients then a test for zero is not performed on each coefficient at each step of the Chien search. The test to determine if there are any zero coefficients in the error locator polynomial is performed initially and after each divide down operation.

Future new releases of the software will be posted at www.ChannelScience.com that incorporate significant new speedups for software encoding and decoding. One of the new speedups will significantly increase the speed of root finding for long codes when there are more than a few errors. When more that a few errors exist root finding is a major contributor to decode time.

5. Average Error Correction Time

Error Free Decode Time Dominates Average Decode Time

In most applications few errors are more likely than many. And if errors are random and the probability that a bit is in error is the same for all bits then it would be very rare to have, in a sector, the maximum number of errors supported by the error correcting code. In most applications we will see mostly error free sectors and a few sectors with one error and fewer still sectors with two errors and so on. It will be very rare to see the maximum number of errors. So in most cases the error free decode time has the most influence on average decode time.

Lets put this in perspective with an example. If we are using a binary BCH code and we have 1024 data bytes in each sector and t=14 and m=14 then there would be about 8388 bits in each codeword. If we want our uncorrectable error rate to be less than 1.E-15 (uncorrectable sectors per bit) then for our given parameters the raw error rate must be less than 1.5152 E-4 (bit errors per bit). Assume that this is precisely the raw error rate, then the probability for each number of errors (0 to 14) in a sector is given in column 1 of Table 1 (units are events per sector). Column 2 lists, for the parameters given above, the decode time in seconds for a 2.67 GHz i7 920 processor. Column 3 lists the decode times normalized to the decode time for the zero error case. Now we can multiply frequency of occurrence (column 1) times decode time, normalized to the time to decode the zero error case (column 3), for each number of errors to get the contribution of each to normalized average decode time. The results are given in column 4. From

column 4 it can be seen that, for this example, the zero error case is the largest contributor to average decode time. The next largest contributor is the one error case and so on. The sum of these contributions (1.3545) is the average decode time normalized to the time to decode the zero error case. So, for this example, the average decode time is only about 35% higher than the time to decode the zero error case.

Table 1

Probability of given number of errors in a sector	Decode time in seconds	Normalized decode time	Contribution to Normalized average decode time
0 -- 7.1946e-001	0 -- 7.20e-005	0 -- 1.0000	0 -- 7.1946e-001
1 -- 3.6286e-001	1 -- 8.20e-005	1 -- 1.1389	1 -- 4.1326e-001
2 -- 1.3624e-001	2 -- 8.40e-005	2 -- 1.1667	2 -- 1.5894e-001
3 -- 4.0239e-002	3 -- 8.40e-005	3 -- 1.1667	3 -- 4.6945e-002
4 -- 9.7427e-003	4 -- 8.60e-005	4 -- 1.1944	4 -- 1.1637e-002
5 -- 1.9934e-003	5 -- 1.22e-004	5 -- 1.6944	5 -- 3.3777e-003
6 -- 3.5263e-004	6 -- 1.54e-004	6 -- 2.1389	6 -- 7.5424e-004
7 -- 5.4895e-005	7 -- 1.82e-004	7 -- 2.5278	7 -- 1.3876e-004
8 -- 7.6259e-006	8 -- 2.06e-004	8 -- 2.8611	8 -- 2.1819e-005
9 -- 9.5612e-007	9 -- 2.68e-004	9 -- 3.7222	9 -- 3.5589e-006
10 -- 1.0920e-007	10 -- 3.24e-004	10 -- 4.5000	10 -- 4.9141e-007
11 -- 1.1450e-008	11 -- 3.80e-004	11 -- 5.2778	11 -- 6.0433e-008
12 -- 1.1096e-009	12 -- 4.36e-004	12 -- 6.0556	12 -- 6.7195e-009
13 -- 9.9943e-011	13 -- 4.85e-004	13 -- 6.7361	13 -- 6.7322e-010
14 -- 8.4074e-012	14 -- 5.40e-004	14 -- 7.5000	14 -- 6.3056e-011

6. Average Throughput

For some applications, real time audio or video streaming for example, it is worst case throughput that is important . For other applications like writing to or reading from a thumb drive perhaps it is average throughput that is important. It would seem that software encoding and decoding is most applicable when it is average throughput that is important and the throughput requirement is modest. However when adequate buffering is available and

adequate delay is allowed then software encoding and decoding may work for some streaming applications as well.

Use Elastic Buffer for Streaming

An elastic buffer can be used for streaming applications. The buffer would hold both corrected and uncorrected sectors. The average correction speed would be faster than the speed of moving sectors from the buffer to the streaming application. Normally most of the sectors in the buffer would be corrected sectors. But, if a sector is encountered that has lots of errors then the ratio of corrected sectors to uncorrected sectors would decrease until correction of the problem sector is complete. Then the ratio would increase again until again the buffer contains mostly corrected sectors. The buffer should be large enough so that the supply of corrected sectors is not exhausted during the correction of a sector with the maximum number of errors. The size of the buffer determines the delay in delivering data to the streaming application.

7. Measuring Throughput Times

The FastBchEnDecR300 software incorporates features that facilitate timing measurements. If you choose to bypass encode, the time measurement will apply to decode only. The measurement will be more accurate if you choose not to compare the decoded codeword with the original codeword. To measure encode time, run the program with and without encode and subtract the measured times. If you are given the option, do not choose to generate random data and do not choose to compare.

8. Hybrid Binary BCH Decoders

The decode functions of "FastBchEnDecR300" can be used in hybrid binary BCH decoders as well as in software decoders. There are two techniques for implementing a hybrid binary BCH decoder. With the first technique, read remainder computation is performed in hardware and syndrome computation and error correction are performed in software or firmware. With the second technique, read syndrome computation is performed in hardware and only error correction is performed in software of firmware. The second technique is faster. The decoding speed of Hybrid BCH decoders is between that of hardware decoders and that of software decoders.

In a hybrid implementation, if it is only the remainder that is computed in hardware on decode, the zero error decode time of Table 1 would be subtracted from all decode times. So, for the example employing Table 1, the improvement in average decode

time would be almost four to one. The improvement would be even greater if syndromes were computed in hardware.

9. Managing Errors

In many error correction systems errors are managed. The frequency of correctable error events in each sector may be monitored to see if errors are repeating. If errors are repeating then the sector may be rewritten and if the errors persist then the sector may be retired in some way. An alternative strategy would be to allow a few errors, say one or two, to exist in any sector and when a greater number of errors exist in a sector and persist after a rewrite then to retire the sector. Of course ideally all this would be done before the error situation for the sector becomes severe enough for the sector to become uncorrectable.

10. Conclusions

Software encoding and decoding of binary BCH codes is possible for a range of throughput requirements. The FastBchEnDecR300 software implements several features that extend the range of software encoding and decoding throughput by increasing encoding and decoding speed. Perhaps software encoder/decoders are most applicable to non-streaming applications but may be applicable for some streaming applications if a suitable elastic buffering strategy is employed.

When errors occur at random intervals and the probability distribution for the number of bits between errors can be approximated by a normal probability distribution, the error free decode time is likely to be the dominate contributor to average decoding time. The time to decode an error free sector can be reduced by using a hybrid strategy where encoding and syndrome computation are performed in hardware and all other decoding functions are performed in software.

New Releases Coming

A future release of the software will introduce a new faster root finder. This will reduce worst case decode time significantly. Another future release will further reduce the error free decoding time.

11. References

1) N. Glover and T. Dudley, *Practical Error Correction Design for Engineers* - Revised Second Edition, Cirrus Logic, 1991.
2) ChannelScience web site www.ChannelScience.com – FREE downloadable software: FastBchEnDecR300.

About the Author

Neal Glover was a driver in moving the hard disk drive industry to more powerful error correction codes in the early 1990s. He has been interested in the practical application of error correcting codes for more than 30 years.

Neal has taught short courses on the subject and holds a number of patents in the field. He started two small businesses to provide error correction products and services to the magnetic and optical storage industries.

He enjoys programming in MATLAB® and "C" and has developed an extensive finite field function library for prototyping error correction algorithms in MATLAB®.