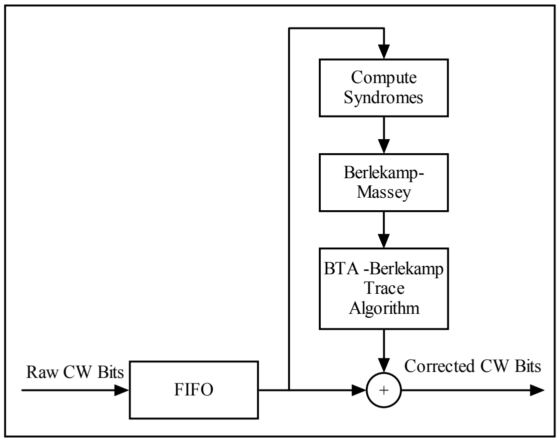
Fast Software BCH Encoder and Decoder -FastBchEnDecR400-



A ChannelScience White Paper

Written by

Neal Glover

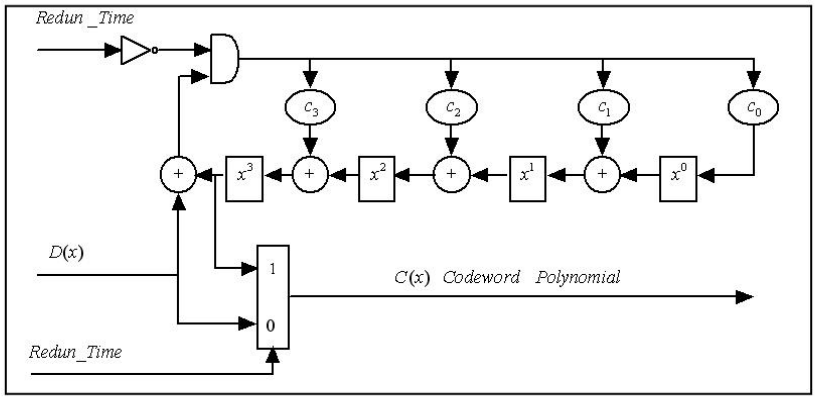
October 10, 2011

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 FastBchEnDecR400, hereafter called R400. 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 R400. 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

R400 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 supports "t", the maximum number of errors correctable by the code, from one to sixty-four. The "C" project can be found at www.ChannelScience.com/ecc.html.

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 Preferred if Bit Errors

Binary BCH codes are generally preferred over Reed-Solomon codes when the errors seen by the code are random 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.

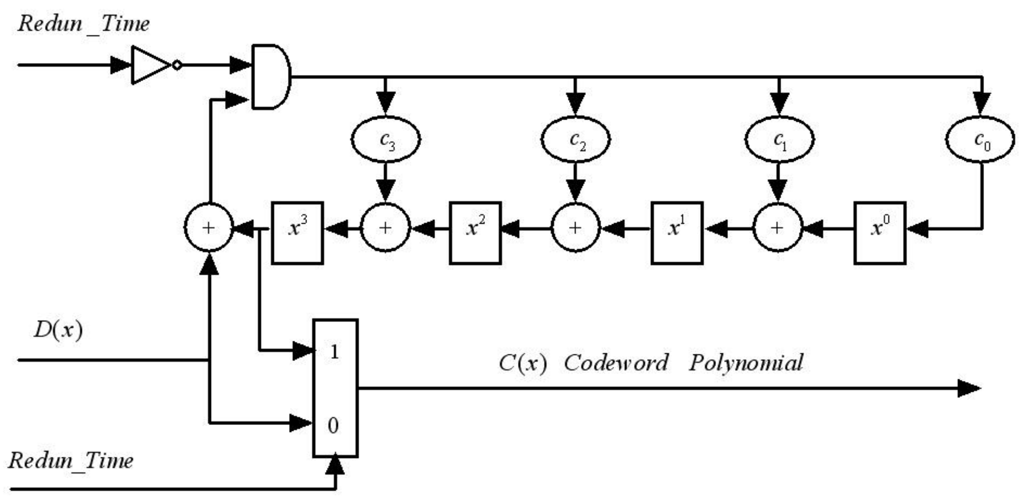


Figure 1

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.

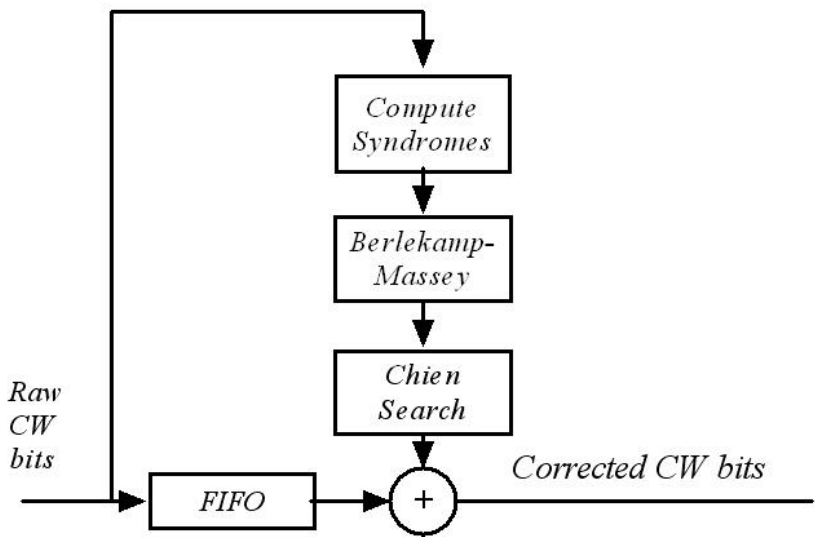


Figure 2

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, such as the BTA (Berlekamp Trace Algorithm) which is incorporated in R400.

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.

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

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

Algorithm 2

4. Design Overview

Faster Encoding and Decoding

The R400 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.

5. Root Finding

In R400, root finding is accomplished with either a fast Chien search or with the BTA (Berlekamp Trace Algorithm). For some parameter sets of practical interest the BTA algorithm is much faster than the Chien search. But for others the Chien search is faster.

Divide Down for Speed

In R400 the speed of the fast Chien search 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 of the fast Chien search 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.

R400 also incorporates a BTA root finding algorithm. The BTA was first described by Berlekamp around 1970 [5]. The algorithm as defined by Berlekamp, splits the polynomial to be factored into two factors using a polynomial greatest common divisor (gcd) function. Each resulting factor is also split into two factors and so on until there exist only degree one factors. To split a polynomial the greatest common divisor function is performed on the polynomial and a trace polynomial that has as its roots about half the elements of the finite field employed.

Stop splitting polynomial factors when the degree of a factor falls below a threshold

It is faster to stop splitting when the degree of a factor falls below a threshold and instead to find the roots of such factors by even faster methods for low degree polynomials. In R400 the splitting is stopped at degree four for even "m" ("m" or GF(2^m)) and at degree two for odd "m". Currently special root finding algorithms are used for linear, quadratic, cubic and quartic polynomials. Stopping the splitting at degree four or less for both odd and even m would speed up root finding. A further speed improvement could be obtained by stopping the splitting at degree six or less by using special fast root finding algorithms for quintic and sextic polynomials as well. Such algorithms are known. An example algorithm is described in [8]. It can be used to find the roots of both quintic and sextic polynomials.

Trace Polynomial Is Key

Key to the operation of BTA is the fact that the trace polynomial $Tr(x)$ has as its roots about half the elements of the finite field. So performing the gcd function on the polynomial to be factored and the trace polynomial has a good chance of splitting the polynomial to be factored. Performing the same gcd function on $Tr(\beta_k x)$ and the polynomial to be factored may split the polynomial differently. β_k is an element of the basis of the field. In practice β_k is different for the subsequent splits of a polynomial. Lets say that β_1 is used to split the original polynomial then each factor resulting from the split is again split using β_2 and so on.

Two steps of gcd

The computation of a gcd can be divided into two steps. The first step computes a residue by computing the larger polynomial mod the smaller polynomial and then the second step computes the gcd of this residue and the smaller polynomial. This is done in the R400 implementation of the BTA algorithm.

For GF(2^m) the trace polynomial of x has the following form,

$$x^{2^{m-1}} + \ldots + x^{2^3} + x^{2^2} + x^{2^1} + x^{2^0} .$$

Note that all powers of x are powers of 2.

The trace polynomial for $\beta_k x$ has the form,

$$\beta_k^{2^{m-1}} x^{2^{m-1}} + \ldots + \beta_k^{2^3} x^{2^3} + \beta_k^{2^2} x^{2^2} + \beta_k^{2^1} x^{2^1} + \beta_k^{2^0} x^{2^0} .$$

In our implementation of BTA we first compute

$$Tr(\beta_k x) \ mod \ \ p(x),$$

where p(x) is the polynomial to be factored. So we compute

$$(\beta_k^{2^{m-1}} x^{2^{m-1}} + \ldots + \beta_k^{2^3} x^{2^3} + \beta_k^{2^2} x^{2^2} + \beta_k^{2^1} x^{2^1} + \beta_k^{2^0} x^{2^0}) \bmod p(x)$$

This is computed by computing each term mod p(x) and summing the results. To compute the residue mod p(x) of the example term

$$\beta_k^{2^3} x^{2^3},$$

first compute the residue of

$$(x^{2^3}) \bmod p(x)$$

and then multiply the resulting vector by $\beta_k^{2^3}$ to get

$$(\beta_k^{2^3} x^{2^3}) \bmod p(x).$$

So compute and retain the residues for each term of $Tr(x)$. Then multiply each residue vector by a function of β_k and sum the results to get the residue of

$Tr(\beta_k x) \bmod p(x)$ for a particular value of k.

The residues of terms whose powers of x are powers of 2 can be computed by repeated squaring, for example

$$(x^{2^3}) \bmod p(x) = ((x^{2^2}) \bmod p(x))^2 \bmod p(x)$$

Algorithm 4 is a high level representation of the BTA algorithm implemented in R400.

```
// $Berlekamp\ Trace\ Algorithm\ (BTA)$
// $p(x)$ is Error Locator Polynomial
// $k$ is $k$ of $T_k$
// $j$ is pointer to entry of FactorTbl to be further factored
// $n$ is pointer to next entry of FactorTbl to be filled
// $T_k = (Tr(\beta_k x))\ mod\ p(x)$
function BTA($p(x)$)
        for $k = 1:m$
            $T_k = \sum_{j=0}^{m-1} \beta_k^{2^j} (x^{2^j} \bmod p(x))$
        end
//
        $j = 1; n = 1;$
        $FactorTbl(1) = p(x);$
        $kTbl(1) = 1;$
        while $j < n$
            $f(x) = FactorTbl(j);$
            if $\deg(f(x)) \le 4$
                call quartic, cubic, quatratic, or linear;
            else
                $k = kTbl(j);$
                $g(x) = \gcd(f(x), T_k);$
                $h(x) = f(x)/g(x);$
                $FactorTbl(n) = g(x);$
                $kTbl(n) = k+1; n = n+1;$
                $FactorTbl(n) = h(x);$
                $kTbl(n) = k+1; n = n+1;$
            end
            $j = j+1;$
      end
      return
end
```

Algorithm 4

For simplicity, the pseudo code for Algorithm 4 shows pre-computing of all the T_k vectors. In practice it is faster to compute each T_k when it is needed the first time. Once a T_k is computed, it should be saved for possible use again. Referring to the Algorithm 4 pseudo code, the vectors $x^{2^j} \, mod \;\, p(x)$ should be computed just once per error case, but will be used up to m times in computing the T_k vectors.

6. Error Correction Time

In this section the speed of error correction for R400 will be discussed. The two root finders of R400 are compared using several speed measures. Table 1 compares the number of finite field multiplies and adds required for decoding three particular cases. Table 2 compares root finding time (only) for three cases. Table 3 compares total decode time for three cases.

The number of multiplies and adds for Table 1 were measured in MATLAB® simulators. The timings in Table 2 were measured using a standalone “C” implementation of the algorithms. The timings for Table 3 were measured using R400 itself.

From Table 1 you can see that the finite field multiply and add counts only provide a rough indication of the speed difference between the two root finders. From Table 2 you will see that when it is only the root finding times that are compared, the timing differences are significant. But this large difference does not translate directly into an as large a difference in total decode time. See Table 3. This is because root finding is not the only contributor to total decode time. Still Table 3 does show that there is a significant speed benefit to using the BTA instead of the fast Chien search for particular parameter sets when the application must be designed to make error correction fast for the maximum number of errors that are correctable.

Table 1. Average Number of Finite Field Multiplies and Adds Required just for Root Finding using the BTA Root Finder and using a Fast Chien Search. For each case there are t random errors

	512 Data Bytes (t=14, m=13)	1024 Data Bytes (t=14, m=14)	1024 Data Bytes (t=25, m=14)	1024 Data Bytes (t=40, m=14)
BTA				
Finite field add count	2,488	2,131	6,362	16,002
Finite field mult count	2,727	2,267	6,710	16,780
Fast Chien search				
Finite field add count	27,461	53,123	103,500	175,040
Finite field mult count	27,461	53,123	103,500	175,040

Table 2. Time just for root finding for 100,000 polynomials using the BTA root finder and using a fast Chien search root finder. In each case the degree of the polynomial is equal to t and the roots are at random locations. The roots may take on any of the possible finite field values. Times are for a 2.67 GHz i7 920 processor

	(t=14, m=13)	(t=14, m=14)	(t=25, m=14)	(t=40, m=14)
BTA				
Time to find roots	4.1 Seconds	3.4 Seconds	9.8 Seconds	23.9 Seconds
Fast Chien search				
Time to find roots	42 Seconds	87 Seconds	188 Seconds	317 Seconds

Table 3. Time for 100,000 full decodes using the BTA root finder and using a fast Chien search root finder. For each decode the number of errors simulated is equal to t. Times are for a 2.67 GHz i7 920 processor

	512 Data Bytes (t=14, m=13)	1024 Data Bytes (t=14, m=14)	1024 Data Bytes (t=25, m=14)	1024 Data Bytes (t=40, m=14)
BTA				
Total decode time	9 Seconds	13 Seconds	25 Seconds	55 Seconds
Fast Chien search				
Total decode time	27 Seconds	54 Seconds	116 Seconds	203 Seconds

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.

Error Free Decode Time Dominates Average Decode Time

This can be put in perspective with tables 4 and 5. Table 4 assumes that the fast Chien search is used for root finding and Table 5 assumes that the BTA algorithm is used for root finding. 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 tables 4 and 5 (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 the frequency of occurrence (column 1) times the 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 the 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 is the average decode time, normalized to the time to decode the zero error case. The sum is 1.3545 for Table 4 and 1.3667 for Table 5. So, for this example, the average decode time is only about 35% higher than the time to decode the zero error case. And this is true for both the fast Chien search and the BTA algorithm. However the average decode time for the correction of the maximum number of errors correctable by the code (14 in this case) is much greater for the Chien search. It is 5.40e-4 for the Chien search and only 1.24e-4 for the BTA algorithm. So for the parameters that we have assumed, if the decoding needs to be fast for each correctable error, as would be the case for a worst case optimized system, then the BTA algorithm is a better solution.

Table 4 - Fast Chien Search Root Finder

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 -- 1.00	0 -- 7.19e-001
1 -- 3.6286e-001	1 -- 8.20e-005	1 -- 1.14	1 -- 4.13e-001
2 -- 1.3624e-001	2 -- 8.40e-005	2 -- 1.17	2 -- 1.59e-001
3 -- 4.0239e-002	3 -- 8.40e-005	3 -- 1.17	3 -- 4.69e-002
4 -- 9.7427e-003	4 -- 8.60e-005	4 -- 1.19	4 -- 1.16e-002
5 -- 1.9934e-003	5 -- 1.22e-004	5 -- 1.69	5 -- 3.38e-003
6 -- 3.5263e-004	6 -- 1.54e-004	6 -- 2.14	6 -- 7.54e-004
7 -- 5.4895e-005	7 -- 1.82e-004	7 -- 2.53	7 -- 1.39e-004
8 -- 7.6259e-006	8 -- 2.06e-004	8 -- 2.86	8 -- 2.18e-005
9 -- 9.5612e-007	9 -- 2.68e-004	9 -- 3.72	9 -- 3.56e-006
10 -- 1.0920e-007	10 -- 3.24e-004	10 -- 4.50	10 -- 4.91e-007
11 -- 1.1450e-008	11 -- 3.80e-004	11 -- 5.28	11 -- 6.04e-008
12 -- 1.1096e-009	12 -- 4.36e-004	12 -- 6.06	12 -- 6.72e-009
13 -- 9.9943e-011	13 -- 4.85e-004	13 -- 6.74	13 -- 6.73e-010
14 -- 8.4074e-012	14 -- 5.40e-004	14 -- 7.50	14 -- 6.31e-011

Table 5 - BTA Root Finder

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--1.00	0--7.19e-001
1 -- 3.6286e-001	1 -- 8.40e-005	1--1.17	1--4.23e-001
2 -- 1.3624e-001	2 -- 8.50e-005	2--1.18	2--1.61e-001
3 -- 4.0239e-002	3 -- 8.63e-005	3--1.20	3--4.82e-002
4 -- 9.7427e-003	4 -- 8.71e-005	4--1.21	4--1.18e-002
5 -- 1.9934e-003	5 -- 9.14e-005	5--1.27	5--2.53e-003
6 -- 3.5263e-004	6 -- 9.43e-005	6--1.31	6--4.62e-004
7 -- 5.4895e-005	7 -- 9.71e-005	7--1.35	7--7.40e-005
8 -- 7.6259e-006	8 -- 1.00e-004	8--1.39	8--1.06e-005
9 -- 9.5612e-007	9 -- 1.04e-004	9--1.44	9--1.38e-006
10 -- 1.0920e-007	10 -- 1.08e-004	10--1.50	10--1.64e-007
11 -- 1.1450e-008	11 -- 1.12e-004	11--1.56	11--1.78e-008
12 -- 1.1096e-009	12 -- 1.16e-004	12--1.61	12--1.79e-009
13 -- 9.9943e-011	13 -- 1.18e-004	13--1.64	13--1.64e-010
14 -- 8.4074e-012	14 -- 1.24e-004	14--1.72	14--1.44e-011

7. Average Throughput

Use Elastic Buffer for Streaming

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.

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.

8. Measuring Throughput Times

The R400 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. For more accurate timing measurements, if you are given the option, do not choose to generate random data and do not choose to compare.

9. Hybrid Binary BCH Decoders

The decode functions of R400 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 or 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 4 and 5 would be subtracted from all decode times.

10. 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.

11. Conclusions

Software encoding and decoding of binary BCH codes is possible for a range of throughput requirements. The R400 software implements several features that extend the range of software encoding and decoding throughput by increasing encoding and decoding speed.

BTA may give Better Performance

The BTA (Berlekamp Trace Algorithm) has been incorporated into R400. Root finding can be accomplished with either the BTA algorithm or a fast Chien search. In systems where it is the worst case decode time that is important, the BTA root finding algorithm may give significantly better performance than the fast Chien search.

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. In systems where it is average decode time that is important then the fast Chien search root finding algorithm may be satisfactory.

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.

Consider a hybrid solution

A hybrid decoding strategy has a performance between an all hardware decoding strategy and an all software decoding strategy. With a hybrid strategy, some of the decoding functions are performed in hardware and some are performed in software. Many of the software functions of R400 can be used with a hybrid strategy.

12. References

My references for the BTA algorithm are a paper [5] by Berlekamp and three papers [1,2,3] that discuss a BTA derivative algorithm called BTZ. In the R400 program I implemented much of the math from BTZ, but instead of implementing the special root finding methods of low degree polynomials by Zinoviev, I implemented alternative methods that have been know to me for many years. References [1-5] can be found on the internet, but there may be a fee for some of them.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

1. V. Herbert. Efficient root finding of polynomials over fields of characteristic 2, WEWoRC 2009, INRIA Paris-Rocquencourt, 2009.

2. V. Herbert. Efficient root finding of polynomials over fields of characteristic 2, INRIA Paris-Rocquencourt, 2008 or later.

3. B. Biswas, V. Herbert. Efficient root finding of polynomials over fields of characteristic 2, CRI INRIA Paris-Rocquencourt, 2008 or later.

4. V.A. Zinoviev. On the solution of equations of degree 10 over finite fields GF(2^q). In Rapport de recherche INRIA 2829, 1996.

5. E. Berlekamp (1970), Factoring polynomials over large finite fields, Mathematics of Computation, v. 24, 1970, pp. 713-735.

6. ChannelScience web site www.ChannelScience.com – FREE downloadable software: FastBchEnDecR400.

7. N. Glover and T. Dudley, *Practical Error Correction Design for Engineers* - Revised Second Edition, Cirrus Logic, 1991.

8. Cox, Hassner, Trager, and Winograd, Root solver and associated method for solving finite field polynomial equations, Patent US 6,792,569 B2, 2004

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®.