

Reed-Solomon Decoding on the StarCore Processor

By Dana Taipale, Iantha E. Scheiwe, Tina M. Redheendran

An increasingly popular way to speed up signal processors is to place multiple processor units on a single chip. One challenge in programming applications for such designs is to make effective use of the multiple processing units. This application note discusses how to meet this challenge in an application that implements a Reed-Solomon decoder on a signal processor with four data arithmetic units and two address generation units. The problem posed here is uniquely instructive in that the decoding is a commonly needed function and requires operations not typically performed using specialized operation codes. Therefore, we can focus on the use of multiple processors to accomplish the task efficiently.

The Reed-Solomon decoder is implemented on the StarCore, which is the core of Motorola's new family of DSPs. The use of StarCore for the Reed-Solomon algorithm yields two main benefits. First, the multiple data and address processors allow many of the Reed-Solomon computations to complete in parallel. Second, the modulo arithmetic of the address arithmetic units allows easy log/alog computation using table look-up for finite fields.

This document presents the theoretical background for the Reed-Solomon algorithm, including finite fields, irreducible and primitive polynomials, and a general discussion of error correcting codes and block codes. The steps required for this implementation of Reed-Solomon decoding are discussed in detail and illustrated with code examples.

Contents

1 Theory	2
1.1 Error Correcting Codes.....	2
1.2 Finite Field Theory	2
1.3 Reed-Solomon Theory.....	8
2 Implementation	9
2.1 Multiplication Method.....	10
2.2 Reed-Solomon Program	13
3 Conclusion	23
4 References	23



1 Theory

This section presents background information on error correcting codes, a development of finite fields, and theory on the functioning of Reed-Solomon codes.

1.1 Error Correcting Codes

Several different error correcting codes are used in communication systems. Two common code types are convolutional codes and block codes. Viterbi decoding is the most popular way to decode convolutional codes. In convolutional coding, each bit depends on the current bit as well as on some number of previous bits. Therefore, convolutional codes have memory. Redundancy in convolutional codes is added by increasing the memory. Reed-Solomon decoding is not based on convolutional codes, but instead is a type of linear block code. In block coding, each code word block is independent of previous code words. Redundancy in block codes is achieved by adding redundant bits to help combat noise. Reed-Solomon codes were developed through work completed in 1959 and 1960 [1].

1.2 Finite Field Theory

This section covers both simple finite fields and field extensions.

1.2.1 Simple Finite Fields

For block error correcting codes, the arithmetic is done in finite fields. These constructs are very similar to the most widely recognized and used (nonfinite) field, the field of real numbers. All fields have many common properties, including addition and multiplication operations, commutativity and associativity of both operations, a distributive property, and inverse elements for all elements except a multiplicative inverse for the additive identity (that is, you cannot divide by 0) [2].

Finite fields differ from the real number field in that they have only a finite number of elements. They are therefore easier to understand and manipulate than real numbers, but they may seem unusual at first. We use finite fields for block codes in order to work with finite precision yet still guarantee exact results. There must be a finite number of elements in the field so that finite precision can represent every possible result exactly. We want to take as many of the familiar properties as possible to the new field. We also need to set up and solve

Definitions

Cyclic codes Codes for which cyclic shifts of elements in a codeword are codewords.

Error Location Polynomial (ELP) A polynomial whose roots indicate the position of errors in the receive vector.

Field A set of elements for which we can do addition, subtraction, multiplication, and division without leaving the set.

Finite Field A field with a finite number of elements.

Galois Field Another term for a finite field, named in honor of the discoverer of finite fields.

Generator Polynomial A nonzero code polynomial of minimum degree that is unique in a given cyclic code. Multiplying the message to be encoded with the generator polynomial creates the code word to be transmitted.

Irreducible Polynomial (over GF[2]) A $P(x)$ that cannot be factored into a product of polynomials of smaller degree with coefficients from GF[2].

Minimum distance The minimum number of places in which two distinct code words differ.

Primitive Polynomial An irreducible polynomial $P(x)$ of degree m where the smallest positive integer n for which $P(x)$ divides X^n+1 is $n=2^m-1$.

Shortened Code A method to reduce processing requirements while remaining within a large field. A code is shortened so that only a portion of the received codeword has the transmitted message, and the remainder of the codeword is padded with zeroes and is not decoded.

Syndrome The result of a parity check performed on the received codeword to determine whether it is a valid member of the codeword set. The syndrome equals zero only if the received data is a code word.

Note: For more information on these terms, consult the References section at the end of this application note.

equations and add, subtract, multiply and divide. We can keep the familiar properties (commutative, associative, and distributive) for these operations. In general, this might be hard to do (or verify), but it is possible to create new fields that bring these properties with them by starting with familiar objects.

One back door approach to finite fields is to begin with a different problem. Suppose that for the two integers 490 and 2142, we want to find the largest common factor. We could use Euclid's algorithm, an example of which appears in **Table 1-1**.

Table 1-1. Euclid's Algorithm, Example

large =	small * factor +	remainder
2142	490*4	182
490	182*2	126
182	126*1	56
126	56*2	14
56	14*4	0

The largest common factor is 14. It is easy to see why this works. Since the largest common factor divides the numbers in the first two columns evenly (this is what largest common factor means), it must also divide the remainder evenly. It is not possible to divide both sides of an equality by the same factor to obtain a zero remainder for one result and a nonzero remainder for the other. Therefore, the largest common factor evenly divides entries in all columns of the table. After filling in the first line, we take the "small" entry from column two and the remainder entry from column three and move them to columns one and two of the second line, respectively. Since the largest common factor divides the "small" entry and the remainder evenly, we can repeat the process. Because remainders are always smaller than their divisors, the table entries must decrease. Eventually, we must get to our desired factor. Once we do, it will of course divide evenly, so we can use a zero remainder as the condition to stop the process. The largest common factor appears in column three immediately above the zero remainder entry.

Now, let's consider integers. Integers have nice addition and multiplication properties, but there is an infinite number of integers. To make a new finite construction, we do addition, subtraction and multiplication on integers using the customary rules, but we divide all results by one chosen integer N and keep only the remainders as results. The resulting construct is called the integers modulo N , or $\mathbf{Z} \bmod N$, where \mathbf{Z} is understood to be a symbol for the integers.

A concrete example is the integers modulo 5. For this example, the elements (possible remainders after division) are 0, 1, 2, 3, and 4. Therefore, $25=0$, $21=1$, $-1=4$, and so on. We need only consider the remainders. We can multiply, add, and subtract just as we normally would. We take the remainder of the result, and everything works out. For multiplication, $2*3=6=1$, $4*2=8=3$, and $0*2=0$. For addition, $2+3=5=0$, and $2+4=6=1$ and for subtraction $2-3=-1=4$, and $4-1=3$. Our only problem is dividing. If we consider a multiplication table for the integers modulo 5, shown in **Table 1-2**, we see that this is not a problem either.

Table 1-2. Integers Modulo

xx xx	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Note: $1*1=1$, $2*3=1$, and $4*4=1$. In particular, each nonzero element has a multiplicative inverse, $2=3^{-1}$, $4=4^{-1}$, and so on. To do division, we multiply by the multiplicative inverse. Another example that does not work out as neatly is the element 2 in the integers modulo 6. The multiples of 2 are 0, 2, 4, 0, 2, 4, no 1. Thus, 2 has no inverse, which means that we cannot divide by 2. This could be a major problem.

To ensure that the new constructs are ones in which division can occur, we start with Euclid’s algorithm. We can start at the bottom line of the **Table 1-1**, and work up to express 14 (our largest common factor) as a combination of our original two numbers:

$$126 - 56 \times 2 = 14$$

However, the third line of **Table 1-1** can express 56 differently:

$$126 - [182 - 126 \times 1] \times 2 = -182 \times 2 + 126 \times 3 = 14$$

Of course, we can use line two of the table and repeat this process to get the following:

$$-182 \times 2 + [490 - 182 \times 2] \times 3 = 490 \times 3 - 182 \times 8 = 14$$

and finally:

$$490 \times 3 - [2142 - 490 \times 4] \times 8 = -2142 \times 8 + 490 \times 35 = 14$$

It is not important to remember the process. The main thing to remember is that given any two integers, X and Y, we can always find two other integers, A and B, such that $AX+BY = \text{LCF}(X,Y)$, that is, their largest common factor.

Suppose we set $Y=5$, or whatever we want as a modulus (which is the number by which we are dividing to obtain a new field). If X is a number that is relatively prime to Y (no common factors except 1, so $\text{LCF}(X,Y) = 1$), it is always possible find an A and a B so that $AX+B5=1$, or $AX=1-5B$. Since $1-5B$ is a multiple of 5 with remainder 1, $AX=1$ in our modulo 5 construct. This means that $X^{-1} = A$, and so X has an inverse. Therefore, to ensure that all numbers (less than 5) have an inverse, they must all be relatively prime to 5. Since 5 is a prime number, this is true. In general, if the divider (modulus) is a prime, we can always get inverses. The result is called a finite field. A finite field of p elements is also called a Galois field of p elements, denoted $\text{GF}[p]$.

1.2.2 Field Extensions

In almost all coding cases, we want to begin with the field with only two elements in it, namely zero and one. Arithmetic for this binary field is $\mathbf{Z} \bmod 2$ arithmetic. The addition and multiplication tables are as shown in **Table 1-3** and **Table 1-4**.

Table 1-3. Addition Table

+ Zmod2	0	1
0	0	1
1	1	0

Table 1-4. Multiplication Table

X Zmod2	0	1
0	0	0
1	0	1

Now we take the binary field and extend it (instead of starting with plain integers) in a slightly different way. It is still important to be able to bring along all of the number properties, and in particular to ensure that division is possible. To extend the binary field, consider the set of all polynomials with binary coefficients:

$$F[x] = \sum_{i=0}^{\infty} b_i \cdot x^i$$

where each b^i is an element of the binary field. To make the result finite again, we divide and keep only the remainders. We divide by prime polynomials. A polynomial is prime over the binary field if it cannot be factored into polynomials of smaller degree with only binary coefficients. Such a polynomial is also called *irreducible*. This is the equivalent of dividing by prime numbers when working with integers.

For example, we can create a field with 16 elements by taking all polynomials with binary coefficients, and keeping their remainders after dividing by a degree four polynomial, which is a polynomial that does not factor. One such polynomial is $P(x)=x^4+x+1$. Remainders have the form $R(x)=b_3x^3+b_2x^2+b_1x+b_0$ where coefficients are again binary. Addition and subtraction are easy. We just use binary field ($\mathbf{Z} \bmod 2$) addition on the coefficients of terms with matching exponents for x , keeping the powers of x unchanged. Addition in this field is easy because it is just an EXCLUSIVE-OR of the arguments. Subtraction, too, is easy because in the binary field, subtraction is the same as addition. Multiplication is more involved. We can multiply the polynomials, divide the result by the prime polynomial, and take the remainder to obtain the result. Alternatively, we can let α be a solution to $P(x)$, $\alpha^4+\alpha+1=0$ (alpha is defined to be a solution to the equation). We substitute alphas for x 's in all calculations and rearrange the equation for alpha, $\alpha^4=\alpha+1$ to keep the exponents under four. For example,

$$\alpha^3(\alpha^2 + \alpha) = \alpha^5 + \alpha^4 = \alpha \cdot \alpha^4 + \alpha^4 = \alpha \cdot (\alpha + 1) + (\alpha + 1) = \alpha^2 + \alpha + \alpha + 1 = \alpha^2 + 1.$$

Finite Field Theory

Probably, you have already been doing this for years. This technique is the only way to extend fields (for finite dimensional extensions). Consider polynomials with real coefficients, and divide by the polynomial x^2+1 . The result is remainders of the form $ax+b$, where a and b are real. It is customary to replace the polynomial variable x by a symbol that is defined as the solution to the dividing polynomial (note that the polynomial is not factorable over the real numbers). Our solution is the square root of -1 (a symbol that means “the solution to $x^2+1=0$ ”) arbitrarily denoted as i , so the remainder is written as $ai+b$. Again, we keep the exponents of i down in this case by noting that the square of i is -1 . The result is the complex field, with the complex arithmetic that is probably familiar. Since this is the only way to extend fields, one consequence is that it is not possible to extend the complex numbers into a larger field because all polynomials factor into linear terms. Thus, we see vector spaces and the like, but nothing with a nice multiplication and division beyond the complex numbers.

Another convenient way to multiply in finite fields is to use a logarithm table. We construct the field using powers of alpha and again keep the exponents under four (so they look like remainders). Notice that the prime polynomial $P(x)$ has the property that every nonzero element of the field is a power of alpha. Polynomials with this additional property are called primitive polynomials. We can always find primitive polynomials of any degree over $GF[2]$. All primitive polynomials are prime. Not all prime polynomials are primitive. In general, polynomials with this property are not easy to identify by any technique other than creating consecutive powers of the alpha defined by the polynomial and determining whether the number of distinct results corresponds to the number of nonzero field elements. If true, the polynomial is primitive and α is called a primitive element. However, in many cases, the primitive polynomial is already supplied—for example, by a standards body that specifies the use of the code.

One way to multiply is to perform a table look-up for the exponents, add exponents (for this example, modulo 15), and then perform a table look-up to undo the log. **Table 1-5** shows a log table for the $GF[16]$ field. The first column shows the exponent of alpha, and the second column represents alpha. This representation is:

$$\alpha^i = \sum_{k=0}^3 b_k \alpha^k$$

Table 1-5. $GF[16]$ Logarithms

i	α^i	
	Binary form	Polynomial form
0	0001	1
1	0010	α
2	0100	α^2
3	1000	α^3
4	0011	$\alpha + 1$
5	0110	$\alpha^2 + \alpha$
6	1100	$\alpha^3 + \alpha^2$

Table 1-5. GF[16] Logarithms (Continued)

i	α^i	
	Binary form	Polynomial form
7	1011	$\alpha^3 + \alpha + 1$
8	0101	$\alpha^2 + 1$
9	1010	$\alpha^3 + \alpha$
10	0111	$\alpha^2 + \alpha + 1$
11	1110	$\alpha^3 + \alpha^2 + \alpha$
12	1111	$\alpha^3 + \alpha^2 + \alpha + 1$
13	1101	$\alpha^3 + \alpha^2 + 1$
14	1001	$\alpha^3 + 1$
15	0001	1

A log table promotes easy multiplication and division. For example, $0011 * 1001$ can be accomplished by noting that the logs are 4 and 14, respectively. Multiplying, we add the logarithms, so $4+14=18$. We notice that the table starts repeating at 15, so multiples of 15 can be subtracted without changing the result (it is like dividing by $0001=1$). Therefore, a log of 18 is the same as a log of $18-15=3$. Taking the alog by finding the table entry that matches in column one and finding the corresponding entry in column two, we get an answer of 1000. Addition is the same as subtraction and is an EXCLUSIVE-OR of the component vectors. Therefore, $0011+1001=1010$. We can also use a log notation to represent numbers, that is, $0011=\alpha^4$ and $1001=\alpha^{14}$. We can then rewrite the examples as $\alpha^4 * \alpha^{14} = \alpha^3$, and $\alpha^4 + \alpha^{14} = \alpha^9$. Both kinds of representation are used, depending on convenience. Multiplication is easy using the log notation, and addition is easy using the alog notation. Division is like multiplication, except that we subtract the logarithms instead of adding.

We performed a log multiplication in which the logs were added, resulting in an integer that goes beyond the limits of the log table. However, logs can be reduced because the table begins to repeat. In general, for a field of size N, logarithm results can be taken modulo N-1.

In summary, by extending the fields as described here, we inherit all of the properties from polynomials with binary field coefficients [2]. Multiplication and addition have the following properties:

- They are closed (operation results stay in the field they start in).
- They have identity elements ($0=0000$ for addition and $1=0001$ for multiplication).
- They are both commutative and associative.
- The pair of operations has the distributive property.

Since we modulo by a prime, we can also divide by any nonzero field element. All of these properties allow us to perform arithmetic in this field much as we do for real numbers.

Reed-Solomon Theory

However, this field can be represented by a finite binary word length. In fact, for extensions of the binary field such as those developed here, the field elements can be exactly represented by a binary word of length equal to the degree of the polynomial used to create the field. Each bit of the word is a coefficient of the remainder representation shown earlier. This convenience is part of what makes binary field extensions popular for digital processing applications. In addition, the ability to represent all field elements exactly in a fixed word length is quite useful for applications such as error correction coding, where we do not want to introduce more errors because of precision problems.

1.3 Reed-Solomon Theory

This section considers the parameters of error-correcting code, the decoding process, and the overall benefits of the Reed-Solomon algorithm.

1.3.1 Parameters

The following parameters define a t error correcting Reed-Solomon code with symbols from $GF[q]$:

- Block length: $n=q-1$
- Number of parity-check digits: $n-k=2t$
- Minimum distance: $d_{\min}=2t+1$

q is a prime number to the power of m ($q=\text{Prime}^m$). In this application note, $q=2^m$. The length of a Reed-Solomon code is one less than the size of the code symbol alphabet. The minimum distance is one greater than the number of parity-check symbols. This application note uses a code for which the generator polynomial is of the form:

$$g(x) = \prod_{i=1}^4 (1 + x\alpha^i)$$

Determining a generator polynomial is not required to decode Reed-Solomon code. However, it is needed as part of the encoder and is therefore useful for debugging a decoder program since the generator creates valid codewords. Variations on how a codeword can be formed are beyond the scope of this application note. Consult references [4] or [1] for further details.

Codewords can be formed by using the information symbols to create an information polynomial (partition bits of the input to create elements of the finite field as coefficients of the information polynomial):

$$i(x) = \sum_{k=0}^{q-2t-2} i_k x^{k+4}$$

Divide this polynomial by the generator $g(x)$ to find the four remainder symbols. Append the remainder symbols to $i(x)$ to create a codeword of length $m+4$ field elements or $(m+4)\log(N)$ bits. The polynomial form of the generator is not required to implement the Reed-Solomon decoder as described in this application note. However, this knowledge is useful in creating a test environment for the decoder.

1.3.2 Decoding Process

Three questions must be answered when a message is decoded using Reed-Solomon decoding:

1. How many errors exist?
2. Where are they located?
3. What are the error values?

Answers to these questions can be used to correct any errors that exist in a received message.

The roots of the generator polynomial ($\alpha, \alpha^2, \dots, \alpha^{2t}$) are used to calculate syndromes, which is the first step in Reed-Solomon decoding. Syndrome calculation uses knowledge of the roots of the generator polynomial to determine whether any errors occurred during a message transmission. If any of the syndromes are nonzero, then additive errors were introduced into the message and must be corrected. The equation for calculating syndromes is as follows:

$$S_j = \sum_{i=0}^{n-1} v_i \cdot \alpha^{i(j+1)}$$

and the syndrome polynomial is:

$$S(z) = \sum_{j=1}^{2t} S_j z^{j-1}$$

Every Reed-Solomon codeword has $2t$ consecutive syndromes that are zero. Any nonzero components in the syndromes of the received word are solely due to additive errors. Once the syndromes are calculated, they are used to generate solutions to the equation $\Lambda(z) \cdot S(z) = \Omega(z) \text{ mod } z^{2t}$. Λ is a nonzero polynomial of the smallest possible degree that satisfies this equation with the constraint that the resulting polynomial Ω must have degree smaller than Λ . Once Λ is found, the next step is to factor Λ into linear components. The zero of each component is a power of α , and this exponent is the position of an error. We can then use Ω and Λ to determine what the error values are and subtract them to correct the codeword.

1.3.3 Reed-Solomon Benefits

Using Reed-Solomon codes, a system designer can exactly specify decoder parameters based on the number of errors to be corrected. When a two-error correcting Reed-Solomon coder is chosen, the system designer knows that the decoder can correct two errors or less. Some other types of coders have less definite error correcting characteristics. This ability to definitively specify the number of errors to be corrected is one of the benefits of Reed-Solomon codes. Another benefit is that Reed-Solomon codes give the largest minimum distance for any linear code with the same encoder input and output block lengths [3]. Therefore, Reed-Solomon codes can definitively correct a larger number of errors with the same encoder input and output lengths than many other linear codes.

2 Implementation

This section discusses implementation decisions made to perform the finite field arithmetic. We consider the possible implementation methods and the trade-offs of each. We state our chosen method and show supporting tables.

2.1 Multiplication Method

There are several ways to support arithmetic in finite fields. When a log notation is chosen, multiplication is easy and addition is difficult. Conversely, when an alog notation is chosen, addition is easy and multiplication is difficult. These choices are somewhat arbitrary, and both forms appear where convenient in the code (first shown in **Section 2.2**). For ease of presentation, this section assumes an alog form.

There are a number of ways to do multiplication. One technique is to treat the alog forms as polynomials with binary coefficients, perform the polynomial multiplication, and either divide by the prime polynomial used for this field representation, or use a reduction technique. The advantage of this technique is that it requires almost no memory. The disadvantage is that it can take multiple processing steps to do the reduction back to a standard remainder form, as well as to perform the polynomial multiply. Therefore, it can be slow.

A second approach is to move to the other extreme and create a table indexed by the arguments of the multiplication whose entries are the product. This solution is very fast, but it can require a large memory. For example, in industry today, the field size is often 256, which means a look-up table is $256^2 = 65,536$ entries long. This number of entries is certainly achievable but is impractically large for typical DSP memories today.

A third approach is to perform a log table look-up to convert from alog to log forms and back. This approach still demands tables (one for log and one for alog), but each table is only the size of the field. 256 entries per table is an acceptable size for our implementation. This approach is not as fast as table look-up, since an example multiply would involve looking up log forms for the two arguments, adding the arguments, reducing the result to fit in the table, and taking the alog of the result. However, it requires far fewer operations than the first approach, and it is the approach taken for the implementation described in this document.

Implementation of log table look-up for alog/log conversions has advantages on the StarCore processor, since some of the work can be parallelized (as discussed later), and some of the computation can occur in the address generation units, which have a natural modulo capability that is quite useful. If we did this calculation in a DALU part of the processor, this modulo would require more processing cycles. However, we can do this part of the calculation in the AGUs of the processor and make use of the fact that address registers can be configured to do modulo N-1 reduction automatically with no performance penalty. One problematic area for log multiplication is that field values of zero do not have finite logarithms. On the StarCore, this problem is handled by creating mask words at the same time that the logarithms are being found. These masks can be created simultaneously because they are created in the DALU, not the AGU. An example multiply process proceeds as follows:

1. Data is moved to the address generation unit, and simultaneously an all 1's or all 0's mask is created in a DALU.
2. The address generation unit does a log table look-up.
3. The result is ANDed to another log to multiply, and simultaneously the alog table look-up on the result is performed, with results going to a data register.
4. The result is ANDed with the mask, which is all 1's if the original values are nonzero, and all 0's if one of the arguments was zero.

The result of the ANDing is either the product in alog form for nonzero arguments or 0 if one of the arguments was zero. This is the desired result.

2.1.1 Generating Look-up Tables

Our implementation of the decoder requires three tables to be generated. Two of the tables are related to performing finite field arithmetic. These are the alog-to-log form table look-up and the log-to-alog table look-up. The third table is a shortcut way to factor the quadratic polynomials with finite field elements, which is needed as part of the error correction process.

2.1.1.1 Log and Alog Table Generation

The key to generating log and alog tables is to create a method for multiplying a field element in alog form, by alpha, and obtaining a result in alog form. The multiplication can then iterate by alpha on the result. Each iteration generates another line in the alog table (in the correct order) by filling the alog table location that corresponds to the iteration index (0 to N-2) with the alog result of the multiplication. The log table can be filled in at the same time by taking the alog result of the iteration as the index of the log table and writing the iteration number in that location. Conversions can now be done in either direction by taking the number to be converted as the index into the conversion table and reading the converted number out of the table as the result.

Multiplication by alpha is relatively simple and need not be optimized since table generation can occur long before program execution. If the alog form is used, multiplication by alpha is equivalent to a left shift of the binary coefficients of the alog form. This is not surprising, since this form is a representation of:

$$\alpha^i = \sum_{k=0}^3 b_k \alpha^k$$

If our alog representation is (b_3, b_2, b_1, b_0) , then moving each coefficient one position to the left is like multiplying the corresponding α 's in the sum by alpha. The problem is that the leftmost coefficient is beyond the representation and needs to be reduced. We use the reduction rule obtained from the prime polynomial. For the example $\alpha^4 = \alpha + 1$, take the binary representation of the right side of the equation (0011) and exclusive-or it with the shifted result. Then mask off extra leftside bits to complete the multiply. Performing the multiply to generate an alog table yields the following:

- Start with 0001 (start with the number 1).
- Left shift to get 0010 (alpha)—no extra 1 off the left side, so we are done.
- Left shift to get 0100 (alpha squared)—no extra 1 off the left side, so we are done.
- Left shift to get 1000 (alpha cubed)—no extra 1 off the left side, so we are done.
- Left shift to get 10000—extra 1 on left, so exclusive-or 10000 with 0011 (right justified) and mask off leading 1 to get 0011 (alpha to the fourth).
- Left shift to get 0110.
- Left shift to get 1100.
- Left shift to get 11000—extra 1 so exclusive or 11000 with 0011 and mask off leading 1 to get 1011 (alpha to the fifth).

These results are identical to the table entries in our table and are a partial listing of the alog table for this field. We can fill in a log table at the same time by switching the iteration number and the result.

Multiplication Method

2.1.1.2 Factor Table

While performing the error correction, it is necessary (for correcting two errors) to factor quadratic polynomials over the finite field. This is a bit more difficult for finite fields than for real numbers because the quadratic formula breaks down and cannot be used here. Exhaustive search for roots is one approach, but this application note, which considers only codes that can correct up to two errors, uses a table look-up.

As before, the size of the field is a major consideration in determining whether this technique is practical. The field size we consider is 256. A straightforward table look-up for roots quadratic polynomials would require at least 256^2 entries, but there is a good strategy for reducing the table.

In general, quadratic polynomials over the field have the form:

$$q(x) = f_2x^2 + f_1x^1 + f_0x^0$$

where the f_i 's are elements of the finite field (not binary). We can easily reduce this a bit for the special case of root determination by noting that:

$$q(x) = f_2x^2 + f_1x^1 + f_0 = 0 \Rightarrow f_0 \left(\frac{f_2}{f_0}x^2 + \frac{f_1}{f_0}x^1 + 1 \right) = g_2x^2 + g_1x^1 + 1 = 0$$

Thus, there are only two parameters instead of three to index the table. In fact, this is the form of the quadratic that we start with in the routine (the algorithm that determines this polynomial determines it in this form for this implementation). However, two parameters of field size still make a table that is too large. It turns out that we can make a further simplification, reducing the table to a one parameter size, while slightly increasing the computation. This approach still requires far less execution time than a trial and error root search. Our strategy is to change variables, substituting $x = y/g_1$ into the equation and obtaining the following:

$$g_2 \left(\frac{y}{g_1} \right)^2 + g_1 \frac{y}{g_1} + 1 = g_2 \left(\frac{y}{g_1} \right)^2 + y + 1 = hy^2 + y + 1 = 0$$

This result has only one parameter of field size, $h = g_2/g_1^2$. Notice that this substitution fails if g_1 is zero, but results in a special form for the quadratic polynomial that implies a decoding failure. Hence, we can check for this exception and report a decoding failure if necessary. Using this substitution, we can create a field size look-up table, indexed by the parameter h , to list the roots y_1, y_2 of the polynomial $hy^2 + y + 1 = 0$. Once we have those roots, we determine the corresponding roots x_1 and x_2 equal to g_1/y_1 and g_1/y_2 , respectively.

To create the table, we start with linear polynomials of a special form that, when multiplied together, result in a quadratic of the form $hy^2 + y + 1$. It is not difficult to determine what this form is. Since the constant term is 1, we start with linear polynomials of the form $(Ay + 1)$, and $(By + 1)$. Multiplying to get a quadratic, we obtain $ABy^2 + (A+B)y + 1 = hy^2 + y + 1$. Equating the terms, we obtain $AB = h$, and $A+B=1$. This second equality gives us the relation $B=A+1$. To create the table, we perform the following operations for each nonzero element A of the field:

1. Determine $A+1$.
2. Multiply by A to determine $A(A+1)$.
3. Save the values of A and $A+1$ in the table index corresponding to the log form of $A(A+1)$.

To use the table, we determine h (log form) and use it as an index into the factor table to obtain A and $A+1$. In this application note (and most of the prevalent applications of Reed-Solomon coding in use), the field size is 256 or less, so both roots can be stored in a word of size 16.

Some table entries are not used. In fact, when polynomial parameters corresponding to decoding errors are excluded, the number of table entries is $N/2-1$, for a field size of N . Nonvalid entry table entries are filled with zero values. A zero value corresponds to a decoding error if that table entry is read.

2.2 Reed-Solomon Program

The Reed-Solomon decoder implementation described in this section is defined by the following parameters:

Block length: $2^8-1=255$; $n=64$, $k=60$

Number of parity-check digits: $n-k=4$

Minimum distance: $d_{\min}=5$

Generator polynomial: $g(x)=(x+\alpha)(x+\alpha^2)(x+\alpha^3)(x+\alpha^4)$

Primitive polynomial: $P(x)=1+x^2+x^3+x^4+x^8$

Notice that n does not equal the block length. We are implementing a shortened code in which 64 received symbols in the block have useful information. The rest of the block is padded with zeros and is not decoded. This method greatly reduces the computational requirements of the decoder. Since there are four parity-check digits, $t=2$, we can correct two or fewer errors. The encoder uses the generator polynomial, $g(x)$, shown. We use this information when testing the decoder. The primitive polynomial for GF[256], $P(x)$, generates the log/alog tables used in decoding.

2.2.1 Calculating the Syndromes

To determine whether errors occurred during transmission, the syndromes must be calculated. If the syndromes equal zero, then no errors are detected and no corrections are needed. However, if the syndromes do not equal zero, we use them to determine the error location polynomial. In our implementation, syndrome calculation is the most compute-intensive portion of the code. Cycle count varies proportionately with the specified codeword size. The generator used by the transmitter encoder is $g(x)=(x+\alpha^1)(x+\alpha^2)(x+\alpha^3)(x+\alpha^4)$. Therefore, we must calculate four syndrome values to determine if any errors were received. The StarCore AGU allows us to retrieve two pieces of data each cycle, so we partition the syndrome calculation into two steps. The first step calculates two syndrome values. The process repeats for the remaining two syndromes. We can thus retrieve values required for each syndrome at each cycle, while the four available ALUs complete calculations required for each syndrome.

It would be natural to calculate S_0 and S_1 first, followed by S_2 and S_3 . However, looking ahead to calculation of the error location polynomial and analyzing register usage, we determine that it is most efficient to calculate the syndromes in a different order. First, we calculate S_0 and S_3 . In the second iteration, we calculate S_1 and S_2 . This allows us to leave S_1 and S_2 in the registers for immediate use by the error-location polynomial calculations.

Example 2-1. Code for Calculating Syndromes 0 and 3

move #ALOGTBL1+2,r1	move #ALOGTBL1+14,r3	;r1&r3->ALOGTBL plus offsets
move #LOGTBL,r2	move #509,m0	;r2->LOGTBL, r1&r3 mod 255 words
move.l #00008080,MCTL		;r1,r3 (ALOGTBL ptrs) are mod m0
move #000,b1	move #000,b3	;b1,b3 base address at \$0
dosetup3 _SYN03LP		
doen3 #CWLENGTH/2-1		
move #MASK,r12	move.2w (r8)-,d8:d9	;d8=v62,d9=v63
move.w #1,d10	move #-LOGTBL,d12	
move.4w (r12),d4:d5:d6:d7		
tsteg d8	subl d12,d8	;check if v62=0
sub #1,d7	deceq d4 IFT	;setup 0masks
move d8,r4	move d8,r0	
tsteg d9	subl d12,d9	;check if v63=0
sub #1,d6	deceq d5 IFT	;setup 0masks
move d9,r5	move d9,r6	
move.2w (r8)-,d8:d9	neg d7 neg d6	;d8=v60,d9=v61

The code shown in **Example 2-1** begins the Reed-Solomon syndrome calculation and is also the beginning of the complete decoder code. As discussed in **Section 2.1.1**, we use log tables to calculate the syndromes. Therefore, the first line of code in **Example 2-1** initializes the pointer to the appropriate offset in the alog table for the α values. R1 is used in the S_0 calculations while r3 is used in the S_3 calculations. Each syndrome uses a different alpha offset and therefore requires a different offset from the beginning of the alog table.

After setting the alog table offset we initialize r2 to point to the beginning of the log table. We also initialize registers r1 and r3 to be modulo so that calculations using those registers remain within the boundaries of the alog table. The base address of the modulo is set at \$0, the address of the alog table in memory. The syndrome calculations require looping, so the loop address and counter are also initialized.

The first step in calculating the syndromes is to set up the loop. The loop length is equal to the codeword length, 64, divided by two. The length is divided by two because we calculate two iterations of the equation in a single loop—odd and even iterations of the loop. To make the syndrome inner loop as efficient as possible, we set some registers prior to entering the loop. The equations to calculate the syndrome are:

$$\text{odd: } [(\dots(((v63 * \alpha^{2^i} + v61) * \alpha^{2^i + v59}) * \alpha^{2^i + v57}) * \alpha^{2^i + \dots}) * \alpha^{2^i + v1}] \alpha^i$$

$$\text{even: } (\dots(((v62 * \alpha^{2^i} + v60) * \alpha^{2^i + v58}) * \alpha^{2^i + v56}) * \alpha^{2^i + \dots}) * \alpha^{2^i + v0}$$

where α^i is the log table entry at offset i ($\alpha^{2^i} = \alpha^2$ for S_0 and $\alpha^{2^i} = \alpha^8$ for S_3) and v63 through v0 are the received codeword values. The results of these two equations are added together to form the syndrome value.

The first thing we do is retrieve v62 and v63, which are used for both S_0 and S_3 . These received values are placed in registers d8 and d9. They are also copied to r0, r4, r5, and r6 for later use in the loop calculation. We calculate the masks as discussed in section **Section 2.1**. Also, we retrieve the first four log table entries and place them in registers d[4-7]. Immediately before entering the loop we retrieve the next two receive data values, v60 and v61, and place them in d8 and d9, respectively. We are now ready to enter the syndrome loop calculation.

Since we have the received information, v63..v60, and alpha information, α^{2^i} , we can begin the calculation shown in the preceding equation. The syndrome loop calculation is shown in **Example 2-2**.

Example 2-2. Syndrome Inner Loop Calculation

```

sub d12,d7,d7   sub d12,d6,d6   neg d4         neg d5         move (r4),r4   move (r5),r5
sub d12,d4,d4   sub d12,d5,d5   move (r0),r0   move (r6),r6   ;masks are 0 or $ffff
move (r1+r4),d3 and d7,d3        and d6,d2      move (r1+r5),d2 move (r3+r0),d0 move (r3+r6),d1
; d3=log(v62*a2), d2=log(v63*a2)
; d0=log(v62*a8), d1=log(v63*a8)
eor d8,d3      eor d9,d2      and d4,d0      and d5,d1      move.4w (r12),d4:d5:d6:d7
; mask results to 0 if prev term=0
subl d12,d3    subl d12,d2    eor d8,d0      eor d9,d1      move.2w (r8)-,d8:d9
; d3=v62a2+v60, d2=v63a2+v61
; d0=v62a8+v60, d1=v63a8+v61
min d3,d7      min d2,d6      subl d12,d0    subl d12,d1    move d3,r4     move d2,r5
min d0,d4      min d1,d5      neg d7        neg d6        move d0,r0     move d1,r6
; prepare to log result

```

The **sub** instructions perform table offset calculations, then the **move** instructions actually execute the syndromes. R4 holds v62 coming into the loop, while r5 holds v63. The **move** instruction then takes the log of the received value. The second line of code repeats this process by taking the log of v62 and v63 again. The first log is required for the odd iteration of the loop, while the second log is for the even iteration of the loop. Next we calculate the alog of $v62 \cdot \alpha^2$. α^2 is pointed to by r1. Adding in the log domain is the same as multiplying in the alog domain. Therefore, by adding r1 and r4, we are actually multiplying the values $v62 \cdot \alpha^2$. The **move** instruction takes the alog of the multiplied value and places it in d3. The same is done for $v63 \cdot \alpha^2$, with the value placed in d2. The next line repeats this process using α^8 for the S_3 calculation. The results using α^8 are stored in d0 and d1. Now we EXCLUSIVE-OR the multiplied values with the received value, so d2 then holds $(v63 \cdot \alpha^2) + v61$. and d3 holds $(v62 \cdot \alpha^2) + v60$. The same is true of d0 and d1, although they have α^2 replaced with α^8 . We also retrieve the next log table entries and the next two received data, v59 and v58. The next received data are again placed in d8 and d9. Notice that everything is done four times; that is two of the multiplies are calculated from the syndrome calculation equation in each loop iteration. After the loop is complete, these values must be added together to complete the syndrome calculation. Also, the equation for two syndrome calculations executes in each loop. This process repeats until the two halves of the entire codeword are computed.

Example 2-3. Syndrome Storage

```

sub d12,d6,d6   neg d5         move (r5),r5     move #ALOGTBL1,r1 ;r1->ALOGTBL offset for a1
sub d12,d5,d5   move (r6),r6     move #ALOGTBL1+6,r3 ;r3->ALOGTBL offset for a4
add d12,d3,d3   add d12,d0,d0    move #SYNST,r10  ;r10->syndrome storage
lsr d3         lsr d0         move (r1+r5),d2  move (r3+r6),d1  ;d2=a*odd d1=a4*odd
tfr d10,d6     and d6,d2      tfr d10,d5       and d5,d1        ;mask to 0 if prev term=0
eor d3,d2      eor d0,d1      ;d2&d1=odd+even terms
min d2,d6      move d2,r5     min d1,d5        move d1,r6       ;prepare to log result
neg d6         neg d5         ;masks are 0 or $ffff
zxt.w d6,d5    zxt.w d5,d3    move (r2+r5),d4  move (r2+r6),d2  ;log of syndromes
move.2w d4:d5,(r10)+
move.2w d2:d3,(r10)
;store S0,0mask
;store S3,0mask

```

Once the loop is complete, we have almost finished calculating the two syndromes, S_0 and S_3 . In **Example 2-3**, we take the log of the odd portion of the syndrome calculation. We also reset the alog table pointers, r1 and r3. We need to set a pointer to the destination of our syndrome calculation. R10 points to the syndrome storage area. The fourth line of the **move** instructions multiplies the odd portion of the syndrome calculation by α^1 again, as the equation requires. These moves also take the alog so that the value is back in the alog domain. Now we perform an EXCLUSIVE-OR to add the even and odd portions of the calculation together. d1 holds the S_3 calculation while d2 holds the S_0 calculation. We prepare to log the results before storing them by setting their mask values. We then take the log of S_3 , which has been moved to r6, and the log of S_0 , which is in r5. These results are stored with their masks at the location to which r10 points.

Reed-Solomon Program

Calculation of S_0 and S_3 is complete, and we are ready to calculate S_1 and S_2 . The code in **Example 2-4** is almost identical to the previous syndrome calculations. The only changes are due to table offsets for the syndromes being calculated. Also, after the syndrome loop calculations, there is some code that sets up registers for the next sections of code.

Example 2-4. Code for Calculating Syndromes 1 and 2

```

move #ALOGTBL1+6,r1          move #ALOGTBL1+10,r3          ;r1&r3->ALOGTBL plus offsets
move #RECDATA-2,r8          ;reset input ptr
dosetup3 _SYN12LP
doen3 #CWLLENGTH/2-1

move.w #1,d10      move.2w (r8)-,d8:d9          ;d8=v62,d9=v63
move.4w (r12),d4:d5:d6:d7
tstreq d8          subl d12,d8          ;check if v62=0
sub #1,d7          deceq d4          IFT          ;setup 0masks
move d8,r4          move d8,r0
tstreq d9          subl d12,d9          ;check if v63=0
sub #1,d6          deceq d5          IFT          ;setup 0masks
move d9,r5          move d9,r6
neg d7             neg d6             move.2w (r8)-,d8:d9          ;d8=v60,d9=v61

loopstart3
_SYN12LP
sub d12,d7,d7      sub d12,d6,d6      neg d4          neg d5          move (r4),r4      move (r5),r5
sub d12,d4,d4      sub d12,d5,d5      move (r0),r0    move (r6),r6; masks are 0 or $ffff
move (r1+r4),d3    ;d3=log(v62*a4),d2=log(v63*a4)
; d0=log(v62*a6),d1=log(v63*a6)
move (r3+r0),d0    move (r3+r6),d1and d7,d3          and d6,d2      and d5,d1      move.4w (r12),d4:d5:d6:d7
eor d8,d3          eor d9,d2          and d4,d0      ;mask results to 0 if prev term=0
subl d12,d3        subl d12,d2          eor d8,d0      eor d9,d1      move.2w (r8)-,d8:d9
; d3=v62a4+v60,d2=v63a4+v61
; d0=v62a6+v60,d1=v63a6+v61
min d3,d7          min d2,d6          subl d12,d0     subl d12,d1     move d3,r4      move d2,r5
min d0,d4          min d1,d5          neg d7          neg d6          move d0,r0      move d1,r6
;prepare to log result

loopend3
sub d12,d6,d6      neg d5          move (r5),r5    move #ALOGTBL1+2,r1
; r1->ALOGTBL offset for a2
; r3->ALOGTBL offset for a3
sub d12,d5,d5      move (r6),r6      move #ALOGTBL1+4,r3
add d12,d3,d3      add d12,d0,d0     move #SYNST+8,r10
move (r1+r5),d2    move (r3+r6),d1   lsr d3          and d5,d1      ;d2=a2*odd, d1=a3*odd
tfr d10,d6         and d6,d2         tfr d10,d5      and d5,d1      ;mask to 0 if prev term=0
eor d3,d2          eor d0,d1         ;d2&d1=odd+even terms
min d2,d6          move d2,r5        min d1,d5        move d1,r6      ;prepare to log result
neg d6             neg d5            ;masks are 0 or $ffff
move (r2+r5),d0    move (r2+r6),d2   zxt.w d6,d5      zxt.w d5,d6     ;log of syndromes
move d0,r4         move d2,r5        tfr d5,d1        tfr d6,d3      ;r4=log(S1),r5=log(S2)
asl d0,d4          asl d2,d1         move.2w d0:d1,(r10)+
tfra r4,n0         tfra r5,n1        ;store S1,0mask
add #ALOGTBL,d4    add #ALOGTBL,d1   move.2w d2:d3,(r10)
; n0=log(S1), n1=log(S2)
; move #SYNST,r10
; store S2,0mask
move d1,r3         move d4,r1        ;r2->SYNST, r1=log(S1)+tbl offset

```

After calculating S_1 and S_2 , we leave them in registers $r4$ and $r5$. We also transfer them to registers $n0$ and $n1$ to use as address offsets in the next section of code. Later, for the error-location polynomial calculation, these values will be required to determine the polynomial coefficients.

2.2.2 Finding the Error Location Polynomial (ELP)

Once the syndromes are found, the next step is to determine the coefficients for the error location polynomial. For our two-error example, there are several ways to determine the coefficients, including using the Berlekamp-Massey algorithm, the Modified Sugiyama (Euclid's) algorithm, or the matrix techniques of the Peterson-Gorenstein-Zierler (PGZ) decoder [4]. For larger numbers of errors, the first

two techniques require less computation, but for the two-error case, it is actually easier to solve the equations of the PGZ decode technique. Correcting two errors and assuming the syndromes are S_0 , S_1 , S_2 , and S_3 , the equations to solve (for Λ_1 and Λ_2) are $S_2 + \Lambda_1 S_1 + \Lambda_2 S_0 = 0$ and $S_3 + \Lambda_1 S_2 + \Lambda_2 S_1 = 0$.

Solving, we get the following two results:

$$\Lambda_1 = \frac{S_2 S_1 - S_0 S_3}{S_1^2 - S_0 S_2}$$

$$\Lambda_2 = \frac{S_3 S_1 - S_2^2}{S_1^2 - S_0 S_2}$$

There are six intermediate products to find for these terms. Since there are two address processors to use, we determine the products in pairs. Noting that S_1 and S_2 appear in all but one of the six products, we choose to start with S_1 and S_2 . These two syndromes and their mask values are calculated last in the previous section, so they are available for immediate use here. **Example 2-5** shows the error-location polynomial code.

First, the alogs of the squares of S_1 and S_2 are determined using the table look-up. We calculate the two products, $S_1 S_2$ and $S_0 S_2$, after loading S_0 into the AGU and its mask value into the DALU. Each product is calculated using four instructions:

- An **adda** instruction to compute the log addition (equivalent to a regular multiply)
- A **move** instruction to take the alog of the added result using the table look-up
- An **and** instruction to combine the mask values for the two terms that are being added
- Another **and** instruction to mask the final alog result

The first two instructions require modulo arithmetic, so they execute in the AGU. The **and** instructions execute in the DALU concurrently with the first two instructions.

The DALU subtracts $S_0 S_2$ from S_1^2 using an **eor** instruction to get the complete denominator for both Λ s. Loading S_3 into the AGU and its mask value into the DALU, we find the two remaining products, $S_1 S_3$ and $S_0 S_3$, using the same procedure as for the previous two products. The sequence of four instructions described above repeats four times to calculate all four products, $S_1 S_2$, $S_0 S_2$, $S_1 S_3$, and $S_0 S_3$. The DALU calculates the numerators for the Λ s using two more **eor** instructions. Meanwhile, the AGU gets the log of the denominator and the log of the numerators using **move** instructions for the table look-up. Finally, both Λ s are calculated using **sub** instructions to divide the numerators by the denominator.

This process demonstrates one of the challenging aspects of this kind of multiprocessor programming. Determining the order of operations is more often a task of determining a way to keep the processors loaded with the needed information, rather than figuring out how to do the arithmetic. The order we chose for the preceding calculation was determined by data movement considerations.

Before we combine the numerators and the denominator, it is useful to do a few tests. If the denominator is zero, we know that there are not two errors. Since we can correct only error patterns of two or less, this condition should assume there is one or zero errors and proceed to that processing. In the case of one error, $\Lambda_2=0$, $\Lambda_1=S_1/S_0$ is absorbed into the error evaluator equation. The code jumps to NOTTWO and begins the error evaluation processing assuming one error.

Reed-Solomon Program

At this point, it is important to test for decoding failures, which occur when there are more than two errors in the received code word. These failures manifest themselves in many different ways. Thus, the decoder tests for failures frequently. If a failure is detected, the code jumps to FAIL, where the program ends and no errors are corrected. The value in register d0 at FAIL holds a number that describes the decoder failure mode. If either of the Λ numerators are zero, the corresponding Λ is zero and a decoding failure occurred:

- If the numerator for Λ_1 is zero, the value in register d0 is one, indicating that the degree of the error location polynomial is less than two.
- If the numerator for Λ_2 is zero, the value in register d0 is two, indicating that the two errors are in the same place.

Example 2-5. ELP Calculating Code

move #000,b4	move #000,b5				
addlla n1,r3	move (r1+n0),d0				;r4 and r5 have mod base at \$0
move.l #00999080,MCTL					;r3=address alog(S2^2),d0=alog(S1^2)
move (r3),d1	move #ALOGTBL,r1				;make r3,r4,r5 modulo m1, r1 mod m0
move #ff,m1					;r1->ALOGTBL, d1=alog(S2^2)
adda r4,r5	tfra r5,r3				;m1 mod 256
and d5,d0	and d6,d1				;r5=log(S1*S2), r3=log(S2)
		tfr d6,d2			move (r10)+,r5 move (r1+r5),d4
					;d0=S1^2 masked, d1=S2^2 masked, d2=
					;S2 0mask, r5=log(S0),d4=alog(S1*S2)
and d5,d2	adda #8,r8,r8				;d2=S1*S2 0mask, r8->adjust RECDATA
and d2,d4	moveu.w (r10)+,d2	adda r5,r3			;d4=S1*S2 masked, d2=S0 0mask
					;r3=log(S0*S2)
move (r1+r3),d3		move (r10)+,r3			;r3=log(S3), d3=alog(S2*S0)
and d2,d6					;d6=S2*S0 0mask
and d6,d3	moveu.w (r10),d6	adda r3,r4			;d3=S0*S2 masked, d6=S3 0mask
					;r4=log(S3*S1)
eor d3,d0	and d6,d5	and d6,d2			adda r3,r5 move (r1+r4),d6
					;d0=S1^2-S0*S2,d5=S3*S1 0mask
					;d2=S3*S0 0mask,d6=alog(S3*S1)
					;r5=log(S3*S0)
and d5,d6	move (r1+r5),d5	move d0,r5			;d6=S1*S3, d5=alog(S0*S3), r5=D
and d2,d5	eor d6,d1				;d5=alog(S0*S3), d1=L2N
eor d5,d4	move d1,r4				;d4=L1N, r4=L2N
move d4,r3	move (r2+r5),r5				;r3=L1N, r5=log(D)
tsteg d0					;see if D=0
tfr d4,d2	move (r2+r4),r4	move (r2+r3),r3			;d2=L1N, r4=log(L2N), r3=log(L1N)
jt NOTTWO					;D=0, not 2 errors, go to 1 error
tsteg d1	move #1,d0	move #FACTBL,r6			;D!=0,L2N=0->decoding failure
jt FAIL					;D!=0, deg Lambda<2 is fail #1
					;d0=error flag
tsteg d4	inc d0	suba r5,r4			suba r5,r3
					;tst L1N=0, r4=log(L2),r3=log(L1)
jt FAIL	suba r3,r4				;D!=0, L1N=0 -> common root. Fail #2

2.2.3 Factoring the ELP

The next step in the process is to factor the error-location polynomial. We assume that there are two errors because the one and zero error (and failure) cases are straightforward. For the two-error case, we can do a Chien search that systematically evaluates the error location polynomial looking for zeros. This general algorithm can be used for Reed-Solomon codes designed to find any number of errors. For our two-error correcting example, we take a shortcut that enables us do a table look-up without an excessively large table size. We assume that the field size is 256 or less. In general, doing a table look-up on the error location polynomial $\Lambda_2x^2 + \Lambda_1x + 1$ requires that we have entries for all values of Λ_1 and Λ_2 , which for a field size of 256 means a table size of $256^2 = 65,536!$ We reduce this table to 256 entries by dividing the task into two steps. The first is to make the substitution:

$$\frac{y}{\Lambda_1} = x$$

The modified error location polynomial becomes:

$$\frac{\Lambda_2}{\Lambda_1^2} y^2 + y + 1$$

We index the table from the single value:

$$\frac{\Lambda_2}{\Lambda_1^2}$$

Assuming we are working with 16-bit words, the table is a list of precomputed logarithms of the two roots of all possible polynomials, indexed by the value Λ_2/Λ_1^2 . For field sizes of 256 or less, the log of both roots are stored in the same word, so a table size of 256 words is sufficient. For a nonextended Reed-Solomon code, there are at most $255*254$ possible root values, and taking the normalization by Λ_1 into account, there are 127 possible index values that correspond to valid root combinations.

Our table lists all combinations and places the value 0 in the 129 locations that are nonvalid values. A table look-up that returns 0 is interpreted as a decoding failure (error pattern with more than two errors). The code that factors the error location polynomial is shown in **Example 2-6**. The first three lines compute the index and perform the table look-up of the roots. The roots are tested to ensure that they do not indicate decoding failures. If both roots are zero, the error-location polynomial failed to factor into two linear terms and a decoding failure occurred. The code jumps to FAIL, where the program ends, no errors are corrected and the value in register d0 is five.

Although this table look-up method makes the factoring problem small in code size and execution time, there is still an opportunity to use the parallel processing units. The data is partitioned into its separate roots in one step by assigning one ALU to mask off the upper byte and another ALU to shift the word right by one byte into another register. Then we adjust the root values by adding the log of Λ_1 to both products to undo our normalization. Again, this is done in parallel, using multiple processors.

For the shortened code, the results must be compared to the largest codeword symbol index to check for a pattern with more than two errors masquerading as a two-error pattern with one or both error positions larger than the transmitted codeword. Both error positions are tested, and if either is greater than 63 (the length of the codeword), the code jumps to FAIL, where the program ends and no errors are corrected. The value placed in register d0 is six, indicating that there are two error positions with at least one of them in a position greater than the codeword length.

Example 2-6. ELP Factoring Code

suba r3,r4									
suba r3,r4									;r4=log(L2/L1)
move (r6+r4),d1									;r4=log(L2/L1^2)
tsteg dl									;d1=factors
jt FAIL									;check factors !=0, else fail
move #SYNST,r10	asrr #8,d1			move #5,d0					;failure to factor. Fail #5
move dl,r4	move d2,r5								;split the factors
move #CWLENGTH-1,r6									;move to address for log arith
inc d0	adda r3,r4			adda r3,r5					;r6=code length
cmpgta r6,r4	move r4,d4								;error positions in r4 & r5
jt FAIL									;fail #6. Factor to pos >63
cmpgta r6,r5	move r5,d5								
jt FAIL									;same fail, error positions in d4&d5

2.2.4 Evaluating the Errors

Once the error positions are determined, they are passed to the error-evaluation section to determine the value of the error in each position and to correct for the errors. First, the error-evaluator coefficients, Ω_x , are calculated. The error magnitudes are calculated using the error locations and the error-evaluator coefficients. The error magnitudes are used to correct the errors in the codeword at the location defined by the error positions. The error magnitudes contain a one where a bit error is located in the codeword and zeros elsewhere. Thus, when the error magnitudes are EXCLUSIVE-ORed with the codeword, the incorrect bits are changed and the correct bits are unchanged.

2.2.4.1 Two-Error Case

For the two-error case, the error-evaluator coefficients are as follows:

$$\Omega_1 = \frac{S_1}{\Lambda_1} + S_0$$

$$\Omega_0 = \frac{S_0}{\Lambda_1}$$

The code that corrects the errors for the two-error case is shown in **Example 2-7**. First, the error-evaluator terms are calculated. The value of the Λ_1 is left in the AGU from the previous calculations. We load the S_0 and S_1 values into the DALU and subtract Λ_1 from both syndromes using **suba** instructions. We take the alog of the results, placing the alogs into the DALU. These computations are completed simultaneously, using the two address processors. Meanwhile, we obtain the alog of S_0 , which is added to the S_1/Λ_1 value to get Ω_1 . All alog values are masked in the DALU using **and** instructions.

The error magnitudes are calculated by the following equation, using the roots of the error-location polynomial for the error positions:

$$\frac{\Omega_1}{\text{errorposition}} + \Omega_0$$

First, we compute the log of Ω_1 and a mask value for Ω_1 . The mask computation requires a **min** instruction with register d0 as the source. The **min** instruction is very restrictive with its register combinations, and in this case, the destination for the **min** instruction must be register d4. Thus, the error location in register d4 is temporarily moved to register d6 while the Ω_1 mask is calculated. We subtract both error positions from Ω_1 using **suba** instructions and take the alog using **move** instructions for the table look-up. We add Ω_0 using **eor** instructions to get the final error magnitudes. Both error magnitudes are calculated simultaneously using the two address processors.

Once the error magnitudes are calculated, the symbols at the locations defined by the two error positions are retrieved and EXCLUSIVE-ORed with the error magnitudes. The corrected symbol is moved back to the codeword to give the correct transmitted data words, and the decoding is complete.

Example 2-7. Two Error Evaluation Code

move.2w (r10),d0:d1	adda #8,r10,r10		;d0=S0, d1=S0 0mask
move.2w (r10),d2:d3	move d0,r4		;d2=S1, d3=S1 0mask, r4=S0
move d2,r5			;r5=S1
move (r1+r4),d0			;d0=log(S0)
and d1,d0	suba r3,r5	suba r3,r4	;d0=S0 masked, r5=S1/L1, r4=S0/L1
move (r1+r4),d2	move (r1+r5),d6		;d2=log(S0/L1), d6=log(S1/L1)
and d1,d2	and d3,d6	move d4,r3	move d5,r6
			;0mask alogs, d2=Omega0
eor d6,d0	tfr d4,d6	clr d4	;d0=Omegal
inc d4	move d0,r4		;r4=Omegal
min d0,d4			
move (r2+r4),r4	move (r2+r4),r5		;r4=r5=log(Omegal)
neg d4			;d4=0mask
suba r3,r4	suba r6,r5		;r4=Omegal/err1, r5=Omegal/err2
xzt.w d4,d1	tfr d6,d4	move (r1+r4),d0	move (r1+r5),d3
			;d1=01 0mask, d4=err1, d0=log
			; (01/err1), d3=log(01/err2)
			;mask d0 and d3
			;d0=Omegal/err1+Omega0
			;d3=Omegal/err2+Omega0
			;prep to log Omega sums
			;r2 & r6 = error positions
			;r3 & r4 = log(Omega sums)
			;get errors to correct
			;d0 & d1 = error values
			;correct errors
			;write corrections
			;well? There aren't any now!

2.2.4.2 One-Error Case

The code that corrects the errors for the one-error case is shown in **Example 2-8**. This code is located at NOTTWO, and we jump to it from the error location polynomial calculation. For the one and zero error cases, the Λ denominator and both numerators are zero. We know that the denominator is zero at this point because we jump to NOTTWO based on a zero value for the denominator. Thus, the code first tests both of the Λ numerators (already calculated during the error location polynomial phase and stored in registers d1 and d4) to ensure that they are both equal to zero. If both numerators are not equal to zero, the code jumps to FAIL and register d0 is three. Now we must test for any other decoding failures. **Table 2-1** shows the values of the syndromes (0 for zero and x for nonzero) and the numerator values based on these syndromes. The table also shows which cases are for zero and one errors and which cases are failures. Some of the syndrome combinations are missing from the table because these combinations are not possible at this point. Only combinations in which the denominator is zero or $S_1^2=S_0S_2$ are possible.

We use the fact that the following equation applies for one error to get zero values for the numerators in the one error case row:

$$\frac{S_3}{S_2} = \frac{S_2}{S_1} = \frac{S_1}{S_0} = \Lambda_1$$

The rows that are gray were already eliminated by the previous test (at least one of the numerators is not zero). Thus, we need to test only for the two failure rows. The code tests whether both or neither S_0 and S_3 are equal to zero (using an **eor** instruction) and jumps to FAIL if the result is false, after placing three in register d0. The code also tests for the zero error case (when S_2 equals zero) and ends the processing if there are zero errors to correct.

Table 2-1. Syndrome and Numerator Values

S ₃	S ₂	S ₁	S ₀	Λ ₁ numerator	Λ ₂ numerator	Comment
0	0	0	0	0	0	Zero error case
0	0	0	x	0	0	Failure
0	x	0	0	0	x	
0	x	x	x	x	x	
x	0	0	0	0	0	Failure
x	0	0	x	x	0	
x	x	0	0	0	x	
x	x	x	x	0	0	One error case

Meanwhile, we calculate the error-evaluator and error location. For the one error case, the error-evaluator is a constant, Ω₀=S₀, and the error location is as follows:

$$\Lambda_1 = \frac{S_1}{S_0}$$

We load the S₀ and S₁ values into the AGU and subtract S₁ from S₀ to get Λ₁. For the shortened code, the error location must be compared to the largest codeword symbol index to check for a decoding failure. The error location is tested, and if it is greater than 63 (the length of the codeword), the code jumps to FAIL where the program ends and no errors are corrected. The value placed in register d0 is four, indicating that there is one error, but it is in a position greater than the codeword length.

We calculate the error magnitude using the following:

$$\frac{\Omega_0}{error\ position} = \frac{S_0}{\Lambda_1}$$

We subtract Λ₁ from S₀ and take the alog of the result, using a **move** instruction to get the final error magnitude. Once the error magnitude is calculated, the symbol at the location defined by the error position is retrieved and EXCLUSIVE-ORed with the error magnitude. The corrected symbol is moved back to the codeword to give the correct transmitted data word, and the decoding is complete.

Example 2-8. One Error Evaluation Code

```

or d1,d4      move #SYNST,r10      ;to check if both are 0
tstq d4      move #3,d0          ;if D=0 and N1 or N2 !=0, Fail #3
jf FAIL      move.2l (r10)+,d0:d1 ;d0=S0+mask, d1=S3+mask
eor d0,d1    zxt.w d0,d0         ;d1=S0+S3, r5=S1
asrw d1,d1   move.w (r10),d2     ;d2=S2, r4=S0
tstq d1
move #3,d0   suba r4,r5          ;r5=S1/S0=L1
jf FAIL      ;fail #3
tstq d2     move r5,d1          ;tst for 0 error case, d1=r5=L1
jt NOERRORS
inc d0      cmpgt.w #CWLENGTH-1,d1 ;if d1<=63, all is well
jt FAIL     suba r5,r4          ;fail #4
move (r1+r4),d0      move (r8+r5),d1 ;d0=alog(S0/L1), get error to fix
eor d0,d1    ;correct error
move d1,(r8+r5) ;write correction
jmp NOERRORS ;well? There aren't any now!

```

3 Conclusion

The two-error Reed-Solomon decoder described in this application note requires a small number of MIPS when implemented on the StarCore processor. The cycle count is data dependent because there is a memory contention issue. The contention issue arises because the table look-ups for finite field logarithms and antilogarithms are (for this example) in GF(256). Since we use words for each entry in the table, each table takes 512 bytes. Typically, this is not a problem. While the syndrome determination loop performs a parallel access to the tables in the same instruction, saving two cycles when compared to code that does not do the access in parallel, it does so at the cost of a memory access contention stall for some accesses. Modeling the table look-up addresses as uniformly distributed random variables, this contention stall occurs with a probability of 1/16 for each relevant access attempt. Consequently, the cycle count for the code presented here is the sum of a constant and a binomially distributed random variable. Although the best-case cycle count is 819 and the worst-case count is 1115, the average cycle count is heavily biased toward the lower number; 99.9 percent of the time, the cycle count is less than 854.

The program is 1178 bytes, the log tables are 1024 bytes, and the factor table is 512 bytes. Therefore, the total amount of memory used for the program, tables, and received codeword is 2.714 Kbytes.


Memory contention stalls can be avoided simply by creating second copies of the logarithm and antilogarithm tables with adequately spaced address differences. Creating these copies reduces the cycle count to 819 and raises the memory use to 3.738 Kbytes.

The study of error correcting codes promotes great flexibility in the type of coding used, the parameters chosen to implement in a given code, and mathematical strategies to improve coding performance. For Reed-Solomon codes, this work can be expanded to explore the performance effects of correcting more than two errors. While this work would create a more powerful decoder, it would also greatly increase the complexity and perhaps the cycle count of the decoder. Additionally, this Reed-Solomon decoder could be compared with convolutional codes of a similar capability to determine performance benefits and drawbacks for each implementation in a given system.

4 References

- [1] Lin, Shu and Costello, *Error Control Coding: Fundamentals and Applications*, Prentice-Hall: Englewood Cliffs, NJ, 1983.
- [2] Herstein, I.N., *Topics in Algebra*, Xerox College Publishing: Lexington, MA, 1975.
- [3] Sklar, B., *Digital Communications Fundamentals and Applications*, Prentice-Hall: Englewood Cliffs, NJ, 1988.
- [4] Blahut, R.E., *Theory and Practice of Error Control Codes*, Addison Wesley: Reading, MA, 1984.

StareCore is a trademark of Motorola, Inc.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

How to reach us:

USA/Europe/Locations Not Listed:

Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1 (800) 441-2447
1 (303) 675-2140

Motorola Fax Back System (Mfax™):

RMFAX0@email.sps.mot.com
TOUCHTONE (602) 244-6609
USA and Canada ONLY:
1-800-774-1848
<http://sps.motorola.com/mfax/>

Asia/Pacific:

Motorola Semiconductors H.K. Ltd.
8B Tai Ping Industrial Park
51 Ting Kok Road
Tai Po, N.T., Hong Kong
852-26629298

Technical Resource Center:

1 (800) 521-6274

DSP Helpline

dsphelp@dsp.sps.mot.com

Japan:

Nippon Motorola Ltd.
SPD, Strategic Planning Office141
4-32-1, Nishi-Gotanda
3-14-2 Tatsumi Koto-Ku
Shinagawa-ku, Tokyo, Japan
81-3-5487-8488

Customer Focus Center:

1-800-521-6274

Internet:

<http://www.mot.com/SPS/DSP/>

