Polynomial Multipliers and Dividers, Shift Register Generators and Scramblers

Phil Lucht
Rimrock Digital Technology, Salt Lake City, Utah 84103
last update: June 12, 2013
rimrock@xmission.com

Maple code is available upon request. Comments and errata are welcome.
The material in this document is copyrighted by the author.

The graphics look ratty in Windows Adobe PDF viewers when not scaled up, but look just fine in this excellent freeware viewer: http://www.tracker-software.com/pdf-xchange-products-comparison-chart .

The table of contents has live links.

Overview and Summary **3**

Chapter 1: Polynomial Processors **4**

1.1 Review of the Z Transform 4
1.2 The Scrambler-Style Polynomial Divider 5
Symbols 5
Analysis of the Scrambler-Style Divider in the z-domain 7
Interpretation of Polynomial Division 9
Example 1: $(z^2+1) / (z+1)$ 10
Example 2: $(1 + z^{-1}) / (z^2 + z + 1)$ 11
Example 3: $(1 + z^{-1}) / (z^2 + z)$ 12
Example 4 : SMPTE Scrambler 13
1.3 More Interpretation of Polynomial Division 13
Finite Division 14
The Hidden Remainder 15
Impulse Response 16
1.4 The Standard-Form Polynomial Divider 16
Analysis of the Standard-Form Divider in the z-domain 17
1.5 How Polynomial Dividers Actually Work 18
Operation of the Standard-Form Polynomial Divider 18
Operation of the Scrambler-Style Polynomial Divider 19
1.6. The Descrambler-Style Polynomial Multiplier 21
Analysis of the Descrambler-Style Multiplier in the z-domain 21
Interpretation of Polynomial Multiplication 22
Impulse Response 23
Example : SMPTE Descrambler 23
1.7 The Standard-Form Polynomial Multiplier 24
Analysis of the Standard-Form Multiplier in the z-domain 25
1.8 How Polynomial Multipliers Actually Work 26
Operation of the Decrambler-Style Polynomial Multiplier 26
Operation of the Standard-Form Polynomial Multiplier 28
1.9 Polynomial Processors in the Time Domain 29

The Descrambler-Style Multiplier in the Time Domain 29
The Scrambler-Style Divider in the Time Domain 30
Convolution Theorem Approach 31
1.10 Simultaneous Polynomial Multiply and Divide 33
1.12 Application: Cyclic Redundancy Check 35
Appendix 1.1. Processing polynomials vs. processing integers. 37
Chapter 2: Shift Register Generators 42
2.1. The Maximum Period of a Shift Register Generator 42
2.2 The State Vector Sequence of a Shift Register Generator 43
2.3 The Output Sequence of a Shift Register Generator 51
2.4. Characteristic Sequences 58
2.5 Autocorrelation, Probability, and Statistics for a sequence in GF(2) 63
2.6 The Spectral Power Density of a Shift Register Generator 70
Appendix 2.1: Proof of Fact 4 of Section 2.3 75
Appendix 2.2: A List of Primitive Polynomials over GF(2) 78
Chapter 3: The Matrix Approach 79
3.1 Review of earlier Chapters 79
3.2 Matrix Solution of Figure 1.5. 80
3.3 Matrix Solution of Figure 1. 82
3.4 Shift Register Generators Revisited 83
3.5 The Output of a Scrambler 84
3.6 The Spectral Density of a Scrambler Output 88
3.7 The NRZI Mini-Scrambler 90
3.8. Matrix analysis of the polynomial multiplier (de-scrambler) of Figure 7 94
3.9 Matrix analysis of the standard polynomial multiplier of Figure 8 97
3.10. Proof that a Descrambler really descrambles the output of a Scrambler. 98
3.11 The Kill Sequence Problem and Strings of Zeros 104
Appendix 3.1: Proof of Eq. (3.10.15) 109
References 110

Overview and Summary

This section is not yet written.

Chapter 1: Polynomial Processors

1.1 Review of the Z Transform

The Z Transform is discussed in Chapter 3 of Ref [32], see e.g. (24.37). Let f_n describe the values of a digital signal at times $t = n\Delta t$ where Δt is the clock spacing between the samples. The Z Transform of the sequence f_n is given by (≡ means "is defined as")

$$F(z) \equiv \sum_{n=-\infty}^{\infty} f_n z^{-n} \ . \tag{1.1.1}$$

One thinks of F(z) as being a polynomial in the variable z (positive and negative powers allowed), and the coefficients of this polynomial are in fact the sequence values f_n. As things develop below, one realizes that this variable z is basically just an inert carrier that allows us to talk about F(z) as a polynomial. In Z Transform theory, z is related to angular frequency ω by $z = e^{i\omega\Delta t}$. Thus, z is a dimensionless complex number which, for real frequency, lies on the unit circle in the complex z-plane. Since z is dimensionless, the dimensions of F(z) are the same as those of samples f_n . We speak of (1.1.1) as transforming a signal from the time-domain to the z-domain.

Consider now the sequence f_{n-1}. This is simply f_n *delayed* 1 time unit (shifted 1 time unit Δt into the future). For example, if sequence f_n has a strong peak at n=0, then f_{n-1} peaks at n=1, so the peak moved one clock later in time. We denote the Z transform of f_{n-1} by $F_{-1}(z)$ where the -1 suggests a delay of 1 unit.

How is $F_{-1}(z)$ related to F(z)? Let n' = n-1 so that

$$F_{-1}(z) = \sum_{n=-\infty}^{\infty} f_{n-1} z^{-n} = \sum_{n'=-\infty}^{\infty} f_{n'} z^{-(n'+1)} = z^{-1} \sum_{n=-\infty}^{\infty} f_n z^{-n} = z^{-1} F(z) \ .$$

Similarly, we write the Z transform for the sequence delayed by m clocks, this time with n' = n-m,

$$F_{-m}(z) = \sum_{n=-\infty}^{\infty} f_{n-m} z^{-n} = \sum_{n'=-\infty}^{\infty} f_{n'} z^{-(n'+m)} = z^{-m} \sum_{n=-\infty}^{\infty} f_n z^{-n} = z^{-m} F(z) \ . \tag{1.1.2}$$

Here f_{n-m} is the sequence f_n which has been delayed m clocks. Its transform is z^{-m} times the transform of f_n. Thus, in the z-domain, we can associate a factor of z^{-1} for each flip-flop (register) delay of a signal. This is such an important point, and there are going to be so many powers of z floating around, we will repeat:

Fact: One can interpret z^{-m} as a delay of a digital signal by m clocks.

Example: If we run f_n into a chain of m registers, then f_{n-m} is what emerges from the last register.

Here is the Z transform of sequence f_{n+m} which is *advanced* m clocks relative to f_n:

$$F_m(z) = \sum_{n=0}^{\infty} f_{n+m}\, z^{-n} = z^m\, F(z) = z^m\, F_0(z)\,. \qquad (1.1.3)$$

We can summarize these results as follows:

time domain		z-domain
f_n	↔	$F(z)$
f_{n-1}	↔	$z^{-1}\, F(z)$
f_{n-m}	↔	$z^{-m}\, F(z)$
f_{n+1}	↔	$z\, F(z)$
f_{n+m}	↔	$z^m\, F(z)$

(1.1.4)

1.2 The Scrambler-Style Polynomial Divider

Consider the following circuit:

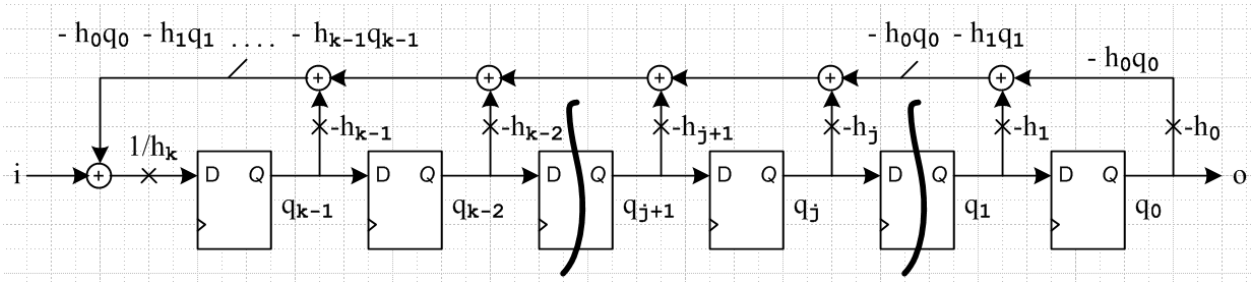


Figure 1.1: A scrambler-style polynomial divider circuit.

For want of a better name, we refer to this circuit having outboard adders along the single feedback path as a "scrambler style" divider. Later we shall introduce another form for the divider we call the "standard form". It has many feedback paths with an adder between each register. In fact, either circuit could be used as a "scrambler", an example of which will be given below.

Before analyzing the above circuit, and others similar to it, we wish to make a clear statement of the degree of generality these circuits have. The reader should not jump to the conclusion that these circuits process only bits. Rather, they process *symbols*. Basically, we are dealing with a topology here that is not dependent on the specific nature of the symbols.

Symbols

We shall call the basic signals in the above circuit *symbols*. All lines in the drawing carry such symbols. The adders add two such symbols using an operation + ; the X's indicate places where a symbol being carried on a line is multiplied by some constant symbol $-h_j$ using an operation • ; each register holds a symbol; the input and output data streams consist of symbols.

A general way to interpret a symbol is to regard it as an m-tuple of numbers which are elements of the ring Mod(p). (If you are unfamiliar with rings and fields, see Chap 1 of Ref [31] for definitions.) In this case, we can think of each register as consisting of m "p-ary flip-flops" which are special in that they can store not just 2 but p different integers: 0,1,2.....p-1. One can regard such an m-tuple symbol as an element of a ring we will just call Ring(p^m, •, +). The definition of the circuit would then be complete once we specified how the operations • and + act on the p^m ring elements.

signal sample = a symbol = an m-tuple
= {n_1, n_2..... n_p} = element of Ring(p^m, •, +) where each n_i = element of ring Mod(p)

Notice that there are p * p * p ... * p = p^m possible m-tuples, which is the order of Ring(p^m, •, +) .

Definition: A p-ary flip-flop stores a *pit*. If p=2, a pit is a bit. That is, a binary pit is a bit. Current computer circuits tend to deal with bits and not p>2 pits; the future may be different. A p=3 pit could be represented by three voltage levels, for example.

Example 1: One specification for the operations • and + would be to say Ring(p^m, •, +) = Mod(p^m). More explicitly, we might consider p=2, m=8 and write Ring(2^8, •, +) = Mod(2^8) = Mod(256) = Z_{256}. In this case, the symbols in the above Figure are 8-bit binary numbers (bytes), and we might then refer to the circuit as a "digital filter". As we shall soon see, the transfer function for this digital filter is 1/H(z) , where H(z) is a polynomial whose coefficients are the h_j shown in Fig 1.1.

Notice that in such a circuit, there is a "mixing" of the individual bit lines at each place where + or • is performed. For example, an adder is not just 8 independent 1-bit adders. The adders have "carry" between the bit positions. Similarly for the • operations. This is the way Mod(p^m) works.

There is a subtle restriction on the circuit of this example. Notice that the circuit involves multiplication by $1/h_k = h_k^{-1}$. In general, not all elements of Mod(p^m) have inverses. For example, in Mod(4), there is no element 2^{-1} since 2•0 = 0, 2•1 = 2, 2•2 = 0, 2•3 = 2. Thus, we would have to make sure that we choose some h_k which has an inverse. A good candidate is h_k = 1.

If we are interested in having h_k be an *arbitrary* one of our symbols, we need to restrict our interest to cases where Ring(p^m, •, +) is a *field*. In a field, every non-zero element always has a • inverse. As shown in Ref [31], Ring(p^m, •, +) will be a field if and only if p is a prime number. In this case, we have a special notation : Ring(p^m, •, +) = GF(p^m), where GF stands for Galois Field. This leads to:

Example 2: Let Ring(p^m, •, +) = GF(p^m) where p = prime. In this case, our symbols are m-tuples of numbers which are elements of Mod(p). For p=prime, Mod(p) is itself a field which we call GF(p) or Z_p. It is just the modulo-p integer field we are all familiar with.

So in this example, our symbols are elements of GF(p^m), and each number in the symbol m-tuple is an element of GF(p). The operations • and + for GF(p^m) are specific to GF(p^m) and are studied in Ref [32]. In the m-tuple basis we are using here for our symbols, the rule for addition + is very simple: it is performed independently on each element of the m-tuple using the + table for GF(p). In this case, there is no "carry" between the "pit positions" of the adders in Fig 1.1.

With the symbols of Example 2, Fig 1.1 is still a "digital filter", and, as we shall show below, that filter still has the transfer function 1/H(z). In our two examples, the topology of the circuit is the same, only the symbols are different.

Observation: GF(p^m) ≠ Mod(p^m). Both these rings have p^m elements, but their + and • operations are completely different. Moreover, GF(p^m) is a field, whereas Mod(p^m) is not a field except for m=1.

Example 3: This example is just Example 2 with m=1. In this case, a symbol is a 1-tuple -- just a single number. The symbol is an element of GF(p). If p=2, the symbols of GF(2) are just bits. Examples of this kind of circuit are binary polynomial dividers and scramblers.

Analysis of the Scrambler-Style Divider in the z-domain

Now we resume our consideration of Figure 1.1 which we replicate here:

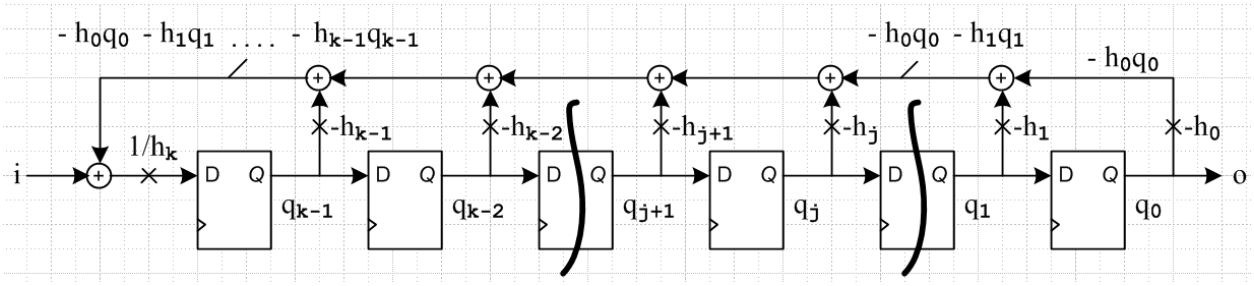


Figure 1.1: A scrambler-style polynomial divider circuit.

Fig 1.1 shows a set of k registers numbered q_0 to q_{k-1}. We denote by $q_j(n)$ the output of register j at time n, meaning t = nΔt where Δt is the clock period. The clock lines to the registers are not shown. Similarly, we denote by $d_j(n)$ the value of the symbol pressing up against the D input of register j at time n. Our intent is that the registers are made of p-ary "D flip-flops" which have the property that $q_j(n+1) = d_j(n)$. This says that the input of a register becomes the output of the register one clock later. The registers might be "clocked" on the positive edges of a square-wave clock signal of period Δt.

The input symbol sequence is i(n) on the left, and the output sequence is o(n) on the right.

The equations that go with the above figure are quite straightforward. Notice that all the feedback accumulates and ends up at the input of the leftmost register. Inspecting Fig 1.1 we find that,

$$q_{k-1}(n+1) = d_{k-1}(n) = (1/h_k)\left[\, i(n) - \sum_{j=0}^{k-1} h_j\, q_j(n) \,\right] \quad . \tag{1.2.1}$$

We now have a small notational conflict. In the Z transform discussion above, the sequence index n was treated as a subscript, for example, f_n. *Here*, we wish to reserve the subscript location to label a particular register of the shift register. We have to put the n somewhere, so we put it as an argument (n). Hopefully, this should cause no confusion.

If we project the above equation into the z-plane, we get at once (see Z transform discussion above):

$$z\,Q_{k-1}(z) = D_{k-1}(z) = (1/h_k)\,[\;I(z) - \sum_{j=0}^{k-1} h_j\,Q_j(z)\;] \qquad (1.2.2)$$

The left side expression is an application of the 4th line of (1.1.4). Since this is the first of many Z transforms of equations from the time domain to the z domain, the reader should stare at (1.2.1) and (1.2.2) until a Zen level of comfort is achieved.

In Figure 1.1 the symbols just march left to right through the registers unaltered. Thus for example,

$$q_0(n+2) = q_1(n+1) = q_2(n) \qquad \text{and} \qquad q_0(n+j) = q_j(n)$$

According to the last line of (1.1.4) this last equation transforms into

$$Q_j(z) = z^j\,Q_0(z) \qquad Q_{k-1}(z) = z^{k-1}\,Q_0(z) \quad . \qquad (1.2.3)$$

where in the second equation we set j = k-1. We have chosen to use the rightmost register q_0 as our reference point. We can then write (1.2.2) as,

$$z^k\,Q_0(z) = (1/h_k)\,[\;I(z) - \sum_{j=0}^{k-1} h_j\,z^j\;Q_0(z)\;] \quad . \qquad (1.2.4)$$

It is an easy matter to solve this equation for $Q_0(z) = O(z)$, our output function, in terms of I(z), the input function. First, get the two $Q_0(z)$ terms on the left,

$$z^k\,Q_0(z) + (1/h_k)\,Q_0(z) \sum_{j=0}^{k-1} h_j\,z^j = (1/h_k)\,I(z)$$

$$Q_0(z)\,[\;h_k z^k + \sum_{j=0}^{k-1} h_j\,z^j\;] = I(z)$$

$$Q_0(z)\,[\;\sum_{j=0}^{k} h_j\,z^j\;] = I(z) \quad .$$

Here then is the final result for $O(z) = Q_0(z)$,

$$O(z) = I(z) / H(z) \qquad (1.2.5)$$

where

$$H(z) = h_k\,z^k + h_{k-1}\,z^{k-1} + \ldots + h_1\,z + h_0 = \sum_{j=0}^{k} h_j z^j \quad . \qquad (1.2.6)$$

We have therefore arrived at the undeniable conclusion that the circuit shown in Figure 1.1 does in fact divide the incoming polynomial I(z) by the polynomial H(z) to generate an output polynomial O(z). This circuit is a *polynomial divider*. This conclusion is independent of the nature of the symbols which are the coefficients of the polynomials.

Interpretation of Polynomial Division

Normally one thinks of a "polynomial" as something like $z^3 + 2z^2 - z + 2$. There is some highest power known as the degree of the polynomial (here 3), and there are no negative powers of z. We shall refer to such a polynomial as a "proper" polynomial. In dealing with z-domain functions like F(z) in (1.1.1), F(z) in general can have both positive and negative powers of z. If there are negative powers (these arise from samples f_n for n > 0), we shall refer to F(z) as an "improper" polynomial.

We see from (1.2.6) that H(z) is a proper polynomial of degree k. But what are the polynomials I(z) and O(z)?

To get a handle on I(z), we shall assume that i(n) = i_n is a finite sequence of non-zero samples such that the first non-zero sample is i_0 and the last is i_r, so the input is then a finite string of r+1 samples proceeded and followed by all-zero samples. The Z transform of this sample stream is then

$$I(z) = i_0 + i_1 z^{-1} + i_2 z^{-2} + ... + i_r z^{-r} .$$

Looking at Figure 1.1, and assuming the registers were pre-cleared, the first k output samples o_0 through o_{k-1} will be zero since we are just draining the shift register. The first non-zero output sample is then o_k. Thus, the Z transform of the output sample stream has this form

$$O(z) = o_k z^{-k} + o_{k+1} z^{-k-1} + o_{k+2} z^{-k-2} + ...$$

So, both I(z) and O(z) are improper polynomials.

The actual polynomial division done by Figure 1.1 in the z-domain is this:

$$[o_k z^{-k} + o_{k+1} z^{-k-1} + o_{k+2} z^{-k-2} + ...]$$

$$= [i_0 + i_1 z^{-1} + i_2 z^{-2} + ... + i_r z^{-r}] / [h_k z^k + h_{k-1} z^{k-1} + ... + h_1 z + h_0]$$

This is not something familiar to an algebra student who divides polynomials. We are used to getting a quotient and a remainder when we do such a division, but here we seem to get only a quotient. Moreover, this quotient seems to go on forever, even though the numerator and denominator each have just a finite number of terms.

In the world of proper polynomial division, we normally "stop dividing" when we have a remainder whose degree is less than that of the divisor. But if negative powers are allowed, then one need not stop at that particular point. In fact one could stop at any point one wanted, or one could just keep going forever.

Example 1: $(z^2+1) / (z+1)$

Consider the routine polynomial division $(z^2+1) / (z+1)$ with three different stopping points, where in the latter two cases we allow negative powers of z to appear in the quotient:

$$\begin{array}{r|l} & z - 1 \\ z+1 & z^2 + 1 \\ & \underline{z^2 + z} \\ & \quad -z+1 \\ & \quad \underline{-z -1} \\ & \qquad 2 \end{array}$$

$$\Rightarrow \quad \frac{z^2+1}{z+1} = (z - 1) + \frac{2}{z+1} \qquad \text{// the usual stopping point}$$

$$\begin{array}{r|l} & z - 1 + 2z^{-1} \\ z+1 & z^2 + 1 \\ & \underline{z^2 + z} \\ & \quad -z+1 \\ & \quad \underline{-z -1} \\ & \qquad 2 \\ & \quad \underline{2 + 2z^{-1}} \\ & \qquad -2z^{-1} \end{array}$$

$$\Rightarrow \quad \frac{z^2+1}{z+1} = (z - 1 + 2z^{-1}) - \frac{2z^{-1}}{z+1}$$

$$\begin{array}{r|l} & z - 1 + 2z^{-1} - 2z^{-2} \\ z+1 & z^2 + 1 \\ & \underline{z^2 + z} \\ & \quad -z+1 \\ & \quad \underline{-z -1} \\ & \qquad 2 \\ & \quad \underline{2 + 2z^{-1}} \\ & \qquad -2z^{-1} \\ & \quad \underline{-2z^{-1} - 2z^{-2}} \\ & \qquad\quad 2z^{-2} \end{array}$$

$$\Rightarrow \quad \frac{z^2+1}{z+1} = (z - 1 + 2z^{-1} - 2z^{-2}) + \frac{2z^{-2}}{z+1}$$

Maple can quickly verify these three results:

```
f := (z^2+1)/(z+1) - (z-1)-2/(z+1): simplify(%);
                                0
f := (z^2+1)/(z+1) - (z-1+2/z)+(2/z)/(z+1): simplify(%);
                                0
f := (z^2+1)/(z+1) - (z-1+2/z-2/z^2)-(2/z^2)/(z+1): simplify(%);
                                0
```

For each stopping point, we end up with a certain "remainder" as shown. If we continue the division process forever, we end up with a quotient that never stops and no remainder. If $z >> 1$, we could regard the division process shown above as developing a large-z series approximation for $(z^2+1) / (z+1)$.

Example 2: $(1 + z^{-1}) / (z^2 + z + 1)$

Consider our Figure 1.1 polynomial divider with k = 2 registers.

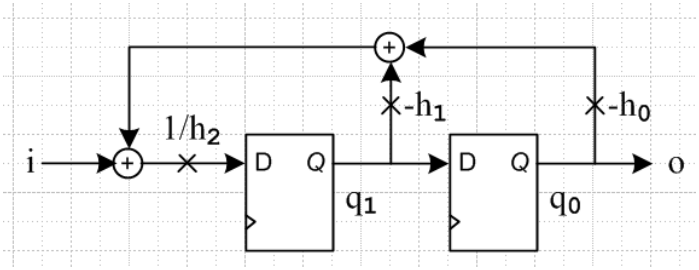


Figure 1.2: A scrambler-style divider with k = 2.

Assume $H(z) = z^2+z+1$ so $h_2= h_1= h_0 = 1$. Assume that the input sequence is just i_0, $i_1 = 1,1$ so that $I(z) = 1 + z^{-1}$. We can than examine the $O(z) = I(z)/H(z)$ result of our polynomial divider. We could use "long division" method as in the previous example, but instead we use a slightly different technique. We know that our first output sample will be o_2 so here is the situation:

$$[o_2z^{-2} + o_3z^{-3} + o_4z^{-4} + \ldots\ldots] = [1 + z^{-1}] / [z^2 + z + 1] .$$

Multiply through by z^2+z+1,

$$[z^2 + z + 1][o_2z^{-2} + o_3z^{-3} + o_4z^{-4} + \ldots\ldots] = [1 + z^{-1}] .$$

The left side may be written

$$(o_2 + o_3 z^{-1} + o_4z^{-2} + \ldots\ ) + (o_2z^{-1} + o_3 z^{-2} + o_4z^{-3} + \ldots\ ) + (o_2z^{-2} + o_3z^{-3} + o_4z^{-4} + \ldots\ldots)$$

$$= o_2 + (o_3+o_2)z^{-1} + (o_4 + o_3 + o_2)z^{-2} + (o_5 + o_4 + o_3) z^{-3} + \ldots\ldots$$

Setting this equal to $[1 + z^{-1}]$ and matching powers of z we learn that

$o_2 = 1$
$(o_3+o_2) = 1 \Rightarrow o_3 = 0$
$(o_4 + o_3 + o_2) = 0 \Rightarrow o_4 = -1$
$(o_5 + o_4 + o_3) = 0 \Rightarrow o_5 = +1$
$(o_6 + o_5 + o_4) = 0 \Rightarrow o_6 = 0$
$(o_7 + o_6 + o_5) = 0 \Rightarrow o_7 = -1$
etc.

Thus, the result of the polynomial division is this

$$O(z) = z^{-2} + 0z^{-3} - z^{-4} + z^{-5} + 0z^{-6} - z^{-7} + z^{-8} + ..$$

$$= (z^{-2} - z^{-4}) + (z^{-5} - z^{-7}) + (z^{-8} - z^{-10}) + (z^{-11} - z^{-13}) + ...$$

The output O(z) continues forever in this pattern.

Observation: If we take a snapshot of the division process of Figure 1.1 at any clock, we find that the k registers always hold the next k symbols of the quotient polynomial O(z). Just stare at Figure 1.1. Nothing can alter the contents of these registers as their contents shift to the right.

Example 3: $(1 + z^{-1}) / (z^2 + z)$

This is another k = 2 divider situation with $i_0, i_1 = 1,1$ but now $H(z) = z^2 + z$.

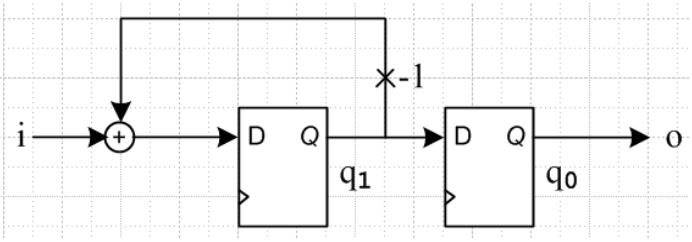


Fig 1.3. Another scrambler-style divider with k = 2

The result of this division is easily found to be,

$$[1 + z^{-1}] / [z^2 + z] = o_2z^{-2} + o_3z^{-3} + o_4z^{-4} + ... = z^{-2} .$$

We include this example just to show that it is *possible* for the output polynomial to truncate. Anticipating discussion to come later, if we think of q_1q_0 as the "state vector" of the above divider, we see that the state vector follows this pattern : .....00, 10, 01, 00 ..... . Here we are thinking of a symbol as being just a 1-tuple with value say in Mod(3), so that -1 = 2. In this example, the state vector is extinguished to 00 by the incoming symbol pattern! On the other hand, the unit impulse pattern caused by just $i_0 = 1$ gives this state vector sequence:

....00, 10, -11, -2-1, -3-2, -4-3.....

or

....00, 10, 21, 12, 01, 10.....

and the pattern continues cycling around forever, as appropriate for an Infinite Impulse Response filter. In Section 24 (f) of Ref [Fourier], it is shown that the transfer function of any IIR filter has non-zero poles in the z plane, and since here that transfer function is $1/[z^2+z]$, we see a non-zero pole located at z = -1. Also, any IIR filter has feedback, as in our current example, whereas FIR filters have no feedback and no non-zero poles.

Example 4 : SMPTE Scrambler

If the symbols are just bits, then if we take the polynomial (k=9)

$$H(z) = z^9 + z^4 + 1$$

the circuit of Figure 1.1 becomes exactly the left part of the following double scrambler which appears the ANSI/ SMPTE 259M standard for a Serial Digital Interface (Ref [33]) ,

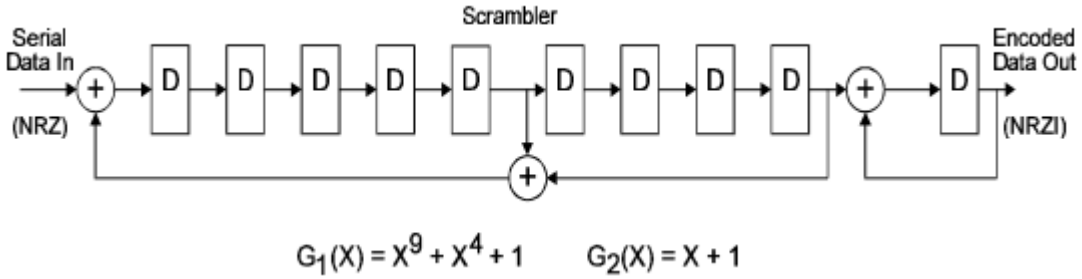


Fig 1.4: Two scramblers in series.

The right end of this circuit is a k = 1 "NRZI mini-scrambler" which we shall mention later. It is another example of Fig 1.1 with the simple polynomial H(z) = z + 1.

One can interpret the action of the first scrambler on the incoming signal as division of the incoming signal's polynomial by $z^9 + z^4 + 1$. Then the second scrambler divides the output of the first scrambler by z+1. The reason for using such scramblers is explained later in this document.

1.3 More Interpretation of Polynomial Division

We commented above on the interpretation of the polynomials I(z) and O(z), and here we revisit that discussion. Some ideas of the previous section are repeated here for emphasis.

In general, the input data stream entering the circuit of Fig 1.1 starts being non-zero at some time (which we take here to be n = 0) and goes on forever, so its Z Transform has this form,

$$I(z) = i_0 + i_1 z^{-1} + i_2 z^{-2} + \ldots\ldots \qquad (1.3.1)$$

Even if i_n consists of a finite number of non-zero symbols followed by all zeros, we can still regard it as going on forever. As the divider circuit clocks each input symbol in, it clocks one quotient symbol out. Even if the input stream terminates and becomes all zeros, we have no guarantee that the output stream might not go on forever with non-zero symbols. We can make the following distant analogy with numbers:

$$514.000000... /37 = 13.891891891891... \qquad (1.3.2)$$

Although the dividend 514 ends with all zeros, the quotient in this case goes on forever.

Finite Division

It is helpful now to think of *stopping* the circuit after a finite number of input symbols have been shifted in. In this case, we can regard the input polynomial as finite, namely,

$$I(z) = i_0 + i_1 z^{-1} + i_2 z^{-2} + ... + i_r z^{-r} \quad . \tag{1.3.3}$$

Here, we have shifted in r+1 symbols { i_0, ... i_r }, and then we stop the shift clock. We can rewrite this input sequence as:

$$I(z) = z^{-r} [\; i_0 z^r + i_1 z^{r-1} + i_2 z^{r-2} + ... + i_r \;] \equiv z^{-r} I_r(z) \tag{1.3.4}$$

where we have defined the bracketed polynomial as $I_r(z)$, a proper polynomial of degree r. One should understand that there is no absolute significance to the size of the powers, what matters is the relativity between powers. We have written $I(z) = z^{-r} I_r(z)$ so we can deal with something we are comfortable with, namely $I_r(z)$ which is a proper polynomial of degree r. Regardless of how we think of I(z), we know that i_0 is the first non-zero symbol (coefficient) to enter the circuit.

Since H(z) has degree k, and since I(z) has degree 0 (largest exponent in (1.3.3)), we know from (1.2.5) that O(z) must have degree 0-k = -k. This means that its first k symbols all vanish. This is easily interpreted in terms of Figure 1.1 since one has to wait k clocks before any non-zero symbols emerge from the circuit output, after i_0 hits the input. (This will also be true of our second circuit to be given below.)

Note: We assume that all registers are initialized to 0 at some time in the past and therefore an incoming stream of zeros up to time n = 0 maintains zeros in all the registers. Alternatively, we could just clear the registers at n = 0 and start the input stream there and forget about the past.

Since we have shifted in r+1 symbols, we must have shifted out r+1 symbols, of which the first k vanish. Thus, the output sequence contains a total of r+1-k non-zero symbols. So here is O(z):

$$O(z) = o_k z^{-k} + o_{k+1} z^{-k-1} + ... o_{k+r} z^{-r} \quad . \tag{1.3.5}$$

$$= z^{-r} [\; o_k z^{r-k} + o_{k+1} \; z^{r-k-1} + ... + o_{k+r} \;] \quad = \; z^{-r} O_{r-k}(z) \quad .$$

As long $r \geq k$, $O_{r-k}(z)$ is a proper polynomial of degree r-k. So far then we have:

$$O(z) = I(z) / H(z) \qquad I(z) = z^{-r} I_r(z) \qquad O(z) = z^{-r} O_{r-k}(z)$$

Insert the last two into the first, cancel the z^{-r} and we get:

$$O_{r-k}(z) = I_r(z) / H(z) \quad . \tag{1.3.8}$$

The three polynomials appearing in (1.3.8) are now all proper polynomials ($r \geq k$). We can take the above equation, sit down, and do our usual long division by hand and try to make a comparison with what the circuit of Figure 1.1 is doing. We shall entertain this task a few sections hence.

The Hidden Remainder

We have seen that if we shift in a polynomial of degree r, and we divide by a divisor H(z) of degree k, we end up with a quotient of degree r-k. There is of course a *remainder* that is hidden somehow within the circuit. In other words, the *information* of the remainder is somehow stored in the k registers. We know that the remainder is a polynomial of degree $\leq$ k-1 . Mathematically, we can write:

$$I_{\mathbf{r}}(z) / H(z) = O_{\mathbf{r-k}}(z) + R(z)/H(z) \qquad (1.3.9)$$

where O is the quotient polynomial, and R is the remainder polynomial. The circuit, however, does not output the second term, it only outputs the first term, the quotient.

Question: What happens if we now shift in one more input symbol?

Answer: In this case, we redo the above analysis replacing r with r+1 everywhere. We end up with the circuit then computing:

$$O_{\mathbf{r+1-k}}(z) = I_{\mathbf{r+1}}(z) / H(z)$$

and mathematically, we know there is again some hidden remainder, so we write:

$$I_{\mathbf{r+1}}(z) / H(z) = O_{\mathbf{r+1-k}}(z) + R'(z)/H(z)$$

The quotient has just gained one more symbol, but now the remainder is likely to be completely different, so we put a prime on R'.

The maximum possible degree of the remainder never changes, the degree is always $\leq$k-1. So the remainder has k coefficients, some of which may be zero. And the circuit has k registers. The remainder is some linear combination (we suspect) of the symbols in the registers. If at some point all registers become 0, the remainder will be zero, and then the sequence of quotient symbols terminates. We saw this happen in Example 3 above.

So we can now summarize what happens when the input sequence contains only a finite number r+1 of non-zero symbols. For any number of clocks n after the input sequence goes to zero, we get

$$I_{\mathbf{r+n}}(z) / H(z) = O_{\mathbf{r+n-k}}(z) + R^{(\mathbf{n})}(z)/H(z)$$

On each clock we get a new quotient symbol, and some new hidden remainder $R^{(\mathbf{n})}(z)$. If at some point the remainder goes to zero, the quotient sequence terminates and the system returns to the idle state which it had at negative time. The other possibility is that symbols keep circulating forever, and the quotient sequence then is infinite.

Impulse Response

This "circulating forever" possibility is why our circuit is sometimes called an infinite impulse response filter (IIR). In fact, if the input sequence consists of a single non-zero 1 symbol, then the "filter" output is by definition the impulse response. In this case we would write (using (1.1.1) as $I(z) = \Sigma_{n=-\infty}^{\infty} i_n z^{-n}$ with $i_n = \delta_{n,0}$)

$$I(z) = 1 \qquad O(z) = I(z)/H(z) = 1/H(z) \tag{1.3.10}$$

so the circuit is now dividing H(z) into an ever growing polynomial whose leading power has coefficient 1, and whose later powers have zero coefficients. If we wait for r clocks after the unit symbol has shifted in, we have $I_r(z) = z^r\ I(z) = z^r$ so (1.3.9) becomes

$$z^r / H(z) = O_{r-k}(z) + R(z)/H(z) .$$

Unless H(z) is a single power of z, the impulse response will go on forever. We shall have more to say about this subject later on.

1.4 The Standard-Form Polynomial Divider

In the previous section we have given a "scrambler" type implementation of a polynomial divider. We proved that it really does divide polynomials.

Here, we present the more standard form of such a polynomial divider:

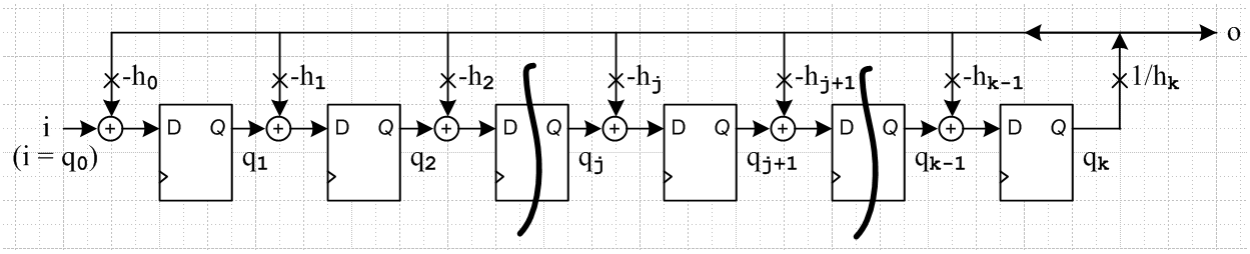


Figure 1.5. The Standard Form polynomial divider circuit.

It is going to turn out that O(z) = I(z)/H(z) exactly as before, but this is certainly not obvious at this point. Comparing this Fig 1.5 to Fig 1.1, we make these observations:

- There are still k registers, but now they are numbered q_1 to q_k going left to right, whereas in Fig 1.1they were numbered q_0 to q_{k-1} going right to left.

- In Fig 1.1 only the leftmost register got any feedback, but now in Fig 1.5 all registers get feedback.

Analysis of the Standard-Form Divider in the z-domain

Looking at Figure 1.5, we see that for register j+1,

$$q_{j+1}(n+1) = d_{j+1}(n) = q_j(n) - h_j\ o(n) \qquad j = 0,1,2,...k-1 \quad . \tag{1.4.1}$$

There is no register $q_0(n)$, but we can imagine such a register off the left end which produces i(n). Thus, we have $q_0(n) = i(n)$ and, after Z transform, $Q_0(z) = I(z)$.

Similarly, there is no register $q_{k+1}(n+1)$ but it is convenient to *define* q_{k+1} by the above equation so that

$$q_{k+1}(n+1) \equiv\ q_k(n) - h_k\ o(n) \ .$$

But since $o(n) = (1/h_k)\ q_k(n)$, we have $q_{k+1}(n+1) = 0$ so $q_{k+1}(n) = 0$ and therefore $Q_{k+1}(z) = 0$.

Projecting (1.4.1) into the z-domain yields, again using (1.1.4),

$$z\ Q_{j+1}(z) =\ Q_j(z) - h_j\ O(z). \tag{1.4.2}$$

Now multiply both sides by z^j to get

$$z^{j+1}\ Q_{j+1}(z) =\ z^j\ Q_j(z) - h_j\ z^j\ O(z).$$

This is a difference equation we can solve by inspecting the first several iterations:

$$\begin{aligned} zQ_1(z) &= Q_0(z) - h_0O(z) && \text{// j = 0} \\ &= I(z) - h_0O(z) \end{aligned}$$

$$\begin{aligned} z^2Q_2(z) &= z\ Q_1(z) -\ h_1zO(z) && \text{// j = 1} \\ &= [I(z) - h_0O(z)] -\ h_1z\ O(z) \\ &= I(z)\ - (h_1z\ + h_0)\ O(z) \end{aligned}$$

$$\begin{aligned} z^3Q_3(z) &= z^2\ Q_2(z) - h_2\ z^2\ O(z) && \text{// j = 2} \\ &= [I(z)\ - (h_1z\ + h_0)\ O(z)] - z^2\ h_2O(z) \\ &= I(z)\ - (h_2z^2 + h_1z\ + h_0)\ O(z)] \end{aligned}$$

Evidently the general solution is this:

$$z^{j+1}Q_{j+1}(z) =\ I(z)\ - [\ h_j\ z^j\ +... + h_1\ z +\ \ h_0\ ]O(z) \quad . \tag{1.4.3}$$

Setting j = k then gives

$$z^{k+1}Q_{k+1}(z) =\ I(z)\ - [\ h_k\ z^k\ +... + h_1\ z +\ \ h_0\ ]O(z)$$

$$= I(z) - H(z)O(z) .$$

But above we showed that $Q_{k+1}(z) = 0$, so the above says $I(z) = H(z)O(z)$ or

$$O(z) = I(z) / H(z) . \qquad (1.4.4)$$

Thus, the circuit in Figure 1.5 is functionally identical to that in Figure 1.1.

Registers: If we take a snapshot of the division process of Figure 1.5 at any clock, we *claim* that the k registers of Figure 1.5 always contain the most significant k symbols of the "current dividend". We can always interpret the current dividend (contents of the k registers of Figure 1.5) as being the remainder of the polynomial which so far has been shifted in. We will elaborate on this in the next section. This is very different from what we said about Figure 1.1. There, the registers contained the next k symbols of the quotient-to-be. Thus, although the I/O of these circuits is the same, their internal registers have different meanings.

1.5 How Polynomial Dividers Actually Work

Operation of the Standard-Form Polynomial Divider

Although one can do the analysis for general k, things are very much clearer of we take a specific k value, so we assume k = 3. As we have seen, the polynomial dividers generates a quotient of this form

$$O(z) = I(z) / H(z) . \qquad (1.2.5)$$

Recall that the input polynomial, output polynomial, and divisor polynomial have these forms, where for each we put the highest power of z on the left (and we assume k = 3):

$$I(z) = i_0 + i_1 z^{-1} + i_2 z^{-2} + ...$$
$$O(z) = o_3 z^{-3} + o_4 z^{-4} + o_5 z^{-5} + ...$$
$$H(z) = h_3 z^3 + h_2 z^2 + h_1 z + h_0 .$$

Recall that the divider registers are all initialized to zero. So here is our truncated Figure 1.5:

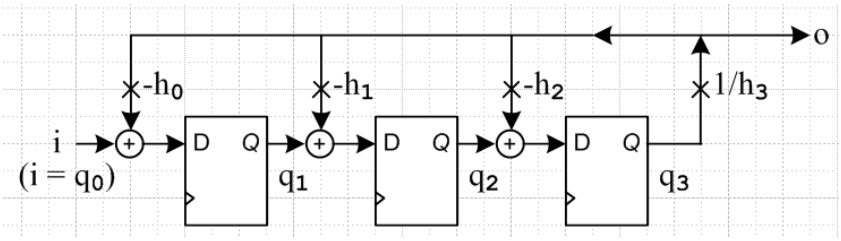


Figure 1.6: The standard-form divider circuit with k = 3.

To demonstrate the operation, we first write out our long division, then explain things below:

$$o_3 z^{-3} + o_4 z^{-4} + o_5 z^{-5} + o_6 z^{-6} + o_7 z^{-7} + o_8 z^{-8} + ...$$

$h_3z^3 + h_2z^2 + h_1z^1 + h_0z^0 \mid i_0z^0 + i_1z^{-1} + i_2z^{-2} + i_3z^{-3} + i_4z^{-4} + i_5z^{-5} + ...$

$- o_3h_3z^0 - o_3h_2z^{-1} - o_3h_1z^{-2} - o_3h_0z^{-3}$

Current Dividend #1 $a_1z^{-1} + a_2z^{-2} + a_3z^{-3} + i_4z^{-4} + i_5z^{-5} + ...$

$- o_4h_3z^{-1} - o_4h_2z^{-2} - o_4h_1z^{-3} - o_4h_0z^{-4}$

Current Dividend #2 $b_2z^{-2} + b_3z^{-3} + b_4z^{-4} + i_5z^{-5} + ...$

$- o_5h_3z^{-2} - o_5h_2z^{-3} - o_5h_1z^{-4} - o_5h_0z^{-5}$

Current Dividend #3 $c_3z^{-3} + c_4z^{-4} + c_5z^{-5} + ...$

$- o_6h_3z^{-3} - o_6h_2z^{-4} - o_6h_1z^{-5} - o_6h_0z^{-6}$

---

and on forever

The divisor H(z) and dividend I(z) are written in the usual manner with the highest power to the left. We then perform the usual grade-school long division process. In every vertical column, the powers of z match. The original dividend appears under the radical (Current Dividend #0). At each stage of the long division process we have a new Current Dividend as shown. The first three terms of each current dividend are shown in green.

To give the above long division layout a compact look, we have defined a lot of new symbols along the way, in this order (down each column, then to the next column):

$o_3 \equiv i_0/h_3$	$o_4 \equiv a_1/h_3$	$o_5 \equiv b_2/h_3$	$o_6 \equiv c_3/h_3$
$a_1 \equiv i_1 - o_3h_2$	$b_2 \equiv a_2 - o_4h_2$	$c_3 \equiv b_3 - o_5h_2$	etc
$a_2 \equiv i_2 - o_3h_1$	$b_3 \equiv a_3 - o_4h_1$	$c_4 \equiv a_3 - o_5h_1$	etc
$a_3 \equiv i_3 - o_3h_0$	$b_4 \equiv i_4 - o_4h_0$	$c_5 \equiv i_5 - o_5h_0$	etc

Notice how the output sequence o_3, o_4, o_5 ..... is being computed in the first row of this little table. The three green terms (without the z powers) indicate the contents of the three registers, but in the order (q_3, q_2, q_1) which is backwards from the ordering in Figure 1.6. Looking at the sequence of symbol definitions above, one realizes that only the green register contents are necessary to compute all the output coefficients o_n.

It is assumed as usual that the shift register is cleared before i_0 comes in. The input symbols just shift right in and appear as (i_2, i_1, i_0) in registers (q_1, q_2, q_3). This is so because up to this point there is no feedback on the o bus. These register contents are indicated by $i_0z^0 + i_1z^{-1} + i_2z^{-2}$ in the original dividend, and as just noted, the order is reversed. After this point, the feedback is activated, and things progress as shown above. Even if the input sequence truncates after some i_r, the output sequence in general continues forever. The exception of course is the case that I(z) is an exact multiple of H(z).

Operation of the Scrambler-Style Polynomial Divider

Again we consider the case k = 3 and we assume the registers are pre-cleared. When input i_0 appears on the i bus, the D input to register q_2 is then (i_0/h_3) and on the next clock edge this becomes the contents of

q_2. In two more clocks, this value will appear as q_0 so (i_0/h_3) must be o_3 ! In fact, at any instant in time, the three registers always hold the next three o_j outputs since the registers form a simple shift register. This fact is crucial to understanding how this circuit works.

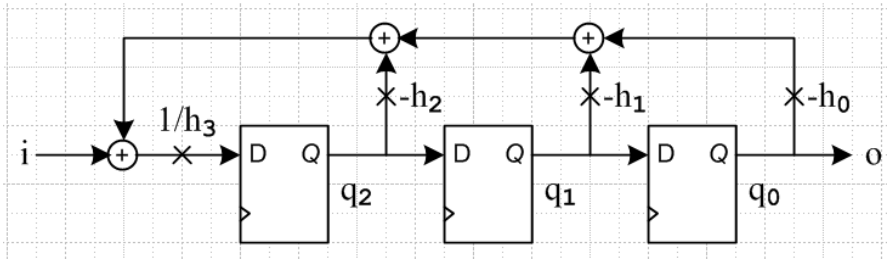


Figure 1.7: The scrambler-style divider circuit with k = 3.

We repeat here the long division layout of the previous section, but with different "coloration":

$$\begin{array}{rl} & o_3z^{-3} + o_4z^{-4} + o_5z^{-5} + o_6z^{-6} + o_7z^{-7} + o_8z^{-8} + \ldots. \\ h_3z^3 + h_2z^2 + h_1z^1 + h_0z^0 \;|\; & i_0z^0 + i_1z^{-1} + i_2z^{-2} + i_3z^{-3} + i_4z^{-4} + i_5z^{-5} + \ldots.. \\ & -o_3h_3z^0 - o_3h_2z^{-1} - o_3h_1z^{-2} - o_3h_0z^{-3} \\ & \text{---} \end{array}$$

Current Dividend #1

$$\begin{array}{rl} & a_1z^{-1} + a_2z^{-2} + a_3z^{-3} + i_4z^{-4} + i_5z^{-5} + \ldots.. \\ & -o_4h_3z^{-1} - o_4h_2z^{-2} - o_4h_1z^{-3} - o_4h_0z^{-4} \\ & \text{---} \end{array}$$

Current Dividend #2

$$\begin{array}{rl} & b_2z^{-2} + b_3z^{-3} + b_4z^{-4} + i_5z^{-5} + \ldots. \\ & -o_5h_3z^{-2} - o_5h_2z^{-3} - o_5h_1z^{-4} - o_5h_0z^{-5} \\ & \text{---} \end{array}$$

Current Dividend #3

$$\begin{array}{rl} & c_3z^{-3} + c_4z^{-4} + c_5z^{-5} + \ldots. \\ & -o_6h_3z^{-3} - o_6h_2z^{-4} - o_6h_1z^{-5} - o_6h_0z^{-6} \\ & \text{---} \end{array}$$

Current Dividend #4

$$\begin{array}{rl} & d_4z^{-4} + d_5z^{-5} + d_6z^{-6} + \ldots. \\ & -o_7h_3z^{-4} - o_7h_2z^{-5} - o_7h_1z^{-6} \ldots. \\ & \text{---------------------------------} \end{array}$$

Current Dividend #5

$$\begin{array}{rl} & e_5z^{-5} + e_6z^{-6} + \ldots \\ & -o_8h_3z^{-5} - o_8h_2z^{-6} \\ & \text{----------------------------} \end{array}$$

Let's start in the middle of things to see what the adders in Figure 1.7 are doing. Consider the column of red expressions which has i_4 at the top. At this time, the output is o_4 and the input is i_4. The adders are therefore computing the following sum (based on the fact just stated about outputs-to-be)

D input to q_2 = $(1/h_3) * [i_4 + (-h_0)o_4 + (-h_1)o_5 + (-h_2)o_6]$

where the expressions inside [...] match those of the red column just noted. On the next clock edge, this value is strobed into register q_2 and it is then destined to become o_7, which is shown in blue underneath the red column of expressions just noted. With the previous Figure 1.6 circuit, this same sum was in effect

computed, but in a set of steps associated with the current dividends. Here the sum is computed in a single shot. The circuit does not store any current dividends, it stores the next three o_n outputs.

If we look one clock later, we have the next column of red expressions being added to determine o_8. And one clock earlier, we have a column adding up expressions to determine o_6. Before this time, some of the adder inputs are 0 so the column of red expressions added has fewer than 4 elements.

ok to here

1.6. The Descrambler-Style Polynomial Multiplier

Consider the following circuit:

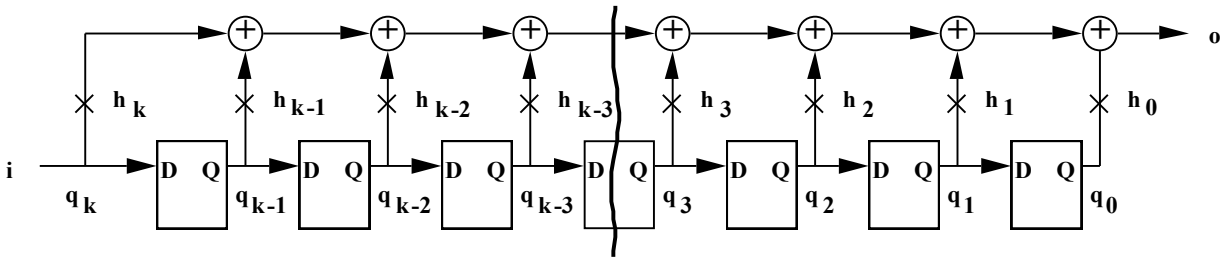


Figure 1.8: A descrambler-style polynomial multiplier circuit.

Again, we arbitrarily call this circuit with outboard adders by the name "descrambler-style" since it is sometimes used in descrambler applications. Below we shall encounter a "standard form" multiplier as well.

Analysis of the Descrambler-Style Multiplier in the z-domain

The analyses of this and the next multiplier circuit are similar to the analyses of the divider circuits discussed above, so we shall proceed with a minimum of comment and shall mimic the divider discussion as much as possible. As with the divider circuits, we assume that the registers were all cleared at some time in the past during which the input stream was all zeros prior to a first non-zero sample i_0. Alternatively, we can assume that the registers are cleared just prior to the input of i_0.

The equation for the output node is:

$$o(n) = \sum_{j=0}^{k} h_j \, q_j \, (n) \quad . \tag{1.6.1}$$

Project into the z domain using (1.1.1) to get

$$O(z) = \sum_{j=0}^{k} h_j Q_j \, (z) \quad . \tag{1.6.2}$$

Translate all $Q_j(z)$ back to the input node using (1.1.4),

$$Q_j(z) = z^{-(k-j)} Q_k(z) = z^{-(k-j)} I(z) . \qquad (1.6.3)$$

The result is

$$O(z) = \sum_{j=0}^{k} h_j [z^{-(k-j)} I(z)] = z^{-k} \left(\sum_{j=0}^{k} h_j z^j \right) I(z) = z^{-k} H(z) I(z)$$

or

$$z^k O(z) = H(z) I(z) . \qquad (1.6.4)$$

The circuit shown in Figure 1.8 must indeed by a *polynomial multiplier*. The output of the physical circuit is O(z), but the product we see is z^k O(z) which is O(z) advanced by k clocks as in (1.1.3). To get the complete product then we have to clock the registers k more times after the last input bit has been fed in. As a simple example, suppose we had H(z) = 1, so (1.6.4) then says $O(z) = z^{-k} I(z)$. In z-domain language, this means the output is delayed from the input by k clocks. This is exactly what Figure 1.8 does when H(z) = 1, since $h_0 = 1$ and we can ignore all the other X's.

Since I(z) is degree 0 as in (1.3.1), and H(z) is degree k, we expect O(z) to be degree 0.

Interpretation of Polynomial Multiplication

Proceeding as in Section 1.3, we consider a "finite multiplication". We shall assume that the input data polynomial I(z) has r+1 non-zero coefficients starting with i_0, and this is followed by all zeros.

It takes r+1 clocks to clock in I(z), and most of the product is then formed. It will take another k clocks to clock out the final k bits of the product, which have been left in the registers. So we assume that we are going to do a total of r+1+k clocks to get our answer.

As before, we write the input polynomial in this form,

$$I(z) = z^{-r} [i_0 z^r + i_1 z^{r-1} + i_2 z^{r-2} + ... + i_r] = z^{-r} I_r(z) . \qquad (1.6.5)$$

The output polynomial, noted above to be of degree 0, has this form after r+1+k clocks,

$$O(z) = o_0 + o_1 z^{-1} + ... o_{r+k} z^{-(r+k)} . \qquad (1.6.6)$$

We factor out $z^{-(r+k)}$ to get a proper polynomial,

$$O(z) = z^{-(r+k)} [o_0 z^{r+k} + o_1 z^{r+k-1} + ... o_{r+k}] \equiv z^{-(r+k)} O_{r+k}(z) . \qquad (1.6.7)$$

So far then we have from (1.6.4), (1.6.5) and (1.6.7),

$$z^k O(z) = H(z) I(z) \qquad I(z) = z^{-r} I_r(z) \qquad O(z) = z^{-(r+k)} O_{r+k}(z)$$

Installing the last two expressions in the first equation then yields,

$$O_{r+k}(z) = H(z)\, I_r(z) \ . \qquad (1.6.8)$$

This now looks like the kind of proper polynomial multiplication we are familiar with. We can take the above equation, sit down, and do the long multiplication by hand if we want, and try to make a comparison with what the circuit of Figure 1.8 is doing. We shall carry out this task below.

Impulse Response

If we inject an I(z) which has $i_0 = 1$ as its only non-zero coefficient, we have I(z) = 1 and (1.6.8) says

$$z^k\ O(z) \ = H(z) \ . \qquad (1.6.9)$$

The impulse response is just $O(z) = z^{-k}\ H(z)$, which is to say

$$O(z) = z^{-k}\ [\ h_k z^k + h_{k-1} z^{k-1} + ... + h_1 z + h_0] \ = h_k + h_{k-1} z^{-1} + h_{k-2} z^{-2} + ... + h_1 z^{-k+1} + h_0 z^{-k} \ .$$

This fact is pretty clear looking at Figure 1.8. As the unity symbol $i_0 = 1$ shifts to the right, each h_j coefficient is activated one at a time and drives the output for one clock period. Right off the bat when i_0 is at the q_k input, we already have $o_0 = h_k$. The sliding one just scans out the h_j coefficients. Each clock of the slide adds another delay factor of z^{-1}.

The circuits of Figure 1.8 and Figure 1.10 (to come) have no feedback, so they are FIR "filters". They have a finite impulse response, as we have just seen. The impulse response lasts k clocks and is then gone, and the system is back to its steady state of idleness.

Registers: The registers in the circuit of Figure 1.8 hold the most recent k input symbols of I(z).

Example : SMPTE Descrambler

If the symbols are just bits, then if we take the polynomial

$$H(z) = z^9 + z^4 + 1 \ ,$$

the circuit of Figure 1.8 becomes exactly the right part of the following double descrambler which appears the ANSI/ SMPTE 259M standard for a Serial Digital Interface (Ref [33]) ,

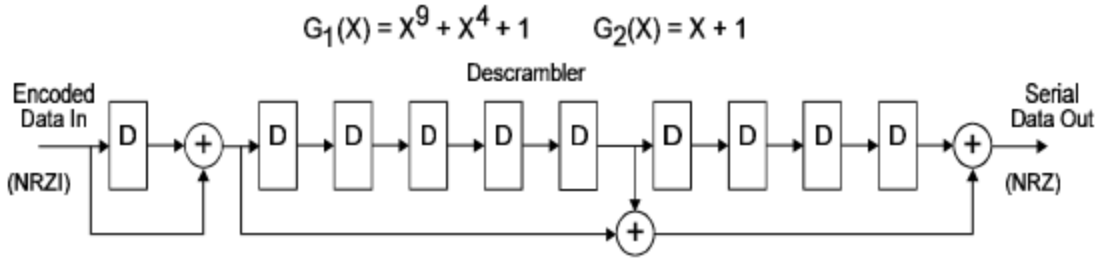


Fig 1.9. Two descramblers in series.

The left end of this circuit is an "NRZI mini-descrambler" which we shall mention later. It is another example of Fig 1.8 with the simple polynomial H(z) = z + 1.

One can interpret the action of the first descrambler on the incoming signal as multiplication of the incoming signal's polynomial by z + 1. Then the second descrambler multiplies the output of the first descrambler by $z^9 + z^4 + 1$.

We can then examine the z-domain overall effect of scrambling a signal S(z) according to Fig 1.4, transmitting that signal through a coaxial cable, and then descrambling the signal according to Fig 1.9:

T(z) ≡ transmitted signal = [S(z) / ($z^9 + z^4 + 1$)] / (z+1)

descrambled signal = [T(z) * (z+1)] * ($z^9 + z^4 + 1$) = S(z)

so the original signal is recovered with no change. Digital signals usually contain embedded synchronization codes so (apart from latency issues) one can ignore the fact that a signal is delayed or advanced by a small number of clocks. The reason for using such scramblers and descramblers is explained later in this document.

1.7 The Standard-Form Polynomial Multiplier

In the previous section we have given a "descrambler" type implementation of a polynomial multiplier. We proved that it really does multiply polynomials.

Here, we present the more standard form of such a polynomial multiplier:

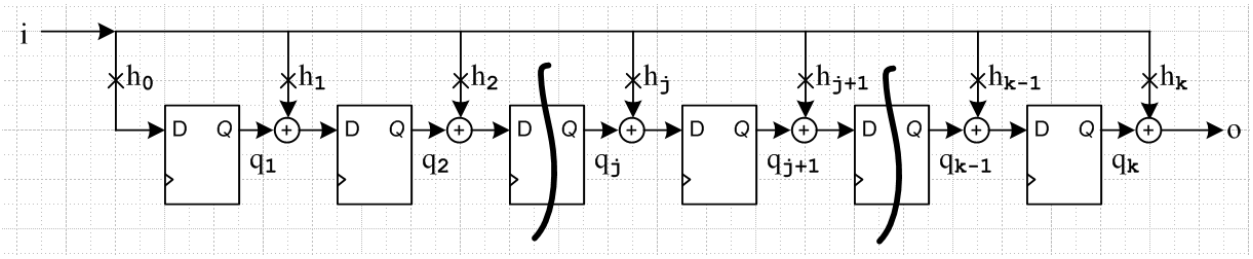


Figure 1.10: The standard-form polynomial multiplier circuit.

It is going to turn out that $z^k O(z) = I(z)\ H(z)$ exactly as before, but this is probably not obvious at this point. Just as Figure 1.8 was very similar to Figure 1.1, Figure 1.10 above is very similar to Figure 1.5. Notice that in Fig 1.10 there are no minus signs in front of the h_j coefficients.

Analysis of the Standard-Form Multiplier in the z-domain

The time-domain equation for register q_{j+1} is seen by inspection of Fig 1.10 to be,

$$q_{j+1}(n+1) = d_{j+1}(n) = q_j(n) + h_j\ i(n) \qquad j = 0,1,2,...k-1 \quad . \tag{1.7.1}$$

Projecting this into the z-domain using the fourth line of (1.1.4) on the left gives,

$$z\ Q_{j+1}(z) = Q_j(z) + h_j\ I(z) \quad . \tag{1.7.2}$$

Notice that this is identical to the division equation (1.4.2) except for the sign in front of h_j.

There is no register q_0, but we can define q_0 using (1.7.1) with j = 0,

$$q_1(n+1) = d_1(n) = q_0(n) + h_0\ i(n)$$

Since Fig 1.10 shows that $d_1(n) = h_0\ i(n)$, we conclude that $q_0(n)$ defined in this manner must be zero. Thus we also have $Q_0(z) = 0$.

Similarly, there is no register q_{k+1} but once again we define q_{k+1} using (1.7.1) with j = k

$$q_{k+1}(n+1) = d_{k+1}(n) = q_k(n) + h_k\ i(n)$$

But Fig 1.10 shows that $q_k(n) + h_k\ i(n) = o(n)$, so we have then that $q_{k+1}(n+1) = o(n)$. In the z domain this says, again using (1.1.4),

$$zQ_{k+1}(z) = O(z). \tag{1.7.3}$$

Now multiply both sides of (1.7.2) by z^j to get

$$z^{j+1}\ Q_{j+1}(z) = z^j\ Q_j(z) + h_j\ z^j\ I(z).$$

This is a difference equation we can solve by inspecting the first several iterations:

$zQ_1(z) = h_0\ I(z)$ // j = 0 (and $Q_0(z) = 0$)

$z^2\ Q_2(z) = z\ Q_1(z) + h_1 z\ I(z)$ // j = 1

$= h_0\ I(z) + h_1 z\ I(z)$

$= (h_1 z + h_0)\ I(z)$

$$z^3\ Q_3(z) = z^2\ Q_2(z) + h_2\ z^2\ I(z) \qquad // j = 2$$
$$= (h_1\ z + h_0)\ I(z) + h_2\ z^2\ I(z)$$
$$= (h_2 z^2 + h_1 z + h_0)\ I(z)$$

Evidently the general solution is this:

$$z^{j+1}\ Q_{j+1}(z) = (\ h_j\ z^j\ + ... h_1\ z + h_0\ )\ I(z)\ . \qquad (1.7.4)$$

Setting j=k we get from (1.7.3) and (1.7.4),

$$z^{k+1}\ Q_{k+1}(z) = z^k\ O(z) = H(z)\ I(z)\ .$$

Our result is then

$$z^k\ O(z)\ = H(z)\ I(z) \qquad (1.7.5)$$

which is the same as that obtained from Figure 1.8. Thus, the circuit in Figure 1.10 is functionally identical to that in Figure 1.8.
ok to here

1.8 How Polynomial Multipliers Actually Work

Operation of the Decrambler-Style Polynomial Multiplier

As we have seen, the polynomial multiplier generates a product of this form

$$O(z)\ = z^{-k}\ H(z)\ I(z)\ .$$

Recall that the input polynomial, output polynomial, and divisor polynomial have these forms, where for each we put the highest power of z on the left: (once again, k = 3)

$$I(z) = i_0 + i_1\ z^{-1} + i_2\ z^{-2} + ...$$
$$O(z) = o_0 + o_1\ z^{-1} + o_2\ z^{-2} + ...$$
$$H(z) = h_3 z^3 + h_2 z^2 + h_1 z + h_0\ .$$

Recall that the registers are all initialized to zero.

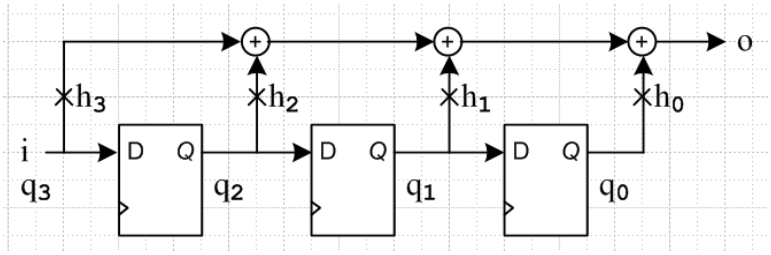


Figure 1.11. Descrambler-style multiplier with k = 3

To demonstrate the operation, we first write out a bunch of symbols, then explain them below:

$$
\begin{array}{l}
\qquad h_3z^3 + h_2z^2 + h_1z^1 + h_0z^0 \\
\ldots. + i_3z^{-3} + i_2z^{-2} + i_1z^{-1} + i_0z^0 \\
\text{----------} \\
\quad i_0h_3z^3 + i_0h_2z^2 + i_0h_1z^1 + i_0h_0z^0 \\
\qquad\quad i_1h_3z^2 + i_1h_2z^1 + i_1h_1z^0 + i_1h_0z^{-1} \\
\qquad\qquad\quad i_2h_3z^1 + i_2h_2z^0 + i_2h_1z^{-1} + i_2h_0z^{-2} \\
\qquad\qquad\qquad\quad i_3h_3z^0 + i_3h_2z^{-1} + i_3h_1z^{-2} + i_3h_0z^{-3} \\
\qquad\qquad\qquad\qquad\quad i_4h_3z^{-1} + i_4h_2z^{-2} + i_4h_1z^{-3} + i_4h_0z^{-4} \\
\qquad\qquad\qquad\qquad\qquad\quad i_5h_3z^{-2} + i_5h_2z^{-3} + i_5h_1z^{-4} + i_5h_0z^{-5} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad **** + ***** + ***** + \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ***** + ***** + \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ***** + \ldots \\
\text{----------} \\
\quad o_0z^3 + o_1z^2 + o_2z^1 + o_3z^0 + o_4z^{-1} + o_5z^{-2} + o_6z^{-3} + o_7z^{-4} + o_8z^{-5} \ldots \\
\\
= z^3 [\, o_0z^0 + o_1z^{-1} + o_2z^{-2} + o_3z^{-3} + o_4z^{-4} + o_5z^{-5} + o_6z^{-6} + o_7z^{-7} + o_8z^{-8} \ldots]
\end{array}
$$

First of all, when multiplying two polynomials by hand, one is certainly allowed to arbitrarily order the powers in each polynomial as one wants. In the above we have H(z) with its highest power on the left, but we show I(z) with its highest power on the right. We then mechanically do the multiplication as we were taught in grade school. The rightmost element i_0z^0 of I(z) is multiplied by H(z) as shown to get the first row. Then the next multiplier element i_1z^{-1} of I(z) is multiplied by H(z) as shown to get the second row, and so on. The rows are aligned so that powers of z match in each column. Normally these rows march to the left, but because I(z) has negative powers, they march to the right.

After all the rows are written out, we *add up the columns* to get the overall product, as shown under the second dotted line. Since $z^3O(z) = H(z)\,I(z)$, we factor out z^3 on the very last line to expose O(z) in the bracket. The idea here is that, for example, $o_2 = i_0h_1 + i_1h_2 + i_2h_3$. To get each o_j symbol, we have to add up the column of expressions above it. This column addition is performed by the set of adders appearing in Figure 1.11!

We can associate a clock tick with each column going left to right. Each clock tick causes another input symbol to shift into the left end of the shift register, and causes all the symbols already in the register to shift to the right one position. At clock tick 0 the output is $o_0 = i_0h_3$. At clock tick 1 the output is $o_1 = i_0h_2 + i_1h_3$, and so on. By clock tick 3, the registers contain (i_3,i_2,i_1,i_0) and all three adders are "active" to produce a sum of four numbers as shown in the column above o_3. As time moves on, the three adders continue to add four symbols of a column. If the input polynomial is finite so the last non-zero input is i_r and all subsequent inputs are 0, the rows taper off the way they began. For example, if i_5 were the last input, the above picture applies if we delete all the rows with asterisks. In this case, looking at the two polynomials being multiplied, it is clear that the largest power in the result is z^3 and the smallest is z^{-5}, so there will be 3-(-5)+1 = 9 symbols in the product, namely o_0 through o_8. In the general case where the last input is i_r and H(z) has degree k, the product will have k-(-r)+1 = k+r+1 symbols.

Operation of the Standard-Form Polynomial Multiplier

Again we consider the case k = 3 and this time we assume a finite input stream (i_0, i_1, i_2, i_3, i_4).

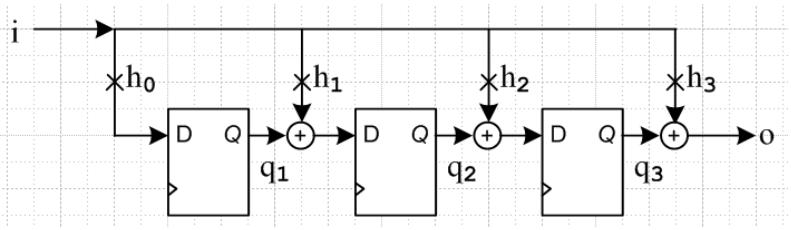


Figure 1.12. Standard-form multiplier with k = 3

We first write down the math, and then explain it below:

$$
\begin{array}{l}
\qquad\qquad\qquad

As we go down, we make up names for coefficients. For example, $a_0 \equiv i_0h_0 + i_1h_1$ and later $b_0 \equiv a_0 + i_2h_2$. When i_0 is at the input, the output is $o_0 = i_0h_3$ as shown in red on the partial product #0 line.

In each partial product, the three expressions shown in green are the contents of the three registers q_1, q_2, q_3 and we could regard the expression in red as the output o captured by some register not shown off to the right. The black expression underneath each partial product represents the other set of inputs to the array of adders in Fig 1.12. On each clock, the result of an addition is strobed into the registers (green), an output in red is strobed out off the o bus, and the blue expressions are no longer stored anywhere since we don't need them.

The first output (ignoring powers of z) is $o_0 = i_0h_3$ and the second is $o_1 = a_2 = i_0h_0 + i_1h_1$, both in agreement with the Figure 1.11 circuit analysis.

Since i_4 is assumed to be the last non-zero input symbol, the machinery keeps running in its endgame with all-zero lines added in, and during this phase the three register contents are just shifted right to produce the last 3 output symbols, out after which the output goes to zero.

1.9 Polynomial Processors in the Time Domain

In the previous sections of this Chapter we concentrated on the various circuits from a polynomial point of view. The circuits divided or multiplied two polynomials to produce an output polynomial.

In this section, we focus directly on the symbols of the input and output data streams in both cases, and we give a very concise restatement of the operation of our various circuits. We first do some mechanical derivations to convince the reader that the results are valid, then at the end, we show why these are the right results.

The Descrambler-Style Multiplier in the Time Domain

Already from Eq. (1.6.1) we have most of our desired result,

$$o(n) = \sum_{j=0}^{k} h_j q_j(n) \, . \qquad (1.6.1)$$

Recall that q_j refers to register number j, and n is a time index. We are in the time domain here. Looking at Figure 1.8, we see that

$$q_1(n) = q_2(n-1) = q_3(n-2) = \text{etc} \, .$$

This is because in Fig 1.8 data samples just march from one register to the next with no alteration. What *is* at location q_1 at time n is what *was* at q_2 at time n-1. If we start with register j and move in this way to the left (going backwards in time) we get

$$q_j(n) = q_{j+1}(n-1) = q_{j+2}(n-2) \ldots\ldots = q_k(n-k+j) = i(n-k+j) \qquad (1.9.1)$$

where notice that the subscript and argument always add up to j+n. Here, we have mapped a given q_j all the way to the left side of Figure 1.8 where we identify q_k with the input data stream. If we insert this expression (1.9.1) for $q_j(n)$ into (1.6.1) for o(n) we get,

$$o(n) = \sum_{j=0}^{k} h_j \; i(n-k+j) \; .$$

And since n is an arbitrary time index on both sides, we can take $n \rightarrow n+k$. At the same time, we put our time index as a subscript instead of as an argument. This gives our final result:

$$o_{k+n} \; = \; \sum_{j=0}^{k} h_j \; i_{n+j} \; . \qquad (1.9.2)$$

For each integer n, we get a very simple equation relating the output sequence to the input sequence. If we were solving for i_m in terms of o_m, this would be called a set of difference equations. However, for the multiplier of Figure 1.8, we are really interested in o_m as a function of i_m. Nevertheless, we shall loosely refer to this as a difference equation.

Claim: Since the Figure 1.10 multiplier performs the same function as the Figure 1.8 multiplier, the above equation applies to it as well.

Proof: We leave it to the reader to directly derive the above result for the Figure 1.10 multiplier. We know the result will be the same, and soon this will become very obvious.

The Scrambler-Style Divider in the Time Domain

We start with (1.2.1) which reads

$$q_{k-1}(n+1) = \; (1/h_k) \; [\; i(n) - \sum_{j=0}^{k-1} h_j \; q_j(n) \;] \qquad \textit{(1.2.1)}$$

Notice that Fig 1.1 has the same property noted in the previous section: the symbols move left to right from each register to the next without alteration. Although there is no q_k appearing in Fig 1.1, we can define it to be the D input of register q_{k-1} so that

$$q_k(n) \; \equiv d_{k-1}(n) = q_{k-1}(n+1)$$

since what appears at a D register input always appears at the Q output one clock later. Thus, we can rewrite (1.2.1) in this more convenient form,

$$q_k(n) = (1/h_k)\left[\, i(n) - \sum_{j=0}^{k-1} h_j\, q_j(n) \,\right]$$

Move h_k to the left side, then realize that the left side is just the kth term of the sum. Thus we have

$$\sum_{j=0}^{k} h_j\, q_j(n) = i(n) \qquad (1.9.3)$$

Now map the $q_j(n)$ to the *right* by going *forward* in time, see Figure 1.1:

$$q_j(n) = q_{j-1}(n+1) = q_{j-2}(n+2) = \ldots\ldots = q_0(n+j) = o(n+j) \qquad (1.9.4)$$

where the subscript and argument always add up to j+n. Inserting this into (1.9.3) we get

$$\sum_{j=0}^{k} h_j\, o(n+j) = i(n)$$

Again going to subscripts for the time indices, we get our final result,

$$i_n = \sum_{j=0}^{k} h_j\, o_{n+j} \quad . \qquad (1.9.5)$$

This is amazingly similar to the multiplier result (1.9.2) given above. Here, however, since we are solving for o_m in terms of i_m, we really do have a set of difference equations.

Claim: Since the Figure 1.5 divider performs the same function as the Figure 1.1 divider, the above equation applies to it as well. The reader is welcome to derive this fact by brute force.

Convolution Theorem Approach

Consider equation (1.6.4) which says that a polynomial multiplier does in fact multiply polynomials:

$$z^k\, O(z) = H(z)\, I(z) \quad . \qquad \textit{(1.6.4)}$$

We showed that the circuits of Figures 7 and 8 each implement this operation.

The reader is now referred to the summary box (24.37) at the end of Section 24 of Ref[32]. This box summarizes the properties of the Z Transform. Like all derivatives of the underlying Fourier Integral Transform, there is a convolution theorem which operates with respect to the Z transform. If we set our time step $\Delta t = 1$, meaning 1 clock period, the convolution theorem states:

$$A(z) = B(z)C(z) \quad \Leftrightarrow \quad a_n = \sum_{j=-\infty}^{\infty} b_{n-j}\, c_j \qquad (1.9.6)$$

We now make the following selections for A,B and C:

$$A(z) = z^k O(z) \quad \Rightarrow \quad a_n = o_{n+k} \qquad \text{// see Eq. (1.1.3)}$$
$$B(z) = I(z) \quad \Rightarrow \quad b_n = i_n$$
$$C(z) = H(z^{-1}) \quad \Rightarrow \quad c_n = h_{-n}$$

The last item deserves some comment. Recall that the Z transform is set up for polynomials in "Z transform format", as in Eq. (1.1.1) for example, where coefficients are aligned with negative powers of z. Our I(z) and O(z) have always been in this format, but H(z) is in "proper polynomial format" where we align with positive powers of z.

$$H(z) = h_0 + h_1 z + h_2 z^2 + \ldots + h_{k-1} z^{k-1} + h_k z^k \qquad (1.2.6)$$

Thus, we write $H(z^{-1})$ to put H back into the Z transform format.

$$H(z^{-1}) = h_0 + h_1 z^{-1} + h_2 z^{-2} + \ldots + h_{k-1} z^{-(k-1)} + h_k z^{-k}$$

A glance at Eq. (1.1.1) shows that $z \rightarrow z^{-1}$ in a Z transform means that $f_n \rightarrow f_{-n}$ in the time domain.

Installing now the above time-domain sequences into the convolution sum of (1.9.6), then taking $j \rightarrow -j$ on the dummy summation index, we get

$$o_{n+k} = \sum_{j=-\infty}^{\infty} i_{n-j}\, h_{-j} = \sum_{j=-\infty}^{\infty} i_{n+j}\, h_j \quad .$$

Now we know that h_j vanishes for $j<0$ and for $j>k$, so our final result is then

$$o_{k+n} = \sum_{j=0}^{k} h_j\, i_{n+j}$$

which agrees with our derivation (1.9.2) above.

We quickly repeat the process for the division equation (1.2.5),

$$I(z) = O(z)H(z) \; . \qquad (1.2.5)$$

This time make the following selections for A,B and C:

$$A(z) = I(z) \quad \Rightarrow \quad a_n = i_n$$

$$B(z) = O(z) \quad \Rightarrow \quad b_n = o_n$$
$$C(z) = H(z^{-1}) \quad \Rightarrow \quad c_n = h_{-n}$$

Installing these into the convolution sum of (1.9.6) gives,

$$i_n = \sum_{j=-\infty}^{\infty} o_{n-j}\ h_{-j} = \sum_{j=-\infty}^{\infty} o_{n+j}\ h_j = \sum_{j=0}^{k} h_j\ o_{n+j}$$

which duplicates (1.9.5) above. We now summarize the results of this section:

Polynomial Processors (1.9.7)

	z-domain	time-domain
Polynomial Divider	$O(z) = I(z)/H(z)$	$i_n = \sum_{j=0}^{k} h_j\ o_{n+j}$
(Figure 1.1 or Figure 2)		
Polynomial Multiplier	$z^k\ O(z) = I(z)\ H(z)$	$o_{k+n} = \sum_{j=0}^{k} h_j\ i_{n+j}$
(Figure 7 or Figure 8)		

1.10 Simultaneous Polynomial Multiply and Divide

The following circuit is both useful in cyclic code hardware, and serves as a check on all previous work of this chapter. Consider,

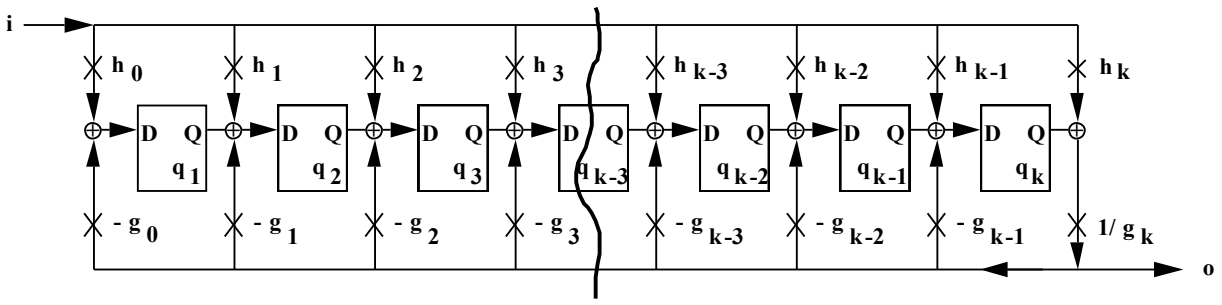


Figure 1.13: A simultaneous polynomial multiplier and divider.

This circuit looks like a superposition of a multiplier on the top by polynomial H(z), and a divider on the bottom by polynomial G(z). Both polynomials are of degree k.

Before analyzing this circuit, it is useful to consider the limits that give a simple divider or a simple multiplier.

First, if we set $g_k = 1$ and all other $g_i = 0$, Figure 1.13 becomes the multiplier of Figure 1.10. Notice that in this case we have $G(z) = z^k$ and not $G(z) = 1$.

Second, if we set $h_0 = 1$ and all other $h_i = 0$, Figure 1.13 becomes the divider of Figure 1.5 in which the h_i are labeled g_i. In this case we have $H(z) = 1$.

Analysis of the simultaneous multiply/divide circuit in the z-domain

We are by now familiar with the technique. We start with a specific case like this one,

$$q_3(n+1) = q_2(n) + h_2\, i(n) - g_2\, o(n)$$

and then generalize to an arbitrary register,

$$q_{j+1}(n+1) = q_j(n) + h_j\, i(n) - g_j\, o(n) \quad . \tag{1.10.1}$$

Project into the z plane to get:

$$zQ_{j+1}(z) = Q_j(z) + h_j\, I(z) - g_j\, O(z) \quad . \tag{1.10.2}$$

We now examine the two extremal cases j = 0 and j = k. For j = 0, we take $Q_0(z) = 0$ at the left end, since an imagined q_0 register to the left of q_1 contributes 0 to the adder going to the q_1 input. For j = k, equation (1.9.2) states that $zQ_{k+1}(z) = Q_k(z) + h_k\, I(z) - g_k\, O(z)$. Looking at the rightmost adder node in Fig 1.13 we see that $Q_k(z) + h_k\, I(z) = g_k O(z)$. Thus, although there is no q_{k+1} register, we must take $Q_{k+1}(z) = 0$.

The brute force solution of the difference equation proceeds as usual:

$$zQ_1(z) = h_0\, I(z) - g_0\, O(z) \qquad // \; j = 0$$

$$z^2Q_2(z) = zQ_1(z) + h_1\, z\, I(z) - g_1\, z\, O(z) \qquad // \; j = 1 \text{ (times z)}$$
$$= (\, h_1 z + h_0\,)\, I(z) - (\, g_1 z + g_0\,)\, O(z)$$

...

$$z^{j+1}\, Q_{j+1}(z) = (\, h_j z^j + \ldots h_0\,)\, I(z) - (\, g_j z^j + \ldots g_0\,)\, O(z) \quad . \tag{1.10.3}$$

Setting j=k and using the above limit that $Q_{k+1}(z) = 0$ we get:

$$0 = H(z)\, I(z) - G(z)\, O(z)$$

which we can then solve to get

$$O(z) = I(z)\, H(z) / G(z) \,. \qquad (1.10.4)$$

Sure enough, Figure 1.13 multiplies by H(z) at the same time it divides by G(z). We have seen in the derivation that it keeps track of the successive additions due to the multiplication, as well as the successive subtractions of the division.

Now lets reexamine the two limiting cases mentioned above. If we select H(z) = 1, so that $h_0 = 1$, then we recover the divider of Figure 1.5:

$$O(z) = I(z) / G(z) \,.$$

On the other hand, to get a multiplier, we need to select G(z) such that $g_k = 1$. This means that we have $G(z) = z^k$. Then we duplicate the results of Figure 1.10,

$$O(z) = I(z)\, H(z) / z^k \qquad \text{or} \qquad z^k\, O(z) = I(z)\, H(z) \,.$$

Application: In an encoder for a cyclic code, one needs to compute the remainder of a power z^s times a an incoming polynomial. See for example Ref [GAL] Chapter 8 (c), where s = n-k is the number of parity check symbols for the code (n,k). The Figure 1.13 circuit can be used for this purpose. When the input stream is done shifting in, the registers will contain the desired remainder, as discussed earlier. We have:

$$O(z) = [\, I(z)\, z^s \,] / G(z)$$

Thus, $H(z) = z^s$, which means that the input stream is injected only into stage s in Figure 1.13. That is, h_s=1, and all other h_i vanish. If this point happens to align with a vanishing coefficient of g(z), then no 3-input adders are needed in the circuit!

We leave as an exercise for the reader to build a simultaneous multiplier/divider out of Figure 1.1 and Figure 7. It seems that it ought to be possible.

1.12 Application: Cyclic Redundancy Check

In this little section the degree of polynomial H(z) is K ≡ n-k instead of k. This K is then the number of registers in a Figure 1.5 polynomial divider set up to divide a polynomial by H(z). The symbols n and k have the following new meaning: a set of k data symbols is used to construct a proper polynomial d(z). A set of n code symbols (known as a block) is transmitted where n > k. The block contains the k data symbols and n-k "parity check symbols" which are used to allow detection of errors. At the transmitting end, a Figure 1.5 polynomial divider is used to divide the polynomial $[z^{n-k}d(z)]$ by the polynomial H(z) of degree K = n-k. In this context, H(z) is referred to as a "generator" polynomial. This division can be thought of this way

$$z^{n-k}d(z) = D(z)H(z) - \gamma(z)$$

where D(z) is the division quotient and -γ(z) is the division remainder (which has degree ≤ n-k). The quotient D(z) has degree k, the same as that of d(z). A "code word polynomial" (degree n) is given by

$$C(z) \equiv D(z)H(z) = \gamma(z) + z^{n-k}d(z)$$

where we just use the previous line to get the right side. If we write out the coefficients of C(z) (from low to high power), we find that the first n-k of them are the coefficients of γ(z), and the last k of them are the data symbols of d(z). For example, if n=5 and k=3,

$$C(z) = (\gamma_0+\gamma_1 z) + z^2(d_0+d_1 z+d_2 z^2) = \gamma_0+\gamma_1 z+d_0 z^2+d_1 z^3+d_2 z^4 \rightarrow (\gamma_0, \gamma_1, d_0, d_1, d_2)$$
$$= (c_4, c_3, c_2, c_1, c_0).$$

Thus in this scheme, the K = n-k parity check symbols γ_i are the negative of the remainder coefficients obtained by doing the division $[z^{n-k}d(z)]/H(z)$. This remainder is obtained using a Figure 1.5 divider with K = n-k registers. When the proper polynomial division has completed, the registers contain the remainder -γ(z). By grounding the feedback line in Figure 1.5 and directly accessing the output of the rightmost register, the next n-k clocks shift out -γ(z) and the coefficients are then negated and appended to the data stream as shown in the example above to create code polynomial C(z). The data stream of coefficients c_i, which we think of as the polynomial C(z), is then transmitted (symbol d_2 first in our example). At the receiving end, the received C(z) is divided by H(z) by another Figure 1.5 divider. Since C(z) ≡ D(z)H(z), there should be no remainder! If there was an error in the transmission so that some erroneous code word C'(z) was received, one gets C'(z) = D(z) H(z) + s(z) where s(z) is an anomalous remainder (the syndrome) which is found sitting in the Figure 1.5 registers after the division. This error detection scheme is known as a Cyclic Redundancy Check (CRC). Notice that both the transmitting end (encoder) and receiving end (decoder) have Figure 1.5 dividers which divide by H(z).

The subject is discussed more in Chapter 8 of Ref [Galois]. There, variable z is called x, and H(z) is called g(x).

Appendix 1.1. Processing polynomials vs. processing integers.

In this section, we will consider a series of questions which will bring out some distinctions between dealing with polynomials and dealing with integers.

Question 1: Can we make a correspondence between multiplying polynomials and multiplying base-10 integers?

For example, we might try to associate $5z^2 + 1z + 4$ with the base-10 integer 514. If we think of z = 10, this seems to be a reasonable association. Now let's multiply two polynomials:

$(3z+8)(5z+9) = (15z^2 + 67\,z + 72)$

On the other hand, we could write down this base-10 integer product:

$38 \bullet 59 = 2242$

If we evaluate $(15z^2 + 67\,z + 72)$ at z=10, we do in fact get 2242. *However*, note the following:

$38 \bullet 59 = 2242 \quad \not\Rightarrow \quad (3z+8)(5z+9) = (2z^3 + 2z^2 + 4z + 2)$

We will now give a formal statement of why $A \not\Rightarrow B$.

(1) We can represent a base-10 integer as an m-tuple of elements of the field Z_{10} = Mod(10). Such m-tuples themselves form a ring which has a 1-to-1 correspondence with the ring of integers Z. Let's call this ring Z-base-10. We have a certain familiar rule for the multiplication of such m-tuples. The rule is that you convert each m-tuple to an integer, get the product, and then convert the result back to an m-tuple. This rule is very effectively implemented by a device known as a calculator.

(2) On the other hand, we can represent a polynomial f(z) as an m-tuple of elements of the integer ring Z. Such m-tuples themselves form a ring known as Polys(z,Z). We have a rule for multiplying these m-tuples, it is what we do when we do a long multiplication of polynomials. The result can also be represented as an m-tuple.

(3) What we have pointed out above in our $A \not\Rightarrow B$ counterexample is that the two rings Z and Polys(z,Z) are not the same. Once again,

Z-base-10: $\{3,8\} \bullet \{5,9\} = \{2,2,4,2\}$

Polys(z,Z): $\{3,8\} \bullet \{5,9\} = \{15,67,72\}$

Well, the reader might say, it is pretty obvious that these two rings are different. For example, we cannot represent 15 or 67 as a single base-10 digit. Also, we would be in trouble if we tried to deal with a polynomial with a negative coefficient:

$(3z^2 - 7z + 2) = ?= 3[-7]2$

The thing on the right does not look much like a base-10 integer. This leads to a refinement of our previous question:

Question 2: Can we make a correspondence between multiplying polynomials defined over the field Z_{10} = Mod(10) and multiplying base-10 integers?

Now consider the same product of polynomials above, but coefficients are in Z_{10}:

$(3z+8)(5z+9) = (15z^2 + 67\,z + 72) = (5z^2 + 7z + 2)$

Now we have eliminated the abovementioned objections, namely, we no longer have negative coefficients, nor do we have coefficients that are larger than one digit. However, we still have no correspondence between the above polynomial product and this integer product 38•59 = 2242. So we now have:

Z-base-10: $\{3,8\}\bullet\{5,9\} = \{2,2,4,2\}$

Polys(z,Z_{10}): $\{3,8\}\bullet\{5,9\} = \{5,7,2\}$

Why are these rings not the same? At the point $(15z^2 + 67\,z + 72)$ we at least stood a chance, but then when we map each coefficient into Z_{10} to get $(5z^2 + 7z + 2)$, we throw out information. There is no way to connect this polynomial with the integer 2242. This leads to the following observation:

Fact: Doing polynomial multiplication in the ring Polys(z,Z_{10}) is the same as multiplying the corresponding base-10 integers *if you ignore all carries*.

For example,

```
  59
  38
 ---
  02
57  .
---
572
```

This is exactly what you are doing when you multiply (3z+8)(5z+9) with coefficients in Z_{10}. So we can now generalize the above to say:

Fact: Multiplying polynomials with coefficients in the ring Z_n corresponds to multiplying integers base-n provided that all carries are thrown out.

Corollary: Multiplying polynomials with coefficients in the field GF(2) = Z_2 corresponds to multiplying binary numbers provided all carries are thrown out.

We are now ready for,

Question 3: Is there some corresponding statement we can make about *dividing* polynomials?

Our first observation is one that we probably should have made earlier. Looking back at our long hand division of polynomials in Figure 5, or at the circuits of Figure 1.1 or 2, we realize that the coefficient h_k of H(z) must have an inverse, or we are dead in the water. For example, the very first quotient symbol is $(i_0/h_k) = i_0 \bullet (h_k)^{-1}$. This leaves us with two options if we want to be thinking about integers:

(1) Make sure polynomial coefficients are in a *field*, not just a ring. In a field, all elements have inverses.

(2) Restrict to H(z) which have $h_k = 1$. Such polynomials are called *monic*.

Now, assuming we do (1) or (2), we move to the analog for division of Question 2 above, namely:

Question 4: Can we make a correspondence between *dividing* polynomials defined over the field Z_{10} = Mod(10) and dividing base-10 integers?

Here is an example:

$$(5z^2 + 3z + 1) / (z+2) = (5z + 3) + 5/(z+2)$$

where we give both the quotient and remainder polynomials. Note that coefficients are not in Z, they are restricted to Z_{10}. Now consider the division of two corresponding base-10 integers:

$$531/12 = 44.25 = 44 + 25/100$$

It is hopefully clear that there is not much connection between (5z+3) and "44", and (5) and ".25". To make this very clear we write:

$$531/12 = 44.25 \quad \not\Rightarrow \quad (5z^2 + 3z + 1) / (z+2) = (4z + 4) + (2z+5)/(z+2)$$
$$= (4z + 6) + 1/(z+2)$$

This is no doubt quite obvious to the reader. However, the next fact may be less obvious:

Fact: Doing polynomial division in the ring Polys(z,Z_{10}) is the same as dividing the corresponding base-10 integers *if you ignore all carries and borrows*.

First, we review the above example:

$$(5z^2 + 3z + 1) / (z+2) = (5z + 3) + 5/(z+2)$$

Now we do it with no-carry, no-borrow long division:

```
   .53   .
12 | 531
     50.        / 5*12 = 50 if no carry
      31
      36
       5        / 31-36 = 5 if no borrow [   1 - 6 = 1 + (-6) = 1 + 4 = 5  ]
```

With multiplication of polynomials in Polys(z,Z_{10}) we never encountered borrows because there were never any subtractions. Multiplication consists of only additions. In contrast, the process of long division involves both multiplications (involving possible carries) and subtractions (involving possible carries).

We now generalize,

Fact: Doing polynomial division in the ring Polys(z,Z_n) is the same as dividing the corresponding base-n integers *if you ignore all carries and borrows*.

Corollary: Doing polynomial division in the ring Polys[z,GF(2)] is the same as dividing the corresponding binary numbers *if you ignore all carries and borrows*.

Summary: When dealing with multiplication and division of integers, there is an interaction between the digit positions. This interaction is known as borrow and carry. For example, when the coefficient in a base-10 digit becomes "too large", part of the information is transferred to the next digit on the left via a "carry". One can think of an integer base-z as a polynomial in powers of z, but there is always this interaction implied by the arithmetic rules +, - • /.

In contrast, in the multiplication and division of polynomials in z, everything is carefully aligned with powers of the variable z. Information is *never* transferred between two unequal powers.

We close this section with one final question:

Question 5: How do we know that, in polynomial division over some field F, it is possible that the quotient may never terminate?

Back in Section 1.3 we made this claim, and appealed to an analogy with integer division by observing that

514.000000... /37 = 13.891891891891...

gives a remainder which never terminates. The analogy is that dividing polynomials is "like" dividing integers, and we might compare a non-terminating remainder polynomial with this situation involving integer division:

514/37 = 13	Remainder = 33
5140/37 = 138	Remainder = 34
51400/37 = 1389	Remainder = 7
514000/37 = 13891	Remainder = 33
5140000/37 = 138918	Remainder = 34

51400000/37 = 1389189 Remainder = 7
514000000/37 = 13891891 Remainder = 33

In retrospect, we must now admit that dividing integers is really quite different from dividing polynomials, so our analogy is not very convincing.

A true answer to Question 5 will have to await the next chapter.

Chapter 2: Shift Register Generators

2.1. The Maximum Period of a Shift Register Generator

Figure 1.1 of Chapter 1 shows the "scrambler-style" polynomial divider circuit, and Figure 1.5 shows the "standard" divider circuit. In either circuit, if we set the input to zero, preset the k registers to some values, and just let the thing run, we get some pattern out the output. Such a device is usually referred to as a *shift register generator*.

Definition: The *state vector* of any state machine circuit consisting of k registers, each of which can store any of Q symbols, is a k-tuple consisting of the symbols stored in each register. There are therefore Q^k possible values that this k-tuple (state vector) can have. In other words, there are Q^k states.

Definition: The *state vector period* of a state machine is the minimum number of clocks it takes to get from some starting state vector back to that same state vector. This period may or may not depend on the starting state.

Definition: The *output period* of a state machine is the minimum period of the output symbol sequence. This period is the same no matter where you start counting in the output symbol sequence. Note that the output sequence can in general be a combinatoric function of the state vector, but it is often just the output of one of the registers, as in Figure 1.1 or Figure 1.5.

Fact 1: For a state machine with k registers which can store any of Q symbols, neither the state vector period nor the output period can exceed Q^k.

Proof: The state machine takes some periodic path through its state space. Assume there are N states on this path. In this case, the state vector period is N, and the output period must be some number which integrally divides N, call it M = N/k. Thus, $M \leq N$. In any event, since $N \leq Q^k$, we know that both the state vector period N and the output period M cannot exceed Q^k.

Example: Consider a decade counter built from 4 flip-flops. In this circuit, the state vector period is N = 10, while $Q^k = 2^4 = 16$. We can consider any register to be an "output". The output period of the least significant counter bit is M = 2. The output period of each of the other bits is M = 10.

Now we apply the above notions for general state machines to shift register generators. As is clear from Figure 1.1 and Figure 1.5, if the state vector becomes zero, the generator "dies". All future state vectors are 0, and the output sequence is 0. All periods are 1.

We therefore *exclude* this state vector from the following discussion. We shall for the moment assume that this state vector is never "hit", so we do not worry about the machine dying. Later we shall show why it is never hit. Thus, we have:

Fact 2: In a shift register generator with k registers each of which can store Q symbols, neither the state vector period nor the output period can exceed $Q^k - 1$.

Proof: This is just Fact 1, but we are saying that there are at most Q^k - 1 states, since the zero state is excluded.

2.2 The State Vector Sequence of a Shift Register Generator

In this section we discuss only the "standard" Figure 1.5 style shift register generator. We will apply our Ref [GAL] knowledge of Galois fields to learn what the state vector does after it is initialized in some non-zero state. In subsequent sections, we will discuss the output symbol sequence and also the state vector sequence of the Figure 1.1 type generator.

A spinoff of this section will be a knowledge of what happens to the sequence of remainders of a polynomial division when, after there has been some non-zero input, the input sequence of a polynomial divider is set to zero.

ok to here

The Figure 1.5 Iteration Equation

Figure 1.5 shows k registers q_1 through q_k. Construct the following polynomial:

$$q(x) = q_1 + q_2\, x + q_3\, x^2 + \ldots q_k\ x^{k-1} \ = \ \Sigma_{i=0}^{k-1}\, q_{i+1} x^i \qquad (2.2.1)$$

One can regard q(x) as a representation of the *state vector* of the shift register generator.

As shown in (1.4.1), the equation which determines the behavior of a register q_j in Figure 1.5 is

$$q_{j+1}(s+1) = q_j(s) - h_j\ o(s)$$

where the subscript denotes a register stage, and the argument s is a time index. We have changed this time index from n to s, because we have another use for symbol n below. We now simplify this as:

$$q'_{j+1} = q_j - h_j\ o \qquad (2.2.2)$$

where prime means one clock after no-prime. Thus, q(x) above records the register state before a clock, and q'(x) records the state after one clock,

$$q'(x) = q'_1 + q'_2\, x + q'_3\, x^2 + \ldots q'_k\ x^{k-1} \ = \ \Sigma_{i=0}^{k-1}\, q'_{i+1} x^i. \qquad (2.2.3)$$

If we insert (2.2.2) into (2.2.3) .

$$q'(x) = \ \Sigma_{i=0}^{k-1}\, q'_{i+1} x^i$$

$$= \Sigma_{i=0}^{k-1}\, [q_i - h_i\ o] x^i$$

$$= \Sigma_{i=0}^{k-1}\, q_i\, x^i - o\, \Sigma_{i=0}^{k-1}\, h_i\, x^i$$

$$= x \sum_{i=0}^{k-1} q_i x^{i-1} - o \sum_{i=0}^{k-1} h_i x^i$$

we arrive at this iteration equation for Figure 1.5:

$$q'(x) = x\, q(x) - o\, h(x) + i \qquad (2.2.4)$$

Here, o = output which is q_k/h_k, and i = input = q_0. See Figure 1.5. h(x) is the polynomial we called H(x) in Chapter 1, namely

$$h(x) = h_o + h_1 x + ... + h_k x^k \qquad (2.2.5)$$

The fact that q(x) and h(x) have a relative offset in their indices is just a quirk of the way we chose to label the registers in Figure 1.

Equation (2.2.4) describes how the state vector of a shift register generator moves in time.

A note on Symbols

In Section 1.2 we discussed the generality of our various circuits and the meaning of the *symbols* that flow around in these circuits. In the most general case, each register could hold one of $Q = p^m$ different symbols, as discussed in Example 1 and Example 2 of Section 1.2. This is the Q used in Section 2.1, and this Q is also applicable to the present section up to this point.

For the rest of this Chapter, we shall consider the simpler Example 3 situation where each register holds only Q = p different values, so the symbols are now elements of GF(p). This means, for example, that the coefficients of h(x) lie in GF(p). Eventually, we will only care about p=2.

We can generalize everything below to the fully general case $Q = p^m$ by making a global substitution of $p \rightarrow p^m$. This would be necessary, for example, if we were analyzing a Reed-Solomon style shift register generator. In this case, our Galois construction generates a higher order extension field from a lower order extension field.

This seems like an unnecessary level of complexity to maintain in all the following work, so we settle for symbols in GF(p).

Galois Field Review

Recall the Galois Field construction discussed at great length in our Galois notes. [We replace m with M in order to avoid confusion with the m used in the above note on symbols.]

$$GF(p^M) = Polys[x,GF(p)] / (f(x))$$

By way of review, here is what this says:

" If you are given *any* polynomial f(x) of degree M which is irreducible over GF(p) -- meaning that it cannot be factored -- , you can build a structure whose elements and operations + and • are equivalent to

the abstract Galois Field GF(p^M). The elements of this structure are the remainders of polynomials of any degree defined over GF(p) that one gets by dividing these polynomials by f(x). The ring of such polynomials of any degree over the field GF(p) is Polys[x,GF(p)], and then the set of remainders is denoted as this ring / (f(x)). The implied operations + and • are what naturally arise when you add or multiply polynomials and then take a remainder."

The usual notation in dealing with the above structure is to put the raw polynomials of Polys[x,GF(p)] inside curly brackets. Then the curly bracketed thing represents one of the remainders which is then regarded as an element of GF(p^M). The following line should be clearly understood:

$$\{x\ f(x)\} = \{f(x)\} = \{0\} = \mathbf{0}$$

Clearly, all three polynomials xf(x) , f(x) , and 0 have remainder 0 when divided by f(x). This element of GF(p^M) must be the additive identity 0. If you add f(x) or a multiple of f(x) to some other polynomial, you do not change the remainder.

This is of course just the beginning of Galois Theory and the reader is referred to earlier notes, especially in regard to things like minimum polynomials, primitive elements and polynomials, etc.

We shall replace f(x) with the h(x) of our shift register generator, and M with k to get:

$$GF(q = p^k) = Polys[x,GF(p)] / (h(x))$$

This implies that the degree of h(x) is k, so we must have some $h_k \neq 0$. If p>2 in GF(p), it is possible to have $h_k \neq 1$. In this case, one could divide through all the coefficients of h(x) and write

$$h(x) = h_k\, h'(x)$$

where $h'_k = 1$. If h(x) was irreducible, then so is h'(x). Since h'(x) has its highest degree coefficient equal to one, it is called a *monic* polynomial. From now on, we shall restrict our interest to monic h(x). Our real interest in doing this is that we can then identify h(x) with some minimum polynomial of an element of GF(q), since all minimum polynomials are monic. We use q as a shorthand for p^k. The following fact is a combination of Fact 3 of Galois Chapter 5, and the first Fact of Chapter 6:

Fact 3: Any monic h(x) of degree k that is irreducible over GF(p) is the minimum polynomial of the element {x} of GF(q=p^k).

Proof: We know that {h(x)} = 0 (see above for f(x)), and therefore h({x}) = 0. Thus, {x} is a root of h(x). Since h(x) is monic and irreducible, it must be (from Fact 3 of Chapter 5) a minimum polynomial of some element α of GF(q). But we have found a candidate element {x}. In general, a given minimum polynomial m(x) is the minimum polynomial for several elements of GF(q), namely, all elements of the conjugate set of α. This set can contain up to k elements.

Observation: The coefficient h_0 of h(x) must be non-zero. Otherwise you could factor out x from h(x) and then h(x) would be reducible. In terms of Figure 1.5, the implication is that some feedback must

always go *to* the leftmost register. And in Figure 1, some feedback must always come *from* the rightmost register. Thus, an implication of h(x) being irreducible is that all k registers are in the feedback loop.

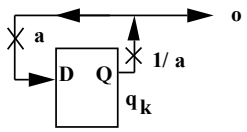
Reminder: In Chapter 4 is was found that the field GF(q) was cyclic and that therefore there exist certain elements called primitive elements in terms of which all q elements of GF(q) can be enumerated as follows:

$$GF(q) = \{ 0, 1, \alpha, \alpha^2, \alpha^3, \ldots\ldots \alpha^{q-2} \} \qquad \alpha^{q-1} = 1 \quad \alpha$$

In general, any non-zero element of GF(q) has an *order* -- a minimum integer n such that $\alpha^n = 1$. Primitive elements have order n = q-1. The order of any element must integrally divide q-1.

Fact 4: The order of {x} must exceed 1.

Proof: If n=1, then α = 1, so that α = {x} = {1}. But this is a contradiction, because {x}•{f(x)} = {xf(x)} ≠ {f(x)} for arbitrary f(x). One loophole is if k = 1, so that h(x) = ax+b and all remainders are constants, so {f(x)} = K. In this case only, it is possible to have {xK} = {K}. This means Rem(xK/h(x)) = K which in turn means that h(x) = ax-a. Here is a sketch of this fascinating special case of Figure 1.5:



We are not surprised to learn that the period of this circuit is 1. If we are willing to ignore this less than interesting case, we may conclude that n > 1.

The Figure 1.5 Galois Iterator

In light of the Galois field GF(p^k) discussion above, and the {..} notation so defined, we can write the above state vector iteration equation (2.2.4) as

$$\{q'(x)\} = \{x\}\bullet\{q(x)\} - \{o\}\bullet \{h(x)\} + \{i\}$$

{o} = o{1} and {i} = i{1} are multiples of the Galois • identity element {1} [why?]. Since {h(x)} = 0, we get

$$\{q'(x)\} = \{x\}\bullet \{q(x)\} + \{i\} \qquad (2.2.7)$$

To make a closer connection to the Galois fields, we now use Greek letters to denote field elements, as was done in our Galois notes. Define

$$\alpha = \{x\} \qquad \beta = \{q(x)\} \qquad \{i\} = i\{1\} = i\ 1 \qquad (2.2.8)$$

Then we get this Galois statement of what happens with a single clocking of the shift register:

$$\beta' = \alpha \bullet \beta + i1 = \alpha\,\beta + i\,1$$

Here, $\beta = \{q(x)\}$ represents the starting state vector of the register set. And i is the input bit at this time. We can now go back add back in our time s subscripts as follows:

$$\beta_{s+1} = \alpha\,\beta_s + i_s\,1 \qquad (2.2.9)$$

We are now able to study the progression of the state vector, starting at time s=0. After examining the first few terms, we quickly arrive at the general solution of our difference equation (2.2.9) for β_s :

$$\begin{aligned}
&\beta_1 = \alpha\,\beta_0 + i_0 \\
&\beta_2 = \alpha\,\beta_1 + i_1 \quad = \alpha^2\ \beta_0\ + [\,\alpha\ i_0\ + i_1\,]\,1 \\
&.... \\
&\beta_s = \alpha^s\,\beta_0\ +\ [\,\alpha^{s-1}\,i_0\ + \alpha^{s-2}\,i_1 + ... + \alpha\ i_{s-2}\ +\ i_{s-1}\ ]\,1
\end{aligned} \qquad (2.2.10)$$

This last equation is an exceedingly powerful result and says a lot about what is going on in our polynomial divider circuit of Figure 1.5. We shall refer to it as the Galois Iterator for Figure 1.5.

If there is no input, as in a "shift register generator" , the result simplifies:

$$\beta_s = \alpha^s\,\beta_0 \qquad (2.2.11)$$

Fact 5: The state vector of a (Figure 1.5) shift register generator steps periodically through some subset N of the non-zero elements of GF(p^k), provided it is initialized in some non-zero state. The state vector period N must integrally divide $p^k - 1$, so $N \le p^k - 1$. The integer N is the order of {x}.

Proof: If $\beta_0 = 0$, then $\beta_s = 0$ and the shift register has "died", so assume $\beta_0 \neq 0$. In this case, we know that $\alpha \neq 0$. The non-zero elements of GF(q) form a cyclic subgroup under •, therefore no product of non-zero elements can ever give the zero element. Thus, we can never have $\beta_s = 0$ if $\beta_0 \neq 0$.

We know that the element $\alpha = \{x\}$ has some order N which integrally divides $p^k - 1$. Thus, N is the smallest integer such that $\alpha^N = 1$. Therefore we can write,

$$\beta_{s+N} = \alpha^{s+N}\,\beta_0 = \alpha^s\,\beta_0 = \beta_s$$

which says that N is the period of the state vector.

Fact 6: If $p^k - 1$ happens to be a prime number, the state vector period is $p^k - 1$.

Proof: From Fact 5, the period n must evenly divide $p^k - 1$. In Fact 4 we have rejected N=1, therefore we conclude that $N = p^k - 1$.

Example: For GF(2^3) we have $p^k - 1 = 2^3 - 1 = 7$ which is a prime number. Thus, if you start a 3-register version of Figure 1.5 in any non-zero state, it will cycle through all 7 non-zero states. This is true regardless of what irreducible h(x) is selected.

Example: For GF(2^9) we have $2^9 - 1 = 511 = 7 \bullet 73$ = non-prime. In this case, it we start a 9-register version of Figure 1.5 in any non-zero state, it will cycle through some number N of states before returning to the initial state. There are only three possibilities : N = 7, N=73, N=511.

Comment: The preceding example shows the power of the Galois analysis. It would be extremely painful and time-consuming to obtain this kind of result by trial and error analysis.

Fact 7: If $\alpha = \{x\}$ is a *primitive element* of GF($q=p^k$), then the state vector period is q-1. In this case, the state vector "counts" through all q-1 non-zero elements of GF(q).

Proof: By definition, the powers of a primitive α enumerate all q-1 elements of { GF(q) - 0,•}, and $\alpha^{q-1} = 1$. Since α = primitive element is a generator of the cyclic group { GF(q) - 0,•}, we can represent β_0 as some power of α, say $\beta_0 = \alpha^r$. Then from $\beta_s = \alpha^s \beta_0$ we have:

$$\{ \beta_0, \beta_1, \beta_2, \beta_3, \beta_3, \ldots \beta_{q-2}\} = \{ \alpha^r, \alpha^{r+1}, \alpha^{r+2}, \ldots, \alpha^{q-2}, \alpha^{q-1} = 1, \alpha, \alpha^2, \ldots \alpha^{r-1} \}$$

There are q-1 elements in each set. All elements in the right set are different, so this is true of the left set as well. The next element on the left would be $\beta_{q-1} = \alpha^r = \beta_0$, so the period is q-1. In this case $\beta_s = \alpha^s \beta_0$ counts through all non-zero elements of GF(q) so the period β_s is then equal to the maximum set in Fact 5, n = q-1.

Fact 8: If h(x) is a *primitive polynomial* of GF($q=p^k$), then the state vector period is p^k-1.

Proof: If h(x) is a primitive polynomial of GF(q), then all its roots are primitive elements. Since {x} is a known root of h(x), {x} is then a primitive element, so the results follow from Fact 7.

An example : GF($q=2^9$)

Consider again GF($q=2^9$) = GF(512). If we select $h(x) = 1 + x^4 + x^9$ or $1 + x^5 + x^9$, these being the primitive polynomials having the fewest non-zero coefficients, then if we start our shift register off in any non-zero state β_0, it will not return to that state until 511 clocks have gone by. In other words, the state vector period is $N = q-1 = 2^9 - 1 = 511$.

We know that the number of primitive polynomials of a Galois Field is equal to the number of distinct conjugate sets which contain primitive elements, this was Fact 11 of Galois Chapter 5. Off hand, we do not know this number for GF(512), but it may be somewhat large, since there are 512 elements to deal with.

For assistance, we now peruse Appendix C of Peterson and Weldon. In this list, we count a total of 24 primitive polynomials for GF(2^9), and this does not include the reflected ones mentioned in Galois Chapter 5 Fact 12, so there are 48 primitive polynomials in all. Each set of conjugates containing a

primitive element contains m=9 distinct elements (Fact 8 of Chapter 5), so we conclude that there are a total of 9*48 = 432 primitive elements out of the total number of 512 elements of GF(2^9).

In Appendix C, the primitive polynomial with the *fewest* non-zero coefficients is listed first, let us call its corresponding primitive element a. The remaining primitive polynomials are then given, along with the smallest power of a which is one of their roots. The polynomials are given in an octal notation. Here are the first two table entries:

1 1021 h(x) = (x - a) ... = minimum polynomial containing a
1021 = 001,000,010,001 $\Rightarrow$ $h(x) = 1 + x^4 + x^9$
reflected version is: $h(x) = 1 + x^5 + x^9$

3 1131 $h(x) = (x - a^3)$ ... = minimum polynomial containing a^3
1131 = 001,001,011,001 $\Rightarrow$ $h(x) = 1 + x^3 + x^4 + x^6 + x^9$
reflected version is: $h(x) = 1 + x^3 + x^5 + x^6 + x^9$

We are obviously interested in the h(x) with the fewest non-zero coefficients since this minimizes the number of adders needed in Figure 1.5 (or Figure 1). So we shall select $h(x) = 1 + x^4 + x^9$, which means we are choosing our $\alpha = \{x\} = a$.

The non-terminating Remainder in polynomial division.

In Chapter 1 we discussed the use of either Figure 1.1 or Figure 1.5 as a polynomial divider. Both circuits perform the same function. In Figure 1, the remainder is "hidden", but in Figure 1.5 it is exposed: the state vector always holds the "current remainder". In equation (1.3.9) we wrote this as:

$$I_r(z) / H(z) = O_{r-k}(z) + R^{(0)}_{k-1}(z)/H(z)$$

To review, $I_r(z)$ was an Input polynomial of degree r, H(z) was the divisor polynomial of degree k, and $O_{r-k}(z)$ was the Output quotient polynomial of degree r-k. The Remainder $R^{(0)}_{k-1}(z)$ is always of degree k-1 . If some of its leading coefficients happen to vanish, it is of degree less than k-1.

In Chapter 1 we used upper-case polynomial names to stress the notion of Z Transform, and for the same reason we used the variable z. We were also trying to show explicitly how the degrees of all the polynomials come out right.

Here we shall ignore all this baggage and rewrite the above in the simple form

$$i_0(x)/ h(x)= o_0(x) + r_0(x)/h(x)$$

We have not changed anything, we are just simplifying the notation. The 0 subscript is a relative time index. We think now of time t=0 just after the last non-zero symbol of the input stream has been shifted in to the divider. If the input stream is always zero from this point on, we can think of the circuit as the *shift register generator* discussed above which has its register state initialized with the remainder appearing in

the above equation, $r_0(x)$. That is, the coefficients of this polynomial are the starting contents of the k registers. After one clock, we have:

$$i_1(x) / h(x) = [\ x\ i_0(x)\ ] / h(x) = o_1(x) + r_1(x)/h(x)$$

The input polynomial $i_1(x) = x\ i_0(x)$, because we have added a trailing 0 to the input symbol stream. After s clocks we would have

$$i_s(x) / h(x) = [\ x^s\ i_0(x)\ ] / h(x) = o_s(x) + r_s(x)/h(x)$$

If at some point the remainder vanished, the subsequent quotient symbols would be zeros. Conversely, if the remainder were never to vanish, the quotient symbol stream would go on forever.

Fact 9: If h(x) of degree k is irreducible over GF(p), and if the first remainder

$$r_0(x) = \text{Rem}[\ i_0(x) / h(x)\ ]$$

is non-zero, then all subsequent remainders

$$r_s(x) = \text{Rem}[\ x^s\ i_0(x) / h(x)\ ]$$

are non-zero, and the quotient sequence never terminates. Moreover, the sequence of remainders has period N which is the state vector period discussed above. This period N divides evenly into $p^k - 1$. Furthermore, if h(x) is a primitive polynomial of GF(p^k), then the state vector period is exactly $p^k - 1$.

Proof: Once we identify the remainder with the shift register state, everything follows directly from the Facts given above.

Corollary: We have therefore answered Question 5 posed in Appendix 1.1

> **Question 5**: How do we know that, in polynomial division over some field F, it is possible that the quotient may never terminate?

We have seen that, not only is it possible, but for irreducible h(x), it is very likely.

Fact 10: The *state vector period* of a Figure 1.5 shift register generator is the same as the *period* of h(x) .

Proof: The *period* of a monic irreducible h(x) was defined rather mysteriously in Chapter 5 as being the smallest value of n such that h(x) divides evenly into $x^n - 1$. Let us assume that h(x) has *period* n, so we can then write $(x^n - 1)/h(x) = g(x)$, where g(x) is whatever quotient polynomial we get by this division. Then consider the remainder sequence noted above, for time s clocks after the input polynomial coefficients resume being 0,

$$r_s(x) = \text{Rem}[\ x^s\ i_0(x) / h(x)\ ]$$

Then we can evaluate

$$r_{s+n}(x) - r_s(x) = \text{Rem}[\,(x^n - 1)x^s\, i_0(x) / h(x)\,] = \text{Rem}[\,x^s\, g(x)i_0(x)] = 0,$$

This shows that $r_{s+n}(x) = r_s(x)$, which says that the state vector period is n. Recall that the *period* n refers to the smallest power n such that $(x^n - 1)/h(x) = g(x)$. Similarly, the *state vector period* is the smallest integer n that makes the register state (= remainder) repeat. Thus, we see that these two entities are one in the same.

Summary: This section has focused on the state vector as being an element of a Galois Field GF(p^k). For the Figure 1.5 circuit, this state vector can be identified with the remainder of polynomial division. We learned that the state vector period N of a Figure 1.5 style shift register generator is determined by the nature of h(x).

We did not in this section have much to say about any particular register. For example, the output symbol sequence is basically the contents of the rightmost register in Figure 1.5. One wonders what the periodicity of the contents of a particular register might be, and thus, one wonders about the periodicity of the output data stream of a shift register generator.

For the moment, we settle for the following:

Fact 11: The period of any particular register of the Figure 1.5 circuit must integrally divide the state vector period N.

Proof: Certainly the period of one register cannot exceed the period of the register set as a whole. If the whole register state repeats with period N, then so must any one register. However, it is easy to imagine that a particular register might have a smaller period. To be compatible with the state vector period, the state vector period would have to consist of an integral multiple of these smaller periods.

Corollary: The *output period* of a Figure 1.5 shift register generator must be a divisor of the state vector period N.

In the next section, we shall find a much stronger statement about the output period.

2.3 The Output Sequence of a Shift Register Generator

In the previous section, we examined the behavior in time of a *state vector* representing the contents of the k registers of a Figure 1.5 shift register generator.

In this section, we concentrate on the *output sequence* of a shift register generator, rather than on the state vector of its register set.

In contrast with the state vector situation, the output sequences of both types of shift register generators -- Figure 1.1 style and Figure 1.5 style --obey the *same* fundamental equation.

The Figure 1.1 state vector behavior is then obtained by considering a grouping of k consecutive output symbols.

The fundamental equation of interest here is the time domain description of the polynomial divider which was derived in Section 1.8 and appears there in a summary box,

$$i_n = \sum_{j=0}^{k} h_j \; o_{n+j} \qquad n = \text{any integer} \tag{2.3.1}$$

Recall that this is the time-domain view of polynomial division O(z) = H(z)/I(z).

For the scrambler style divider circuit of Figure 1, we can identify the output stream o with the contents of the rightmost register q_0. If we set the input sequence to zero, $i_n = 0$, we have the equation which governs a shift register generator:

$$\sum_{j=0}^{k} h_j \; o_{n+j} = 0 \qquad n = \text{any integer} \tag{2.3.2}$$

If we separate out the highest term and put it on the left, and set time index n=0, we get:

$$h_k \, o_k = -h_0 \, o_0 - h_1 \, o_1 - \ldots - h_{k-1} \, o_{k-1} \tag{2.3.3}$$

The reader should stare at Figure 1.1 to see how it directly implements the above equation. The output at time k is determined entirely by the previous k outputs, which are stored in the registers of Figure 1.

The above equation is a difference equation. If we solve it iteratively, as we have done with other equations several times earlier, we end up explaining once again how polynomial division works, and we end up again with Figure 6, with symbols explained in Figure 3. The dividend polynomial in this case takes its leading coefficients from the starting contents of the shift register generator.

Our purpose here is not to solve the difference equation, but to make some fairly astounding observations about the properties that solutions of the difference equation have.

Fact 1: There are exactly p^k solutions to the difference equation (2.3.2). One of these solutions is all zeros, so there are $p^k - 1$ non-zero solutions.

Proof: By "solution" we mean an infinite sequence

$$\text{solution} = \{o_j\} = \{ o_0, o_1, o_2, \ldots\ldots o_{512,453} \ldots\ldots \}$$

Consider Eq. (2.3.3) above. This shows that o_k is fully determined by the numbers $\{ o_{k-1}, \ldots o_0 \}$. If we iterate, we find that o_{k+1} is then determined by the set of numbers $\{ o_k, \ldots o_1 \}$. But we just noted that o_k is determined by $\{ o_{k-1}, \ldots o_0 \}$, therefore o_{k+1} is determined by $\{ o_{k-1}, \ldots o_0 \}$. Similary, all subsequent o_{k+j} are determined by $\{ o_{k-1}, \ldots o_0 \}$. In other words, the entire solution is completely determined by the starting set $\{ o_{k-1}, \ldots o_0 \}$. Thus, the number of solutions is equal to the number of starting sets. Since each symbol o_i is a number in GF(p), it can take p values. Thus, there are p^k starting

sets, and hence there are p^k solutions. No two of these solutions can be the same because they had different starting sets, and the starting set is part of the solution sequence.

Fact 2: The p^k solutions of the difference equation can be regarded as elements of a k-dimensional vector space V^k. The k "basis vectors" in this space can be regarded as the k solutions which have the starting sets { 10000..}, (01000..}, {00100..}, etc.

Proof: Since the difference equation is linear, any multiple of a solution or any sum of solutions is also a solution. Thus, the solutions form a vector space. Since we have found k linearly independent basis vectors which span all solutions, the dimension of the space is k. Note that any solution can be written as a sum of the basis vector solutions simply because any starting set can be written as a linear combination of the basis vector starting sets.

Assumption : Assume for the moment that we can find an integer n such that h(x) of degree k divides evenly into x^n- 1. It is not necessary that n be the smallest such integer (the so-called *period* of h(x)). Also, we put off any discussion as to whether or not an n always exists for an arbitrary h(x). For those h(x) that will be of interest to us, we will *know* that n exists.

Definition: Assuming that n exists, we define g(x) to be the quotient: $g(x) = (x^n - 1)/h(x)$.

Aside: If h(x) is of degree k *and* is the minimum polynomial of an element of GF(q=p^k), then we know that q-1 = p^k - 1 is a workable candidate for the n mentioned in the above Assumption. See Galois Chapter 5. Any monic h(x) of degree k that is irreducible in GF(p) is such a minimum polynomial, so for any such h(x) our candidate n is n = p^k - 1.

Fact 3: Since we then have $g(x)h(x) = x^n - 1$, we can consider g(x) to be the generator of a *cyclic code*. All coefficients are assumed to lie in field GF(p).

The subject of cyclic codes was elaborated in Galois Chapter 7. We summarize some basic facts:

(a) h(x) is of degree k, and g(x) is of degree n-k, and $g(x)h(x) = x^n - 1$.

(b) Code polynomials are formed as c(x) = d(x)g(x), where the d(x) have degree k-1 and are called data polynomials. Thus, code polynomials have degree n-1, and so have n coefficients in GF(p).

(c) Since the d(x) have k coefficients in GF(p), there are p^k possible data polynomials. Therefore, there are p^k possible code words.

(d) All cyclic permutations of a code word are code words.

Since the coefficients of a code polynomial lie in GF(p), we can think of an n-dimensional vector or n-tuple consisting of the coefficients of c(x). We write this as $c = \{c_0, c_1, ...c_n\}$, and we refer to it as a code word. Sometimes we will loosely refer to c(x) as a code word, but we really mean this vector c.

We now make the following unusual claim:

Fact 4: If c is any code word of the cyclic code generated by g(x), then the infinite sequence

$$\{o_i\} = \{c,c,c,c \ldots\}$$

is a solution of the difference equation (2.3.2).

Proof: See Appendix 2.1. This is the crucial fact, so naturally the proof is more than two lines. It might be good to look at this proof to make sure the above notation is understood.

Fact 5: All p^k solutions of the difference equation (2.3.2) are of the form $\{o_i\} = \{c,c,c,c \ldots\}$.

Proof: In Fact 4 we found a solution of (2.3.2) for each code word c. We know from cyclic code theory that there are p^k such code words, so we have found p^k solutions of (2.3.2). According to Fact 1, (2.3.2) has exactly p^k solutions, so we have found them all, and they are therefore all of the form claimed.

Fact 6: Any solution of (2.3.2) has a period which evenly divides n. We do not rule out the possibility here that different solutions might have different periods which divide n.

Proof: Since $\{o_i\} = \{c,c,c,c \ldots\}$, we know that all solutions have period n. If, for some particular solution, it happens that the symbols of the code word c are periodic with some period which divides n, then of course that solution $\{o_i\}$ will have this smaller period.

An example : GF(16)

According to the Peterson and Weldon Appendix C, here are some candidate h(x) polynomials, and α is a primitive element.

1 23 $h(x) = (x - \alpha)(x - \alpha^2)(x - \alpha^4)(x - \alpha^8)$ order 15

$23 = 010,011 \quad \Rightarrow \quad h(x) = 1 + x + x^4$

3 37 $h(x) = (x - \alpha^3)(x - \alpha^6)(x - \alpha^{12})(x - \alpha^9)$ order 5

$37 = 011,111 \quad \Rightarrow \quad h(x) = 1 + x + x^2 + x^3 + x^4$

5 07 $h(x) = (x - \alpha^5)(x - \alpha^{10})$ order 3

$07 = 000,111 \quad \Rightarrow \quad h(x) = 1 + x + x^2$

We have enumerated some of the minimum polynomials. For each one, we indicate the order of the field elements that go with that polynomial.

Consider $h(x) = 1 + x + x^2 + x^3 + x^4$ which is of degree 4, and which corresponds to a field element α^3 which is blatantly *not* a primitive element (since its order is 5 and not 15). Thus, this h(x) is *not* a primitive polynomial.

We know that h(x) divides into $x^{15} - 1$ to give g(x). We would like to find this g(x). Consider,

$h(x) = 1 + x + x^2 + x^3 + x^4 = (x^5 - 1) / (x-1)$

$g(x) = (x^{15} - 1) / h(x) = (x-1) [(x^{15} - 1) / (x^5 - 1)] = (x-1) (x^{10} + x^5 + 1)$

$g(x) = x^{11} + x^{10} + x^6 + x^5 + x + 1$

Now we can write down some of the code words of g(x). Our first one is g(x) itself, and then we do all cyclic permutations. Notice for this g(x) code word we start at the right and then fill with three zeros on the left to get it from n-k to n coefficients. Please do not miss the point here. Code words are multiplies of g(x), so the unit multiple is one of the code words.

{ 0,0,0,1,1, 0,0,0,1,1, 0,0,0,1,1}

We have artificially introduced gaps to reveal the repeating pattern that results. We can do cyclic permutations on the above to get 5 of the 16 codes words.

{ 0,0,0,1,1, 0,0,0,1,1, 0,0,0,1,1}
{ 0,0,1,1,0, 0,0,1,1,0, 0,0,1,1,0}
{ 0,1,1,0,0, 0,1,1,0,0, 0,1,1,0,0}
{ 1,1,0,0,0, 1,1,0,0,0, 1,1,0,0,0}
{ 1,0,0,0,1, 1,0,0,0,1, 1,0,0,0,1}

To get the other code words, add two of the above and then take its 5 cyclic forms, and then add another two and get its 5 cyclic forms. The 16th code word is all zeros.

Now, construct a shift register generator with 4 registers and use $h(x) = 1 + x + x^2 + x^3 + x^4$. We know that all output sequences are of the form $\{o_i\} = \{c,c,c,c ...\}$. Looking at the above code words, we quickly conclude that the period of all possible non-zero output streams is 5 !

This result is consistent with Fact 5 of Section 2.1. There, we showed that the state vector period of the shift register was equal to the order of the Galois element {x} which is a root of h(x). For our h(x) here, $\{x\} = \alpha^3$ which has order 5. Since the state vector repeats every 5 clocks, it is not too surprising that the output sequence has this same period.

• end of example

Fact 7: If the integer n happens to be the *period* of h(x) , then at least n solutions of the difference equation (2.3.2) have period n.

Proof: We already know from Fact 6 that the period of any solution must divide n. The question here is whether there are some solutions which have the full period n.

Consider the code word c(x) obtained by multiplying the generator g(x) by unity. We claim that this c has the full period n. To show this, assume first that it does not, and that it has some smaller period m = n/N (as happened in our example above). In this case we must be able to group the elements of c into N

identical groups of m symbols. (In the example we had 3 identical groups of 5 symbols) . In this case we can write:

$$c(x) = g(x) = f(x) [\ 1 + x^{m} + x^{2m} + x^{3m} + ... + x^{(N-1)m}]$$

where f(x) is a polynomial of degree m representing the pattern that repeats N times. We recognize the above [...] as $(x^{n} - 1) / (x^{m} - 1)$ where n = Mm. And of course $(x^{n} - 1) = g(x)h(x)$ as usual. Thus we have

$$g(x) = f(x) (x^{n} - 1) / (x^{m} - 1) = f(x)g(x)h(x) / (x^{m} - 1)$$

Cancelling g(x) we find that $f(x)h(x) = (x^{m} - 1)$ where m<n . But by definition, assuming that the *period* of h(x) is n means that there is no such m < n such that h(x) divides $(x^{m} - 1)$. Thus we arrive at a contradiction, so it must be that this c(x) = g(x) has the full period n, and therefore the corresponding solution to the difference equation (2.3.2) must have the full period n as well.

Furthermore, we know we can form n-1 distinct other code words by taking cyclic permutations of this g(x) code word, and all of these code words will have period n as well.

Fact 8: If h(x) of degree k is a *primitive polynomial* of GF(p^{k}), then *all* p^{k}- 1 non-zero solutions of the difference equation (2.3.2) have period $n = p^{k} - 1$.

Proof: This is the big result we have been waiting for. The proof is so simple that it is deceptive. From Galois Chapter 5 Fact 4, we know that if h(x) is a primitive polynomial of GF(p^{k}), then h(x) has *period* n = p^{k}-1. From Fact 7, we know that in this case we can produce n solutions of the difference equation (2.3.2) which have the full period n -- these are just the n cyclic permutations of the code word g(x) (filled out to have n coefficients) .

On the other hand, from the discussion following Fact 3, we know that the cyclic code generated by g(x) has exactly p^{k} code words. One of these is the zero code word, so this leaves p^{k} - 1 non-zero code words. But this is our number n. We know from Fact 5 that every solution of (2.3.2) corresponds to a code word. Since there are n non-zero code words, there are only n non-zero solutions, and we have found them all, and they all have period n.

Fact 9: If h(x) of degree k is a *primitive polynomial* of GF(p^{k}), then there is essentially only *one* characteristic non-zero output pattern of the shift register generator, and its period is p^{k} - 1. This pattern is often called a *maximum length sequence*, but we will just call it the *characteristic sequence*.

Proof: Pick any of the p^{k}- 1 non-zero solutions mentioned in Fact 8. Write it down. Then all the other solutions are sub-sequences obtained by doing delayed starts on the original solution. For example,

```
Selected solution =     abcdefgabcdefgabcdefg.............
   another solution =     bcdefgabcdefgabcdefga.............
   another solution =      cdefgabcdefgabcdefgab............
   another solution =       defgabcdefgabcdefgabc...........
   another solution =        efgabcdefgabcdefgabcd..........
   another solution =         fgabcdefgabcdefgabcde.........
   another solution =          gabcdefgabcdefgabcdef........
```

In this sketch, we have k = 3, and the period of all (non-zero) solutions is $2^3 - 1 = 7$. The 7 solutions are all listed. The 8th solution is the boring one that is all zeros.

Example: We know that $h(x) = 1 + x^4 + x^9$ is a primitive polynomial of $GF(2^9) = GF(512)$. If we implement a shift register generator with nine registers using this h(x), we know that there is one characteristic output sequence and it has period $n = 2^9 - 1 = 511$ bits.

Moreover, we know exactly what this sequence looks like: divide h(x) into $x^{511} - 1$ to get g(x) which has degree n-k = 511-9 = 502. The output pattern is then the 503 coefficients of g(x) with a filler of 8 zeros on the end to make the full characteristic pattern of 511 bits.

This suggests the following fact;

In Section 2.1 we found that in the Figure 1.5 type shift register generator, if h(x) is a primitive polynomial, then the register state vector cycles through all possible non-zero elements of GF(q), so the *state vector period* was then $q-1 = p^k - 1$. We concluded that the pattern of any particular register must have a period that divides evenly into q-1. It seemed quite possible that some particular register might have some smaller period than q-1.

What we have learned is that, for the Figure 1.5 circuit, the rightmost register must have the full period n = q-1. This is because it generates the output sequence directly ($h_k = 1$ since h(x) monic) .

With regard to Figure 1, we know that *all* registers have period n = q-1 ! This is because all registers contain the output sequence, we just have a time delay between them. We have thus proved:

Fact 10: For a Figure 1.1 shift register generator, the *state vector period* is the same as the *output period*.

Corollary: If h(x) is a primitive polynomial of $GF(p^k)$, then the state vector period of a Figure 1.1 shift register generator is the maximal period $p^k - 1$.

Here is a drawing to illustrate this fact, which one can relate to the previous sketch above:

```
output sequence=      abcdefgabcdefgabcdefg......
state vectors =       abc
(Figure 1.1 circuit)   bcd
                     cde
                      def
                       efg
                        fga
                         gab
                          abc
```

There is not much distinction between the state vector sequence and the output sequence. In the following discussion, we will refer to the state vectors as being obtained by "sifting through" the output sequence, one symbol at a time. What we mean by this inexact language is what is illustrated in the figure above. In the next section, we shall explore the nature of a characteristic sequence

2.4. Characteristic Sequences

In the previous section, we considered a shift register generator having k registers, having symbols in GF(p), and having for its h(x) a primitive polynomial of GF(p^k). Such a shift register generator has essentially one output sequence known as a characteristic or maximum-length sequence. The circuit can be implemented as either Figure 1.1 or Figure 1.5.

Here we shall see what we can learn about such a characteristic sequence.

Observation: Different primitive polynomials h(x) of degree k are likely to produce different characteristic sequences. We do not want to leave the impression that there is only one such sequence. In our Section 2.2 example of GF(512), we noted that there were 48 distinct primitive polynomials. One usually selects one of the two possible primitive polynomials which have the minimum number of non-zero coefficients.

Fact 1: If one were to sift through one period of a characteristic sequence, moving one symbol position at a time, one would find each k-symbol *state vector* once and only once. In other words, each possible state vector occurs exactly once somewhere in the characteristic sequence.

Aside: What we do not know is the exact order in which these state vectors occur.

Proof: Fact 1 was demonstrated at the end of the previous section. If the same state vector were found twice within one period of the output sequence, then the period would equal the difference between the two findings of the same state vector, which is a contradiction. The last figure in the previous section exactly demonstrates this fact.

We now add another fact which we basically proved already:

Fact 2: In the characteristic sequence of a primitive h(x) shift register generator, the maximum number of consecutive zeros is k-1, and this sequence occurs only once per period.

Proof: We can think of the characteristic sequence as belonging to code word g(x) which has n-k+1 coefficients. The total period is n, so we have to fill with n - (n-k+1) = k-1 zeros. This pattern must occur once per period. It cannot occur twice, because if it did, and we including an adjacent 1, we would have a k-bit state vector pattern repeating twice per period, which we showed in Fact 11 is impossible.

By the way, it should be obvious that we could never have a string of k sequential zeros. This would kill the sequence altogether.

Fact 3: The maximum number of consecutive identical non-zero symbols in the characteristic output sequence is k, and this pattern occurs once per period.

Proof: Since every non-zero state vector must be included once in the characteristic sequence (Fact 11), the vector consisting of k identical symbols for any non-zero symbol in GF(p) [such as 1] must occur once. If there were a string of k+1 identical symbols, then the vector of k identicals would appear twice, but Fact 11 says this cannot happen.

Fact 4: In a characteristic sequence, every non-zero symbol appears exactly (p^{k-1}) times, and the 0 symbol appears exactly (p^{k-1} - 1) times. As a check on these numbers ,

$$(p^{k-1})\,(p-1) + (p^{k-1} - 1) = p^k - 1 = \text{total symbols in characteristic sequence}$$

Proof: Take all p^k state vectors and lay them side by side in a huge set which then has kp^k symbols. Since each symbol can take p values, and since all symbols are treated equally in this full enumeration of the state vectors, in this huge set there must be (kp^k) / p occurrences of each symbol. Thus, each symbol appears kp^{k-1} times.

Now form another set which includes all state vectors except the all-zero vector. All non-zero symbols occur in this set kp^{k-1} times, just as in the last set, because we did not eliminate any of them. However, we removed k occurrences of the symbol 0, so it occurs kp^{k-1} - k times in this new set.

Consider once again the "sifting" process where you walk through the entire characteristic sequence looking at one vector's worth of symbols (k-symbols) at a time, but you shift only one symbol on each step of this walk. During this "walk" you encounter precisely all the symbols of our second set described above. Ie, we encounter all non-zero vectors. On the other hand, each symbol of the characteristic sequence is repeated k times by this process.

Therefore, each non-zero symbol appears in the characteristic sequence exactly (kp^{k-1})/k times, and the zero symbol appears (kp^{k-1} - k)/k times.

Note on the minimum weight and distance of the cyclic code generated by g(x).

We have seen that any period's worth of a characteristic sequence can be regarded as a code word of the code generated by g(x), where h(x)g(x) = x^n - 1 with n = p^k - 1. The entire cyclic code consists of cyclic permutations of this code word, plus the zero code word. Thus, all non-zero code words have the same *weight*, which is the number of non-zero elements. From Fact 5 above, this weight is $(p-1)p^{k-1}$. Thus, this is also the minimum weight of the code, which we know is the code *distance* as described in Galois Chapter 7.

Fact 5: If we line up our characteristic sequence with any delayed version of itself and compare over one period, we will find that the two sequences differ in exactly $(p-1)p^{k-1}$ places.

Proof: A characteristic sequence and a delayed version of itself over one period may be regarded as two code words of the cyclic code generated by g(x), as discussed in the above paragraph. Assume that two code words differ in exactly N places. When you subtract these two code words, you then get a result which has N non-zero places. But this result is also a code word, and it has weight N. But we know this weight must be $(p-1)p^{k-1}$, as discussed above. Q.E.D.

Binary Characteristic Sequences (p=2).

First, we restate all the facts derived in Section 2.3, but specialized to the case p=2.

Fact 1': If one were to sift through one period of a binary characteristic sequence, moving one bit position at a time, one would find each k-bit *state vector* once and only once. In other words, each possible state vector occurs exactly once somewhere in the characteristic sequence.

Fact 2': In the binary characteristic sequence of a primitive h(x) shift register generator, the maximum number of consecutive zeros is k-1, and this sequence occurs only once per period.

Fact 3': The maximum number of consecutive 1's in a binary characteristic output sequence is k, and this pattern occurs once per period.

Fact 4': In a binary characteristic sequence , 1 appears exactly 2^{k-1} times, and 0 appears exactly $2^{k-1} - 1$ times.

Fact 5': If we line up a binary characteristic sequence with any delayed version of itself and compare over one period, we will find that the two sequences differ in exactly 2^{k-1} places.

Now we add some new facts which apply only to binary characteristic sequences:

Fact 6: A string of (k-1) 1's appears exactly once in a binary characteristic sequence.

Proof: The state vector 01111..11 must appear somewhere, having (k-1) 1's. If the thing on the right of it begins with a 0, then we have found our string of (k-1) 1's, and we have also located the state vector 1111..110. If the thing on the right begins with a 1, then we have found the single allowed occurrence of k 1's (see Fact 3'). In this case we look for the state vector 11111...110 . Then this must have a 0 on the left to avoid a second occurrence of k 1's. Then this is our occurrence of (k-1) 1's.

Example: Consider again GF(512). The characteristic sequence has length 511, as noted above. The number of 1's in this sequence is 256, the number of 0's is 255. The largest string of consecutive 0's is 8. The largest string of consecutive 1's is 9, and there is another string of 8 1's.

The product of two shifted binary characteristic sequences.

We have already discussed the *difference* of two such sequences, which is the same as the sum in the binary world, see Fact 5 of the previous section. Here we wish to examine the *product* of two shifted sequences. Our motivation is that such a product is involved in the autocorrelation function of a sequence, and this in turn is related to the spectral density of a sequence.

Consider our two shifted sequences as adjacent column vectors, each containing $n = 2^k - 1$ components. Of course each component is either a 1 or a 0. To the right imagine a third column vector which will be the sum of the two sequences, and a fourth column which is the product. We know that the first two columns are two code words of the cyclic code generated by g(x), and we know these code words have weight 2^{k-1}, this is Fact 4'. Since the third column (the sum) is also a code word, it too has this same weight.

As we scan down the left two column vectors, there are four possible combinations we can get at each row: (0,0), (0,1), (1,0), and (1,1). Let us assign the following counts to these possibilities. In each case, we note the sum and product that will appear in the third and fourth column.

The Counts	Column 3	Column 4	Shifted n's	Unshifted n's
n_1 = number of (0,0)'s	sum = 0	product = 0	$(2^k - 1) - 3 \cdot 2^{k-2}$	$2^{k-1} - 1$
n_2 = number of (0,1)'s	sum = 1	product = 0	2^{k-2}	0
n_3 = number of (1,0)'s	sum = 1	product = 0	2^{k-2}	0
n_4 = number of (1,1)'s	sum = 0	product = 1	2^{k-2}	2^{k-1}

It is easy to calculate these four counts. Let $w = 2^{k-1}$ = the weight of either code word. Then,

$n_2 + n_4 = w$ = number of 1's in second column
$n_3 + n_4 = w$ = number of 1's in first column
$n_2 + n_3 = w$ = number of 1's in third column

These three equations tell us that $n_2 = n_3 = n_4 = w/2 = 2^{k-2}$. The total number of all combinations is equal to the number of rows, so

$$n_1 + n_2 + n_3 + n_4 = n = 2^k - 1$$

We solve this for n_1, using the previous results for the other n's, to get $n_1 = (2^k - 1) - 3 \cdot 2^{k-2}$. We have added these solved-for counts as the Shifted n's in the above table.

Now look back at the above discussion. We assumed that the two starting sequences were *shifted* by some non-zero amount. More specifically, the sequences had to be shifted by an amount not equal to a multiple of the period $2^k - 1$ of the characteristic sequence. If this were not true, then the sum code word (column 3) would be all zeros, and it's weight would then not be w, as we assumed above. In this case the above counts are wrong, but we know what the right ones are. They appear in the Unshifted n's column of the table, and they just follow from Fact 4'.

We have now proved the following:

Fact 7: Exactly 2^{k-2} ones appear in (one period's worth of) the product of two shifted binary characteristic sequences. If the sequences are unshifted, or are shifted by a multiple of the period, then exactly 2^{k-1} ones appear in the product.

Probability of strings of 1's and 0's.

In the next section we shall argue that, for reasonably large k, any characteristic sequence is very close to being a so-called white sequence. For such a totally random sequence, we can arrive at certain conclusions which we then apply to the characteristic sequence. One of these has to do with the probability of the occurrence of strings of repeated symbols.

Obviously the characteristic sequence is not perfectly white in that it never has more than k ones in a row, and never more than k-1 zeros in a row. These facts were proved above. So we take the following results as reasonable estimates, not exact facts. We shall estimate the error as well.

Consider an operating shift register generator. We let it run for a long time and we store all of the output data. Then later on, we scan through the data and count the strings of 1's of various lengths, and we can then compute the relative probability of each length. In doing this scan of the data, we run our finger along the data until we find a 0-1 transition, then we count the number of 1's in the string.

With the "white" assumption above, we can estimate the relative probability of the occurrence of 1-strings of various lengths. Let p = probability of a 1, and q = probability of a 0. From Fact 4':

$$p = 2^{k-1}/(2^k - 1) \qquad q = (2^{k-1} - 1)/(2^k - 1) \qquad p + q = 1 \tag{2.4.1}$$

For k = 9, p = 256/511 and q=255/511. In general, $p \approx q \approx 1/2$.

Here is a picture outlining the possibilities for 1-strings of lengths 1, 2 and 3. The vertical bar indicates the 0-1 transition in the data stream. On the right we indicate the relative probability of each situation. This probability is conditioned on the fact that we found a 0-1 transition:

...	x	x	0	1	0	x	x	...	q
...	x	x	0	1	1	0	x	...	pq
...	x	x	0	1	1	1	0	...	ppq

In general, the probability of our finding a string of 1's of length n is:

$$P(n) = q\,p^{n-1} = (q/p)\,p^n \tag{2.4.2}$$

Roughly setting p = q = 1/2 , we get

$$P(n) \approx 1/(2^n) \qquad (\approx \text{ for shift register generator, } = \text{ for white sequence})$$

Thus, the probability of finding strings of lengths 1 through 10 is

P(1) = 1/2	P(6) = 1/64
P(2) = 1/4	P(7) = 1/128
P(3) = 1/8	P(8) = 1/256
P(4) = 1/16	P(9) = 1/512
P(5) = 1/32	P(10) = 1/1024

We would now like to sum these probabilities to see how much in error our estimate is for a characteristic sequence. First consider the elementary series sum

$$\sum_{n=1}^{m} p^n = \frac{p - p^{m+1}}{1-p} = (p/q)\,(1 - p^m)$$

Now consider the sum of our 1-string probabilities, and make use of the above result:

$$\text{total probability of all 1-strings} = \sum_{n=1}^{m} P(n) = (q/p) \sum_{n=1}^{m} p^n = 1 - p^m$$

For a true random sequence of any $p < 1$, we would add up all terms to $m = \infty$ and the total probability is 1. In particular, for a white sequence having $p = 1/2$, this is true.

For a characteristic sequence, the longest possible string of 1's is k, see Fact 3'. Thus, we must truncate the sum at k, so we find:

$$\text{total probability of all 1-strings for char sequence} = 1 - p^k \qquad (2.4.3)$$

Since $p \approx 1/2$, the error is $(1/2)^k$. For $k = 9$, this is $1/512 \approx 0.2\%$. Because there are no strings of ones longer than k, we have to raise the probability of all smaller strings by about 0.2%. This then gives an idea of the error involved in comparing a characteristic sequence to a white sequence.

As for strings of *zeros*, we can make an identical argument, except p and q are reversed. We get,

$$P(n) = p\, q^{n-1} = (p/q)\, q^n \approx 1/(2^n)$$

In this case, the longest allowed string is k-1 zeros, so our total probability sum stops one short of the 1's case. Adding up the total probability, we get:

$$\text{total probability of all 0-strings for char sequence} = 1 - q^{k-1} \qquad (2.4.4)$$

Since $q \approx 1/2$, the error is $(1/2)^{k-1}$. For $k = 9$, this is $1/256 \approx 0.4\%$. So here we have to raise all our estimates of 0-string lengths by about 0.4%.

We can summarize the above in the following fact:

Fact 8: In a characteristic sequence of reasonably large k, the distribution of string lengths is very close to the theoretical white sequence distribution, up to a length where strings are no longer allowed to exist. Up to this length, the relative probability of a string of exactly n identical symbols is given by $P(n) \approx 1/2^n$. For $k = 9$, the error in this estimate is $< 0.5\%$.

2.5 Autocorrelation, Probability, and Statistics for a sequence in GF(2)

The autocorrelation function for a continuous signal a(t) was defined in Spectral (6.1.1), where we omitted any kind of normalizing factor,

$$r(t) \equiv \int_{-\infty}^{\infty} dt'\, a(t')\, a(t'+t)$$

Although we did not use this terminology in Spectral Chapter 6, it should be clear that we can define an exactly corresponding "digital" autocorrelation function having the following form:

$$r_k \equiv \sum_{n=-\infty}^{\infty} a_n\, a_{n+k} \qquad (2.5.1)$$

At this point, we still imagine that $a_n = a(t_n)$ is a sample of an analog signal. The sum converges if the sequence tapers off sufficiently in the past and future.

For a sequence of numbers in GF(2), the only way to make something "taper off" is to have it be zero before some time in the past and after some time in the future. For a periodic sequence the above definition cannot possible converge, so we go with an alternative definition,

$$r_k \equiv (1/P) \sum_{n=1}^{P} a_n\, a_{n+k} \qquad (2.5.2)$$

where $P = 2^k - 1$ = the length of a characteristic sequence. Here it is implied that if a_{n+k} takes you off the end of the sequence, you wrap to the beginning of the sequence, just as if the sequence repeated forever in time. The only difference between this and the above definition is an infinite normalization factor.

The sum shown above is the average value of the product $a_n\, a_{n+k}$ for a characteristic sequence. This would also be true for any periodic sequence, or an aperiodic one as well as long as P is taken sufficiently large. If we were to take a statistical ensemble of shift register generators started in random starting states, we would identify the above sum as the statistical expectation value of $a_n\, a_{n+k}$, as it appeared in Spectral Chapter 6. Alternatively, we could examine the output of one shift register generator over random time intervals.

Thus, we now make a connection to Spectral Chapter 6 by associating the quantity $\langle a_n a_m \rangle$ with our digital autocorrelation function above:

$$\langle a_n a_{n+k} \rangle = r_k \equiv (1/P) \sum_{n=1}^{P} a_n\, a_{n+k} \qquad (2.5.3)$$

In particular, in Spectral Chapter 6 we derived a fairly general formula for the spectral density of a random pulse train in terms of $\langle a_n a_m \rangle$. The formula of interest is (6.4.7), and we now relate the coefficients α and β to our digital autocorrelation function:

$$\alpha = \langle a_m a_n \rangle = r_{m-n} \quad \text{for } m \neq n$$

$$\beta = \langle a_n^2 \rangle = r_0 \tag{2.5.4}$$

Formula (6.4.7) assumes that the autocorrelation function r_k is constant for all non-zero k. If this is not the case, we have to back up to the earlier formula (6.3.9) which is completely general. In fact, this was what we had to do for our AMI line code example on Section 6.6 (d).

We shall soon compute the digital autocorrelation function r_k for a characteristic sequence, and then use the result in the formulas of Chapter 6 to obtain the frequency spectrum of the characteristic sequence. But first, we insert the following brief review of "probability and statistics" in relation to $\langle a_m a_n \rangle$ and r_k .

A few Basics of Probability and Statistics Applied to GF(2)

The probability of *events* A and B both being true is denoted by $P(A \cap B)$, or P(A,B). This is known as the joint second-order probability density function (pdf). A related concept is P(A|B) meaning the conditional probability that A is true given that B is true. Here are the relations:

$$P(A,B) = P(A \cap B) = P(A|B)\, P(B) = P(B|A)\, P(A) \tag{2.5.5}$$

If events A and B are *statistically independent*, one gets:

$$P(A,B) = P(A \cap B) = P(A)\, P(B) \quad \Leftrightarrow \quad P(A|B) = P(A), \quad P(B|A) = P(B) \tag{2.5.6}$$

The result P(A|B) = P(A) means that the probability of A being true is not influenced by whether or not B is true.

Similarly one can define joint pdf's of third order and beyond. For example, statistical independence for a third-order pdf is indicated by,

$$P(A,B,C) = P(A \cap B \cap C) = P(A)P(B)P(C) \tag{2.5.7}$$

We now apply these general ideas to a sequence of numbers a_n in GF(2) . The "event" of interest is that $a_n = 1$. The following short hand notation is useful:

$$p(a_n) \equiv P(a_n = 1)$$

$$p(a_n, a_m) \equiv P(a_n = 1,\, a_m = 1) \tag{2.5.8}$$

We can now express various mean values in terms of these pdf's:

$$\langle a_n \rangle = 1 \bullet P(a_n = 1) + 0 \bullet P(a_n \neq 1) = P(a_n = 1) = p(a_n)$$

$$\langle a_n^2 \rangle = 1^2 \bullet P(a_n = 1) + 0^2 \bullet P(a_n \neq 1) = P(a_n = 1) = p(a_n)$$

$$\langle a_m a_n \rangle = 1^2 \bullet P(a_m=1, a_n=1) + 3 \text{ zero terms} = P(a_m=1, a_n=1) = p(a_m, a_n) \quad (2.5.9)$$

If $m \neq n$, it is possible that a_m and a_n could be statistically independent. In this case, one would have:

$$\langle a_m a_n \rangle = p(a_m, a_n) = p(a_m)\, p(a_n) = \langle a_m \rangle \langle a_n \rangle \quad (2.5.10)$$

For $m=n$, it is impossible that a_m and a_n be statistically independent, and one has:

$$\langle a_n a_n \rangle = p(a_n, a_n) = \langle a_n^2 \rangle = p(a_n) \neq p(a_n)\, p(a_n) \quad (2.5.11)$$

In other words, the "diagonal" second-order statistical quantity $p(a_n, a_n)$ is really the first-order quantity $p(a_n)$.

We can write down a few related statistical quantities, using $x = a_n$:

The moments for N = 1,2,3...:

$$m_N = \langle x^N \rangle = \langle a_n^N \rangle = 1^N \bullet P(a_n=1) = p(a_n) = \langle x \rangle \tag{2.5.12}$$

The central moments for N = 1,2,3..:

$$\mu_N = \langle (x - \langle x \rangle)^N \rangle = \sum_{j=0}^{N} \binom{N}{j} (-1)^j \langle x^{N-j} \rangle \langle x \rangle^j \tag{2.5.13}$$

The N=2 central moment (= variance = dispersion = σ^2 = square of standard deviation):

$$\mu_2 = \langle (x - \langle x \rangle)^2 \rangle = \langle x^2 \rangle - \langle x \rangle^2 = \langle x \rangle - \langle x \rangle^2 = p(a_n)[1 - p(a_n)] \tag{2.5.14}$$

The statistical quantities listed above are all expressible in terms of the first order pfd $p(a_n)$. In general, a full characterization of a statistical sequence requires specification of the probability density functions of all orders. However, much useful information is contained in the first two orders, and this information is sufficient for many applications.

Fact 1: The first and second order statistics are completely determined by the autocorrelation function.

Proof: Assume that the autocorrelation function r_k is known. Then,

first order:

$$p(a_n) = \langle a_n \rangle = \langle a_n^2 \rangle = p(a_n) = r_0$$

second order:

$$p(a_n, a_n) = \langle a_n^2 \rangle = p(a_n) = r_0$$

$$p(a_m, a_n) = \langle a_m a_n \rangle = r_{m-n} \qquad m \neq n$$

Corollary 1: Third and higher order statistics are not determined by the autocorrelation function. One needs a fancier entity to get these higher statistics.

Example: Consider the third-order density:

$$p(a_n, a_{n+k}, a_{n+k'}) = \langle a_n a_{n+k} a_{n+k'} \rangle$$

One could relate this to some kind of fancier 2-variable autocorrelation function defined on a characteristic sequence of period P:

$$r_{k,k'} + (1/P) \sum_{n=1}^{P} a_n a_{n+k} a_{n+k'}$$

Fact 2: For a statistically independent sequence, all higher order statistics are determined by the first order statistics:

$$\langle a_m a_n \rangle = p(a_m, a_n) = p(a_m)\, p(a_n) = [p(a_m)]^2 \qquad m \neq n$$

$$\langle a_m a_n a_k \rangle = p(a_m, a_n, a_k) = p(a_m)\, p(a_n)\, p(a_k) = [p(a_m)]^3 \qquad m \neq n \neq k$$

etc.

Proof: This follows from the definition of "statistical independence" given above.

Definition: A *white* sequence over GF(2) is one which is statistically independent and has $p(a_n) = 1/2$.

For such a sequence, we have:

$$\langle a_n \rangle = p(a_n) = 1/2$$

$$\langle a_m a_n \rangle = p(a_m, a_n) = p(a_m)\, p(a_n) = 1/4 \qquad m \neq n$$

$$\langle a_m a_n a_k \rangle = p(a_m, a_n, a_k) = p(a_m)\, p(a_n)\, p(a_k) = 1/8 \qquad m \neq n \neq k$$

etc. (2.5.15)

Here is a plot of the autocorrelation function for a white sequence a_n defined over GF(2):

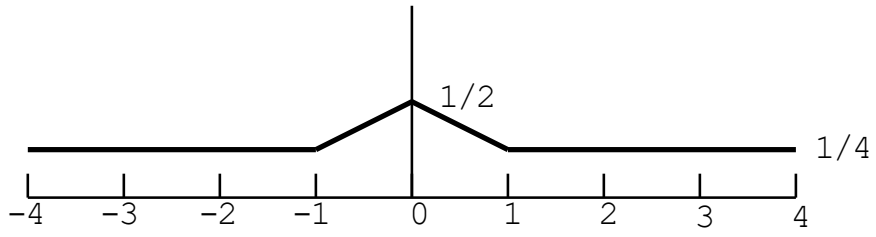


Figure 2.5.1: Autocorrelation function of a white sequence over GF(2) = {0,1}.

Modified GF(2): Coefficients are (+1, -1).

It is sometimes convenient to think of the elements of the sequence a_n as belonging to the set { -1, +1} instead of to the official GF(2) set { 0,1 }. In this case, various statements made above must be modified. For example,

$$P(a_n = 1) = p(a_n)$$

$$P(a_n \neq 1) = P(a_n = -1) = 1 - p(a_n)$$

$$\langle a_n \rangle = 1 \bullet p(a_n) + (-1) \bullet [\, 1 - p(a_n)] = P(a_n=1) = [\, 2\, p(a_n) - 1]$$

$$\langle a_n^2 \rangle = 1^2 \bullet p(a_n) + (-1)^2 \bullet [\, 1 - p(a_n)] = 1$$

$$\langle a_m a_n \rangle = 1^2 \bullet P(a_m=1, a_n=1) + 1 \bullet (-1)\, P(a_m=1, a_n=-1) + (-1) \bullet 1 \bullet P(a_m=-1, a_n=1) + (-1)^2\, P(a_m=1, a_n=-1) \qquad (2.5.9)'$$

The moments for N = 1,2,3...:

$$m_N = \langle x^N \rangle = \langle a_n^N \rangle = 1^N p(a_n) + (-1)^N [\, 1 - p(a_n)] \qquad (2.5.12)'$$

$= 1$ for N even
$= [\, 2\, p(a_n) - 1]$ for N odd

The N=2 central moment (= variance = dispersion = σ^2 = square of standard deviation):

$$\mu_2 = \langle\, (x - \langle x \rangle)^2 \,\rangle = \langle x^2 \rangle - \langle x \rangle^2 = 1 - [\, 2\, p(a_n) - 1]^2 \qquad (2.5.14)'$$

For a *white sequence*, we have:

$$\langle a_n \rangle = p(a_n) = 0$$

$$\langle a_m a_n \rangle = p(a_m, a_n) = p(a_m)\, p(a_n) = 0 \qquad m \neq n$$

$$\langle a_m a_n a_k \rangle = p(a_m, a_n, a_k) = p(a_m)\, p(a_n)\, p(a_k) = 0 \qquad m \neq n \neq k$$

etc. (2.5.15)'

And here is a plot of the autocorrelation function for a white sequence a_n defined over { -1, +1}:

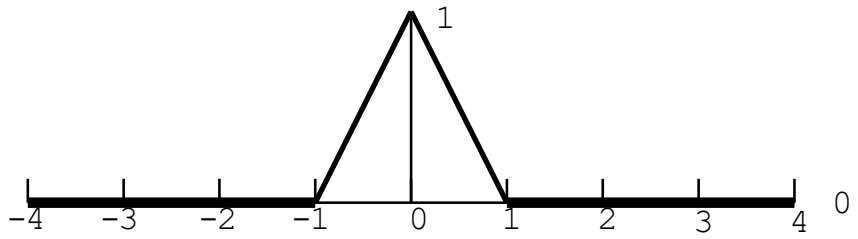


Figure 2.5.2: Autocorrelation function of a white sequence over {-1,+1}.

2.6 The Spectral Power Density of a Shift Register Generator

Consider a shift register generator of k stages having a primitive feedback polynomial h(x). What is the frequency spectrum of the characteristic sequence output by this piece of hardware? There are really two

different answers to this question, depending on what the output driver looks like. We consider the following two cases:

Case 1: (single-ended) On a 1 we make a pulse $x_{pulse}(t)$, and on a 0 we make no pulse. For 1101,

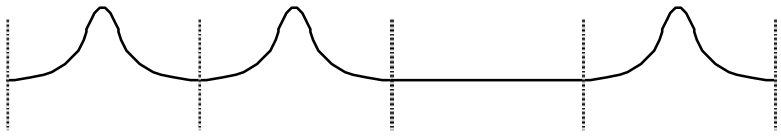


Figure 2.6.1: Encoding of 1101 using arbitrary pulse shape, singled ended drive.

Case 2: (differential) On a 1 we make a pulse $x_{pulse}(t)$, and on a 0 we make a pulse $-x_{pulse}(t)$. For 1101,

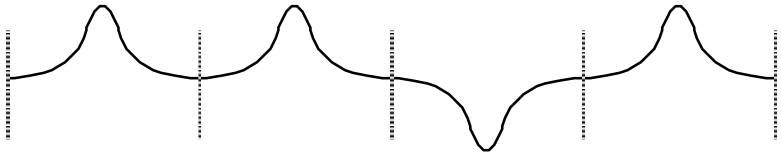


Figure 2.6.2: Encoding of 1101 using arbitrary pulse shape, differential drive.

When $x_{pulse}(t)$ is a box of width T_1 (as it would be for an ideal NRZ line code), Case 1 and Case 2 are related by a factor of 2 scale change and a DC offset. T_1 is the spacing between dotted lines in the above figures. We prefer to deal with an arbitrary pulse shape in the following analysis, so Case 1 and Case 2 are quite different.

In either case, our method of analysis will be the same. We compute the two quantities $\alpha = \langle a_m a_n \rangle$ and $\beta = \langle a_n^2 \rangle$, and we make use of our general formula given in Spectral Chapter 6, boxed equation (6.4.7). We anticipate that these averages will be independent of the subscripts, and so (6.4.7) applies. If this were not the case, we would have to back up to equation (6.3.9) as we did in the analysis of the AMI line code in Spectral Section 6.6(d). In quantity $\alpha = \langle a_m a_n \rangle$ it is implied that $m \neq n$.

Case 1 Analysis: Singled Ended Output

We replicate our table from Section 2.4 above. Recall that this table describes the product of two characteristic sequences, and just such a product sequence is what appears summed in the definition of the autocorrelation function, Equation (2.5.2).

The Counts	Column 3	Column 4	Shifted n's	Unshifted n's
n_1 = number of (0,0)'s	sum = 0	product = 0	$(2^k - 1) - 3 \bullet 2^{k-2}$	$2^{k-1} - 1$
n_2 = number of (0,1)'s	sum = 1	product = 0	2^{k-2}	0
n_3= number of (1,0)'s	sum = 1	product = 0	2^{k-2}	0

n_4 = number of (1,1)'s sum = 0 product = 1 2^{k-2} 2^{k-1}

The product of two characteristic sequences consists of 0's and 1's as shown in Column 4. To get $\alpha = \langle a_m a_n \rangle$, we use the Shifted column for the values of the n_i counts, while for $\beta = \langle a_n^2 \rangle$ we use the Unshifted counts column. In these sums, the product=0 combinations make no contribution, so we add up only the product=1 combinations. We state the results for general k, and for k = 9 as an example:

$$\alpha = \langle a_m a_n \rangle = n_4 / P = 2^{k-2}/(2^k - 1) = (1/4)(1+ 1/P) = 128/511$$

$$\beta = \langle a_n^2 \rangle = n_4 / P = 2 \cdot 2^{k-2}/(2^k - 1) = (1/2)(1+ 1/P) = 256/511$$

$$(\beta - \alpha) = 2^{k-2}/(2^k - 1) = (1/4)(1+ 1/P) = 128/511 \qquad (2.6.1)$$

Recall that $P = 2^k - 1$. Here is a plot:

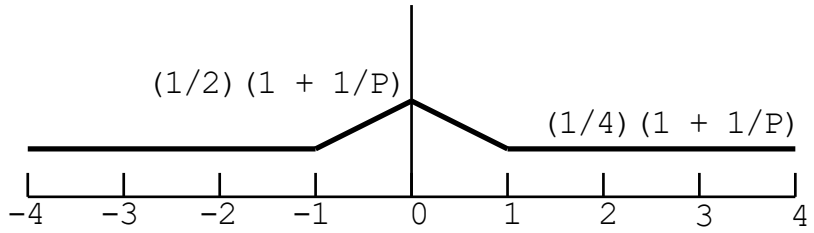


Figure 2.6.3: Autocorrelation function for Case 1 characteristic sequence

Fact 1: Comparison of Figure 2.6.3 with Figure 2.5.1 shows that the characteristic sequence has an autocorrelation function that differs only slightly from that of a white sequence. For k = 9, P = 511 so (1 + 1/P) ≈ 1.002. Thus, the first and second order statistics of a characteristic sequence are very close to those of a a white sequence.

Proof: See Fact 1 of Section 2.5 above.

The corresponding spectral power density from Spectral (6.4.7) is,

$$\frac{\langle |X(\omega)|^2 \rangle}{T} = \frac{|X_{pulse}(\omega)|^2}{T_1} \left\{ (1/4)(1 + 1/P) + (1/4)(1+ 1/P) \sum_{m=-\infty}^{\infty} 2\pi\delta(\omega T_1 - 2\pi m) \right\}$$

(2.6.2)

Notice that the continuous and discrete spectra have exactly equal strengths, and these strengths are just slightly greater than 1/4. For k=9, we get coefficients 128/511 = .0250489.

The above result is in terms of a general pulse shape. If we set P = ∞ in the above formula, we recover the spectral density of a white sequence. If we also specialize to the case of a square wave pulse, we replicate our earlier result for an NRZ line code having p = 1/2. See Spectral equation (6.5.1).

Case 2 Analysis: Differential Output

We modify the above table by replacing (0,1) with (+1,-1). We are basically scaling our coefficients by 2 and giving them an offset of -1 unit compared to Case 1. It was already noted that the resulting analog signals are considerably different between Case 2 and Case 1.

The Counts	Column 3	Column 4	Shifted n's	Unshifted n's
n_1 = number of (-1,-1)'s	sum = -2	product = 1	$(2^k - 1) - 3 \bullet 2^{k-2}$	$2^{k-1} - 1$
n_2 = number of (-1,1)'s	sum = 0	product = -1	2^{k-2}	0
n_3= number of (1,-1)'s	sum = 0	product = -1	2^{k-2}	0
n_4 = number of (1,1)'s	sum = 2	product = 1	2^{k-2}	2^{k-1}

The product of two characteristic sequences consists of -1's and 1's as shown in Column 4. To get $\alpha = \langle a_m a_n \rangle$, we use the Shifted column for the values of the n_i counts, while for $\beta = \langle a_n^2 \rangle$ we use the Unshifted counts column. This time, we get contributions from all four possible combinations. We state the results for general k, and for k = 9 as an example: (recall P = length of sequence = $2^k - 1$)

$$\alpha = \langle a_m a_n \rangle = (n_1 - n_2 - n_3 + n_4) / P = -1/P = -1/511$$

$$\beta = \langle a_n^2 \rangle = (n_1 - n_2 - n_3 + n_4) / P = 1 = 1$$

$$(\beta - \alpha) = 1 + 1/P = 1 + 1/511 \qquad (2.6.3)$$

Here is a plot:

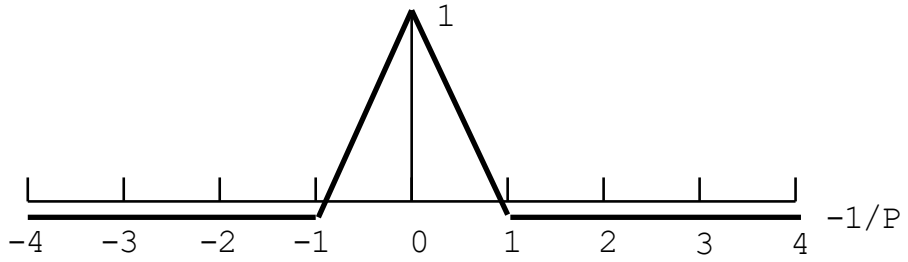


Figure 2.6.4: Autocorrelation function for Case 2 characteristic sequence

Upon comparing Figure 2.6.4 with Figure 2.5.2, one sees again that the autocorrelation function for the characteristic sequence (now defined over +1, -1) is just slightly different from that of a white sequence.

The corresponding Case 2 spectral power density from (6.4.7) is,

$$\frac{\langle |X(\omega)|^2 \rangle}{T} = \frac{|X_{pulse}(\omega)|^2}{T_1} \left\{ 1 + 1/P + (-1/P) \sum_{m=-\infty}^{\infty} 2\pi\delta(\omega T_1 - 2\pi m) \right\} \qquad (2.6.4)$$

where $P = 2^k - 1$. Now almost all the power is in the continuous spectrum. For a white sequence, we set $P = \infty$ and find that all power is in the continuous spectrum .

For an NRZ signal, Case 1 and Case 2 are essentially the same. They differ only by a factor of 4 due to the factor-of-2 scale change on the height of the pulse. The apparent discrete spectrum in (2.6.2) vanishes due to zeros in the quantity $|X_{pulse}(\omega)|^2$. Therefore:

Fact 2: Apart from a possible DC line, the spectrum of the characteristic sequence output by a shift register generator is entirely continuous, and it is the envelope of the pulse shape used.

Appendix 2.1: Proof of Fact 4 of Section 2.3

First, we restate the thing we want to prove:

Fact 4: If c is any code word of the cyclic code generated by g(x), then the infinite sequence

$$\{o_i\} = \{c,c,c,c \ldots\}$$

is a solution of the difference equation (2.3.2).

Proof:

(1) In Section 7.1 we gave the following simple proof that the code words of any linear block code G are orthogonal to the code words of the dual code H,

$$c' \bullet c^T = (d' \bullet H) \bullet (d \bullet G)^T = (d' \bullet H) \bullet G^T \bullet d^T = d' \bullet (H \bullet G^T) \bullet d^T = d' \bullet (0) \bullet d^T = 0$$

Recall that the T transpose symbol is present just to make one vector a column and the other a row, so we can dot them together. We could also write the above as

$$(\mathbf{c'})^T \bullet \mathbf{c} = 0$$

(2) In Section 7.2 we went on to discuss cyclic codes. Certainly h(x) is a code word of the dual code of g(x). Code words are just multiples of their generator, and h(x) is a multiple of h(x). Thus, we can think of the coefficients of h(x) as the **c'** in the above equation. Of course h(x) has k coefficients, and we have then to add n-k zeros to make a true dual code vector **h**. Consider this done. Now, if c is any code word of the cyclic code generated by g(x), we have

$$\mathbf{h}^T \bullet \mathbf{c} = 0$$

If we write this out as a summation, we get

$$\sum_{j=0}^{k} h_j \, c_j = 0$$

where the numbers $\{ c_0 , c_1, c_2, \ldots c_k\}$ are the first k coefficients of any code word of g(x).

We have just proved this fact:

Fact : The dot product of the coefficients of h with the first k+1 coefficients of any code word gives 0.

(3) What happens if we dot h with any k+1 consecutive coefficients of a code word c? We know that there is some other code word c' for which these k+1 consecutive coefficients are the *first* k+1 coefficients. We know this because we know that all cyclic permuations of code words are also code words. Thus we have proved that

Fact: The dot product of the coefficients of h with any consecutive k+1 coefficients of any code word gives 0. Consecutive means in the cyclic sense, so that c_{k-1} , c_k, c_0, c_1 are consecutive.

(4) Now consider what it *means* to show that an infinite sequence {o} is a solution of the difference equation (2.3.2)

$$\sum_{j=0}^{k} h_j \; o_{n+j} = 0 \qquad (2.3.2)$$

Picture the sequence $\{o_i\}$ as an infinite row of numbers. Picture the sequence $\{h_j\}$ as a finite row of numbers. Place the row $\{h_j\}$ anywhere under the infinite row $\{o_i\}$, then take the dot product of the abutting numbers. This is then the left side of (2.3.2), and the result is supposed to be zero. Varying n means varying the position of the row $\{h_j\}$. Here is a picture drawn for k=3, and n=5:

o_0 o_1 o_2 o_3 o_4 o_5 o_6 o_7 o_8 o_9 o_{10} o_{11} ...

h_0 h_1 h_2 h_3

Now consider our candidate solution of the difference equation:

$\{o_i\} = \{c, c, c, c ...\}$

Consider the above process. No matter where we position $\{h_j\}$, we will be dotting h with a consecutive set of k+1 coefficients of the code word c. We have just shown that this gives 0. Thus, $\{o_i\} = \{c,c,c,c...\}$ is a solution of the difference equation (2.3.2). Q.E.D.

Appendix 2.2: A List of Primitive Polynomials over GF(2)

Data is taken directly from E.J. Watson, **Math. Comp**. 16, pp 368-369 (1962).

This list gives a primitive polynomial for each degree k = 1 -100,107,127. As discussed earlier, there are in general many such polynomials for a given k. Recall that reversing the coefficients also gives a primitive polynomial. Notice that many values of k, including odd values, lack 3 term polynomials.

Examples: 9 4 0 $\Rightarrow$ $h(x) = x^9 + x^4 + 1$ 52 3 0 $\Rightarrow$ $h(x) = x^{52} + x^3 + 1$

Chapter 3: The Matrix Approach

Chapter 3: The Matrix Approach

This Chapter will lean heavily on Galois Chapter 8. There, it was shown that it is possible to construct an explicit matrix representation of any Galois Field. Here, as we attempt to solve directly for the output sequence (and state vector sequence) of a scrambler, we will find ourselves eyeball-to-eyeball with exactly the matters discussed in Galois Chapter 8.

3.1 Review of earlier Chapters

Because this document is so lengthy, we feel it is always worthwhile to review our current status.

In Chapter 1 we discussed the polynomial divider circuits of Figure 1.1 and Figure 1.5. Figure 1.1 is what we called "the scrambler style" circuit, whereas Figure 1.5 is the "standard circuit". We showed how these circuits both perform the division of polynomials O(z) = I(z)/H(z). An attempt was made to show how each circuit carries out this task in comparison to the way a human being would carry out the long division of polynomials. The Figure 1.5 circuit was in some ways easier to understand since its registers always contain the current remainder, or current dividend. However, the Figure 1.1 circuit is intrinsically more interesting to use since we are really studying Scramblers.

Polynomial multipliers were also treated in Chapter 1, and again two circuit forms were considered, Figure 7 and Figure 8. We spent less time worrying about the multipliers because they have no feedback, only "feedforward" and are easier to understand. It is a mathematical fact that multiplication is simpler than division.

In Section 1.8 of Chapter 1, we gave a summary box which outlines the behavior of dividers and multipliers in both the Z-domain and the time domain. In the Z-domain, the equations are just the statements of polynomial division and multiplication. In the time domain, we get difference equations which relate the output stream symbols to the input stream symbols.

In Chapter 2 we attempted to characterize the behavior in time of the polynomial divider circuits with their input streams set to zero. Such circuits are called shift register generators. Of particular interest is the periodicity of the register "state vector", and the periodicity of the output stream. We showed that the use of a Galois primitive polynomial for the polynomial h(x) [same as H(z)] resulted in things having the maximum possible periods. We made no comments as to whether this is good or bad, desirable or undesirable.

In Section 2.3 we did a rather thorough analysis of the difference equation which describes the output sequence o_n of a shift register generator, namely,

$$\sum_{j=0}^{k} h_j \; o_{n+j} = 0 \qquad n = \text{any integer} \qquad (2.3.2)$$

Here we were dealing with numbers in GF(p), and with a shift register generator having k registers, and h_i were the feedback coefficients, also the coefficients of h(x).

We showed that there are p^k solutions of the difference equation, one of which is all zeros. In the interesting case that h(x) is a primitive polynomial of GF(p^k), the remaining p^k - 1 solutions turned out to be starting time variations of *one* characteristic solution of the form { c,c,c,c...}. Here, the sequence c has period p^k - 1, and c could be considered any code word of the cyclic code (n=p^k - 1, k). The generator of this code, g(x), is a polynomial of degree n-k = p^k - 1 - k which is obtained by dividing x^n - 1 by h(x). It seemed simplest to consider the sequence c to be the code word consisting of the (p^k - 1 - k) + 1 coefficients of g(x), padded with (k-1) zeros.

We went on to state various properties of the characteristic sequence, also known as a maximal length sequence. Along the way, various Facts from the Galois notes were quoted and used.

In Section 2.2 we used the special polynomial residue class ring representation of GF(p^m) to show that the behavior of the Figure 1.5 divider circuit could be described by a Galois Iterator equation, which we duplicate here:

$$\beta_s = \alpha^s \beta_0 \; + \; [\, \alpha^{s-1} i_0 + \alpha^{s-2} i_1 + ... + \alpha\ i_{s-2} \; + \; i_{s-1} 1 \;\,] \, 1 \qquad \textit{(2.2.10)}$$

The symbol β_0 represented the starting state vector of the registers in Figure 1.5. Then β_s gave the state vector after s clockings of the circuit. We identified α with a certain entity {x}. As can be seen in the above, the input sequence i_n does of course play a role in the subsequent state vector β_s. We then turned off the input stream to get the simpler equation $\beta_s = \alpha^s \beta_0$ which describes the shift register generator.

Perhaps the reader noticed on the part of the writer a certain dissatisfaction with the above equation, and how it was not very much pursued, although it was declared to be important. There were several problems. First, we seemed to have the above equation apply to Figure 1.5, but not to Figure 1. This certainly is unsatisfying, since both circuits do the same thing, and since Figure 1.1 is the circuit used for a scrambler. Secondly, the equation above has a certain abstract feel to it. We have these abstract Galois elements α and β floating around, and we have a dim connection that α = {x} where {x} is an element of some polynomial quotient ring business. But the connection to our circuits is just not very clear. On the other hand, we realize and appreciate how the abstract Galois theory has provided us with all of our most powerful claims made so far, such as those about the period of the characteristic output sequence of a shift register generator.

In the present Chapter, all these deficiencies will be remedied.

3.2 Matrix Solution of Figure 1.5.

We assume from here on out that h(x) is monic, so that h_k = 1. From Section 1.4 we describe the operation of Figure 1.5 as

$$q_{j+1}(n+1) = d_{j+1}(n) = q_j(n) - h_j \ o(n) \qquad j = 0,1,2,...k-1 \qquad \textit{(1.4.1)}$$

As earlier, we write f(n) = f, and f(n+1) = f'. Since $h_k = 1$, o = q_k . And i = q_0. We now explicitly write out the above k equations:

$$
\begin{aligned}
q'_1 &= i - h_0 \quad q_k \\
q'_2 &= q_1 - h_1 \quad q_k \\
q'_3 &= q_2 - h_2 \quad q_k \\
q'_4 &= q_3 - h_3 \quad q_{kk} \\
&\ldots \\
q'_{k-1} &= q_{k-2} - h_{k-2} \quad q_k \\
q'_k &= q_{k-1} - h_{k-1} \quad q_k
\end{aligned}
\tag{3.2.1}
$$

These equations can be trivially re-expressed in matrix form as follows. (we apologize for the vertical bars in place of proper large parentheses)

$$
\left| \begin{matrix} q'_1 \\ q'_2 \\ q'_3 \\ q'_4 \\ \ldots \\ q'_{k-1} \\ q'_k \end{matrix} \right| = \left| \begin{matrix} 0 & 0 & 0 & \ldots & 0 & -h_0 \\ 1 & 0 & 0 & \ldots & 0 & -h_1 \\ 0 & 1 & 0 & \ldots & 0 & -h_2 \\ 0 & 0 & 1 & \ldots & 0 & -h_3 \\ \multicolumn{6}{c}{\ldots\ldots\ldots

(1) The matrix A has exactly the "companion" form C discussed in Galois Chapter 8, Section 8.5. This means that h(x) is the characteristic polynomial of A.

(2) Therefore, from Big Fact 2 of Section 8.1, we know that h(A) = 0, which we can rewrite as,

$$A^k = -h_o I - h_1 A - h_2 A^2 - \ldots - h_{k-1} A^{k-1}$$

(3) Since A is a root of h(x), we know that A is some element of a *matrix representation* of the Galois Field GF(p^m).

(4) It follows from the Section 8.5 Big Corollary that, if h(x) is a *primitive* polynomial of GF(p^m), then the matrix A can be regarded as an explicit matrix representation of an abstract *primitive* element α of the Galois Field GF(p^m). In this case, all non-zero elements of GF(p^m) can be represented as powers of A.

(5) The above solution (3.2.5) is a concrete realization of the abstract Galois Iterator obtained in Chapter 2, Eq. (2.2.10),

$$\beta_n = \alpha^n \beta_0 + [\, i_0 \alpha^{n-1} + i_1 \alpha^{n-2} + \ldots + i_{n-2} \alpha + i_{n-1} 1 \,] 1 \qquad (2.2.10)$$

In (2.2.10) the quantities β_0, β_n , α^k, 1 are abstract field elements, and the i_k are numbers in GF(p). In Eq. (3.2.5) the quantities β_0, β_n , and the outside 1 are replaced with vectors $\mathbf{q}_0$, $\mathbf{q}_n$, **1**. The quantities α^k and the inside 1 are replaced with matrices A^k and I.

Equation (3.2.4) could be put into a computer to generate the state vector sequence, while (3.2.5) gives the result in closed form.

Before going on the draw more conclusions, we pause to derive a similar matrix equation for the scrambler circuit of Figure 1.

3.3 Matrix Solution of Figure 1.

In notation similar to that of the previous section, we now write the equations of motion for the divider circuit of Figure 1

$$\begin{aligned}
q'_{k-1} &= i - h_0 q_0 - h_1 q_1 - h_2 q_2 - \ldots - h_{k-1} q_{k-1} \\
q'_{k-2} &= q_{k-1} \\
q'_{k-3} &= q_{k-2} \\
q'_{k-4} &= q_{k-3} \\
&\ldots \\
q'_1 &= q_2 \\
q'_0 &= q_1
\end{aligned} \qquad (3.3.1)$$

These equations can be trivially re-expressed in matrix form as follows,

$$
\begin{vmatrix} q'_{k-1} \\ q'_{k-2} \\ q'_{k-3} \\ q'_{k-4} \\ \dots \\ q'_1 \\ q'_0 \end{vmatrix}
=
\begin{vmatrix} -h_{k-1} & -h_{k-2} & -h_{k-3} & \dots & -h_1 & -h_0 \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & 0 \end{vmatrix}
\begin{vmatrix} q_{k-1} \\ q_{k-2} \\ q_{k-3} \\ q_{k-4} \\ .. \\ q_1 \\ q_0 \end{vmatrix}
+
\begin{vmatrix} i \\ 0 \\ 0 \\ 0 \\ .. \\ 0 \\ 0 \end{vmatrix}
\tag{3.3.2}
$$

As before, we can write this in compact vector notation as,

$$\mathbf{q'} = B\,\mathbf{q} + i\,\mathbf{1} \tag{3.3.3}$$

Notice that the ordering of the registers in the vector **q** is different here than it was in the previous section. Making the same notational change as before, we get a difference equation and solution:

$$\mathbf{q}_{n+1} = B\,\mathbf{q}_n + i_n\,\mathbf{1} \tag{3.3.4}$$

$$\mathbf{q}_n = B^n\,\mathbf{q}_0 + [\; i_0\, B^{n-1} + i_1\, B^{n-2} + \dots + i_{n-2}\, B + i_{n-1}\, I\,]\,\mathbf{1} \tag{3.3.5}$$

The matrix B above is in one of the standard forms for a companion polynomial (form C_1 of Galois Section 8.5). Thus, all observations made about matrix A in the previous section also apply to matrix B here.

This is a step forward from Chapter 2. We now see that the Galois Iterator equation applies to Figure 1.1 as well as Figure 1.5. These iterator equations are (3.2.5) and (3.3.5).

3.4 Shift Register Generators Revisited

We have learned that, whether implemented as Figure 1.1 or Figure 1.5, the "equation of motion" of the state vector of a scrambler has the basic form exemplified in (3.2.5) and (3.3.5).

We are now in a position to make more powerful statements about the shift register generator than we were able to make in the previous chapter. Consider such a generator. Setting the input sequence to zero in (3.3.5) we get,

$$\mathbf{q}_n = B^n\,\mathbf{q}_0 \qquad n = 0,1,2... \tag{3.4.1}$$

In particular we can wite

$$B^j\,\mathbf{q}_n = B^{j+n}\,\mathbf{q}_0 = \mathbf{q}_{n+j} \tag{3.4.2}$$

Meanwhile, we know that matrix B is a root of the primitive polynomial h(x), h(B) = 0, so we can write:

$$\sum_{j=0}^{k} h_j \; B^j = 0 \qquad (3.4.3)$$

Now apply both sides of (3.4.3) onto the vector $\mathbf{q_n}$, and make use of (3.4.2) to get

$$\sum_{j=0}^{k} h_j \; \mathbf{q_{n+j}} = \mathbf{0} \qquad (3.4.4)$$

Each side of this equation is a k-component vector. We have now proved the following

Fact 1: The entire state vector of a primitive polynomial shift register generator satisfies the difference equation (2.3.2). Thus, each component of the state vector satisfies (2.3.2) as well, and each component is the state of a particular register.

In Chapter 2 we learned a lot about sequences which satisfy (2.3.2), for h(x) being a primitive polynomial of GF(2^k). The main results were stated in Facts 8 and 9 of Section 2.3. The upshot is that any solution of (2.3.2) is a characteristic sequence. Therefore we now know that:

Fact 2: The state of each individual register in a primitive polynomial shift register generator cycles through the characteristic sequence.

Notice how this is an improvement over Fact 11 of Section 2.2.

Corollary 2: The output of a shift register generator cycles through the characteristic sequence.

Proof: This is just because the output happens to be one of the registers.

3.5 The Output of a Scrambler

In this section, finally, we consider the output of our polynomial divider circuit of either Figure 1.1 or Figure 1.5 when there is an input stream present. Such a device is known as a data scrambler. As above, we assume there are k registers, and that the feedback polynomial h(x) is a primitive polynomial of GF(2^k).

First, we need one more technical matter dealt with. Take (3.4.3) and multiply both sides by B^n to get,

$$\sum_{j=0}^{k} h_j \; B^{j+n} = 0 \qquad (3.5.1)$$

This says that difference equation (2.3.2) is also obeyed by the following sequence of *matrices*: { I, B, B^2, ...}. If we examine any particular matrix element of this sequence of matrices, call it the rs element, this sequence of numbers also satisfies (2.3.2), that is,

$$\sum_{j=0}^{k} h_j \, [B^{j+n}]_{rs} = 0 \qquad (3.5.2)$$

We now just proved:

Fact 1: The sequence of numbers $[B^n]_{rs}$ for any fixed rs, and for n=0,1,2.... either cycles through the characteristic sequence, or is identically 0. Thus, $[B^n]_{rs} = g_n$, n = 0,1,2.. where we indicate by $\{g_n\}$ the characteristic sequence.

Aside: I feel certain that this second option is not possible ($[B^n]_{rs}$ identically 0) , but do not know how to prove it. It has little bearing on the following discussion.

Example: If you look at the example at the end of Galois Section 8.5, you see that each matrix element of the set of matrices $\{A^n\}$ for GF(8) runs through the sequence 0010111. Don't forget to include the identity matrix, so there is a total set of 7 matrices.

Now consider (3.3.5) in the case that the starting vector $\mathbf{q}_0 = \mathbf{0}$. In other words, we assume that at some (possibly ancient) time the scrambler was in the zero state, and then some input stream began. We then have:

$$\mathbf{q}_n = [\, i_0 \, B^{n-1} + i_1 \, B^{n-2} + ... + i_{n-2} \, B + i_{n-1} \, I \,] \, \mathbf{1} = \sum_{j=0}^{n-1} i_{n-1-j} \, [B^j] \, \mathbf{1} \qquad (3.5.3)$$

We can express the sth component of this vector equation (corresponding to the state of some register s) as follows,

$$[\mathbf{q}_n]_s = i_0 \, [B^{n-1}]_{s,1} + i_1 \, [B^{n-2}]_{s,1} + ... + i_{n-2}[B]_{s,1} + i_{n-1}[I]_{s,1} = \sum_{j=0}^{n-1} i_{n-1-j} \, [B^j]_{s,1} \qquad (3.5.4)$$

According to Fact 1, the sequence of matrix elements in this sum *is* a characteristic sequence. For convenience, we shall label this sequence as follows:

$$g_{j+1} \equiv [B^j]_{s,1}$$

Then we can rewrite (3.5.4) as ,

$$[q_n]_s = i_{n-1}g_1 + i_{n-2}g_2 + i_{n-3}g_3 + .. \, i_1 \, g_{n-1} + i_0 \, g_n = \sum_{j=1}^{n} g_j \, i_{n-j} \qquad (3.5.5)$$

We have now proved the following:

Fact 2: In the presence of an input stream $\{i_n\}$, and with initial state vector $\mathbf{q}_0 = \mathbf{0}$, the state of any register of a primitive polynomial scrambler equals the "dot product" of a characteristic sequence multiplied by the (time reversed) input sequence.

We now wish to make some comment about the statistical nature of $[\mathbf{q}_n]_s$. Remember that one of the registers (some s) in either Figure 1.1 or Figure 1.5 represents the output stream of the scrambler, so we are really seeking a statement about the scrambler output statistics. To this end we need:

Lemma: If you add up n terms of a sequence $\{r_i\}$ in GF(2), and if each term in the sequence is an independent random variable with probability ε of being 1, then the probability that the sum of the n terms equals 1 is given by

$$P_n = (1/2)\,[\,1 - (1-2\varepsilon)^n\,]$$

Notice that, if $0 < \varepsilon < 1/2$, P_n approaches 1/2 as n keeps increasing. Also, if $\varepsilon = 1/2$, $P_n = 1/2$.

Proof: Let P_n = probability that the sum of n terms is a 1. Then add one more term to get

$$P_{n+1} = P_n(1-\varepsilon) + \varepsilon(1-P_n) = P_n\,(1 - 2\varepsilon) + \varepsilon = \alpha\, P_n + \varepsilon$$

where we define $\alpha = (1 - 2\varepsilon)$. Iterate this difference equation starting with $P_1 = \varepsilon$ to get

$$P_n = \varepsilon\,(1 + \alpha + \alpha^2 + \ldots + \alpha^{n-1}) = \varepsilon\,(1 - \alpha^n)/(1-\alpha) = (1/2)\,(1 - \alpha^n). \qquad \text{Q.E.D.}$$

Now consider the state of any scrambler register as being the (reversed) input sequence dotted with the characteristic sequence, as shown in (3.5.4) above. Consider one particular term in this sequence, for example,

$$g_3\, i_{n-3}$$

Assume that an input bit (like i_{n-3}) has a probability p of being a 1. We know (see Section 2.5, Case 1 Analysis, $\beta = p'$) that the probability of a characteristic sequence bit being 1 is $p' = \beta = (1/2)(1 + 1/P)$, where $P = 2^k - 1$ = the Period of the characteristic sequence. In other words, for reasonably large k, p' is just slightly larger than 1/2. In order for our term $g_3\, i_{n-3}$ to be 1, both factors must be 1, so the probability that this term is 1 is

$$\varepsilon = pp' \qquad p = \text{input sequence prob of 1} \qquad p' = (1/2)(1 + 1/P) \approx 1/2, \qquad P = 2^k - 1$$

We can now use the above Lemma to obtain the probability that the n term sum shown in (3.5.5) is a 1:

$$\{\text{ Prob that } [q_n]_s = 1 \} = (1/2)\,[\,1 - (1-2pp')^n\,]$$

Since $p' \approx 1/2$, we have $\varepsilon = pp' < 1/2$ unless p is unusually close to 1. Thus, as the scrambler continues to run and n increases, the probability that any register of the scrambler is a 1 gets closer and closer to 1/2, *regardless* of the value of the p of the input stream, as long as p is not too close to 1.

Admittedly, we have not been too rigorous in our statistical analysis. With this caveat, we claim to have proven the following fact

Fact 3: If we assume that the input stream is characterized by a p in some reasonable range, say $0.1 < p < 0.9$, then after some number M of clocks has passed, the p of any register of the scrambler is very close to 1/2. It differs from 1/2 in our example by amount $(1/2)\ (0.9)^{M}$. If we let M = 1000 clocks, this difference quantity is on the order of 10^{-46}.

Corollary 3: Barring highly anomalous input streams, and assuming the average value of the input stream (with symbols 0,1) lies in some reasonable range like $0.1<p<0.9$, and assuming there are at least say 6 registers in a primitive polynomial data scrambler, so that $p' \approx 1/2$, then after the scrambler runs for on the order of 1000 clocks, the average value of the output stream will be so close to 1/2 that the variation from 1/2 will be unmeasurable.

Review of Leeper 1973.

A Universal Digital Data Scrambler, by David. G. Leeper, BSTJ 52,10, Dec 1973, pp 1851-1865.

This paper considers the scrambler-descrambler combination just as we are used to seeing it. Earlier papers dealt with scramblers having periodic input streams, while this paper deals with an arbitrary input stream. Subject to a certain technical qualification, the paper concludes that, by choosing a sufficiently large number of stages in the scrambler, the output sequence can be made as "white" as one requires. By this, the author means that the "first and second order statistics" of the scrambler output can be made arbitrarily close to those of white noise.

The "first order statistics" is that the probability of an output bit being a 1 can be made as close to 1/2 as required. We have shown in our own discussion above how this comes out, and we have stolen some of Leeper's arguments in our presentation.

The "second order statistics" are twofold. First, Leeper shows that the autocorrelation function of the output sequence can be made arbitrarily close to that shown in our Figure 2.5.2 of Chapter 2. Second, he shows that any pair of symbols in the scrambler output sequence can be made as "independent" as required. This means that the "joint second-order density" $p(o_m, o_n) = p(o_m)p(o_n) + \varepsilon$, where ε can be made as small as required, again by choosing a suitable number of registers k. Here $\{o_n\}$ is the output sequence. Since the first order statistic says that $p(o_m) \approx 1/2$, he gets $p(o_m, o_n) \approx 1/4$.

The "technical qualification" of the paper is a bit tricky, and we hope that some later paper has been able to bring it a little closer to the practical realm. It is my belief that with k=9 stages, a scrambler designed for use with signals which come from a "live" video source will easily meet this technical qualification. The technical qualification is that the input stream cannot be "too perfect" in some sense, and the author is able to make it less perfect by intentionally adding some "error" to the input stream by simulating a binary

symmetric channel with some small probability of error. Leeper comments that the inherent noise in a digital signal that derives from an analog source is likely to have a sufficient amount of this "imperfection" so that a scrambler with a small number of registers will meet the conditions of his Universal Scrambler Theorem, which results in the conclusions noted above -- namely, that the output stream has the same first and second order characteristics as white noise.

3.6 The Spectral Density of a Scrambler Output

The basic conclusion of the previous section can be summarized as follows:

" Barring anomalous input sequences, the spectral density of the output of an operating scrambler is essentially the same as the spectral density of a random white NRZ signal."

The interesting fact we learned is that, even if $p \neq 1/2$ for the input stream, we end up with $p = 1/2$ for the output stream. This is what "white" means in our current context.

We have already derived the spectral density of such a signal in Spectral Section 6.6(a), and we quote that result here, setting $p = 1/2$:

$$\frac{\langle |X(\omega)|^2 \rangle}{T} = V^2 \, T_1 \, [\, \mathrm{sinc}(\omega T_1/2) \,]^2 \quad \{ \; (1/4) \; + \; (1/4) \; 2\pi \, \delta(\omega T_1) \; \} \qquad \text{Spectral (6.6.1)}$$

If we integrate the DC line, we get power = $(V/2)^2$. Our "power" in effect operates on a 1 ohm load, so this is the power going into a cable due to the average voltage being (1/2) V. In a real system this would be blocked by adding a DC offset to the signal, then there would be no DC line at all.

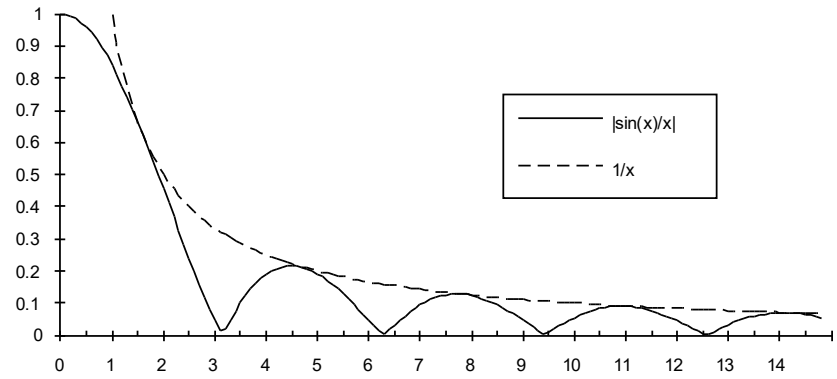
This leaves the interesting part of our scrambler output spectral density,

$$\frac{\langle |X(\omega)|^2 \rangle}{T} = (V/2)^2 \, T_1 \, \mathrm{sinc}^2(\omega T_1/2) \qquad (3.6.1)$$

Here, T_1 is the width of a NRZ "1" or "0", and V is the peak-to-peak amplitude of the NRZ signal.

Interpretation of the Spectrum

It might be helpful to look at the Spectral Figure 17.1 plot of sinc(x) , which we repeat here,



For a periodic square wave pulse train having half-pulses of width T_1, the spectrum was a line spectrum with the first "fundamental" line occurring at $x = \pi/2$ of $\text{sinc}(x = \omega T_1/2)$. This line is at

$$\omega = 2\pi/ 2T_1 \qquad f = 1/(2T_1) = (1/2)\, f_1.$$

If such a signal represented the string 01010101... for a 140 Mbit/sec NRZ signal, one would have f_1 = 140 MHz, and f = 70 MHz. One would normally use a channel with slightly more than 70 MHz bandwidth, and get this line passed through, and ignore the higher lines.

Our scrambler output signal is different. The spectrum is entirely continuous, and it is the area under the above sinc(x) curve squared. In the pure periodic square wave signal, each "line" was at the peak of one of the above humps. In the scrambled real signal, one can imagine that each such discrete line has been broadened out to encompass the entire hump. The right side of the hump is the "upper sideband", and the left side of the hump is the "lower sideband". There is no residual "carrier" line left at the center of each hump. However, there is amplitude at the hump centers, just as there is everywhere except at the zeros. It is just the amplitude of the continuum $\text{sinc}^2(x)$ envelope.

Fact: If you have a 140 Mbit/sec scrambled NRZ signal, and you want to pass the entire main hump, you need a system bandwidth of 140 MHz. To pass the left half of the main hump only, you are back to the 70 MHz bandwidth.

Power Distribution by Hump

It might be interesting to examine the amount of power in various portions of the spectrum. We obtain relative answers by integrating $\text{sinc}^2(x)$ over various ranges. This is not a handy closed form integral, the easiest thing is to do parts and cast the indefinite integral in terms of the Si(x) (sine integral) function. We will then integrate from 0 to various points of interest:

$$\int_0^b dx\, \text{sinc}^2(x) \quad = -\sin^2(b)/b + \text{Si}(2b)$$

Here is a table,

b	2b	Si(2b) =	$-\sin^2(b)/b$	integral
$\pi/2$	π	$Si(\pi) = 1.8519370$	$-2/\pi = -.6366198$	1.2153172
π	2π	$Si(2\pi) = 1.4181516$	0	1.4181516
2π	4π	$Si(4\pi) = 1.4921612$	0	1.4921612
3π	6π	$Si(6\pi) = 1.5180339$	0	1.5180339
4π	8π	$Si(8\pi) = 1.5311313$	0	1.5311313
5π	10π	$Si(10\pi) = 1.5390291$	0	1.5390291
∞	∞	$Si(\infty) = \pi/2 = 1.5707963$	0	1.5707963

And here are the contributions of the various humps, as a percent of the total power:

% Left Half of Main Hump = 1.2153172/1.5707963 =0.7736950	= 77.4%
% Right Half of Main Hump = (1.4181516 - 1.2153172)/1.5707963 =0.1291284	= 12.9%
% Main Hump = 1.4181516/1.5707963 = 0.9028234	= 90.2%
% Second Hump = (1.4921612 - 1.4181516)/ 1.5707963 = 0.0471160	= 4.7%
% Third Hump = (1.5180339 - 1.4921612)/ 1.5707963 =0.0164711	= 1.6%
% Fourth Hump = (1.5311313 - 1.5180339)/ 1.5707963 =0.0083381	= 0.8%

Fact: If, for a 140 Mbit/sec NRZ signal, you pass only the left half of the main hump by using a 70 MHz bandwidth system, you pass 77.4% of the power spectrum. This seems like a reasonable thing to do.

3.7 The NRZI Mini-Scrambler

Typically, a scrambler circuit is followed by an output stage consisting of this circuit:

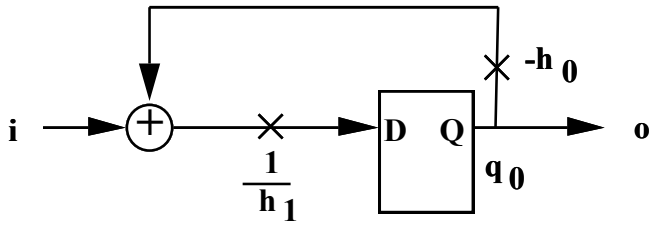


Figure 3.7.1: The NRZI mini-scrambler.

This is a k=1 scrambler (see Figure 1), and the only possible irreducible polynomial is $h(x) = 1 + x$, so $h_0 = h_1 = 1$. Thus, the multiplying x's in the figure can be ignored (-1 = +1 in GF(2)).

Consider the generator matrix B for this scrambler that we would use in equation (3.3.5). The matrix represents a primitive element of $GF(2^k) = GF(2^1) = GF(2) = \{0,1\}$. Thus, the matrix is 1x1 and is simply the number "1".

B = 1 h(B) = 0 = the characteristic polynomial = B + 1

The state vector for this scrambler is the state of its one register. Using this fact, and B = 1, we can write down equation (3.3.5) as follows:

$$o_n = o_0 + [\, i_0 + i_1 + i_2 + \ldots + i_{n-1}] \qquad (3.7.1)$$

Thus, the output of this "mini-scrambler" is simply the *sum* of the input stream plus the initial state of the register. Assume that n-1 bits have been input as shown, and that o_n has some value, either 0 or 1. Consider the effect of the next input bit i_n. If it is a 0, o_n does not change, and if it is a 1, o_n toggles. We have now proven that

Fact 1: The output of the above mini-scrambler changes state each time a 1 is encountered in the input data stream, and it holds its state each time a 0 is encountered in the input stream.

Comment: We have shown this result based on a trivial reduction of our matrix scrambler formalism. The result is of course obvious just looking at the above circuit, since the GF(2) adder is an XOR gate.

Definition: An *NRZI line code* is what emerges from the above circuit when you input a NRZ linecode.

NRZ means "non-return-to-zero", the idea being that on a 1 or mark, the signal stays at 1 for the entire state of the mark, which we have usually called T_1. Various other line codes only hold the mark for half the period and are therefore called RZ or "return to zero" type codes. The I in NRZI = NRZ(I) means that one bit value (in our case a 1) is encoded as an Inversion of the level, ie, as an *edge*.

Corollary 1: In an NRZI signal as output by the above flip-flop, a 1 is indicated by a state which begins with an edge, and a 0 is indicated by a state which begins with no edge. The location of the state relative to the edge it unimportant. The important thing is that a 1 is represented by an edge, and a 0 is represented by the *absence* of an edge.

Observation: Since there is no mention of the polarity of an "edge", we see that the NRZI signal is non-polar. If you have a differential NRZI signal carried on a twisted pair, it does not matter which way you hook up the wires at the receiver. Hooking them up "wrong" just changes the polarity of edges, which does not affect the interpretation of the NRZI signal in terms of 1's and 0's. The desire to have a non-polar signal is the only reason for the use of the mini-scrambler following the main scrambler.

Question: What is the effect of the mini-scrambler on the output of the main scrambler in terms of statistics and frequency spectrum?

On its own, the mini-scrambler cannot add any statistical whiteness to its incoming data stream. It is too weak to do this. We have seen above how the mini-scrambler produces an output stream that is just a sum of the input stream. This should be compared with Equation (3.5.5) of Section 3.5 which shows what a "real" scrambler does to its input stream. The stream is multiplied term by term with the characteristic sequence of the scrambler. For reasonable k, this characteristic sequence $\{g_i\}$ is "pseudorandom" and "induces" onto the output stream its random nature.

For the mini-scrambler, the length of the characteristic sequence is $2^1 - 1 = 1$, so the characteristic sequence is simply { 1,1,1,1,1 ...}. You can imagine in Eq. (3.7.1) that the input sequence is multiplied by this light-weight characteristic sequence. Obviously, the sequence { 1,1,1,1,1 ...} does not carry a lot of whitener. It is like a bad detergent.

Nevertheless, the mini-scrambler is applied after the action of the main scrambler. Thus, the input to the mini-scrambler is already "white". This means that p = 1/2 and the individual bits emerging from the main scrambler are to a high degree statistically independent. We can therefore use the Lemma of Section 3.5.

This Lemma describes exactly the situation we have above. The output sequence of the mini-scrambler is a sum of n terms of the sequence {i} in GF(2), and the input bits are independent. Thus, we get:

output stream probability of 1 $= P_n = (1/2)\,[\,1 - (1 - 2/2)^n\,] = 1/2$

In other words, white-in gives white-out. If the input stream to the mini-scrambler is slightly off white, perhaps $p = 1/2 + \delta$ say, then we get

output stream probability of 1 $= P_n = (1/2)\,[\,1 - (-2\delta)^n\,]$

This shows that the P_n of the mini-scrambler output oscillates around the $P_n = 1/2$, always getting closer to it as time goes by. So in this sense, the whiteness of the main scrambler is somewhat improved by the mini-scrambler. However, the main issue here is the statistical independence of the bits, ie, the "higher order statistics" that you cannot see just looking at the first order statistical quantity P_n. It is the job of the main scrambler to generate this independence.

We summarize these remarks as follows:

Scrambler Summary:

(a) We first assume that the main scrambler does its job of producing a white output sequence. If the output of this scrambler is applied through an output driver to a line, the resulting spectrum is obtained from Eq. (2.6.2) for the single-ended Case 1 signal of Figure 2.6.1, and from Eq. (2.6.4) for the differential Case 2 signal of Figure 2.6.2. In each case, we set $P = \infty$ since we assume full whiteness.

$$\frac{\langle |X(\omega)|^2 \rangle}{T} = \frac{|X_{pulse}(\omega)|^2}{T_1}(1/4)\ \{\ 1\ +\ \sum_{m=-\infty}^{\infty} 2\pi\delta(\omega T_1 - 2\pi m)\ \} \qquad (2.6.2)\ \text{Case 1}$$

$$\frac{\langle |X(\omega)|^2 \rangle}{T} = \frac{|X_{pulse}(\omega)|^2}{T_1}\ \{\ 1\ \} \qquad (2.6.4)\ \text{Case 2}$$

Here, $X_{pulse}(\omega)$ is the Fourier Integral transform of the pulse shape used to drive the line at the output of the scrambler. T_1 is the time duration of one state (high or low). The LHS of each of these equations is the "spectral power density" $P(\omega)$, so that $P(\omega)d\omega$ is measured in watts.

(b) If the main scrambler input is set to all zeros -- a highly non-random data stream -- the above formulas are still correct with a very small amount of error. The error is in the form of the 1/P corrections appearing in (2.6.2) and (2.6.4), where P = length of main scrambler characteristic sequence.

(c) Under the assumption of (a), the insertion of the NRZI mini-scrambler between the main scrambler and the output driver circuit has zero effect on the statistics of the output signal, and zero effect on the above power spectrum of the output signal.

(d) If the line driving pulse is selected to be an ideal box of width T_1 and height V, and if we add a DC offset to the driven signal as needed to eliminate any DC component (add a capacitor), the Case 1 result above reduces as shown in Eq. (3.6.1),

$$\frac{<|X(\omega)|^2>}{T} = (V/2)^2 \; T_1 \; \mathrm{sinc}^2(\omega T_1/2)$$

The power spectrum is continuous, there are no discrete lines whatsoever.

(e) Other pulse shapes can be evaluated using the above formulas and the Fourier Transform discussed in the Spectral notes.

(f) One of the main motivations for using a scrambler in the first place is to obtain the above continuous spectrum. Since power is smoothly distributed over a continuum of frequencies and there are no sharp lines, crosstalk between cables and circuits carrying scrambled signals is minimized. RFI emissions are also held in check.

(g) Notice that this is true even if the input signal to the main scrambler is all zeros. In this case, the main scrambler simply outputs its characteristic sequence.

(h) Another motivation for scrambling is to reduce the frequency of occurrence of large "edge-less" periods in the signal. Such streams make clock recovery more difficult. Because of the NRZI output circuit, only strings of zeros output by the main scrambler are of concern in this regard. According to our discussion in Section 2.4, the probability distribution of strings of n zeros output by the main scrambler (and also by the mini-scrambler) is independent of the distribution of such strings in the source signal, and is given to a high degree of accuracy by the simple formula $P(n) = 2^{-n}$. When the source input stream is all zeros, the maximum length of a zero output string is k-1. However, for real input data, arbitrarily long strings of zeros are possible with relative probability 2^{-n}. This subject will be discussed further in Section 3.11 below.

3.8. Matrix analysis of the polynomial multiplier (de-scrambler) of Figure 7 .

As noted earlier, the polynomial multiplier circuits are easier to analyze than the corresponding divider circuits mainly because the multipliers have no feedback. Since this is a Chapter on the Matrix Approach, and since we later want to prove by brute force that the scrambler-descrambler combination really does pass data unaltered, we pause here to analyze our multiplier circuits.

For the circuit shown in Figure 7, the equation of motion for the state vector is very simple, and the output is a linear combination of the contents of the registers. That is, the output is a function of the state vector. The data simple flows through the flip-flops from left to right, so that $q_0' = q_1$, and $q_1' = q_2$, etc. When this is put in our usual matrix form we get,

$$\begin{vmatrix} q'_{k-1} \\ q'_{k-2} \\ q'_{k-3} \\ q'_{k-4} \\ \dots \\ q'_1 \\ q'_0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ & & & \dots\dots\dots & & \\ 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & 0 \end{vmatrix} \begin{vmatrix} q_{k-1} \\ q_{k-2} \\ q_{k-3} \\ q_{k-4} \\ .. \\ q_1 \\ q_0 \end{vmatrix} + \begin{vmatrix} i \\ 0 \\ 0 \\ 0 \\ .. \\ 0 \\ 0 \end{vmatrix} \tag{3.8.1}$$

The matrix here has the same form as in Eq. (3.3.2) for the scrambler divider circuit of Figure 1, except here all the feedback entries in the first row are zero. As before, we use matrix/vector notation to rewrite the above as $\mathbf{q'} = C\mathbf{q} + i\mathbf{1}$. We then re-install the time index n to get the vector difference equation and its solution:

$$\mathbf{q}_{n+1} = C\,\mathbf{q}_n + i_n\mathbf{1} \tag{3.8.2}$$

$$\mathbf{q}_n = C^n\,\mathbf{q}_0 \quad + [\, i_0\, C^{n-1} \;+\; i_1\, C^{n-2} + \dots + i_{n-2}\, C \;+\; i_{n-1} I \,]\,\mathbf{1} \tag{3.8.3}$$

The column vector $\mathbf{q}_n$ has the following components, where we revert to a former notation of putting the time index as an argument,

$$[\,\mathbf{q}_n\,]_s \;=\; q_{k-s}(n) \tag{3.8.4}$$

For example, with s=1 the first component of the column vector $\mathbf{q}_n$ equals the register q_{k-1} of Figure 7 at time n.

We have named the above matrix "C", not to be confused with the companion C of Galois Chapter 8. The matrix C has the following matrix elements, as can be seen from (3.8.1),

$$C_{ij} = \delta_{i,j+1}\,, \qquad [C^2]_{ij} = \delta_{i,j+2} \qquad , \dots \qquad [C^n]_{ij} = \delta_{i,j+n} \tag{3.8.5}$$

The matrix C^2 has the ones slid down to the next lower subdiagonal, and so on for higher powers. Finally, the matrix C^{k-1} has a single non-vanishing element, a 1 in the lower left corner. All higher powers $C^n = 0$ for $n \geq k$. Thus, in the solution (3.8.3), most terms vanish. Only those terms having powers less than k survive. We therefore rewrite (3.8.3) as,

$$\mathbf{q}_n = C^n \mathbf{q}_0 + [\, i_{n-k}\, C^{k-1} + i_{n-3}\, C^2 + i_{n-2}\, C + i_{n-1} I\,]\, \mathbf{1} \qquad I = C^0$$

$$= C^n \mathbf{q}_0 + \sum_{j=1}^{k} i_{n-j}\, [C^{j-1}]\, \mathbf{1} \tag{3.8.6}$$

Interpretation: If $n \geq k$, the current state vector of the multiplier is affected only by the most recent k input symbols. If $n < k$, the first term in (3.8.6) has not yet vanished, so some components of the starting state vector q_0 still have an effect.

These comments are of course totally trivial, since the circuit is just a shift register. After the first k clocks of operation, the original contents (the starting state vector) have been shifted out and have no influence on anything that happens in the future. This is in sharp contrast to what happens in the polynomial divider circuit, where the starting state vector has a lingering influence that lasts forever.

An exercise in notation.

This section is included only to demonstrate how to deal with the strange matrix/vector notation used above and in earlier sections of this Chapter. The purpose of this exercise is to use the above matrix/vector notation and to try and recover the time-domain statement of how a polynomial multiplier works.

In an upcoming section we will be attempting a rigorous time-domain proof that a descrambler really does recover the source data stream that went into the scrambler. In other words, we will prove that a self-synchronizing scrambler/descrambler combination really works. In that proof, manipulations of the kind appearing below will be used a lot, so the following "exercise" is a sort of warm up.

Consider the sth component of the vector equation (3.8.6),

$$[q_n]_s = \sum_{j=1}^{k} [\, C^n \,]_{s,j}\, [\mathbf{q}_0]_j + \sum_{j=1}^{k} i_{n-j}\, [C^{j-1}]_{s,1} \tag{3.8.7}$$

According to (3.8.4), we have $[\, C^n \,]_{s,j} = \delta_{s,n+j}$ and $[C^{j-1}]_{s,1} = \delta_{s,j}$ so both sums go away giving

$$[\mathbf{q}_n]_s = [\mathbf{q}_0]_{s-n} + i_{n-s} = [\mathbf{q}_0]_{s-n}\, \theta(n<s) + i_{n-s}\, \theta(n \geq s) \tag{3.8.8}$$

Here we have added explicit θ functions [θ(A) = 1 if A true, 0 if A false] to show that only one of the two terms can be non-vanishing for a given s. The vector index s-n must be in the range (1,k) so $s-n \geq 1$

$\Rightarrow$ n<s for the first term. And the input sequence is assumed to begin with i_0 , hence n-s $\geq$ 0 in the second term.

Now use (3.8.4) to rewrite this last result as

$$[\mathbf{q_n}]_s = q_{k-s}(n) = q_{k-s+n}(0)\,\theta(n<s) + i_{n-s}\,\theta(n\geq s) \qquad (3.8.9)$$

This is an elaborate statement of a trivial fact, see Figure 7. Consider register q_{k-s} . If the number of clocks n is less than s, then after n clocks from startup, register q_{k-s} contains the startup contents of the register n to the left. If n > s, register q_{k-s} contains i_{n-s}. If this still seems strange, pick a particular value of s (try s=1) and stare at Figure 7.

The output of the polynomial multiplier can be written in terms of the components of vector q_n:

$$o_n = \sum_{s=0}^{k-1} h_s\,[q_n]_{k-s} + h_k\,i_n \qquad (3.8.10)$$

If the vector solution (3.8.9) is inserted into (3.8.10), we can solve for the output sequence o_n of our polynomial multiplier:

$$o_n = \sum_{s=0}^{k-n-1} h_s\,q_{s+n}(0) + \sum_{s=k-n}^{k} h_s\,i_{n+s-k} \qquad (3.8.11)$$

As a test of this result, lets examine the first two terms in this output sequence,

$$o_0 = \sum_{s=0}^{k-1} h_s\,q_s(0) + h_k i_0 \quad ,$$

$$o_1 = \sum_{s=0}^{k-2} h_s\,q_{s+1}(0) + h_{k-1} i_0 + h_k\,i_1$$

For n $\geq$ k, the first sum contributes nothing, and the second sum has its full range:

$$o_n = \sum_{s=0}^{k} h_s\,i_{n+s-k} \qquad n\geq k \qquad (3.8.12)$$

Setting n:= n+k we get

$$o_{n+k} = \sum_{s=0}^{k} h_s\,i_{n+s} \qquad n\geq 0 \qquad (3.8.13)$$

Finally we have recovered our time domain result appearing in the box at the end of Section 1.8 in Chapter 1.

3.9 Matrix analysis of the standard polynomial multiplier of Figure 8 .

Looking at Figure 8, one can write down the following series of equations,

$$
\begin{aligned}
q'_1 &= h_0\, i \\
q'_2 &= q_1 + h_1\, i \\
q'_3 &= q_2 + h_2\, i \\
&\ldots \\
q'_{k-1} &= q_{k-2} + h_{k-2}\, i
\end{aligned}
\qquad (3.9.1)
$$

These equations may be combined into a single matrix equation,

$$
\begin{vmatrix} q'_1 \\ q'_2 \\ q'_3 \\ q'_4 \\ \ldots \\ q'_{k-1} \\ q'_k \end{vmatrix}
=
\begin{vmatrix} 0 & 0 & 0 & \ldots & 0 & 0 \\ 1 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ \multicolumn{6}{c}{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots} \\ 0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & \ldots & 1 & 0 \end{vmatrix}
\begin{vmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ \ldots \\ q_{k-1} \\ q_k \end{vmatrix}
+ i \times
\begin{vmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ \ldots \\ h_{k-2} \\ h_{k-1} \end{vmatrix}
$$

(3.9.2)

We recognize the matrix above as the same matrix C which appeared in the Figure 7 analysis. Notice that the rightmost vector used to be **1**, but here it is a vector consisting of the first k components of the feedback polynomial coefficients. We call this vector **h**.

As we have done many times before, we write the above as a vector difference equation and state the solution of this difference equation. We also write out a component of the vector $\mathbf{q}_n$:

$$\mathbf{q}_{n+1} = C\,\mathbf{q}_n + i_n\mathbf{h} \qquad (3.9.3)$$

$$\mathbf{q}_n = C^n\,\mathbf{q}_0 + [\, i_0\, C^{n-1} + i_1\, C^{n-2} + \ldots + i_{n-2}\, C + i_{n-1} I \,]\,\mathbf{h} \qquad (3.9.4)$$

$$[\,\mathbf{q}_n\,]_s = q_s(n) \qquad (3.9.5)$$

The comments made about the matrix C all apply here as well, so we can rewrite (3.9.4) as,

$$\mathbf{q}_n = C^n\,\mathbf{q}_0 + [\, i_{n-k}\, C^{k-1} + i_{n-3}\, C^2 + i_{n-2}\, C + i_{n-1} I \,]\,\mathbf{h} \qquad I = C^0$$

$$= C^n \mathbf{q}_0 + \sum_{j=1}^{k} i_{n-j} [C^{j-1}] \mathbf{h} \tag{3.9.6}$$

In Figure 7, the output o_n was a sum of many terms. For Figure 8, the output is the sum of only two terms,

$$o_n = q_k(n) + h_k i_n = [\mathbf{q}_n]_k + h_k i_n$$

$$= \sum_{j=1}^{k} [C^n]_{k,j} [\mathbf{q}_0]_j + \sum_{j=1}^{k} \sum_{s=0}^{k-1} i_{n-j} [C^{j-1}]_{k,s} h_s + h_k i_n \tag{3.9.7}$$

According to (3.8.4), we have $[C^n]_{k,j} = \delta_{k,n+j}$ and $[C^{j-1}]_{k,s} = \delta_{k,j+s}$ so both j sums go away leaving

$$o_n = [\mathbf{q}_0]_{k-n} + \sum_{s=k-n}^{k-1} h_s i_{n+s-k} + h_k i_n$$

$$= \theta(n<k) [q_{k-n}(0) + h_k i_n] + \theta(n \geq k) \sum_{s=0n}^{k} h_s i_{n+s-k} \tag{3.9.8}$$

The left term controls the output for the first n clocks. For example,

$o_0 = q_k(0) + h_k i_0$.

$o_1 = q_{k-1}(0) + h_k i_1$.

Notice that this is completely different from the output of the Figure 7 circuit. However, after n clocks have gone by, the second term takes over and we get the same results as for Figure 7, namely, equations (3.8.12) and (3.8.13).

This concludes our matrix exercises for the polynomial multiplier circuits. The obvious conclusion in either case is that nobody cares about the output of these circuits for the first k clocks. After that, the output is given by the fundamental result (3.8.13).

3.10. Proof that a Descrambler really descrambles the output of a Scrambler.

The proposed serial digital standard involves the following scrambler - descrambler structure:

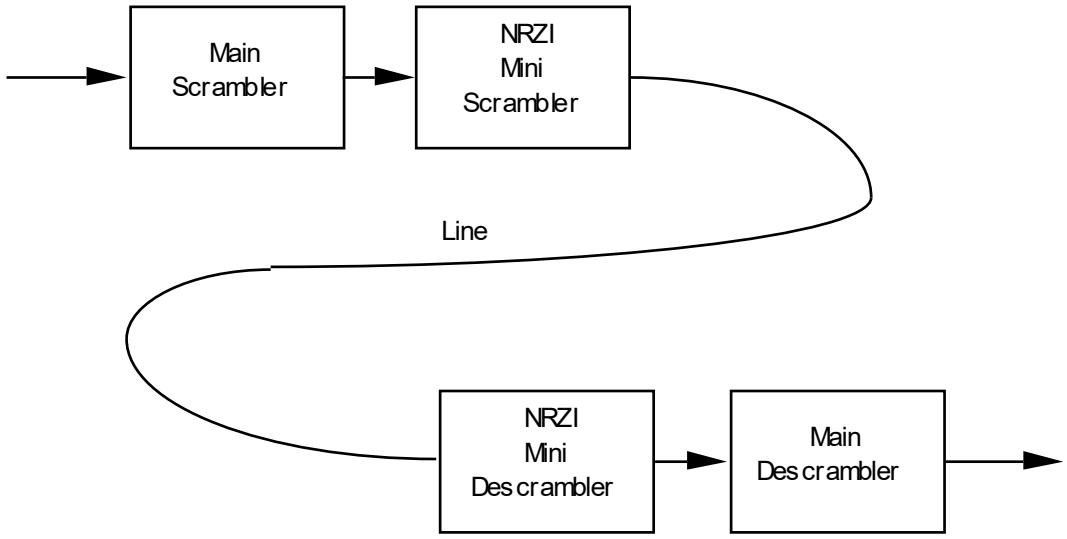


Figure 3.10.1: Proposed scrambler structure for serial digital communication .

One can think of this as an operation $S_1\ S_2\ D_2\ D_1$ acting on the signal.

We want to prove that a "descrambler" really descrambles the output of a scrambler. In terms of operations, this means that S D = 1, or D = S^{-1}. The "1" means there is no change to the signal after passing through a scrambler followed by a descrambler.

We will show this for an arbitrary scrambler. Then, with regard to the above picture, we will have shown that

$$S_1\ S_2\ D_2\ D_1 = S_1\ S_2\ S_2^{-1}S_1^{-1} = S_1(\ S_2S_2^{-1}\ )S_1^{-\ 1} = S_1(\ 1\ )S_1^{-\ 1} = S_1\ S_1^{-\ 1} = 1$$

Here subscript 1 refers to the Main scrambler and descrambler, while 2 refers to the mini NRZI operations as described in Section 3.7 above.

Proof in the Z Domain

Back at the end of Section 1.8 in Chapter 1 we wrote down the z-domain and time-domain equations for polynomial dividers (scramblers) and multipliers (descramblers). Here they are again:

	z-domain	time-domain	
Poly Divider (scrambler)	$O(z) = I(z)/H(z)$	$i_n = \sum_{j=0}^{k} h_j\ o_{n+j}$	(3.10.1)

(Figure 1.1 or Figure 1.5)

Poly Multiplier (descrambler) $z^k\, O(z) = I(z)\, H(z)$ $o_{k+n} = \sum_{j=0}^{k} h_j \; i_{n+j}$ (3.10.2)

(Figure 7 or Figure 8)

In the z domain, our proof is very simple and goes as follows. Let I(z) represent the source data. Let O(z) be the output of a scrambler with coefficients H(z). Then ,

$$O(z) = I(z)/H(z)$$

Now run this O(z) as the input to a descrambler, whose output we will call O'(z):

$$z^k\, O'(z) = \{ O(z) \}\; H(z) = \{ I(z)/H(z) \}\; H(z) = I(z)$$

This says that the output O'(z) of the descrambler replicates the input I(z) to the scrambler, apart from a time delay of k clocks. This delay is characteristic of the multiplier circuit.

Recall that all these z-transform functions of z are just polynomials. Of course the ones other than H(z) can be arbitrarily long, since their corresponding sequences can be long.

The above proof is correct, but somewhat unsatisfying. We would really like to see how this all works out in the time domain.

Proof in the Time Domain

In Section 3.3 we constructed an explicit formula for the output of a Figure 1.1 scrambler, using our matrix formalism. Equation (3.3.5) shows the status of the state vector **q** at time t = n clocks:

$$\mathbf{q}_n = B^n \mathbf{q}_0 + \sum_{j=0}^{n-1} i_{n-1-j}\; B^j \quad \mathbf{1} \qquad (3.10.3)$$

Here $\mathbf{q}_0$ is the starting state vector, **1** is the unit column vector (1,0.0...) , and B is the matrix discussed in Section 3.3. According to Eq. (3.3.2), the state of register k-s is given by the sth component of the vector q_n. Thus we can write,

$$o_n = q_{k-s}(n) = [\mathbf{q}_n]_s = [B^n \mathbf{q}_0]_s + \sum_{j=0}^{n-1} i_{n-1-j}\; [B^j]_{s,1} \qquad (3.10.4)$$

Notice in Figure 1.1 that we can take the scrambled output from *any* of the k flip-flops, the only difference is a time delay. So we will therefore regard $q_{k-s}(n)$ as shown above as the output o_n of our scrambler. Eq. (3.10.4) is an explicit expression for the scrambler output sequence o_n in terms of the scrambler input sequence i_n , and the starting state vector $\mathbf{q}_0$.

During the first k clocks of operation, we know that the *descrambler* clocks out whatever k garbage bits happened to be in its k registers. During this time, the first k bits of the above sequence o_n move into the descrambler registers. We can use (3.10.2) above to compute the descrambler output after this time, assuming that its input is the output of the scrambler:

$$o'_{k+n} = \sum_{r=0}^{k} h_r \; o_{n+r} \qquad n \geq 0 \tag{3.10.5}$$

Our task is now "simply" to insert (3.10.4) with $n \rightarrow n+r$ into (3.10.5), and turn the crank.

The reader is warned that things are going to get a little tricky as we proceed, but in the end, the desired result will be obtained, and some insight will be gained along the way.

Doing the above insertion, we get

$$o'_{k+n} = \sum_{r=0}^{k} h_r \; \{ \; [B^{r+n} \, \mathbf{q}_0 \,]_s \; + \sum_{j=0}^{n+r-1} i^{n+r-1-j} \, [B^j]_{s,1} \; \} \tag{3.10.6}$$

Consider the first term above, which can be expanded as ,

$$\sum_{r=0}^{k} h_r [B^{r+n} \, \mathbf{q}_0 \,]_s = \sum_{i=0}^{k} [\mathbf{q}_0]_i \; \{ \sum_{r=0}^{k} h_r [B^{r+n}]_{s,i} \} = \sum_{i=0}^{k} [\mathbf{q}_0]_i \; \{0\} = 0 \tag{3.10.7}$$

According to Eq. (3.5.2), the sum over r {...} completely vanishes. This may be traced back to the fact that h(B) = 0. The reason this is true is that the matrix B has the "companion" form C_1 shown in Galois Chapter 8, Section 8.5, with $f_i = h_i$. The characteristic polynomial of C_1 is then h(x), and according to Big Fact 2 of Section 8.1, it follows that h(B) = 0. This is the Cayley-Hamilton Theorem.

Since the term containing $\mathbf{q}_0$ completely vanishes in (3.10.6), we conclude that the output sequence o'_{k+n} is *completely independent* of the starting state vector $\mathbf{q}_0$ of the scrambler!! This is the kind of result that is hard to deduce from the z-domain analysis.

We are now left with the following double sum:

$$o'_{k+n} = \sum_{r=0}^{k} h_r \sum_{j=0}^{n+r-1} i_{n+r-1-j} \; [\, B^j \,]_{s,1} \tag{3.10.8}$$

Replace the j summation variable with a new summation variable m = n+r-1-j :

$$o'_{k+n} = \sum_{r=0}^{k} h_r \sum_{m=0}^{n+r-1} i_m \, [B^{r+n-m-1}]_{s,1} \qquad (3.10.9)$$

The motivation for this less-than-obvious replacement was to get i_m. Consider now the shape of the region of the double summation in (3.10.8). Here is a picture:

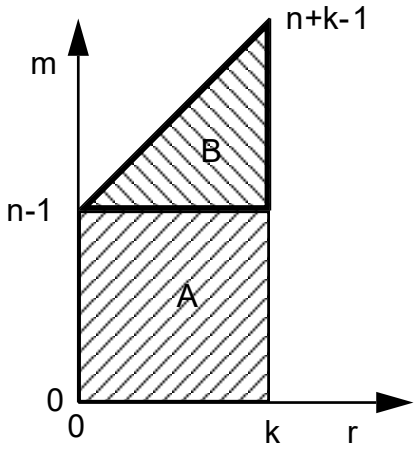


Figure 3.10.2: Summation domain for Eq. (3.10.9)

Consider first the summation over the rectangular region A. Here, r goes 0 to k, and m goes 0 to n-2. The line m=n-1 will be considered part of the upper triangle B. We get,

$$o'_{k+n} \text{ (over region A)} = \sum_{r=0}^{k} h_r \sum_{m=0}^{n-2} i_m \, [B^{r+n-m-1}]_{s,1} \qquad (3.10.10)$$

Because the upper endpoint on the m summation is now independent of r, we can move the r summation to the right where it collides with the B matrix thing to give zero just as happened in (3.10.7):

$$o'_{k+n} \text{ (over region A)} = \sum_{m=0}^{n-2} i_m \left\{ \sum_{r=0}^{k} h_r \, [B^{r+n-m-1}]_{s,1} \right\} = 0 \qquad (3.10.11)$$

Now we have only to worry about the triangular region A. So,

$$o'_{k+n} = o'_{k+n} \text{ (over region B)} = \sum_{r=0}^{k} h_r \sum_{m=n-1}^{n-2} i_m \, [B^{r+n-m-1}]_{s,1} \qquad (3.10.12)$$

The next step is to replace index m with j = m -n+1 which yields,

$$o'_{k+n} = \sum_{r=0}^{k} h_r \sum_{j=0}^{r} i_{j+n-1} \, [\, B^{r-j} \,]_{s,1} \qquad (3.10.13)$$

This time we cannot move in the r summation and have it die against the matrix because r appears as the upper endpoint of the j summation. The summation domain for (3.10.13) is as follows:

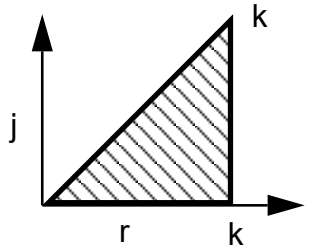


Figure 3.10.3: Summation domain for Eq. (3.10.13)

Based on this picture, we can rewrite the double summation as follows:

$$o'_{k+n} = \sum_{j=0}^{k} i_{j+n-1} \; \{ \sum_{r=j}^{k} h_r \, [\, B^{r-j} \,]_{s,1} \; \} \qquad (3.10.14)$$

In Appendix 3.1 at the end of this Chapter, we prove the following extremely un-obvious fact about the quantity in brackets {..} appearing above,

$$\{ \sum_{r=j}^{k} h_r \, [\, B^{r-j} \,]_{s,1} \; \} = \delta_{s,\, k-j+1} \qquad (3.10.15)$$

Basically this is a property of the B matrices. Inserting (3.10.15) into (3.10.14) we get our final result:

$$o'_{k+n} = \sum_{j=0}^{k} i_{j+n-1} \; \{ \, \delta_{s,\, k-j+1} \, \} = i_{k+n-s} \qquad (3.10.16)$$

Recall that we took our scrambler output from register q_{k-s} of the scrambler of Figure 1. If we now select s=k, the output comes from register q_0 exactly as shown in Figure 1. In this case, we get the result,

$$o'_{k+n} = i_n \qquad n = 0,1,2... \qquad (3.10.17)$$

Conclusions:

(a) Eq. (3.10.17) says that if you run a sequence i_n through a scrambler, and then through the corresponding descrambler, you get out exactly what you put in, with k clock delays due to the way the multiplier starts up. During this startup phase, k garbage bits are emitted by the descrambler.

(b) If you decide to take your scrambler output from an earlier register of Figure 1, then you will lose the first few bits of the input stream during the garbage clock out phase. For example, if (k-s) = 1, so we are taking data from q_1 in Figure 1, we get $o'_{k+n} = i_{n+1}$, so the first non-garbage bit out of the descrambler will be i_1 and i_0 is lost.

(c) Conversely, suppose there are N extra latch delays inserted into the "line" of Figure 3.10.1. The N garbage bits held in these latches will appear right after the k garbage bits are shifted out of the descrambler latches. Then following these N extra garbage bits will come the input sequence starting with the first bit i_0, so nothing is lost.

(d) In general, nobody cares whether the first few bits are lost as in (b), or whether there are a few extra startup garbage bits as in (c). The only possible concern is in the overall delay which is N-(k-s). An interesting idea is to consider taking the scrambler output a few latches back from the end (ie, advanced) and attempt to cancel out any line delay N. That is, one could tune so that N - (k-s) = 0.

(e) Notice that in the above proof we assumed use of the Figure 1.1 scrambler with its B matrices, but we made no assumption about whether Figure 7 or Figure 8 was used as the descrambler.

3.11 The Kill Sequence Problem and Strings of Zeros

It has been noted that a long string of zeros emerging from the Main Scrambler shown in Figure 3.10.1 can cause a problem at the clock recovery circuit which must exist at the end of the "line". Unlike strings of 1's, strings of 0's pass right through the NRZI mini-scrambler and enter the line.

Question: What type of source data can cause the main scrambler to output a long string of zeros?

Answer: At any instant of time, the main scrambler contains some state vector q_n. As will be shown below, for any such state vector, there is a unique corresponding "kill vector" which, if shifted into the scrambler, will bring it to the all-zero state. Subsequent zeros at the input keep the scrambler in its zero state. This leads us to the following:

Fact 1: It is possible that a k-stage scrambler will output a string of N+k zeros if a string of N zeros exists in the input stream. The probability of this happening, given the input stream of N zeros, and assuming the scrambler is in a random state, is 1/P where $P = 2^k - 1$. That's just because P is the number of states the scrambler can be in. For k=9, this probability is 1/511, which is not a very small number in this context.

Proof: Imagine that the input data stream has the following form:

```
....xxxxxxxxx100000000000000...00000000001xxxxxx...
     ↑
```

Suppose that, at the instant in time indicated by the arrow, the scrambler happens to contain exactly the right k-bit vector **a** which gets "killed" by the next k bits of the input stream (underlined). Then, when the first 0 of the long string of zeros shown is at the scrambler input, the scrambler will contain k zeros. Suppose the string of zeros shown above has length N. By the time the "1" which ends the string is at the scrambler input, it will have output N zeros. But its state vector is still 0, so there are k more zeros still to come out. Thus, if N zeros are input, it is possible that N+k zeros will be output.

Example 1: Suppose in D1 video there is a sync pattern consisting of the following four 10-bit words :

3FF, 000, 000, 000 001

Suppose this pattern is followed by a Y or a C sample, with an unknown bit ordering. In the very worst case (probably a few bits worse than could really happen), this pattern could be in effect a 1, and we have added that above. Thus, we have a string of 39 bits (in 10-bit video), and with a k=9 scrambler, there could be a string of 48 zeros in the scrambler output. [We address below the question of what vector is killed by the all-ones input vector.] In 8-bit video, we would lower this estimate from 30+9+9 = 48 to 24 + 7 + 9 = 40.

In 8-bit video, the normal blanking pattern is YC = 0x1080, so the longest zero strings are 10 bits. In 10-bit video, we presume this extends to 12 bits.

Example 2: Suppose in some video spec, one is allowed to have an entire active scanline of ancillary data that is all zeros. In the 601 world with 10-bit video, this would represent a string of 720*20 = 14, 400. Zeros. To this we might add 30 + 9 as above, but who is counting? In HDTV we get 1920*20 = 38400 zeros.

We now prove some of the claims referred to above.

Fact 2: Assume a scrambler is in some state **a**. Consider the k-symbol pattern that next shifts in, call this pattern **i**. After this pattern shifts in, the scrambler is in state **b**. We claim that, given any patterns **a** and **b**, there exists a unique pattern **i** which causes **a** to go to **b**.

Corollary 2: If pattern **b** is the all-zero pattern **0**, there exists a unique pattern **i** which takes **a** to **0**.

Definition: Such a pattern is called a *kill vector*.

Proof of Fact: Our proof makes no assumption about the nature of the symbols carried on the lines of Figure 1. They could be in GF(2), in GF(p), or in GF(p^m). To keep the proof general, we retain the minus signs on the h_i coefficients.

Stare at Figure 1. Assume that at t=0 pattern **a** is in the registers. The rightmost bit of pattern **a** is a_0. Assume the first incoming bit is called i_0. We select i_0 such that the resulting contents of register q_{k-1} will be the rightmost bit b_0 of pattern **b**. This is because b_0 sitting in register q_{k-1} will end up in register q_0 after k clocks.

A clock goes by and we now have some i_1 at the input. The feedback circuit is generating some feedback based on the remaining (shifted) bits of pattern **a** and our b_0 which is sitting in q_{k-1}. We now select i_1 so that when combined with the feedback, it generates the second-from-the-right bit b_1 of pattern **b**.

We keep going in this fashion, and each bit is in turn fully determined. Here are some equations:

$i_0 - h_0 a_0 - h_1 a_1 - \ldots h_{k-1} a_{k-1} = b_0$ solve for i_0

$i_1 - h_0 a_1 - h_1 a_2 - \ldots h_{k-2} a_{k-1} - h_{k-1} b_0 = b_1$ solve for i_1

$i_2 - h_0 a_2 - h_1 a_3 - \ldots h_{k-2} b_0 - h_{k-1} b_1 = b_2$ solve for i_2

etc.

We can group all these equations into a matrix solution for the $\{i_n\}$ as follows:

$$\begin{vmatrix} i_0 \\ i_1 \\ i_2 \\ i_3 \\ i_4 \\ \ldots \\ i_{k-2} \\ i_{k-1} \end{vmatrix} = \begin{vmatrix} a_0 & a_1 & a_2 & a_3 & \ldots & a_{k-3} & a_{k-2} & a_{k-1} \\ a_1 & a_2 & a_3 & a_4 & \ldots & a_{k-2} & a_{k-1} & b_0 \\ a_2 & a_3 & a_4 & a_5 & \ldots & a_{k-1} & b_0 & b_1 \\ a_3 & a_4 & a_5 & a_6 & \ldots & b_0 & b_1 & b_2 \\ a_4 & a_5 & a_6 & a_7 & \ldots & b_1 & b_2 & b_3 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{k-2} & a_{k-1} & b_0 & b_1 & \ldots & b_{k-5} & b_{k-4} & b_{k-3} \\ a_{k-1} & b_0 & b_1 & b_2 & \ldots & b_{k-4} & b_{k-3} & b_{k-2} \end{vmatrix} \begin{vmatrix} -h_0 \\ -h_1 \\ -h_2 \\ -h_3 \\ -h_4 \\ \ldots \\ -h_{k-2} \\ -h_{k-1} \end{vmatrix} + \begin{vmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ \ldots \\ b_{k-2} \\ b_{k-1} \end{vmatrix}$$

Notice that all elements on any / diagonal are the same. We can write this in vector form as:

$$\mathbf{i} = -X(\mathbf{a},\mathbf{b})\,\mathbf{h} + \mathbf{b}$$

where X is the matrix shown, and we note that it is a function of vectors **a** and **b**.

Thus, we have explicitly constructed the input vector **i** which converts state **a** into state **b**. If we are looking for a kill vector, we set **b** = 0, so all elements in X below the diagonal vanish, as does the add-on column vector on the right. Since we have an explicit answer for **i**, it must be unique, given **a** and **b**.

Example: For k=9, X is a 9x9 matrix, and for $h(x) = 1 + x^4 + x^9$ we have $h_0 = h_4 = 1$, all other h_i in the vector **h** vanish. In this case, each equation has only three terms. Here they are:

$i_0 = a_0 + a_4 + b_0$
$i_1 = a_1 + a_5 + b_1$
$i_2 = a_2 + a_6 + b_2$
$i_3 = a_3 + a_7 + b_3$
$i_4 = a_4 + a_8 + b_4$
$i_5 = a_5 + b_0 + b_5$
$i_6 = a_6 + b_1 + b_6$
$i_7 = a_7 + b_2 + b_7$
$i_8 = a_8 + b_3 + b_8$

This is the general result for going from **a** to **b**. To find the *kill vector*, set $b_i = 0$ to get:

$$i_0 = a_0 + a_4 \qquad i_5 = a_5$$
$$i_1 = a_1 + a_5 \qquad i_6 = a_6$$
$$i_2 = a_2 + a_6 \qquad i_7 = a_7$$
$$i_3 = a_3 + a_7 \qquad i_8 = a_8$$
$$i_4 = a_4 + a_8$$

Question: In the above example k = 9, what vector **a** is killed by an incoming vector consisting of all ones?

Answer: Reverse solve the last set of equations above. We quickly get that $a_8 = a_7 = a_6 = a_5 = 1$. Then from the earlier equations we get $a_4 = a_3 = a_2 = a_1 = 0$. Finally, the first equation then says $a_0 = 1$. The answer to the question is therefore:

$$\{ a_8, a_7, a_5. a_4, a_3, a_2. a_1, a_0 \} = \{ 1,1,1,1,0,0,0,0,1\} = 0x1E1$$

Discussion of the zeros problem.

In general, for each video standard, one must find the spec, and study the longest possible string of zeros that is allowed. Officially illegal values do sometimes occur, and there are all kinds of EAV, SAV codes and ancillary data and various indices to worry about, plus embedded sound. The general principle applies: "If it can happen, it will happen." If the clock recovery circuit drops out of lock during a long string of zeros, there will be some period of time determined by the PLL system during which data will be lost. If the lock-up time is very short, perhaps only a few bits at the end of an anomalous scan line will be lost, but this sounds unhealthy. It seems that a long time constant would be better than a short one, so you could live through an anomalous event without losing lock. A time constant that is long compared to a scanline but short compared to a field time would be good. In this case, one could survive an anomalous scanline.

The following fact does not really belong here, but is was omitted from earlier work.

Fact 3: If a k-stage scrambler has an even number of feedback taps, then if the input stream is set to all 1's, the scrambler behaves exactly as if the input stream were all 0's and there were an inverter on the final output.

Thus, a primitive polynomial scrambler with all 1's for input will cycle through an inverted characteristic sequence of the usual length $P = 2^k - 1$. The only pattern that does not appear at the output is then the pattern consisting of k ones. All characteristic sequence theorems quoted earlier then need to be adjusted to account to the effective inversion.

Proof: The input feed of all ones at the input XOR gate can be thought of as an inverter at the input of the first flip flop and an input stream of zeros. We can then mentally move this inverter to the right in Figure 1.1 one step at a time. At the output of a flip flop, the inverter "forks" into a pair of inverters, one

moves to the next flip-flop, and one moves up the feedback path. The lower inverter eventually ends up on the output signal. If there are an even number of feedback paths, then the feedback forked inverters all cancel out. Each such inverter can be regarded as adding a 1, and if you add a 1 an even number of times, it is like doing nothing.

Some final notes:

(1) Long strings of zeros might have some bad effect on a dynamic equalizer.

(2) For each extra bit that is added to a scrambler, the probability of an anomalous event is cut in half. Inside a custom chip, having 9 registers or having 19 in a scrambler does not make much difference, but the 19 bit scrambler has the "zeros problem" reduced by a factor of $2^{-10} = 10^{-3}$.

(3) There is no data propagation problem to speak of with the descrambler. If a data loss occurs, the descrambler will "resynchronize" in k clock periods. In other words, if any garbage bits get into the descrambler, they do not have any long term effect.

Appendix 3.1: Proof of Eq. (3.10.15)

First, we state what we want to prove is true:

$$\{ \sum_{r=j}^{k} h_r \quad [B^{r-j}]_{s,1} \} \quad = \delta_{s,\ k-j+1} \tag{A.1}$$

The matrix B which goes with the Figure 1.1 circuit is shown in Eq. (3.3.2). We can write its elements as,

$$B_{ij} = \delta_{i,\ j+1} + \delta_{i,1}\, h_{k-j} \tag{A.2}$$

We are always working in the GF(2) binary world where - = +. Also, since we are assuming the h(x) is order k, we must have $h_k = 1$.

We will prove (A.1) by induction.

(a) To start, we show that (A.1) is true for j = k. In this case, the LHS of (A.1) has one term and (A.1) says,

$$h_k\, I_{s,1} = \delta_{s,\ 1}$$

But this is trivially true since $h_k = 1$ and I = identity matrix = B^0. So, we are off and running.

(b) Now we show that if (A.1) is true for some j, then it is also true for j-1 . Start with (A.1), and apply the operation $\sum_{s=0}^{k} [B]_{n,s}$ to both sides. On the LHS of (A.1), this new B matrix glues onto the B^{r-j} matrix already present to give $B^{r-(j-1)}$. On the RHS the delta picks out the value s = k-j+1 for the second index of B, so we get:

$$\{ \sum_{r=j}^{k} h_r \quad [B^{r-(j-1)}]_{n,1} \} \quad = B_{n,k-j+1} \quad = \delta_{n,\ k-(j-1)\ +1} \quad + \delta_{n,1}\, h_{j-1} \tag{A.3}$$

where we have used (A.2) to write out the B matrix element. Now, if the sum on the LHS of (A.3) *had* a term for r = j-1, here is what that term would be:

$$h_{j-1}\, [B^0]_{n,1} = h_{j-1}\delta_{n,1}$$

But this is the second term on the right of (A.3). So (A.3) now becomes,

$$\{ \sum_{r=j-1}^{k} h_r \quad [B^{r-(j-1)}]_{n,1} \} \quad = \delta_{n,\ k-(j-1)\ +1} \tag{A.4}$$

If we now set n=s, (A.4) is the same as (A.1) with j replaced by j-1. We have therefore shown that (A.1) implies (A.4), which completes the second part of our induction proof.

Q.E.D.

References

[31] P. Lucht, *Galois Fields and Cyclic Codes* (web "phil lucht documents", 2013).

[32] P. Lucht, *Fourier Transforms and their Application to Pulse Amplitude Modulated Signals* (web "phil lucht documents", 2013).

[33] ANSI/SMPTE 259M-1997 Serial Digital Interface (SDI) standard for digital television signals passing through a coaxial cable (sdi.org.ru/smpte-259m.pdf).