Polynomial Multipliers and Dividers, Shift Register Generators and Scramblers

Phil Lucht
Rimrock Digital Technology, Salt Lake City, Utah 84103
last update: Aug 31, 2013
rimrock@xmission.com

Maple code is available upon request. Comments and errata are welcome.

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

Overview

This monograph describes the theory underlying certain commonly used digital hardware circuits known as polynomial multipliers and dividers. A divider with no input is called a shift register generator, and one with input is known as a scrambler. This latter entity is usually restricted to the case where the fixed divisor polynomial is a "primitive polynomial" of an associated Galois Field.

Just as the rodeo cowgirl stands upon two horses as they prance around a corral, the topics addressed in the current document stand upon two mathematical pillars. A student of these topics can experience the same wobbly uncertainty felt by the cowgirl, and it is not hard to fall off and give up.

The first pillar is the subject of finite fields known as Galois Fields in honor of Évariste Galois. The polynomial processing circuits are interesting in their own right and are not too hard to analyze in either the time domain or the frequency domain (the Z transform domain). Far more interesting is the connection between these circuits and the Galois Fields $GF(p^k)$. This field connection allows certain conclusions to be reached concerning the behavior of polynomial processors which would be very difficult to ascertain by other means. Unfortunately, the subject of Galois Fields is somewhat obscure and often does not appear in the main line curriculum of the undergraduate student in engineering, communications or computer science, though it does appear wherever cyclic codes are discussed. As an aid to the reader, the author has written a separate set of notes on this subject which is referred to in this document as GA, see References.

The second pillar is the universe of topics associated with the Fourier Transform and its digital descendents including the Z Transform. The whole concept of a polynomial processor is that it processes polynomials in the variable z of the Z transform. The circuits are simple to analyze in this "z domain", but in the real world we are also interested in time-domain behavior, and these domains are connected by the Convolution Theorem of Fourier theory. Another Fourier connection involves the ω-domain frequency spectrum and the spectral power density of the output signal of a shift register generator or scrambler, and just understanding the origin of spectral power density formulas is a whole separate effort. In a separate document referred to as FT the author has provided some notes on these matters as well, again see References.

Summary

There is a lot going on in each of the three chapters of this document, and the summaries below are correspondingly fairly detailed. A briefer overview is provided by the Table of Contents.

Chapter 1 : Polynomial Processors

Chapter 1 opens by showing the four basic hardware polynomial processors of interest. The Type A divider and multiplier have outboard adders, while the Type B have inboard adders. The input and output symbol streams are serial streams. More costly parallel implementations, though certainly possible, are not treated.

Section 1.1 notes that any "polynomial" F(z) is the Z Transform of a sequence f_n, so the latter is the **time-domain** representation of a signal while the polynomial is the **z-domain** representation. Certain rules are noted, such as $f_n \leftrightarrow F(z) \Leftrightarrow f_{n-m} \leftrightarrow z^{-m} F(z)$.

DIVIDERS

In **Section 1.2** the **Type A** divider is drawn in Fig 1.1 and the lines are interpreted as carrying **symbols** which could be bits or bytes or something else. In most of the paper, these symbols are treated as elements of the **Galois Field GF(p)** rather than $GF(p^m)$. Normal bits are then elements of GF(2). By starting with a simple time-domain description of the divider, the fact that O(z) = I(z)/H(z) is quickly obtained, verifying that Fig 1.1 in fact implements a polynomial divider. Here I(z) is a possibly infinite polynomial whose coefficients are the symbols i_n which are clocked into Fig 1.1, while O(z) is the normally infinite polynomial whose coefficients are the symbols o_n which are clocked out. H(z) is a polynomial of fixed coefficients h_i which are part of the Fig 1.1 circuit. A distinction is made between **proper** polynomials which have non-negative powers only, and **improper** polynomials which can include negative integer powers. Whereas the divisor H(z) is a proper polynomial of degree k, I(z) and O(z) are improper polynomials each of which in general has an infinite number of terms. Looking at the "quotient" O(z) and its infinite number of terms, one wonders "where is the remainder?" It is seen that if improper polynomials are allowed, the notion of a "remainder" is completely arbitrary and depends on "where you stop" in the division process. Examples of this situation are presented.

Section 1.3 compares the infinite symbol sequence of the result of a polynomial division to the infinite sequence of digits in the division of two real numbers. The output of a polynomial divider is then considered after a fixed number r of input symbols are processed. In this case, the division process can be thought of as $O_{r-k}(z) = I_r(z)/H(z)$ where $I_r(z)$ and H(z) are "proper", but $O_{r-k}(z)$ is still an improper polynomial. One can then write $O_{r-k}(z) = \mathcal{Q}(z) + R(z)/H(z)$ where the quotient $\mathcal{Q}(z)$ is a proper polynomial of degree r-k and the remainder R(z) is a proper polynomial of degree < k. At this point, if another input symbol is processed, the interpretation changes to accommodate r+1 input symbols, and we have a whole new division problem to interpret. In general we find $I_r(z)/H(z) = \mathcal{Q}^{(r)}(z) + R^{(r)}(z)/H(z)$ so as the symbols come in, we get in fact a sequence of quotients and a **sequence of remainders**. Even if the input sequence vanishes after some point, the quotient and remainder sequences continue on in general forever. One is normally used to a division having a unique quotient and remainder, but as the input symbols come in, the division problem keeps changing and so too do the quotient and remainder. It turns out that in the Type A divider, the remainder is not "visible" and in fact is a function of both the register state and some symbols already shifted out as coefficients of O(z). Finally, it is noted that 1/H(z) is the **impulse response** of the divider, since in this case I(z) = 1, meaning the divider is pulsed by a single unity symbol at time t = 0. If the divider were interpreted as a filter, 1/H(z) would be that filter's **transfer function**.

Section 1.4 repeats the above analysis for the **Type B** divider of Fig 1.5 where the adders are "inboard". One again finds that O(z) = I(z)/H(z).

Section 1.5 studies *how* the two kinds of dividers actually "work". The action of each circuit on each clock is related to the process of "grade school division" of polynomials carried out just as one divides numbers by "long division". In either Type A or Type B, there is a "current dividend" at each step. It is

seen that for the Type B divider, the sequence of current dividends is in fact the sequence of remainders discussed just above. For this kind of divider, the remainder is then "visible" and is precisely the state of the registers.

MULTIPLIERS

Section 1.6 addresses the **Type A** polynomial multiplier, Fig 1.8. It is shown that z^k O(z) = H(z) I(z), so apart from the factor z^k, the circuit really does multiply two polynomials. An interpretation of the form $O_{r+k}(z) = H(z)\ I_r(z)$ is then provided in which all three polynomials are "proper". The impulse response is z^k O(z) = H(z). The factor z^k is associated with a simple time shift of the output by k clocks.

Section 1.7 then treats the **Type B** multiplier, and z^k O(z) = H(z) I(z) is again obtained, as is the same proper polynomial interpretation $O_{r+k}(z) = H(z)\ I_r(z)$ and the same impulse response z^k O(z) = H(z).

Section 1.8 shows how these multipliers "work" in terms of "grade school multiplication" of polynomials. For both the dividers and multipliers, we see how the inboard and outboard adder topology affects the method by which contributions are added up to get the desired output. Up to this point, almost everything has been in the z-domain, since this is where the polynomials of the polynomial processors live.

In **Section 1.9**, the circuits are analyzed instead in the **time domain**. The Type A multiplier is treated first, giving a time-domain equation $o_{k+n} = \Sigma_{j=0}^{k}\ h_j i_{n+j}$. The Type A divider is done next, with result $i_n = \Sigma_{j=0}^{k}\ h_j o_{n+j}$ which is a difference equation for the output o_k. Section (c) shows how the Type A time-domain and z-domain equations are consistent with a Z Transform theorem known as **The Convolution Theorem**. Since the Type A and Type B z-domain equations are the same, it is concluded from this theorem that the Type B time-domain results must be the same as for the Type A, so there is no reason to separately compute the Type B time-domain equations.

At this point the basic results are summarized in box (1.9.8).

BOTH AT ONCE

Section 1.10 then considers a circuit Fig 1.13 which does *simultaneous* multiplication of an input polynomial I(z) by polynomial H(z) and division by another polynomial G(z). The result of this combined operation is O(z) = I(z)H(z)/G(z). It is shown how the limits H(z) = 1 and G(z) = z^k reproduce the results of the separate divider and multiplier circuits treated earlier.

Section 1.11 shows how the combined circuit of the previous section is used in **CRC** error detection.

Chapter 2: Shift Register Generators

Section 2.1 defines a state machine and its state vector period N. A shift register generator is defined to be either the Type A or Type B polynomial divider studied in Chapter 1 in which the input stream is set to 0. A shift register generator is an example of a state machine with no inputs and one output. It has k registers each of which holds a symbol of GF(p^m), but in the next section m is restricted to m = 1.

Section 2.2 studies the evolution of the state vector of a Type B shift register generator and seeks to learn about the nature of the state vector period.

(a) The k register values (the state) of a Type B shift register generator are encoded as coefficients of a polynomial q(x), and it is shown how the state updates over one clock period: $q'(x) = x\,q(x) - o\,h(x) + i$.

(b) Our interest from this point on is limited to registers which contain symbols in GF(p). Since there are k registers each holding a GF(p) symbol, the set of registers may be regarded as an element of $GF(p^k)$.

(c) Certain facts about **Galois Fields** $GF(p^k)$ are reviewed.

(d) The initial encoded state vector q(x) is identified with an element β_0 of $GF(p^k)$ and after evolving for s clocks it is shown that $\beta_s = \alpha^s \beta_0$ (the Galois Iterator), where α is a different element of $GF(p^k)$. It is shown that if divisor h(x) is chosen to be a primitive polynomial of $GF(p^k)$, then the state vector period is the maximal possible value $p^k - 1$ since α is then a primitive element of $GF(p^k)$.

(e) The state vector after s clocks is identified with a polynomial division remainder after s clocks. The period n of h(x) is defined as the smallest integer n for which $(x^n-1)/h(x)$ is a polynomial. It is shown using the remainder idea that the Type B state vector period N is equal to this period n of h(x).

Section 2.3 studies the output and the output period of a shift register generator of either type. It is shown that the output sequence $\{o_j\}$ of such a generator satisfies the time-domain equation $\Sigma_{j=0}^{k} h_j o_{n+j} = 0$ for n = 0,1.2... . The solution sequence $\{o_j\}$ of this equation is shown to have interesting properties. There are exactly p^k-1 non-zero solutions, and all solutions can be written as $\{o_i\}$ = {**c**,**c**,**c** ...} where **c** is an n-symbol code word of the **cyclic code** generated by $g(x) = (x^n-1)/h(x)$ where n is the period of h(x). Since cyclic code words have simple properties (any rotation of **c** is another code word and the sum of two code words is a code word), it is easy to find the exact solutions $\{o_i\}$. The case $GF(2^4)$ is treated as a detailed example. It is shown that, if h(x) is a primitive polynomial, then $n = p^k-1$ and all of the p^k-1 solutions $\{o_j\}$ have the same period $P = p^k-1$, and that in fact all solutions are just time-shifts of a single sequence which is called a **characteristic sequence** or **maximal length sequence** (MLS). A 511-bit MLS sequence is displayed for $GF(2^9)$. For a Type A generator, it is shown that the output period P is the same as the state vector period N. The notion of a **sift list** associated with an MLS sequence is defined and it is shown that in one period of an MLS sequence, each symbol appears k times in the associated sift list.

Section 2.4 studies the properties of an MLS sequence produced by any shift register generator whose h(x) is a primitive polynomial of $GF(p^k)$. Any particular k-long sequence can start only once in an MLS period. The largest run of zeros is length k-1, while that of any non-zero symbol is k, and all these runs occur only once per MLS period. In an MLS sequence, 0 occurs p^{k-1}-1 times while each non-zero symbol occurs p^{k-1} times. An MLS sequence differs from a non-trivial shifted version of itself in $(p-1)p^{k-1}$ places. In a small table (2.4.8), the sum and product of an MLS sequence with a shifted version of itself are studied. It is claimed without proof (which comes in the next section) that for reasonably large k, the binary MLS sequence is very close to a white sequence for which the probability of finding runs of length n of either symbol is given by $P(n) = 1/2^n$. The error between a white sequence and an MLS sequence is estimated.

Section 2.5

(a) states the spectral power density of a **statistical pulse train** whose amplitudes form a white sequence. This result is quoted from FT for a general pulse shape and is then specialized to the box pulse shape. The spectrum is stated for symbols in both {1,0} and {1,-1}.

(b) defines the **autocorrelation** sequence r_s for a set of pulse train amplitudes y_n and then quotes a theorem from FT concerning infinite sequences composed of a repeating subsequence of length P. The theorem states the corresponding pulse train's spectral power density provided certain conditions are met concerning r_s.

(c) computes the autocorrelation sequence r_s for an MLS sequence with symbols in {1,0}. It is shown that this r_s meets the conditions of the theorem of the previous section, and then the spectral power density is stated for an MLS sequence, first for a general pulse shape and then for the box pulse shape. Since the pulse train amplitude sequence is periodic (with period P), the spectrum is entirely discrete and has an interesting structure which is plotted in Fig 2.8. As $P \rightarrow \infty$, the spectral lines (apart from the DC line) coalesce into a continuum which is the white sequence spectrum.

(d) repeats the previous section for symbols in {1,-1} and then compares the results of the two sections.

(e) displays a table summarizing the statistics for uncorrelated, MLS and white sequences for both {1,0} and {1,-1} cases, and then calculates the MLS correlation in each case, showing explicitly how MLS sequences are in fact *not* uncorrelated. Finally, the autocorrelation sequence for each case is plotted.

Chapter 3: The Matrix Approach to Scramblers

Section 3.1 reviews the developments of previous sections.

Section 3.2 then develops a linear-algebra style solution for the state vector of a Type B divider given an arbitrary initial state vector and an arbitrary input sequence. The solution involves powers of the kxk **companion matrix** B associated with the divider's primitive polynomial h(x). A connection is made between this solution and a solution previously obtained in Section 2.2 in terms of abstract GF(p^k) Galois Field elements. As noted earlier, the state vector with k elements in GF(p) is interpreted as a GF(p^k) field element.

Section 3.3 finds the corresponding solution of the Type A polynomial divider using an alternative companion matrix A. Later these matrices A and B are generically called C.

Section 3.4 revisits the topic of shift register generators broached in Chapter 2 and develops some new results based on the matrix formalism of the previous sections. It is found that each register of either type of shift register generator cycles through the MLS sequence if h(x) is a primitive polynomial of GF(p^k).

Section 3.5 formally defines a **scrambler**, then comments on its relation to **spread spectrum** communication. It is shown that each matrix element of a matrix representation of a Galois Field based on some primitive field element cycles through the MLS sequence described in Chapter 2. The remainder of the section describes how a scrambler "whitens" the statistics of its input stream.

Section 3.6 then studies the **spectral power density** of a scrambler output in the limit of a very long MLS period. The power density of the scrambler output is "white" and effectively continuous, and this is compared to the discrete spectral power density of a simple pulse train having the same pulse period T_1.

Section 3.7 discusses the NRZI **mini-scrambler** which is a polynomial divider with a single register. This scrambler when appended to a longer one produces an output signal which is non-polar. This section then ends with a summary of scrambler results.

Section 3.8 mimics Section 3.3, finding a linear-algebra solution for the state vector of a Type A polynomial *multiplier* (Section 3.3 did the divider). A new matrix D appears. As a check, the output sequence of the multiplier is computed in this matrix formalism and agrees with a result found earlier.

Section 3.9 repeats this process for the Type B multiplier and the same matrix D appears.

Section 3.10 makes use of the matrix solution found in Section 3.3 for the Type A divider (scrambler) and shows that a **descrambler** really does descramble the output of a scrambler. This fact is trivial to show in the z domain, but is not so easy to show directly in the time domain. This fact is then applied to a standard form of scrambling involving both a "main scrambler" and the NRZI "mini-scrambler".

Section 3.11 notes that a mini-scrambler passes through strings of zeros, and one is then concerned with the length of strings of zeros that can be created by the main scrambler. The concern is that long strings of zeros stress a receiver's **clock recovery** circuitry and can possibly throw it out of lock resulting in data loss. The worse-case situation occurs when the scrambler's input stream forms a **kill vector** for the current scrambler state, throwing the scrambler into the all-zeros state. Examples of strings of zeros are considered in some serial digital video standards. For a given scrambler state, the kill vector is calculated using simple matrix methods. The section concludes with comments on the effect of long strings of zeros including a certain problem known as "video tilt".

Appendix A explores an analogy between real numbers expressed in decimal notation and polynomials whose coefficients lie in Mod(10) = Z_{10}.

Appendix B proves a certain claim made in (2.3.8) concerning the output sequence of a shift register generator.

Appendix C presents E.J. Watson's list of primitive polynomials for GF(2^k) along with other lists.

Appendix D proves a certain sum rule (3.10.15) involving the companion matrix A and the coefficients of the divisor polynomial h(x).

Chapter 1: Polynomial Processors

In this Chapter we deal with two types of polynomial dividers and two types of polynomial multipliers, all implemented in hardware. Below is a preview of these four designs. The Type A circuits have "outboard adders" while the Type B circuits have "inboard adders". The Type B circuits are better for high-speed operation since they don't have multiple adder combinatoric propagation delays.

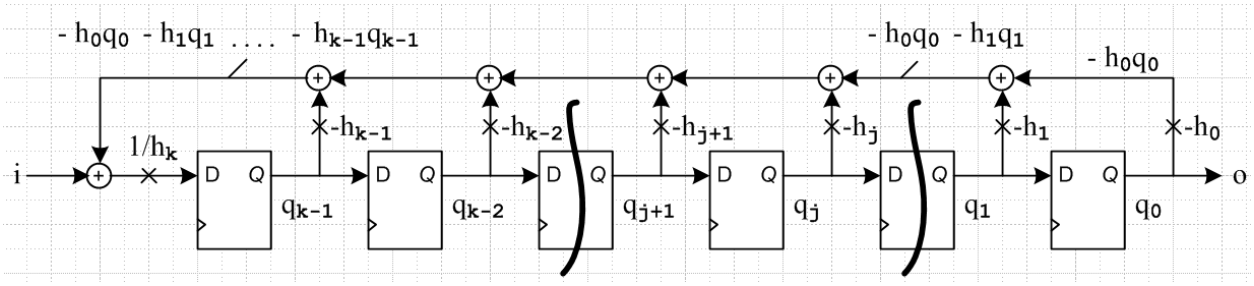


Fig 1.1: Type A polynomial divider.

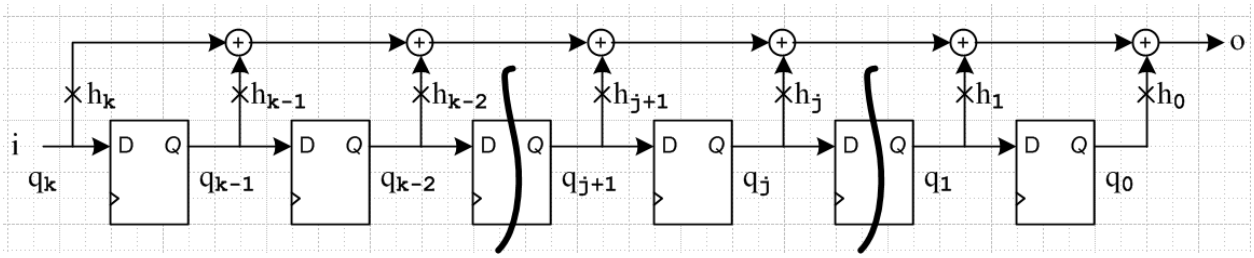


Fig 1.8: Type A polynomial multiplier.

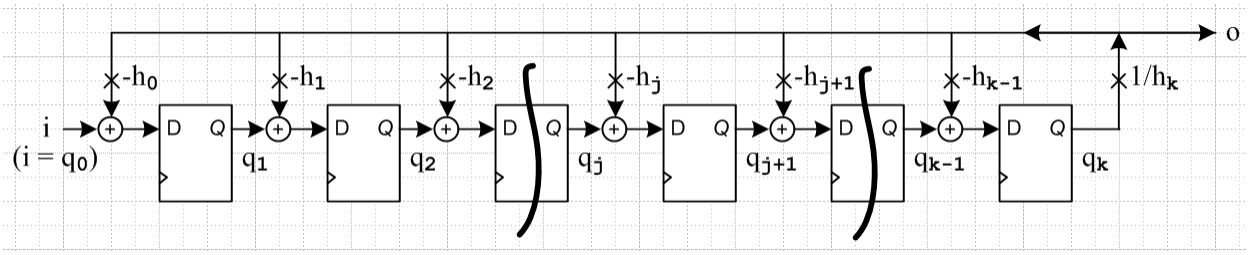


Fig 1.5. Type B polynomial divider.

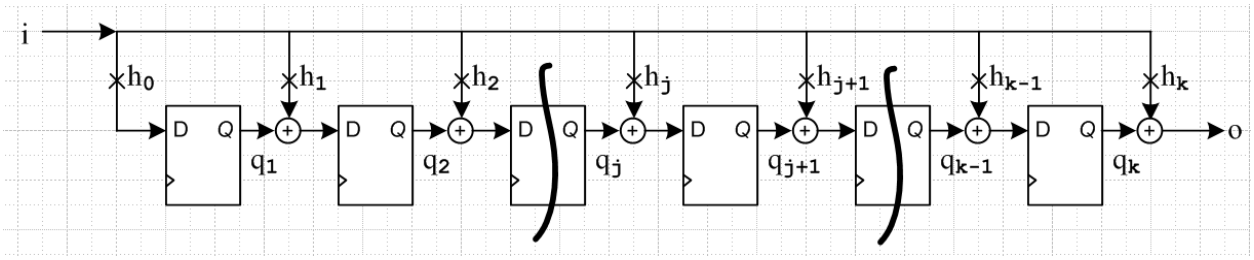


Fig 1.10: Type B polynomial multiplier.

1.1 Review of the Z Transform

The Z Transform is discussed in FT Chapter 3, 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")

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

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

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

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

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

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:

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

We can summarize these results as follows:

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

(1.1.4)

1.2 The Type A Polynomial Divider

Consider the following circuit:

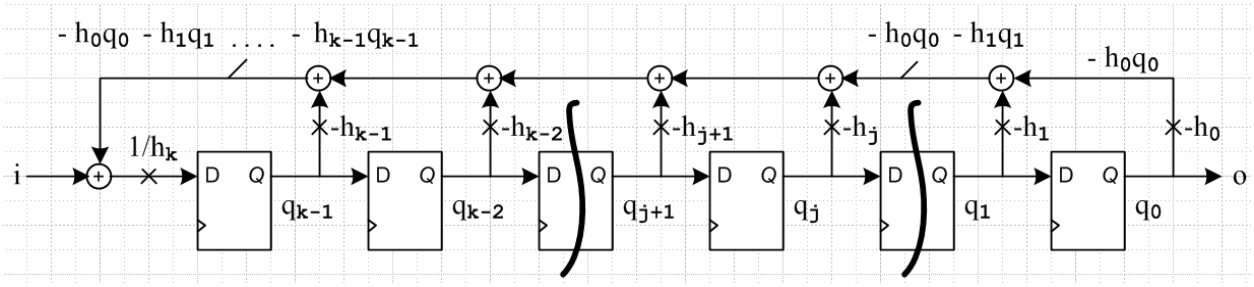


Fig 1.1: Type A polynomial divider.

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.

(a) 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 GA Chap 1 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 GA, 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. Our symbols are still 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 GA. 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.

(b) Analysis of the Type A Divider in the z-domain

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

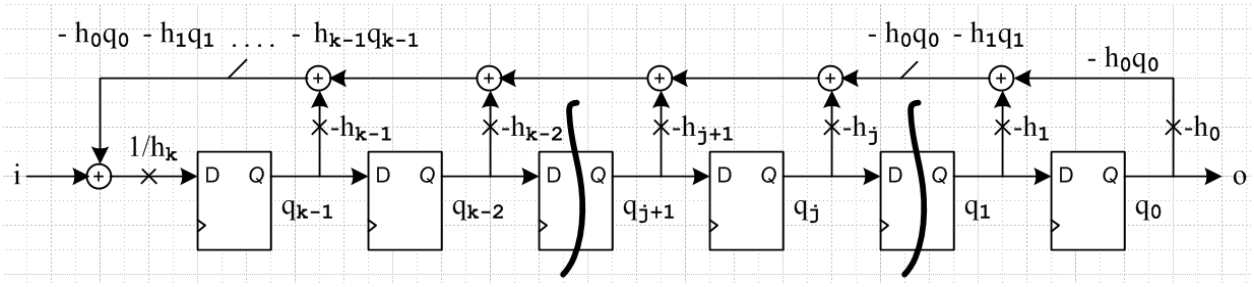


Fig 1.1: Type A polynomial divider.

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 $t = n\Delta t$ 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,

$$z\, Q_{k-1}(z) = D_{k-1}(z) = (1/h_k)\left[\, I(z) - \sum_{j=0}^{k-1} h_j\, Q_j(z) \,\right] \quad . \tag{1.2.2}$$

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

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

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

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

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,

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

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,

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

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

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

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

$$O(z) = I(z) \, / \, H(z) \tag{1.2.5}$$

where

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

We have therefore arrived at the undeniable conclusion that the circuit shown in Fig 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.

(c) 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 Fig 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 meaningful 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, and O(z) is an infinite polynomial, by which we mean it has an infinite number of terms.

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

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

$$= [i_0 + i_1 z^{-1} + i_2 z^{-2} + ... + i_r z^{-r}] / [h_k z^k + h_{k-1} z^{k-1} + ... + h_1 z + h_0] . \qquad (1.2.7)$$

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:

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

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

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

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

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

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

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 $(z^2+1) / (z+1)$. The result in this case is

$$\frac{z^2+1}{z+1} = (z - 1 + 2z^{-1} - 2z^{-2} + 2z^{-3} - 2z^{-4} + ...) \qquad (1.2.8)$$

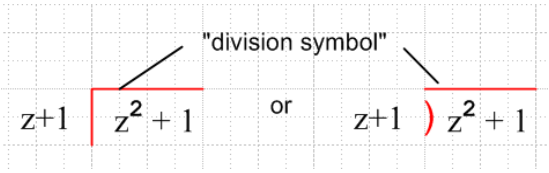
which can be verified as follows:

$$z - 1 + 2z^{-1} - 2z^{-2} + 2z^{-3} - 2z^{-4} + ... = (z-1) + 2\,(z^{-1} - z^{-2} + z^{-3} - z^{-4} + ...)$$

$$= (z-1) + 2z^{-1}\,[1 + (-z)^{-1} + (-z)^{-2} + (-z)^{-3} + ...] \quad = (z-1) + 2z^{-1}\ [\ 1/(1+z^{-1})\ ]$$

$$= (z-1) + 2\ [\ 1/(z+1)\ ] \quad = [\ z^2 -1 + 2\ ]\ /(z+1) \ = \ (z^2+1)/(z+1\ ).$$

Footnote: The "division symbol" used above seems to have no official name:



Some refer to it as a tableau, but surely that word really means the whole layout or "table" that one creates when doing a long division.

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

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

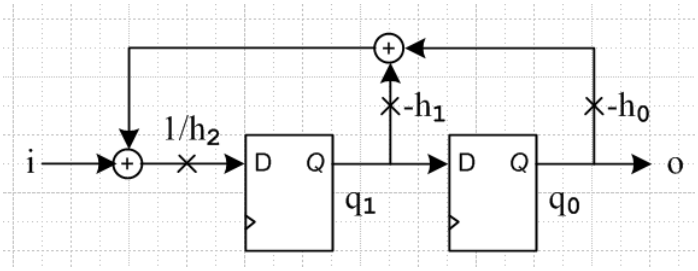


Fig 1.2: Type A divider with k = 2.

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:

$$[o_2 z^{-2} + o_3 z^{-3} + o_4 z^{-4} + ...] \ = \ [1 + z^{-1}] \,/\, [z^2 + z + 1] \ .$$

Multiply through by z^2+z+1,

$$[z^2 + z + 1][o_2 z^{-2} + o_3 z^{-3} + o_4 z^{-4} + \ldots\ldots] = [1 + z^{-1}] \ .$$

The left side may be written

$$(o_2 + o_3\, z^{-1} + o_4 z^{-2} + \ldots\ldots) + (o_2 z^{-1} + o_3\, z^{-2} + o_4 z^{-3} + \ldots\ldots) + (o_2 z^{-2} + o_3 z^{-3} + o_4 z^{-4} + \ldots\ldots)$$

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

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

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

Thus, the result of the polynomial division is this

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

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

$$= \Sigma_{n=1}^{\infty} (z^{-3n+1} - z^{-3n-1}). \qquad (1.2.9)$$

The output O(z) continues forever in this pattern. Here is a Maple verification of this result:

```
f := (1+1/z)/(z^2 + z + 1);
```

$$f := \frac{1 + \frac{1}{z}}{z^2 + z + 1}$$

```
sum( z^(-3*n+1) - z^(-3*n-1),  n= 1..5);
```

$$\frac{1}{z^2} - \frac{1}{z^4} + \frac{1}{z^5} - \frac{1}{z^7} + \frac{1}{z^8} - \frac{1}{z^{10}} + \frac{1}{z^{11}} - \frac{1}{z^{13}} + \frac{1}{z^{14}} - \frac{1}{z^{16}}$$

```
g := sum( z^(-3*n+1) - z^(-3*n-1),  n= 1..infinity);
```

$$g := \frac{\left(\frac{1}{z^2} - \frac{1}{z^4}\right) z^3}{z^3 - 1}$$

```
f-g:  simplify(%);
```

$$0$$

Observation: If we take a snapshot of the division process of Fig 1.1 at any clock, we find that the k registers always hold the next k symbols o_j of the quotient polynomial O(z). Just stare at Fig 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 $H(z) = z^2 + z$.

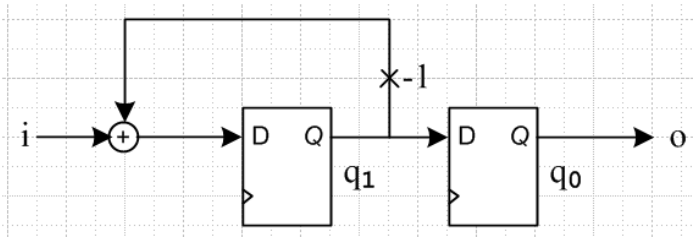


Fig 1.3. Another Type A divider with k = 2

The result of this division is easily found to be,

$$[1 + z^{-1}] / [z^2 + z\,] \;= o_2 z^{-2} + o_3 z^{-3} + o_4 z^{-4} + \;\ldots\ldots\; = z^{-2}\,.$$

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 $q_1 q_0$ 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 FT Section 24 (f) 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 divider circuit of Fig 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 AS) ,

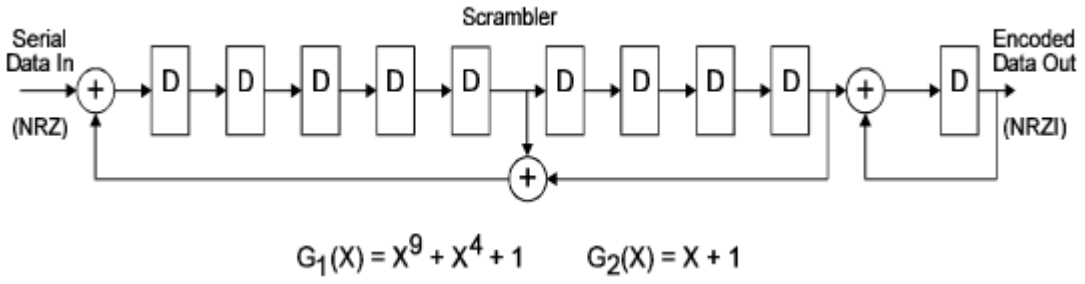


Fig 1.4: Two scramblers in series.

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 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 i_n 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 real numbers:

$$2216.0000 / 342 = 6.\ 479532163742690058\ \ 479532163742690058 \ldots. \qquad (1.3.2)$$

Although the dividend 2216 ends with all zeros, the quotient in this case goes on forever and in fact has a repeating sequence since it is in fact a rational real number.

In Appendix A, we look into this "distant analogy" and we show that if the digits in the numerator and denominator stand for Mod(10) coefficients of powers of z, the polynomial quotient is in fact

$$2216.0000 / 342 = 42.5\ \ 480860620240\ \ 480860620240 \ldots.$$

and the sequence of coefficients really does repeat forever, just as with the division or real numbers. Hopefully this last result seems mysterious enough to motivate the reader to peruse Appendix A.

Semantic Note: Sometimes when we say non-zero symbols, we really mean symbols which *in general* are non-zero. For example, if an input sequence is ....0000$i_0 i_1 i_2 i_3$0000... we refer to i_0,i_1,i_2,i_3 as being the "non-zero symbols", but in practice one or more of them could be zero.

(a) What about The Remainder?

When polynomials are divided, O(z) = I(z)/H(z), and when O(z) is expressed as an infinite polynomial, there *is* no remainder! This corresponds to having the hardware of Fig 1.1 process a finite (or infinite) incoming polynomial I(z) forever. As noted in the examples above, if improper polynomials are allowed, the meaning of the term "remainder" is more or less at the discretion of the person doing the division, but when proper polynomials are divided, the remainder has a standard meaning.

Let us consider an input symbol stream which has this form, such that all inputs are 0 after i_r,

.....0,0,0, i_0, i_1, ...i_r,0,0,0.....

The input polynomial in this case (the Z Transform of the above signal) is this finite polynomial,

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

We can rewrite this input sequence as:

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

Here we have defined $I_r(z)$ to be the *proper* polynomial shown in [..], having degree r.

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 output symbols o_0 through o_{k-1} all vanish. This is easily interpreted in terms of Fig 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 divider circuit to be given below.) Thus we have, with a common time base, and assuming r > k, the following "timing diagram",

i_j: .....0,0,0, i_0, i_1, ...i_r,0,0,0.....
o_j .....0,0,0, 0,0,0.....o_k,o_{k+1},o_{k+2},o_{k+3},o_{k+4},o_{k+5},o_{k+6}, ... (1.3.5)

The output polynomial is then

$$O(z) = o_k z^{-k} + o_{k+1} z^{-k-1} + ... = \Sigma_{i=k}^{\infty} o_i z^{-i} = \qquad (1.3.6)$$

$$= z^{-r} [o_k z^{r-k} + o_{k+1} z^{r-k-1} + ...] \equiv z^{-r} O_{r-k}(z) \qquad (1.3.7)$$

where $O_{r-k}(z)$ in the bracket is a (generally) *infinite* polynomial of degree r-k. Since it generally has negative powers of z, it is an improper polynomial, a characteristic it shares with O(z) and I(z). On the other hand, $I_r(z)$ and H(z) are both proper polynomials.

Since a common factor z^{-r} has been extracted in (1.3.4) and (1.3.5), we can write

$$O(z) = I(z)/H(z) \qquad \Rightarrow \qquad O_{r-k}(z) = I_r(z)/H(z) \tag{1.3.8}$$

where now the polynomial ratio $I_r(z)/H(z)$ is completely conventional since both $I_r(z)$ and H(z) are proper polynomials, of degree r and k respectively. We can then talk about a quotient Q and a remainder R as follows,

$$I_r(z)/H(z) = \mathcal{Q}(z) + R(z)/H(z)\ , \tag{1.3.9}$$

where the quotient $\mathcal{Q}(z)$ has degree r-k and R(z) has degree ≤ k-1 :

$$\mathcal{Q}(z) = q_{r-k}z^{r-k} + q_{r-k-1}z^{r-k-1} + ... q_1 z + q_0 \quad = \quad \Sigma_{i=k}^{r}\ q_{r-i}\ z^{r-i} \tag{1.3.10}$$

$$R(z) = r_{k-1}z^{k-1} + r_{k-2}z^{k-2} + ... + r_1 z + r_0 \quad . \tag{1.3.11}$$

Note: Since our Fig 1.5 registers are already called q_j, we have denoted the quotient coefficients by the symbols q_j and the quotient polynomial correspondingly as $\mathcal{Q}(z)$.

Notice that if it happens that r < k, then $I_r(z)/H(z)$ is already in remainder form and we simply identify $I_r(z) = R(z)$ and $\mathcal{Q}(z) = 0$, so from now on we assume that r ≥ k. Then,

$$O(z) = I(z) / H(z) = [z^{-r} I_r(z)] / H(z) = z^{-r} [I_r(z) / H(z)] = z^{-r}[\mathcal{Q}(z) + R(z)/H(z)]$$

$$= [z^{-r} \mathcal{Q}(z)] \quad + \quad [z^{-r}R(z)] / H(z)$$

$$= \Sigma_{i=k}^{r}\ q_{r-i}\ z^{-i} \quad + \quad [z^{-r}R(z)] / H(z) \quad . \tag{1.3.12}$$

Comparing this sum to the (1.3.6) sum for O(z) decomposed into two pieces,

$$O(z) \quad = \quad \Sigma_{i=k}^{r}\ o_i z^{-i} \quad + \quad \Sigma_{i=r+1}^{\infty}\ o_i z^{-i} \quad , \tag{1.3.13}$$

and matching powers of z, we can identify the sum in (1.3.12) with the first sum in (1.3.13), so that

$$o_i \ = \ q_{r-i} \qquad \text{for i = k to r} \quad . \tag{1.3.14}$$

The quotient of (1.3.10) may then be expressed as

$$\mathcal{Q}(z) = \Sigma_{i=k}^{r}\ o_i\ z^{r-i} \tag{1.3.15}$$

so the first r-k+1 significant terms $o_k....o_r$ of O(z) are in fact the coefficients of the quotient $\mathcal{Q}(z)$.

The second sum in (1.3.13) must then be equal to $[z^{-r} R(z)] / H(z)$,

$$[z^{-r} R(z)] / H(z) = \Sigma_{i=r+1}^{\infty} o_i z^{-i}$$

which says

$$R(z) / H(z) = z^r \Sigma_{i=r+1}^{\infty} o_i z^{-i} \quad . \tag{1.3.16}$$

We then end up with this expression for the proper (finite) remainder R(z) as H(z) z^r times the infinite tail polynomial of O(z),

$$R(z) = H(z)\, z^r \Sigma_{i=r+1}^{\infty} o_i z^{-i} \quad . \tag{1.3.17}$$

In the following example, we see how this finite/infinite conundrum resolves itself.

Example 1 Revisited

We shall now reframe Example 1 in the context of the discussion just concluded. In Example 1 we had

$H(z) = z + 1$, so k = 1, and $I_r(z) = z^2 + 1$, so r = 2

$O_{r-k}(z) = (z - 1 + 2z^{-1} - 2z^{-2} + 2z^{-3} - 2z^{-4} + ...)$

$O(z) = z^{-r} O_{r-k}(z) = z^{-2} (z - 1 + 2z^{-1} - 2z^{-2} + 2z^{-3} - 2z^{-4} + ...)$

$= (z^{-1} - z^{-2} + 2z^{-3} - 2z^{-4} + 2z^{-5} - 2z^{-6} + ...) = \Sigma_{i=1}^{\infty} o_i z^{-i}$.

By doing simple long division in Example 1 we showed that,

$$\frac{z^2 + 1}{z+1} = (z - 1) + \frac{2}{z+1} \quad \Rightarrow \quad \mathcal{Q}(z) = z - 1 \quad \text{and} \quad R(z) = 2$$

We first verify $\mathcal{Q}(z)$ from our general expression (1.3.15),

$\mathcal{Q}(z) = \Sigma_{i=k}^{r} o_i z^{r-i} = \Sigma_{i=1}^{2} o_i z^{2-i} = o_1 z + o_2 = z - 1$ // agrees

Then R(z) can be verified from (1.3.17), with a bit more work,

$R(z) = H(z) z^r \Sigma_{i=r+1}^{\infty} o_i z^{-i}$ // finite R(z) as infinite sum

$= (z+1) z^2 \Sigma_{i=3}^{\infty} o_i z^{-i}$

$$= (z+1)\, z^{2}\, (2z^{-3} - 2z^{-4} + 2z^{-5} - 2z^{-6} + \ldots.)$$

$$= 2\,(z+1)\,(\, z^{-1} - z^{-2} + z^{-3} - z^{-4} + \ldots.)$$

$$= -\,2\,(z+1)\,(-1 + 1 - z^{-1} + z^{-2} - z^{-3} + z^{-4} + \ldots.)$$

$$= -\,2\,(z+1)\,\left(-1 + \frac{1}{1+z^{-1}}\right) = -\,2\,(z+1)\,\left(-1 + \frac{z}{z+1}\right)$$

$$= -\,2\,(z+1)\,\frac{-1}{z+1} = 2 \qquad \text{// agrees}$$

(b) Sequence of Remainders

In the above discussion there is only one remainder, it is R(z). Remainder R(z) is what appears in (1.3.9).

$$I_r(z)/H(z) = \mathcal{Q}(z) + R(z)/H(z) \qquad \textit{(1.3.9)}$$

$$I_r(z) = i_0\, z^{r} + i_1\, z^{r-1} + i_2\, z^{r-2} + \ldots\ldots + i_r\ . \qquad \textit{(1.3.4)}$$

However, both $\mathcal{Q}(z)$ and R(z) are implicitly dependent on the choice of integer r, so it might be better to write

$$I_r(z)/H(z) = \mathcal{Q}^{(r)}(z) + R^{(r)}(z)/H(z)\ . \qquad (1.3.18)$$

For example, if we now think of the last non-zero input sample being i_{r+1} instead of i_r, then we have

$$I_{r+1}(z)/H(z) = \mathcal{Q}^{(r+1)}(z) + R^{(r+1)}(z)/H(z)$$

In this case $I_{r+1}(z)$ is a proper polynomial of degree r+1 and when we divide this by H(z), we get a new and different quotient and remainder. This is true even if it happens that $i_{r+1} = 0$. We are still allowed to *think* of the incoming stream as having r+2 elements.

In terms of the hardware of Fig 1.1, imagine some *general* input sequence i_n whose first non-zero sample is i_0. If we clock in only r+1 of the samples $\{i_0 \ldots i_r\}$ of this input sequence, and then stop, then we have a particular "division problem" in which we have $I(z) = z^{-r} I_r(z)$ where $I_r(z) = i_0\, z^{r} + i_1\, z^{r-1} + \ldots + i_r$. For the proper polynomial division problem $I_r(z)/H(z)$ we then have a quotient $\mathcal{Q}^{(r)}(z)$ and a remainder $R^{(r)}(z)$. If we then do one more clock and clock in i_{r+1}, then we have a whole new proper division problem, and it has a whole new quotient $\mathcal{Q}^{(r+1)}(z)$ and remainder $R^{(r+1)}(z)$.

In this case, suppose it happens that $i_{r+1} = 0$. The conclusion still holds, but in this case we have a relationship between $I_{r+1}(z)$ and $I_r(z)$, namely,

$$I_{r+1}(z) = i_0 z^{r+1} + i_1 z^r + i_2 z^{r-1} + \ldots + i_r z + (i_{r+1}=0)$$

$$= z\,(i_0 z^r + i_1 z^{r-1} + \ldots + i_r) = zI_r(z) \qquad (1.3.19)$$

More generally, if we clock in non-zero symbols i_0 through i_r and then clock in n zero symbols, we have

$$I_{r+n}(z) = z^n I_r(z) \qquad (1.3.20)$$

Conclusion: As we clock in each input symbol, we define a new proper polynomial division problem which has its own particular quotient and remainder, so we have in effect a sequence of division problems with a sequence of quotients and a sequence of remainders. This continues to be true even if all the input symbols are zero after some point. A detailed example of this situation is presented in Section 1.5 (a) below.

(c) Where is the Remainder located in Fig 1.1?

Imagine we have clocked in the set of input symbols i_0 through i_r and we stop the divider. At that point, the output symbols o_0 through o_r have been clocked out. The first k of these vanish, so we have then clocked out a total of r-k+1 meaningful output symbols o_k ..... o_r. These symbols o_k .... o_r are exactly those appearing in the quotient $\mathcal{Q}(z)$ in (1.3.15). So we have our quotient, but where is the remainder?

The remainder as shown in (1.3.11) has k coefficients r_0 through r_{k-1} (some of which may be zero). The set of k registers in Fig 1.1 is at this point holding a set of k symbols q_j for j=0 to k-1. One might conjecture that the remainder coefficients are functions of the register values, but this turns out not to be the case. As we shall see later, each r_j remainder coefficient is a function of some of the register values and of some of the output o_j which have already flown the coop.

Later we shall see that for the Type B divider of Fig 1.5, after the division process of the previous paragraph, the remainder coefficients are exactly the contents of the registers!

(d) Impulse Response

We have seen above that in general O(z) = I(z)/H(z) is an infinite polynomial that never terminates. Values circulate around in the divider's registers and, in light of the feedback of the circuit, keep generating new outputs. 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 $i_n = \delta_{n,0}$)

$$I(z) = 1 \qquad O(z) = I(z)/H(z) = 1/H(z) \qquad (1.3.21)$$

We can interpret this division as a sequence of "division problems" indexed by r, as discussed above, where H(z) is being divided into z^r for ever increasing r, so (1.3.18) becomes,

$$z^r / H(z) = \mathcal{Q}^{(r)}(z) + R^{(r)}(z)/H(z)\,. \qquad (1.3.22)$$

Unless H(z) is a single power of z, the impulse response will go on forever. and we get an endless sequence of quotients and remainders. That is to say, O(z) = 1/H(z) is a polynomial which never terminates.

1.4 The Type B Polynomial Divider

In the previous section we studied the Type A divider implementation. We proved that it really does divide polynomials.

Here, we consider the Type B implementation of a polynomial divider:

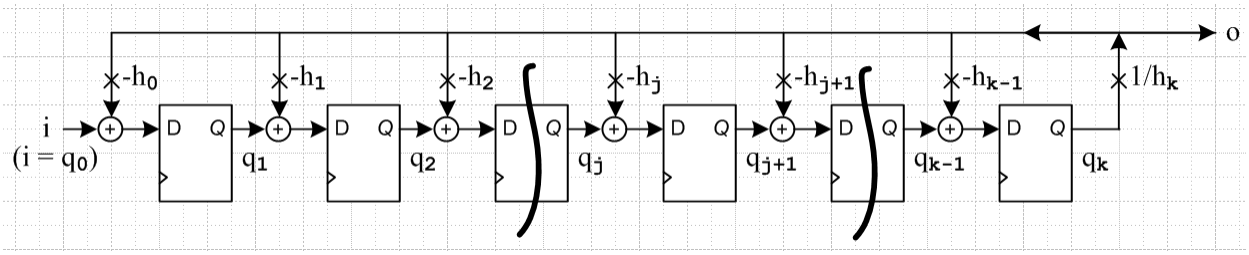


Fig 1.5. Type B polynomial divider.

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 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 1.1 they 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 Type B Divider in the z-domain

Looking at Fig 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),

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

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:

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

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

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

Evidently the general solution is this:

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

Setting j = k then gives

$$\begin{aligned} z^{k+1}Q_{k+1}(z) &= I(z) - [\,h_k\,z^k + \ldots + h_1\,z + h_0\,]O(z) \\ &= I(z) - H(z)O(z)\,. \end{aligned}$$

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

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

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

Registers: If we take a snapshot of the division process of Fig 1.5 at any clock, we *claim* that the k registers of Fig 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 Fig 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 Fig 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

(a) Operation of the Type B Divider

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

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

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

$I(z) = i_0 + i_1 z^{-1} + i_2 z^{-2} + ...$ // input stream i_j starts with i_0
$O(z) = o_3 z^{-3} + o_4 z^{-4} + o_5 z^{-5} + ...$ // output stream o_j starts with o_3
$H(z) = h_3 z^3 + h_2 z^2 + h_1 z + h_0$. // divisor of degree k = 3

The divider registers are assumed to be initialized to zero. So here is our truncated Fig 1.5:

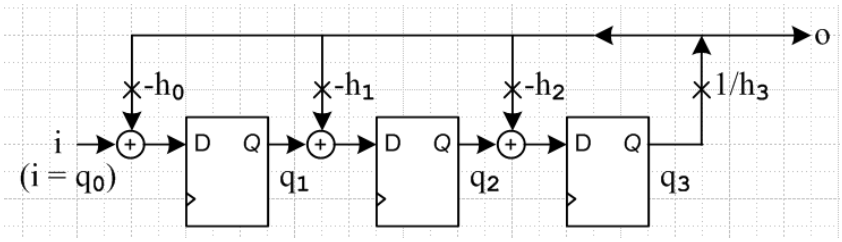


Fig 1.6: The Type B divider circuit with k = 3.

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

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

Current Dividend #1

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

Current Dividend #2

$$\begin{array}{r} b_2z^{-2} + b_3z^{-3} + b_4z^{-4} + i_5z^{-5} + ... \\ -o_5h_3z^{-2} - o_5h_2z^{-3} - o_5h_1z^{-4} - o_5h_0z^{-5} \\ \hline \end{array}$$

Current Dividend #3

$$\begin{array}{r} c_3z^{-3} + c_4z^{-4} + c_5z^{-5} + ... \\ -o_6h_3z^{-3} - o_6h_2z^{-4} - o_6h_1z^{-5} - o_6h_0z^{-6} \\ \hline \end{array}$$

and on forever (1.5.1)

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 inside the division symbol (it is 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 its 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	(1.5.2)

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 Fig 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 registers are cleared before i_0 comes in. The first three input symbols just shift 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).

We can show the temporal relationship between symbols on the i bus and on the o bus

		L1 ↑	L2 ↑	L3 ↑	C1 ↑	C2 ↑	C3 ↑	C4 ↑			
i:	0	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	etc	
o:	0	0	0	0	o_3	o_4	o_5	o_5	o_6	etc	
current dividend:					#0	#1	#2	#3	#4	etc	(1.5.3)

The first line shows clock edges which clock the registers. From the time symbol i_0 is on the input i bus, it takes k=3 initial clocks to get the registers loaded up. These clocks are labeled L1, L2, L3. Only after these clocks does the long division diagram match the hardware and we have our initial dividend which we have called Current Dividend #0. From this point on, each new clock brings the hardware to a new current dividend.

In Section 1.3 we discuss the restatement of the division O(z) = I(z)/H(z) as a proper polynomial division $O_{r-k}(z) = I_r(z)/H(z)$ where $I_r(z) = z^rI(z)$ and $O_{r-k}(z) == z^rO(z)$. Doing that here with r = 5, we obtain

$$I_{r}(z) = z^5 I(z) = i_0 z^5 + i_1 z^4 + i_2 z^3 + i_3 z^2 + i_4 z + i_5$$
$$O_{r-k}(z) = z^5 O(z) = o_3 z^2 + o_4 z + o_5 .$$

Another way to state what we are doing here is just this:

$$[z^5 O(z)] = [z^5 I(z)] / H(z)$$

where O(z) and I(z) area each multiplied by z^5. The long division layout for this proper polynomial division problem is then :

$$\begin{array}{rl} & o_3z^2 + o_4z^1 + o_5 \\ \hline h_3z^3 + h_2z^2 + h_1z^1 + h_0z^0 & \mid i_0z^5 + i_1z^4 + i_2z^3 + i_3z^2 + i_4z^1 + i_5 \\ & -o_3h_3z^5 - o_3h_2z^4 - o_3h_1z^3 - o_3h_0z^2 \\ & \text{---} \end{array}$$

Current Dividend #1
$$\begin{array}{l} a_1z^4 + a_2z^3 + a_3z^2 + i_4z^1 + i_5 \\ -o_4h_3z^4 - o_4h_2z^3 - o_4h_1z^2 - o_4h_0z^1 \\ \text{---} \end{array}$$

Current Dividend #2
$$\begin{array}{l} b_2z^3 + b_3z^2 + b_4z^1 + i_5 \\ -o_5h_3z^3 - o_5h_2z^2 - o_5h_1z^1 - o_5h_0 \\ \text{---} \end{array}$$

Current Dividend #3
$$c_3z^2 + c_4z^1 + c_5 \qquad (1.5.4)$$

We now have a the division of two "proper" polynomials. In this case our timing diagram above looks like this:

		L1	L2	L3	C1	C2	C3		
	↑	↑	↑	↑	↑	↑	↑		
i:	0	i_0	i_1	i_2	i_3	i_4	i_5		
o:	0	0	0	0	o_3	o_4	o_5	o_5	
current dividend:					#0	#1	#2	#3	(1.5.5)

After clock C3, the output o bus has symbol o_5 and the registers hold Current Dividend #3 which is in fact **the remainder polynomial**, since this is the first current dividend of degree less than that of the divisor H(z). The remainder is arranged such that $(q_3, q_2, q_1) = (c_3, c_4, c_5)$ so the highest power coefficient c_3 is sitting in the rightmost register q_3.

Application of another clock C4 will destroy this remainder. However, the registers after clock C4 will hold the remainder of a *new* division problem -- one in which a new symbol i_6 is added to the input stream. In this case we write things using

$$[z^6 O(z)] = [z^6 I(z)] / H(z)$$

and here is the new diagram

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad o_3z^3 \;+\; o_4z^2 \;\;+\; o_5z^1 \;+\; o_6 \\
h_3z^3 + h_2z^2 + h_1z^1 + h_0z^0 \;\big|\; i_0z^6 \;+\; i_1z^5 \;+\; i_2z^4 + \;\; i_3z^3 \;+\; i_4z^2 \;+\; i_5z^1 \;+\; i_6 \\
\qquad\qquad\qquad\quad -o_3h_3z^6 - o_3h_2z^5 - o_3h_1z^4 - o_3h_0z^3 \\
\qquad\qquad\qquad\quad \text{---}
\end{array}
$$

Current Dividend #1

$$
\begin{array}{l}
\qquad\qquad\quad a_1z^5 + a_2z^4 \;+\; a_3z^3 \;+\; i_4z^2 \;+\; i_5z^1 \;+\; i_6 \\
\qquad\quad -o_4h_3z^5 - o_4h_2z^4 - o_4h_1z^3 - o_4h_0z^2 \\
\qquad\quad \text{---}
\end{array}
$$

Current Dividend #2

$$
\begin{array}{l}
\qquad\quad b_2z^4 \;+\; b_3z^3 \;+\; b_4z^2 \;+\; i_5z^1 \;+\; i_6 \\
\qquad -o_5h_3z^4 - o_5h_2z^3 - o_5h_1z^2 - o_5h_0z^1 \\
\qquad \text{---}
\end{array}
$$

Current Dividend #3

$$
\begin{array}{l}
\qquad\quad c_3z^3 \;+\; c_4z^2 \;+\; c_5z^1 \;+\; i_6 \\
\quad -o_6h_3z^3 - o_6h_2z^2 - o_6h_1z^1 - o_6h_0z^0 \\
\quad \text{---}
\end{array}
$$

Current Dividend #4

$$d_4z^2 \;+\; d_5z^1 \;+\; d_6z^0 \qquad (1.5.6)$$

and the new timing diagram

		L1	L2	L3	C1	C2	C3	C4		
	↑	↑	↑	↑	↑	↑	↑	↑		
i:	0	i_0	i_1	i_2	i_3	i_4	i_5	i_6		
o:	0	0	0	0	o_3	o_4	o_5	o_5	o_6	
current dividend:					#0	#1	#2	#3	#4	(1.5.7)

Now after clock C4 the registers contain the Current Dividend #4 which is in fact the remainder of this new division problem, and now we have $(q_3, q_2, q_1) = (d_4, d_5, d_6)$.

Even if it happens that $i_6 = 0$, we still have a new division problem with a new remainder when we clock in $i_6 = 0$. For example, suppose the first proper division problem was this (as shown above)

$(i_0z^5 + i_1z^4 + i_2z^3 + i_3z^2 + i_4z^1 + i_5)/h(z)$ remainder = $c_3z^2 + c_4z + c_5$

If we then clock in i_6 and it happens to be zero, the new proper division problem is this:

$(i_0z^6 + i_1z^5 + i_2z^4 + i_3z^3 + i_4z^2 + i_5z + 0)/h(z)$

$= \{ z\,(i_0z^5 + i_1z^4 + i_2z^3 + i_3z^2 + i_4z^1 + i_5) \} / h(z)$ remainder = $d_4z^2 + d_5z + d_6$

where $d_6 = -\,o_6h_0$. The remainder is different because the dividend has one higher degree and so the division process has to continue through one more current dividend to get a current dividend of degree less than the original dividend.

Conclusion: If we have an endless input stream i_0, i_1..... , then we can regard the contents of the registers of a Type B divider at any clock as holding the remainder of a certain proper polynomial division problem, while the output bus o will have delivered the quotient for that problem. Here is a list of

these division problems for the case k = 3, where the first three rows correspond to 0 quotient and involve only the loading clocks:

I(z)	Remainder	Quotient	Clk
i_0	i_0	0	L1
$i_0z + i_1$	$i_0z + i_1$	0	L2
$i_0z^2 + i_1z + i_2$	$i_0z^2 + i_1z + i_2$	0	L3
$i_0z^3 + i_1z^2 + i_2z + i_3$	$a_1z^2 + a_2z + a_3$	o_3	C1
$i_0z^4 + i_1z^3 + i_2z^2 + i_3z + i_4$	$b_2z^2 + b_3z + b_4$	$o_3z + o_4$	C2
$i_0z^5 + i_1z^4 + i_2z^3 + i_3z^2 + i_4z + i_5$	$c_3z^2 + c_4z + c_5$	$o_3z^2 + o_4z + o_5$	C3
$i_0z^6 + i_1z^5 + i_2z^4 + i_3z^3 + i_4z^2 + i_5z + i_6$	$d_4z^2 + d_5z + d_6$	$o_3z^3 + o_4z^2 + o_5z + o_6$	C4
etc.			

(1.5.8)

(b) Operation of the Type A Divider

Again we consider the case k = 3 and 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.

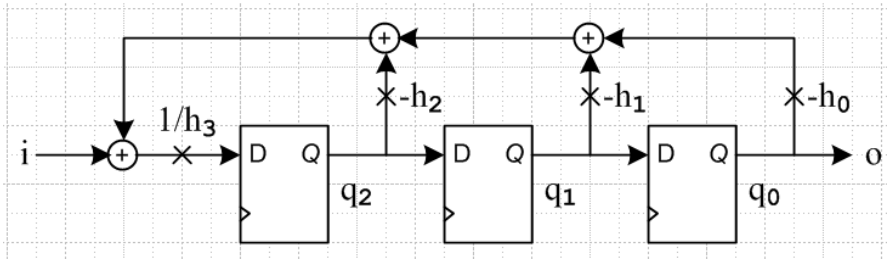


Fig 1.7: The Type A divider circuit with k = 3.

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

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad o_3z^{-3} + o_4z^{-4} + o_5z^{-5} + o_6z^{-6} + o_7z^{-7} + o_8z^{-8} + \ldots. \\
h_3z^3 + h_2z^2 + h_1z^1 + h_0z^0 \;\big|\; i_0z^0 + i_1z^{-1} + i_2z^{-2} + i_3z^{-3} + i_4z^{-4} + i_5z^{-5} + \ldots.. \\
\qquad\qquad\qquad\quad - o_3h_3z^0 - o_3h_2z^{-1} - o_3h_1z^{-2} - o_3h_0z^{-3} \\
\qquad\qquad\qquad\quad \text{----------} \\
\text{Current Dividend \#1} \qquad\qquad\qquad\quad a_1z^{-1} + a_2z^{-2} + a_3z^{-3} + i_4z^{-4} + i_5z^{-5} + \ldots.. \\
\qquad\qquad\qquad\qquad\quad - o_4h_3z^{-1} - o_4h_2z^{-2} - o_4h_1z^{-3} - o_4h_0z^{-4} \\
\qquad\qquad\qquad\qquad\quad \text{----------} \\
\text{Current Dividend \#2} \qquad\qquad\qquad\qquad b_2z^{-2} + b_3z^{-3} + b_4z^{-4} + i_5z^{-5} + \ldots. \\
\qquad\qquad\qquad\qquad\qquad\quad - o_5h_3z^{-2} - o_5h_2z^{-3} - o_5h_1z^{-4} - o_5h_0z^{-5} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{----------} \\
\text{Current Dividend \#3} \qquad\qquad\qquad\qquad\qquad\quad c_3z^{-3} + c_4z^{-4} + c_5z^{-5} + \ldots. \\
\qquad\qquad\qquad\qquad\qquad\qquad - o_6h_3z^{-3} - o_6h_2z^{-4} - o_6h_1z^{-5} - o_6h_0z^{-6} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{----------} \\
\text{Current Dividend \#4} \qquad\qquad\qquad\qquad\qquad\qquad d_4z^{-4} + d_5z^{-5} + d_6z^{-6} + \ldots. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad - o_7h_3z^{-4} - o_7h_2z^{-5} - o_7h_1z^{-6} \ldots. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{----------} \\
\text{Current Dividend \#5} \qquad\qquad\qquad\qquad\qquad\qquad\quad e_5z^{-5} + e_6z^{-6} + \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad - o_8h_3z^{-5} - o_8h_2z^{-6} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{----------}
\end{array}
\qquad (1.5.9)
$$

Let's start in the middle of things to see what the adders in Fig 1.7 are doing. Consider the column of red expressions which has i_3 at the top. At this time, the output is o_3 and the input is i_3. Looking at Fig 1.7, we see that the adders are computing the following sum (based on the fact just stated about outputs-to-be)

$$\text{D input to } q_2 = (1/h_3) * [i_3 + (-h_0)o_3 + (-h_1)o_4 + (-h_2)o_5]$$

$$= (1/h_3) * [i_3 - o_3 h_0 - o_4 h_1 - o_5 h_2]$$

The terms of the bracketed sum here are seen to match the red column just noted. On the next clock edge, this "D input to q_2" value is strobed into register q_2 and it is then destined to become o_6, which is shown in blue underneath the red column. With the previous Type B circuit of Fig 1.6, this same sum for o_6 was also computed, but it was done 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_j outputs.

If we look one clock later, we have the next column of red expressions being added to determine o_7. And one clock earlier, we have a column adding up expressions to determine o_5. Before this time, some of the adder inputs are 0 so the column of red expressions added has fewer than 4 elements.

1.6. The Type A Polynomial Multiplier

Consider the following circuit:

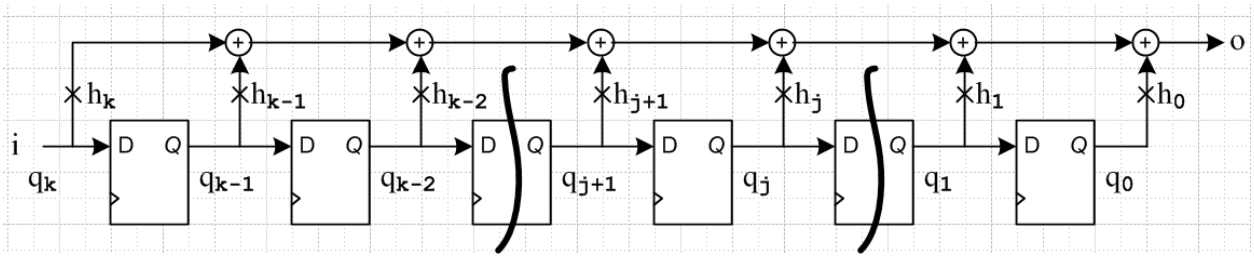


Fig 1.8: Type A polynomial multiplier.

(a) Analysis of the Type A 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),

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

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

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

The circuit shown in Fig 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 (1.1.4). 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 Fig 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.

(b) 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,

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

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,

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

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 Fig 1.8 is doing. We shall carry out this task below.

(c) 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 Fig 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 shifting one just scans out the h_j coefficients. Each clock of the shift adds another delay factor of z^{-1}.

The circuits of Fig 1.8 and Fig 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 Fig 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 Fig 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 AS) ,

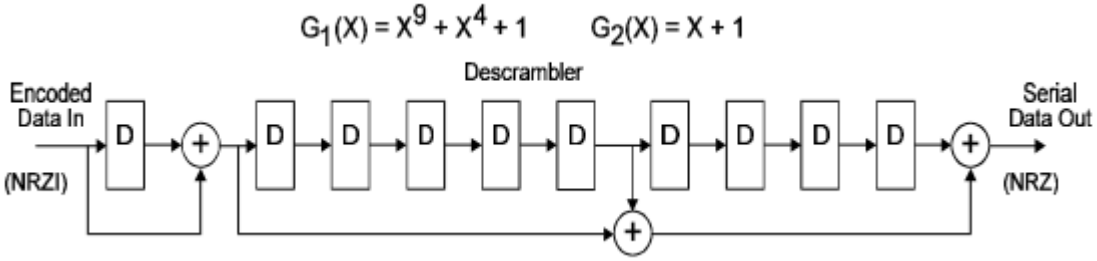


Fig 1.9. Two descramblers in series.

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 Type B Polynomial Multiplier

The Type B polynomial multiplier has this form, with "inboard adders" :

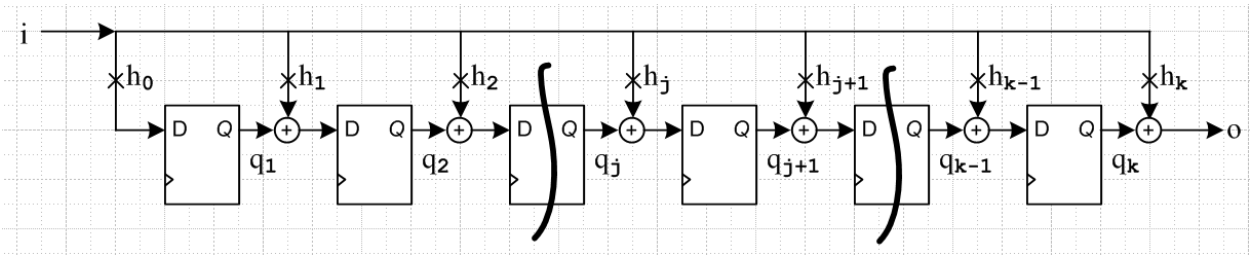


Fig 1.10: Type B polynomial multiplier.

It is going to turn out that z^kO(z) = I(z)H(z) exactly as with the Type A design. Just as the Type A multiplier was very similar to the Type A divider, so too is the Type B multiplier similar to the Type B multiplier (see all four circuits on the first page of Chapter 1). Notice that in Fig 1.10 there are no minus signs in front of the h_j coefficients.

Analysis of the Type B Multiplier in the z-domain

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

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

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 Type B 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),

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

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

$z^3\, Q_3(z) = z^2\, Q_2(z) + h_2\, z^2\, I(z)$ // 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:

$$z^{j+1}\, Q_{j+1}(z) = (\ h_j\, z^j + \ldots . h_1\, z + h_0\ )\, I(z) \quad . \tag{1.7.4}$$

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

$$z^k\, O(z) = H(z)\, I(z) \tag{1.7.5}$$

which is the same as (1.6.4) obtained for the Type A multiplier. Thus, the circuit in Fig 1.10 is functionally identical to that in Fig 1.8.

1.8 How Polynomial Multipliers Actually Work

(a) Operation of the Type A Polynomial Multiplier

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

$$O(z) = z^{-k} H(z) I(z) \quad . \tag{1.8.1}$$

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)

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

As usual, the registers are all initialized to zero. The k = 3 Type A multiplier then looks like this:

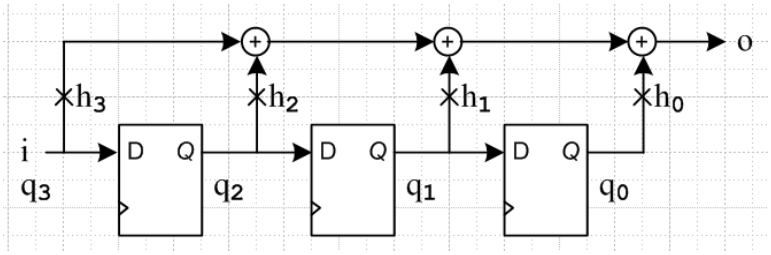


Fig 1.11. Type A multiplier with k = 3

To demonstrate the operation, we multiply H(z) and I(z) "by hand" (comments below):

$$\begin{array}{l}
\qquad h_3z^3 + h_2z^2 + h_1z^1 + h_0z^0 \\
\ldots\ + i_3z^{-3} + i_2z^{-2} + i_1z^{-1} + i_0z^0 \\
\text{---------------------------------------} \\
\qquad i_0h_3z^3 + i_0h_2z^2 + i_0h_1z^1 + i_0h_0z^0 \\
\qquad\quad i_1h_3z^2 + i_1h_2z^1 + i_1h_1z^0 + i_1h_0z^{-1} \\
\qquad\qquad i_2h_3z^1 + i_2h_2z^0 + i_2h_1z^{-1} + i_2h_0z^{-2} \\
\qquad\qquad\quad i_3h_3z^0 + i_3h_2z^{-1} + i_3h_1z^{-2} + i_3h_0z^{-3} \\
\qquad\qquad\qquad i_4h_3z^{-1} + i_4h_2z^{-2} + i_4h_1z^{-3} + i_4h_0z^{-4} \\
\qquad\qquad\qquad\quad i_5h_3z^{-2} + i_5h_2z^{-3} + i_5h_1z^{-4} + i_5h_0z^{-5} \\
\qquad\qquad\qquad\qquad **** + ***** + ***** + \ldots \\
\qquad\qquad\qquad\qquad\quad ***** + ***** + \ldots \\
\qquad\qquad\qquad\qquad\qquad ***** + \ldots \\
\text{---} \\
\qquad o_0z^3 + o_1z^2 + o_2z^1 + o_3z^0 + o_4z^{-1} + o_5z^{-2} + o_6z^{-3} + o_7z^{-4} + o_8z^{-5} \ldots \\
= z^3 [\, o_0z^0 + o_1z^{-1} + o_2z^{-2} + o_3z^{-3} + o_4z^{-4} + o_5z^{-5} + o_6z^{-6} + o_7z^{-7} + o_8z^{-8} \ldots]
\end{array} \tag{1.8.2}$$

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 $i_0 z^0$ of I(z) is multiplied by H(z) as shown to get the first row. Then the next multiplier element $i_1 z^{-1}$ 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 $z^3 O(z) = H(z)\, I(z)$, we factor out z^3 on the very last line to expose O(z) in the bracket. The idea here is that, for example, $o_2 = i_0 h_1 + i_1 h_2 + i_2 h_3$. 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 Fig 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 i_j symbols already in the register to shift to the right one position. At clock tick 0 the output is $o_0 = i_0 h_3$. At clock tick 1 the output is $o_1 = i_0 h_2 + i_1 h_3$, and so on. By clock tick 3, the registers contain (i_2, i_1, i_0) and the input is i_3 so 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.

(b) Operation of the Type B Polynomial Multiplier

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

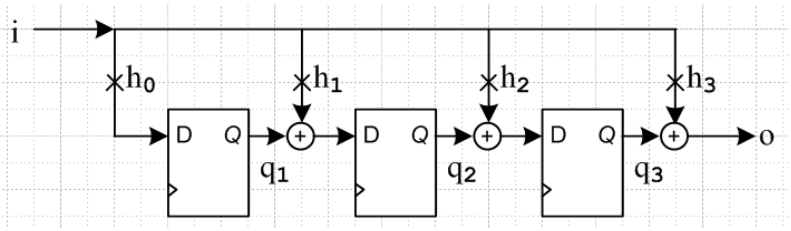


Fig 1.12. Type B multiplier with k = 3

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

$$h_0z^0 + h_1z^1 + h_2z^2 + h_3z^3$$
$$i_4z^{-3} + i_3z^{-3} + i_2z^{-2} + i_1z^{-1} + i_0z^0$$

---

partial product #0 $\quad i_0h_0z^0 + i_0h_1z^1 + i_0h_2z^2 + i_0h_3z^3$
$$i_1h_0z^{-1} + i_1h_1z^0 + i_1h_2z^1 + i_1h_3z^2$$

---

partial product #1 $\quad i_1h_0z^{-1} + a_0z^0 + a_1z^1 + a_2z^2 + i_0h_3z^3$
$$i_2h_0z^{-2} + i_2h_1z^{-1} + i_2h_2z^0 + i_2h_3z^1$$

---

partial product #2 $\quad i_2h_0z^{-2} + b_{-1}z^{-1} + b_0z^0 + b_1z^1 + a_2z^2 + i_0h_3z^3$
$$i_3h_0z^{-3} + i_3h_1z^{-2} + i_3h_2z^{-1} + i_3h_3z^0$$

---

etc. $\quad i_3h_0z^{-3} + c_{-2}z^{-2} + c_{-1}z^{-1} + c_0z^0 + b_1z^1 + a_2z^2 + i_0h_3z^3$
$$i_4h_0z^{-4} + i_4h_1z^{-3} + i_4h_2z^{-2} + i_4h_3z^{-1}$$

$i_4h_0z^{-4} + d_{-3}z^{-3} + d_{-2}z^{-2} + d_{-1}z^{-1} + c_0z^0 + b_1z^1 + a_2z^2 + i_0h_3z^3$
$0 \quad 0 \quad 0 \quad 0$ (since $i_5 = 0$)

$0 \quad i_4h_0z^{-4} + d_{-3}z^{-3} + d_{-2}z^{-2} + d_{-1}z^{-1} + c_0z^0 + b_1z^1 + a_2z^2 + i_0h_3z^3$
$0 \quad 0 \quad 0 \quad 0$ (since $i_6 = 0$)

$0 \quad 0 \quad i_4h_0z^{-4} + d_{-3}z^{-3} + d_{-2}z^{-2} + d_{-1}z^{-1} + c_0z^0 + b_1z^1 + a_2z^2 + i_0h_3z^3$
$0 \quad 0 \quad 0 \quad 0$ (since $i_7 = 0$)

$0 \quad 0 \quad 0 \quad i_4h_0z^{-4} + d_{-3}z^{-3} + d_{-2}z^{-2} + d_{-1}z^{-1} + c_0z^0 + b_1z^1 + a_2z^2 + i_0h_3z^3$
$$o_7z^{-4} + o_6z^{-3} + o_5z^{-2} + o_4z^{-1} + o_3z^0 + o_2z^1 + o_1z^2 + o_0z^3$$
$$z^3\,[\,o_7z^{-7} + o_6z^{-6} + o_5z^{-5} + o_4z^{-4} + o_3z^{-3} + o_2z^{-2} + o_1z^{-1} + o_0]$$

(1.8.3)

This time, both the multiplicand H(z) and the multiplier I(z) are written with the largest power on the right. We then carry out the usual grade-school multiplication method as shown which yields a sequence of partial products. For example, partial product #0 is the product if we account only for i_0's contribution to the full product. Then partial product #1 is the full product if only i_0 and i_1 are accounted for. The additions to obtain these partial products are carried about by the adders in Fig 1.12.

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. 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 Fig 1.11 circuit analysis. The output polynomial is shown at the bottom, and appears in the reverse order compared to the Type A circuit considered earlier. One sees that the output coefficient sequence is $(o_0, o_1, o_2, o_3 \ldots\ldots) = (i_0h_0, a_2, b_1, c_0 \ldots\ldots)$.

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 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.

(a) The Type A 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 Fig 1.8, (replicated here),

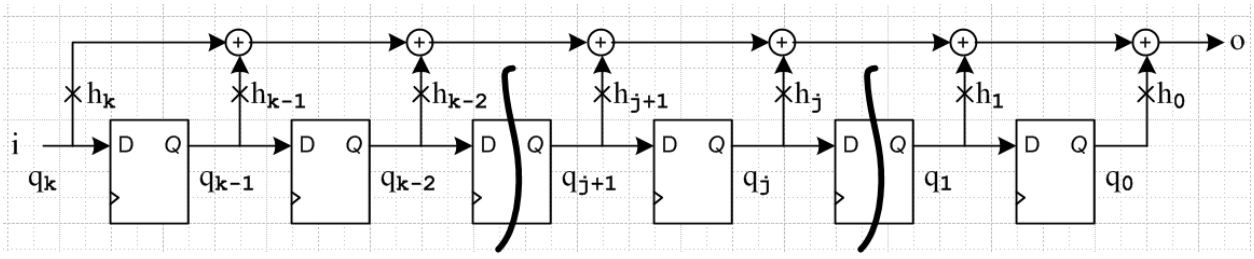


Fig 1.8: Type A polynomial multiplier.

we see that

$$q_0(n) = q_1(n-1) = q_2(n-2) = \text{etc} \, .$$

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_0 at time n *was* at q_1 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 Fig 1.8 where we identify q_k with the input data stream. Inserting expression (1.9.1) for $q_j(n)$ into (1.6.1) for o(n) gives

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

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 back 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 a set of difference equations. However, for the multiplier of Fig 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 Type B multiplier performs the same function as the Type A multiplier, the above equation applies to it as well.

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

(b) The Type A Divider in the Time Domain

We start with (1.2.1) which reads

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

Notice that Fig 1.1 (replicated below) has the same property noted in the previous section: the symbols move left to right from each register to the next without alteration.

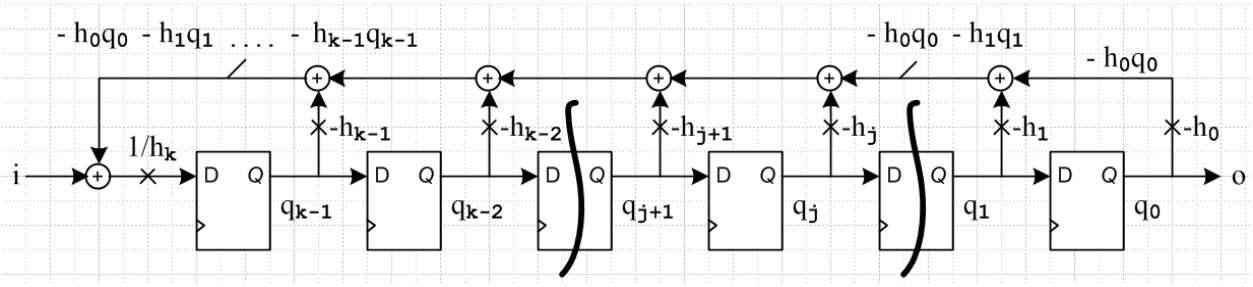


Fig 1.1: Type A polynomial divider.

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 k^{th} 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,

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

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,

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

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 Type B divider performs the same function as the Type A divider, the above equation applies to it as well. The reader is welcome to derive this fact by brute force from Fig 1.8.

(c) The Convolution Theorem Speaks

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

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

We showed that both the Type A and Type B multipliers implement this operation.

The reader is now referred to the box FT (24.37) which summarizes the properties of the Z Transform. Like all descendants of the underlying Fourier Integral Transform, there is a convolution theorem which operates with respect to the Z transform. It states:

$$A(z) = B(z)C(z) \quad \Leftrightarrow \quad a_n = \sum_{j=-\infty}^{\infty} b_{n-j}\, c_j \qquad \text{FT (24.7)} \qquad (1.9.6)$$

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

$$\begin{aligned} A(z) = z^k O(z) \quad &\Rightarrow \quad a_n = o_{n+k} \quad \text{// see Eq. (1.1.4)} \\ B(z) = I(z) \quad &\Rightarrow \quad b_n = i_n \\ C(z) = H(z^{-1}) \quad &\Rightarrow \quad c_n = h_{-n} \end{aligned} \qquad (1.9.7)$$

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.

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

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

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

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, and so $c_n = h_{-n}$ in (1.9.7).

Installing now the above time-domain sequences of (1.9.7) into the convolution sum of (1.9.6), then taking $j \rightarrow -j$ on the dummy summation index, we get

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

Since h_j vanishes for j<0 and for j>k, the final result is

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

which agrees with our derivation (1.9.2) above.

The point here is that, in the time domain, the action of the multiplier on the incoming symbol stream is to create a convolution sum with the h_j coefficients and this sum then naturally "diagonalizes" under the Z transform to the result $z^k O(z) = H(z)\, I(z)$ which is in fact polynomial multiplication.

We can 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 (1.9.8)

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

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 this Type B hybrid circuit:

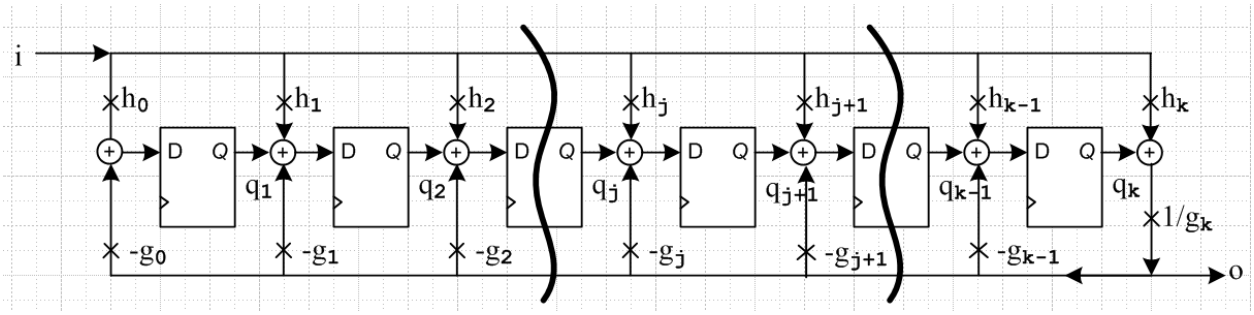


Fig 1.13: A simultaneous polynomial multiplier and divider.

This circuit looks like a superposition of a Type B multiplier on the top (multiply by polynomial H(z)), and a Type B divider on the bottom (divide 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 $g_i = 0$, Fig 1.13 becomes the multiplier of Fig 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 $h_i = 0$, Fig 1.13 becomes the divider of Fig 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

Looking at Fig 1.13 we see that,

$$q_{j+1}(n+1) = d_{j+1}(n) = q_j(n) + h_j\, i(n) - g_j\, o(n) \qquad j = 0,1,2....k-1 \quad . \qquad (1.10.1)$$

Project into the z domain to get, using the fourth line of (1.1.4) on the left, to get

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

Now multiply both sides by z^j,

$$z^{j+1}Q_{j+1}(z) = z^jQ_j(z) + h_jz^j\, I(z) - g_jz^jO(z) \qquad . \qquad (1.10.3)$$

For j=0 equation (1.10.1) says $q_1(n+1) = d_1(n) = q_0(n) + h_0\, i(n) - g_0\, o(n)$. Looking at the Figure we see that in fact $d_1(n) = h_0\, i(n) - g_0\, o(n)$, so it must be that $q_0(n) = 0$ and then $Q_0(z) = 0$.

For j = k we use (1.10.1) to define $q_{k+1}(n+1) = q_k(n) + h_k i(n) - g_k o(n)$. But the Figure shows that $g_k o(n) = q_k(n) + h_k i(n)$, so we conclude that $q_{k+1}(n+1) = 0$ and this $q_{k+1}(n) = 0$ and thus $Q_{k+1}(z) = 0$.

Equation (1.10.3) is a difference equation we can solve by inspecting the first several iterations:

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

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

$$z^3Q_3(z) = z^2Q_2(z) + h_2\, z^2\, I(z) - g_2\, z^2\, O(z) \qquad // j = 2$$
$$= (\, h_2z^2 + h_1 z + h_0\,)\, I(z) - (\, g_2z^2 + g_1 z + g_0\,)\, O(z)$$

Evidently the general solution is this:

$$z^{j+1}\, Q_{j+1}(z) = (\, h_j z^j + \ldots h_0\,)\, I(z) - (\, g_j z^j + \ldots g_0\,)\, O(z) \quad . \qquad (1.10.4)$$

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 find

$$O(z) = I(z)\, H(z) / G(z) \, . \qquad (1.10.5)$$

Sure enough, Fig 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 let's re-examine the two limiting cases mentioned above. If we select H(z) = 1, so that $h_0 = 1$, then we recover the divider of Fig 1.5:

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

On the other hand, to get a multiplier, we need to select G(z) such that $g_k = 1$ with other $g_j = 0$. This means that we have $G(z) = z^k$. Then we duplicate the results of the multiplier of Fig 1.10,

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

In an encoder for a cyclic code, one needs to compute the remainder of a power z^s times an incoming polynomial. See for example GA Chapter 8 (c), where s = n-k (a new k) is the number of parity check symbols for the code (n,k). The Fig 1.13 circuit can be used for this purpose. We have:

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

Thus, $H(z) = z^s$, which means that the input stream is injected into stage s in Fig 1.13. That is, h_s=1, and all other h_i vanish. By doing this injection into the middle of the shift register, we are in effect

"advancing" the input symbol stream by s units, as appropriate for the expression z^s I(z) (see (1.1.3)). If this s^{th} stage happens to have a vanishing coefficient of G(z), then no 3-input adders are needed in the circuit. We shall see an application of this idea in the following section.

Exercise 1. Is it possible to construct a simultaneous multiplier/divider from Figs 1.1 and 1.8 which use the Type A outboard adder structure?

Exercise 2. Carry out the time-domain analysis of Fig 1.13 and show that the resulting time-domain convolution equation is consistent with the Convolution Theorem and (1.10.4).

1.11 Cyclic Redundancy Check (CRC)

In this section the degree of polynomial H(z) is K ≡ n-k instead of k. This K is then the number of registers in a 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 (encoder) a Fig 1.13 Type B polynomial multiplier/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,

$$z^{n-k}d(z) = D(z)H(z) - \gamma(z) \ , \tag{1.11.1}$$

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" C(z) is given by,

$$C(z) \equiv D(z)H(z) = \gamma(z) + z^{n-k}d(z) \ , \tag{1.11.2}$$

where we just use the previous line to get the right side. If we write out the coefficients of C(z) in our standard form for I(z), which is highest power first, we find that the first k of them are the d_i data symbols and the last n-k of them are the γ_i parity check symbols. For example, if n=5 and k=3 then K = 2 and

$$\begin{aligned} C(z) = (\gamma_0+\gamma_1 z) + z^2(d_0+d_1 z+d_2 z^2) &= \gamma_0+\gamma_1 z+d_0 z^2+d_1 z^3+d_2 z^4 \\ &= z^4[\, d_2 + d_1 z^{-1} + d_0 z^{-2} + \gamma_1 z^{-3} + \gamma_0 z^{-4} \,] \\ &= z^4[\, c_0 + c_1 z^{-1} + c_2 z^{-2} + c_3 z^{-3} + c_4 z^{-4} \,] \ . \end{aligned} \tag{1.11.3}$$

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 $[z^{n-k}d(z)]/H(z)$. When the proper polynomial division has completed, the registers contain the remainder -γ(z). By grounding the feedback line in Fig 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). Although it is true that C(z) ≡ D(z)H(z), the encoder does not actually do this multiplication with a polynomial multiplier, it just assembles C(z) as shown above.

The polynomial C(z) (the coefficient set c_i) is then transmitted over some "channel". This could be a long cable of some sort, or a radio transmission, or it could be the "channel" implied by recording data on a CD and then playing it back later from a scratched CD. In our example, $c_0 = d_2$ is transmitted first.

At the receiving end we have a decoder circuit. The received C(z) is divided by H(z) by another Type B 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 non-zero remainder called the syndrome which is found sitting in the Fig 1.5 registers after the division. If a non-zero syndrome is found, a transmission error has been detected.

This error detection scheme is known as a Cyclic Redundancy Check (CRC). The subject is discussed more in GA Chapter 8. There, variable z is called x, and H(z) is called g(x).

Chapter 2: Shift Register Generators

2.1. The Maximum Period of a Shift Register Generator

Fig 1.1 and Fig 1.5 of Chapter 1 show the Type A an d Type B polynomial divider circuits. In either circuit, if we set the input to zero, preset the k registers to some initial values, and start the thing running, we get some output sequence. We shall refer to such a device as a **shift register generator,** or just a **generator** for short. Strictly speaking, only the Type A circuit contains a "shift register" which is a series of registers each directly feeding the next, but we shall use our term for both types. Another common term is a **linear feedback shift register** (LFSR) which again strictly applies only to the Type A design. (2.1.1)

Definition: A generator with k registers is an example of a **state machine** having k registers. In a general state machine, each register's input is some general function of the outputs of all k registers and usually of some external inputs as well. However, in our discussion below, we assume there are no external inputs. Since the state machine is presumed to contain only deterministic elements (no random noise generators for example), the behavior of the state machine will be **periodic**. (2.1.2)

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 can have. In other words, there are Q^k states of the state machine. (2.1.3)

Definition: The **state vector period** N 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. (2.1.4)

Definition: The **output period** P of some register (treated as an output) in a state machine is the minimum period of the output symbol sequence. This period is the same no matter where one starts 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 Fig 1.1. (2.1.5)

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. (2.1.6)

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 P = N/k. Thus, $P \leq N$. In any event, since $N \leq Q^k$, we know that both the state vector period N and the output period P 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 P = 2. The output period of each of the other bits is P = 10.

As is clear from Fig 1.1 and Fig 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 (barring errors perhaps caused by noise). 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 $Q^k - 1$. (2.1.7)

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 Type B Shift Register Generator

In this section we discuss only the Type B shift register generator (Fig 1.5). We will apply our Ref. GA 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 Type A 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.

(a) The State Iteration Equation for a Type B Divider

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

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

One can regard q(x) as a representation of the state vector of the shift register generator. That is to say, the actual state vector $\mathbf{q} = \{q_1, q_2 .... q_k\}$ is formed from the coefficients of q(x).

Comment on notation: In (2.2.1) we are not thinking of the values q_j as forming a temporal sequence the way we thought of the input samples i_j. The q_j all exist at the same moment in time, some clock period of our divider circuit. For this reason, it is not particularly useful to think of (2.2.1) as being a Z Transform of a temporal sequence q_j. To distance ourselves from Z transform polynomials, we use the variable x instead of z, and we use lower case names for the polynomial, here q(x). The same comment applies to our Chapter 1 polynomial H(z). Again, since the h_i are not really a temporal sequence of samples, we wish to steer away from Z transform notation, so this polynomial is here called h(x). Another motivation for variable x is that we are now going to be making use of Galois Field theory, and our entire supporting document GA is written using variable x.

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

$$q_{j+1}(s+1) = q_j(s) - h_j\ o(s) \qquad \textit{(1.4.1)}$$

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 to be:

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

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,

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

Now insert (2.2.2) into (2.2.3) and process the result:

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

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

$$= (\Sigma_{i=0}^{k-1}\ q_i x^i) - (o\ \Sigma_{i=0}^{k-1} h_i x^i) \qquad \text{// two sums}$$

$$= (q_0 + \Sigma_{i=1}^{k}\ q_i x^i - q_k\ x^k) - (o\ \Sigma_{i=0}^{k} h_i x^i - o h_k x^k) \qquad \text{// split each sum}$$

$$= (q_0 + \Sigma_{j=0}^{k-1}\ q_{j+1} x^{j+1} - q_k\ x^k) - (o\ \Sigma_{i=0}^{k} h_i x^i - o h_k x^k) \qquad \text{// } i = j+1 \text{ in first sum}$$

$$= (q_0 + x\ \Sigma_{j=0}^{k-1}\ q_{j+1} x^j - q_k\ x^k) - (o\ \Sigma_{i=0}^{k} h_i x^i - o h_k x^k) \qquad \text{// extract x from first sum}$$

$$= (q_0 + xq(x) - q_k\ x^k) - (oh(x) - o h_k x^k) \qquad \text{// identify } q(x) \text{ and } h(x) \equiv \Sigma_{i=0}^{k} h_i x^i$$

$$= xq(x) - oh(x) + (q_0 + [oh_k - q_k]x^k) \qquad \text{// rearrange}$$

$$= xq(x) - oh(x) + (i + [0]x^k) \qquad \text{// since } i = q_0 \text{ and } o = (1/h_k)q_k$$

$$= xq(x) - oh(x) + i\ .$$

We thus arrive at this state iteration equation for Fig 1.5:

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

h(x) is the polynomial we called H(z) in Chapter 1, namely

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

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 Fig 1.5.

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

(b) 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 be a set of m p-ary flip-flops so that a single register could then hold any of $Q = p^m$ different symbols. We could attempt to maintain this level of generality in the following presentation, but we shall not do so and shall assume from now on that each register consists of a single p-ary flip-flop that holds an element of $\text{Mod}(p) = Z_p$. Furthermore, we shall assume that p is a prime number, in which case we have $\text{Mod}(p) = Z_p = \text{GF}(p)$, the Galois Field with p elements. If one wants to consider a Reed-Solomon type shift register consisting of m p-ary flip-flops, then one must replace GF(p) with $\text{GF}(p^m)$ in the following discussion ($p \rightarrow p^m$).

Now, having limited our interest to the case where a single register holds an element of GF(p), a shift register consisting of k registers then can be regarded as holding an element of $\text{GF}(p^k)$. These elements are the possible state vectors of the shift register, and there are p^k possible states. All the math of the shift register logic occurs in the realm of GF(p). For example, the coefficients of h(x) lie in GF(p) and all the lines in our figures carry GF(p) symbols. Eventually, we will only care about p=2.

(c) Galois Field Review

Our downloadable reference GA contains a rather detailed review of Galois Field Theory. In this section we shall refer to quoted equations and facts from that document using the notation GA (x.x).

GA Chapter 3 discusses a certain construction or "polynomial representation" of the Galois Field $\text{GF}(p^k)$. Here is what that construction looks like:

$$\text{GF}(p^k) = R / (h(x))$$
$$h(x) = \text{irreducible polynomial of degree k, and h(x) is in R} \qquad \text{GA (3.17)} \qquad (2.2.6)$$

Here R is the ring of proper polynomials of any degree (but only non-negative powers) *whose coefficients lie in GF(p)*. In GA, k is called m and h(x) is called f(x), but we shall always use k and h(x) here. The p^k elements of the field $\text{GF}(p^k)$ are "represented" by the p^k possible remainders one can get by dividing an arbitrary polynomial F(x) in R by h(x). Since h(x) is of degree k, remainders must have degree < k, and if we count the number of such remainders (with coefficients in GF(p)) we find there are in fact p^k remainders. The notation R / (h(x)) has a specialized meaning (residue class ring) that is explained in GA and which we don't really need to know much about here. The term "irreducible" means that h(x) cannot be factored into a product of smaller factors whose coefficients lie in GF(p). For example, $x^2 + x$ is reducible, whereas x^2+1 is irreducible (note that imaginary $\pm i$ are not elements of GF(p)). For any prime p, GF(p) always has elements called 0 and 1 which are the additive and multiplicative identities. The p elements of GF(p) can be regarded as {0,1,2...p-1} since GF(p) = Mod(p).

Every Galois Field must have some rules (tables) describing how to do the field operations • and + between any two field elements. Since the field elements here are represented by remainder polynomials which lie in R, we need to know how to add and multiply such polynomials. First, here is the how • and + are defined for general elements a(x) and b(x) of R

$$a(x) + b(x) = (\Sigma_i a_i x^i) + (\Sigma_i b_i x^i) \equiv \Sigma_i (a_i \oplus b_i) x^i$$
$$a(x) \bullet b(x) = (\Sigma_i a_i x^i) \bullet (\Sigma_j b_j x^j) \equiv \Sigma_{i,j} (a_i \otimes b_j) x^{i+j} \quad , \qquad \text{GA (3.2)}$$

where $\oplus$ and $\otimes$ are the field operations for GF(p). If a(x) and b(x) are remainder polynomials, it is possible that the product shown will have degree larger than k-1, in which case we have to obtain the remainder of the product, so for remainders we modify the above to read

$$a(x) + b(x) = \Sigma_i (a_i \oplus b_i) x^i$$
$$a(x) \bullet b(x) = \text{Rem}\,[\, \Sigma_{i,j} (a_i \otimes b_j) x^{i+j} / h(x) \,] \quad . \qquad (2.2.7)$$

So given the + and • operations of GF(p) [called $\oplus$ and $\otimes$ above], the reader now knows exactly how to add or multiply any two elements of GF(p^k). Since GF(p) = Mod(p), the $\oplus$ and $\otimes$ operations are easily determined.

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

$$\{x\, h(x)\} = \{h(x)\} = \{0\} = \mathbf{0} \quad . \qquad (2.2.8)$$

Clearly, all three polynomials xh(x) , h(x) , and 0 have remainder 0 when divided by h(x). This element of GF(p^k) must be the additive identity **0**. If you add h(x) or a multiple of h(x) to some other polynomial, you do not change the remainder. Below is a list of properties involving the {...} notation taken directly from GA. When p is a prime number, GF(p) is the same as $Z_p \equiv$ Mod(p). Remember that GF(p) is only a Galois Field when p is prime, otherwise GF(p) has no meaning (though Mod(p) still has a meaning).

Facts using {} notation:

(a) $\{r_1(x)+r_2(x)\} = \{r_1(x)\} + \{r_2(x)\}$
(b) $\{r_1(x) \bullet r_2(x)\} = \{r_1(x)\} \bullet \{r_2(x)\}$
(c) $\{c\, r_2(x)\} = c\{r_2(x)\}$ for c in $GF(p)_p$
(d) $\{cx^k\} = c\{x^k\}$ for c in GF(p)
(e) $\{x^k\} = \{x\}^k$
(f) $\{p(x)\} = p(\{x\})$ where p(x) is a polynomial with coefficients in GF(p)
(g) $\{r_1(x)\} + \{r_2(x)\} = \{r_3(x)\}$ where $r_3(x) = r_1(x) + r_2(x)$
(h) $\{r_1(x)\} \bullet \{r_2(x)\} = \{r_4(x)\}$ where $r_4(x) = \text{Rem}[\, (r_1(x) \bullet r_2(x))/f(x) \,]$. GA (3.14) (2.2.9)

This is of course just the beginning of Galois Theory and the reader is referred to GA, especially in regard to things like minimum polynomials, primitive elements and other matters. We cannot digress to fully

explain minimal polynomials here, so it will be assumed that the reader knows what these are. If not, they are explained in GA Chapter 5.

When we say h(x) is of degree k, we imply that $h_k \neq 0$. 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 often use q as a shorthand for p^k.

We now quote two "facts" from GA using equation numbers from that reference:

• If monic h(x) in R is irreducible in R and h(α) = 0 for some α in GF(q), then h(x) is the minimum polynomial m(x) of α. GA (5.6)

• h(α) = 0 for α = {x} . GA (6.5)

The proof of this second item is basically that h({x}) = {h(x)} = 0.

We can combine these two statements to obtain the following statement, just using α = {x} :

Fact 3: If monic h(x) in R is irreducible in R, then h(x) is the minimum polynomial m(x) of element {x} of GF(q=p^k). (2.2.10)

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 one could factor out x from h(x) and then h(x) would be reducible. In terms of Fig 1.5, the implication is that some feedback must always go *to* the leftmost register. And in Fig 1.1, 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.

In GA Chapter 4 it is shown that the q-1 = p^k- 1 elements of {GF(q) - 0 } form a cyclic group with respect to • and that therefore there exist certain elements α, called **primitive elements**, in terms of which these q-1 non-zero elements of GF(q) can be enumerated as powers α^i . This then leads to the following complete enumeration of the elements of GF(q), where $1 = \alpha^0$:

$$GF(q) = \{\ 0,\ 1,\ \alpha,\ \alpha^2,\ \alpha^3, \ldots\ldots \alpha^{q-2}\ \},$$
$$\alpha^{q-1} = 1 \quad \alpha^q = \alpha \quad . \qquad \text{GA (4.30), (4.31)} \qquad (2.2.11)$$

Definition: Order of α. As discussed in GA (4.17), any non-zero element α of GF(q) has an **order** -- a minimum integer n such that $\alpha^n = 1$. Primitive elements have order n = q-1 as the enumeration above implies. The order of any element must integrally divide q-1.

Fact 4: The order of {x} must exceed 1. (2.2.12)

Proof: Here we take α as being the element {x} of GF(q), so α = {x}. If this element has order n = 1, that means that $\alpha^1 = \{x\}^1 = 1$, and in our construction above 1 means {1}, the element of GF(1) associated with remainder 1. Then {x} = {1}. If this were true, and if f(x) were an arbitrary polynomial, then we could say {x}•{f(x)} = {1}•{f(x)} which says {xf(x)} = {f(x)}. But this cannot in general be true for arbitrary f(x), so we get a contradiction assuming n = 1 for {x}. One loophole exists 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 h(x) = ax-a ($h_1 = a$, $h_0 = -a$)

$$\begin{array}{rl} & \underline{\quad (K/a)} \\ ax+b \;| & \quad xK \\ & \underline{xK + (K/a)b} \\ & \quad -K(b/a) \qquad = K \qquad \Rightarrow b = -a \end{array}$$

Here is a sketch of this fascinating special case of Fig 1.5:

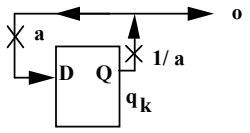


Fig 2.1

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.

(d) The Galois Iterator for a Type B divider

We now assume that our h(x) whose coefficients appear in the Type B Fig 1.5 is irreducible in GF(p), and therefore serves as the defining polynomial for our GF(p^k) = R / (h(x)) Galois Field representation.

Consider again the state iteration equation for a Type B divider derived above,

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

Each side of this equation is a polynomial in x. Recall that o and i are just constants which represent the values of the o and i bus in Fig 1.5 at some particular clock time during which q(x) records the state of the registers, and q'(x) will be that state one clock later. If we apply the Galois representation operation {...} to both sides and use the properties shown in (2.2.9) we obtain

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

Note that {i} = i{1} and {o} = o{1}, meaning that {i} and {o} are just constant multiples of the GF(q) identity {1}. Meanwhile, since {h(x)} = 0, the above line becomes

$$\{q'(x)\} = \{x\} \bullet \{q(x)\} + \{i\} \quad . \tag{2.2.13}$$

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

$$\alpha = \{x\} \qquad \beta = \{q(x)\} \qquad \beta' = \{q'(x)\} \qquad \{i\} = i\{1\} = i\,1 \quad . \tag{2.2.14}$$

Then we get this Galois statement of what happens with a single clocking of our Type B divider,

$$\beta' = \alpha \bullet \beta + i1 = \alpha\,\beta + i\,1 \quad . \tag{2.2.13a}$$

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 add back our time s subscripts as follows:

$$\beta_{s+1} = \alpha\,\beta_s + i_s\,1 \quad . \tag{2.2.15}$$

which is a difference equation for β_s. 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 (2.2.15) for β_s :

$$\begin{aligned}
&\beta_1 = \alpha\,\beta_0 + i_0 \\
&\beta_2 = \alpha\,\beta_1 + i_1 \quad = \alpha^2\,\beta_0 + [\,\alpha\,i_0 + i_1\,]\,1 \\
&\beta_3 = \alpha\,\beta_2 + i_2 \quad = \alpha^3\,\beta_0 + [\,\alpha^2\,i_0 + \alpha\,i_1 + i_2\,]\,1 \\
&\ldots. \\
&\beta_s = \alpha^s\,\beta_0 + [\,\alpha^{s-1}\,i_0 + \alpha^{s-2}\,i_1 + \ldots. + \alpha\,i_{s-2} + i_{s-1}\,]\,1
\end{aligned} \tag{2.2.16}$$

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

In Chapter 1 our time origin t = 0 was associated with the time period that input symbol i_0 is present on the input i bus of Fig 1.1 or Fig 1.5. We now wish to move this **time origin** so the new t = 0 is associated with the time period just after sample i_r has been clocked into the divider and we have i_{r+1} sitting on the input i bus. Thus, we are moving the time origin r+1 clocks into the future from where it used to be. In our new time base, then, we shall call that next input symbol i_{r+1} by the new name i_0. So starting at our new definition of t = 0, we shall have a sequence of input symbols i_0, i_1....and these are the input symbols which appear in (2.2.16).

If we set all these input symbols to 0, the Type B divider becomes a shift register generator whose state was initialized by those earlier input symbols. Since we are interested now in shift register generators, we set all $i_s = 0$ in (2.2.15) and (2.2.16) for $s \geq 0$. This leads to a significant simplification of (2.2.16) :

$$\beta_s = \alpha^s\,\beta_0 \quad . \tag{2.2.17}$$

Fact 5: The state vector of a Type B shift register generator steps periodically through some subset N of the p^k - 1 non-zero elements (states) 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 $N \le p^k - 1$. Furthermore, we claim that the integer N is the order of $\alpha = \{x\}$. (2.2.18)

Proof: If $\beta_0 = 0$, then $\beta_s = 0$ and the shift register has "died", so assume $\beta_0 \neq 0$. We know that $\alpha = \{x\}$ cannot be 0, so β_s will then never vanish. The reason is that 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.

We know that the element $\alpha = \{x\}$ has some order n which integrally divides p^k - 1. Thus, by the definition of "order" given earlier, n is the smallest integer such that $\alpha^n = 1$. Therefore we can write, setting s→s+n in (2.2.17),

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

which says that n is the period of the state vector sequence β_0, β_1..... !! We normally refer to this state vector period as N, so we have just shown that N = n, the order of {x}.

Fact 6: If p^k - 1 happens to be a prime number, the Type B state vector period is $N = p^k - 1$. (2.2.19)

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 $N = p^k - 1$ since there are no other divisors of p^k - 1 (since this was assumed prime).

Example: For GF(2^3) we have $p^k - 1 = 2^3 - 1 = 7$ which is a prime number. Thus, if we start a 3-register version of Fig 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 Fig 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, or N=511.

Comment: The conclusions of the previous Examples seem fairly obvious without the need for any fancy Galois Theory. What is perhaps not so obvious is that the shift register won't hit the zero state and then be completely dead after that point. In contrast, the conclusions of the next set of Facts are definitely non-obvious and Galois Theory shows its muscle.

Fact 7: If $\alpha = \{x\}$ is a *primitive element* of GF($q=p^k$), then the Type B state vector period is N = q-1. In this case, the state vector sequence exhausts all q-1 non-zero elements of GF(q). (2.2.20)

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: (curly brackets here just denote sets)

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

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 of β_s is then equal to the maximum value quoted in Fact 5, N = q-1.

Fact 8: If h(x) is a *primitive polynomial* of GF(q=p^k), then the Type B state vector period is N = p^k-1. (2.2.21)

Proof: If h(x) is a primitive polynomial of GF(q), then h(x) is a minimal polynomial for some primitive element α of GF(q) and h(x) therefore has order q-1. The set of roots of h(x) is known as the conjugate set of α and all these roots must have the same order (see GA (5.35)) which in this case is q-1. Thus, all the roots of h(x) are primitive elements of GF(q). Since {x} is a known root of h(x), {x} is then a primitive element, so the results follow from Fact 7.

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, as shown in GA (5.21). For assistance, we first peruse Appendix C of Peterson and Weldon [PW]. 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 GA (5.32), so there are 48 primitive polynomials in all. Each set of conjugates containing a primitive element contains k=9 distinct elements as noted in GA (5.15), 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). If your copy of Peterson and Weldon is not handy, a list of primitive polynomials for GF(2^9) is available on the Olofsson web site (Ref. OL):

```
Primitive polynomials over GF( 2 ) of degree 9
============================================================
The following list contains all 48 primitive polynomials
over GF( 2 ) of degree 9 .

x^9 + x^7 + x^4 + x^2 + 1
x^9 + x^8 + x^7 + x^6 + x^5 + x + 1
x^9 + x^8 + x^7 + x^5 + x^4 + x^2 + 1
x^9 + x^6 + x^5 + x^4 + x^2 + x + 1
x^9 + x^8 + x^7 + x^6 + x^3 + x^2 + 1
x^9 + x^7 + x^5 + x^2 + 1
x^9 + x^4 + x^3 + x + 1
x^9 + x^8 + x^5 + x^4 + x^3 + x + 1
x^9 + x^8 + x^6 + x^5 + x^4 + x + 1
x^9 + x^7 + x^2 + x + 1
x^9 + x^8 + x^6 + x^5 + x^3 + x^2 + 1
x^9 + x^4 + 1
x^9 + x^8 + x^5 + x^4 + 1
x^9 + x^8 + x^6 + x^3 + x^2 + x + 1
x^9 + x^5 + 1
x^9 + x^7 + x^5 + x^4 + x^2 + x + 1
x^9 + x^8 + x^7 + x^6 + x^4 + x^3 + 1
x^9 + x^8 + x^4 + x + 1
x^9 + x^7 + x^6 + x^3 + x^2 + x + 1
x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + 1
x^9 + x^6 + x^5 + x^4 + x^3 + x^2 + 1
x^9 + x^8 + x^5 + x + 1
x^9 + x^8 + x^7 + x^6 + x^3 + x + 1
x^9 + x^6 + x^4 + x^3 + x^2 + x + 1
x^9 + x^7 + x^5 + x + 1
x^9 + x^6 + x^4 + x^3 + 1

x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
x^9 + x^8 + x^7 + x^6 + x^4 + x^2 + 1
x^9 + x^6 + x^5 + x^3 + x^2 + x + 1
x^9 + x^6 + x^5 + x^3 + 1
x^9 + x^8 + x^6 + x^4 + x^3 + x + 1
x^9 + x^7 + x^6 + x^5 + x^4 + x^3 + 1
x^9 + x^8 + x^7 + x^3 + x^2 + x + 1
x^9 + x^5 + x^4 + x + 1
x^9 + x^8 + x^6 + x^5 + x^3 + x + 1
x^9 + x^8 + x^7 + x^6 + x^2 + x + 1
x^9 + x^7 + x^6 + x^4 + x^3 + x + 1
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x + 1
x^9 + x^5 + x^3 + x^2 + 1
x^9 + x^7 + x^5 + x^4 + x^3 + x^2 + 1
x^9 + x^8 + x^4 + x^3 + x^2 + x + 1
x^9 + x^7 + x^6 + x^5 + x^4 + x^2 + 1
x^9 + x^8 + x^7 + x^2 + 1
x^9 + x^8 + x^7 + x^5 + x^4 + x^3 + 1
x^9 + x^8 + x^6 + x^5 + 1
x^9 + x^7 + x^5 + x^3 + x^2 + x + 1
x^9 + x^7 + x^6 + x^4 + 1
x^9 + x^8 + x^4 + x^2 + 1
```

Fig 2.2

It turns out (see Appendix G of GA) that the number of primitive elements of GF(q) is equal to the count of integers < q-1 which are "coprime" with q-1, and this count is called φ(q-1) where φ is called Euler's totient function. Maple quickly confirms the number of primitive elements of GF(2^9) ,

```
with(numtheory):
phi(511);
                    432
```

The number of primitive polynomials is of GF(q = p^k) is then φ(p^k-1)/k .

Returning to Peterson and Weldon [PW] 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 denoted in an octal notation ordered so the lowest power (always $x^0 = 1$) is on the right, whereas the polynomials are shown with this lowest power on the left. Here are the first two table entries:

1 1021_8 h(x) = (x - a) ... = minimum polynomial containing a

$1021_8 = 001,000,010,001 \Rightarrow h(x) = 1 + x^4 + x^9$

reflected version is: $h(x) = 1 + x^5 + x^9$ (2.2.22)

3 1131_8 h(x) = (x - a^3) ... = minimum polynomial containing a^3

$1131_8 = 001,001,011,001 \Rightarrow h(x) = 1 + x^3 + x^4 + x^6 + x^9$

reflected version is: $h(x) = 1 + x^3 + x^5 + x^6 + x^9$ (2.2.23)

We are interested in the h(x) with the fewest non-zero coefficients since this minimizes the number of adders needed in Fig 1.5. So we select $h(x) = 1 + x^4 + x^9$, which means we are choosing our $\alpha = \{x\}$ = "a" (the "a" of P&W shown in (2.2.22) above) as our primitive element.

Notice that this is exactly the polynomial selected by SMPTE for the 259M standard, as shown in Fig 1.4. However, the circuit shown in Fig 1.4 is a Type A divider, whereas our state sequence theory (so far) applies only to Type B dividers. Also, Fig 1.4 shows a scrambler where the input sequence is non-zero, whereas right now we are talking about shift register generators which have zero input. We have not yet arrived at a theory of scramblers, but it is in the offing.

(e) The non-terminating Remainder in polynomial division

For this section, we first make a few notational changes in line with those already carried out above.

Recall from Section 1.3 (b) "Sequence of Remainders" our interpretation of polynomial division as follows:

$$I_r(z) = i_0 z^r + i_1 z^{r-1} + i_2 z^{r-2} + \ldots\ldots + i_r \ . \qquad (1.3.4)$$

$$I_r(z)/H(z) = \mathcal{Q}^{(r)}(z) + R^{(r)}(z)/H(z) \ . \qquad (1.3.18)$$

If we now bring in another n input symbols $i_{r+1}\ldots\ldots i_{r+n}$ the above becomes,

$$I_{r+n}(z) = \overset{\text{old } t=0}{i_0 z^{r+n} + i_1 z^{r+n-1} + i_2 z^{r+n-2} + \ldots\ldots + i_r z^n} + i_{r+1}z^{n-1} + \ldots + i_{r+n}$$

$$I_{r+n}(z)/H(z) = \mathcal{Q}^{(r+n)}(z) + R^{(r+n)}(z)/H(z) \ . \qquad (2.2.24)$$

If we now shift our time origin as discussed below (2.2.16) so that the new t = 0 occurs just after i_r has been shifted into the divider, and if we then *rename* our input symbols accordingly, we then subtract r+1 from all i_j indices to get

$$I_{r+n}(z) = i_{-r-1} z^{r+n} + i_{-r} z^{r+n-1} + i_{-r+1} z^{r+n-2} + \ldots\ldots + \overset{\text{new } t=0}{i_{-1} z^n} + i_0 z^{n-1} + \ldots + i_{n-1} \ . \qquad (2.2.25)$$

The value of r here is some unspecified number which provides a stream of input samples during $t < 0$ which have been flowing into our divider causing it have some unspecified state vector at t = 0. The divider registers were presumably cleared to 0 some time in the distant past before any inputs arrived.

We next define the following newly-labeled polynomials

$$i_n(z) \equiv I_{r+n}(z) \qquad r_n(z) \equiv R^{(r+n)}(z)$$

$$o_n(z) \equiv \mathcal{Q}^{(r+n)}(z) \qquad h(z) \equiv H(z) \qquad (2.2.26)$$

Notice that $i_n(z)$ is a polynomial, whereas i_j refers to an input stream sample, so our notation is a bit overloaded here. We shall always show the polynomial argument when referring to a polynomial.

The idea here is that $i_n(z)$, $o_n(z)$ and $r_n(z)$ are the input polynomial, quotient polynomial, and remainder polynomial at time t = n at which time "new" symbol i_n is sitting on the input i bus, waiting to be clocked into the divider on the next clock. For example, $i_0(z) = I_r(z)$ as shown in (2.2.25) with n=0 is the input polynomial at the time new sample i_0 (old sample i_{r+1}) is about to get clocked in.

Our final notational change is to switch to variable x to be consistent with the rest of this chapter,

$$i_n(x)/h(x) \;= o_n(x) + r_n(x)/\, h(x) \;. \tag{2.2.27}$$

Now let us return to the time t = 0 when new symbol i_0 is about to be clocked into the divider. At this time we have

$$i_0(x)/\, h(x) = \; o_0(x) + \; r_0(x)/h(x) \;. \tag{2.2.28}$$

If the input samples are 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 $r_0(x)$ appearing in the above equation, as discussed in Section 1.5 (a). That is, the coefficients of this polynomial $r_0(x)$ are the starting contents of the k registers. After one clock, we have:

$$i_1(x) \,/\, h(x) \;\; = \;\; [\;\; x\; i_0(x) \;\;] \,/\, h(x) \;\; = o_1(x) + \; r_1(x)/h(x) \;\; . \tag{2.2.29}$$

The input polynomial $i_1(x) = x\; i_0(x)$, because we have added a trailing 0 to the input symbol stream, as was discussed in (1.3.19) . After s clocks we would have

$$i_s(x) \,/\, h(x) \;\; = \;\; [\;\; x^s\; i_0(x) \;\;] \,/\, h(x) \;\; = o_s(x) + \; r_s(x)/h(x) \;\; . \tag{2.2.30}$$

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 (2.2.31)

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

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 $N = p^k - 1$.

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

Comment: In Appendix A we show an example of polynomial division (coefficients in Mod(10)) where the quotient polynomial never terminates and has a repeating sequence of symbols. Fact 9 gives us many examples of the same thing, where now coefficients are in GF(p) and where h(x) is an irreducible polynomial of degree k.

Definition: The **period** n of h(x).

This definition requires a bit more Galois background material. Recall from Fact 3 (2.2.10) above that if h(x) is monic and irreducible over GF(p), then h(x) is a minimum polynomial of GF(q). Any minimum polynomial h(x) of GF(q) can be factored into factors of the form $(x-a_i)$ where the values of a_i are non-zero elements of GF(q). Here is the general formula for such a minimum polynomial taken from GA,

$$h(x) = (x - \alpha) \bullet (x - \alpha^{p}) \bullet (x - \alpha^{p^2}) \bullet (x - \alpha^{p^3}) \ldots .... (x - \alpha^{p^{s-1}}) \qquad \alpha^{p^s} = \alpha \quad s \le k \quad . \quad \text{GA (5.17)}$$

The set of GF(q) elements appearing here, $\{ \alpha, \alpha^{p}, \alpha^{p^2} ....\}$, is called the conjugate set of α. It happens that if α is a primitive element of GF(q), then s = k. In any event, when the factors in GA (5.17) are multiplied out, h(x) is a polynomial of degree $s \le k$ whose coefficients lie in GF(p) as well as in GF(q). It turns out that the product of $(x-a_i)$ factors with a_i being *all* non-zero elements of GF(q) is $(x^{q-1} - 1)$,

$$(x^{q-1} - 1) = (x - \mathbf{1})\bullet(x - a_2)\bullet(x - a_3)\bullet(x - a_4)\bullet.....(x - a_{q-1}) \; . \qquad \text{GA (4.34)}$$

Since any h(x) contains some subset of these factors, any h(x) divides evenly into $(x^{q-1} - 1)$. It is possible, however, that h(x) might divide into $(x^n - 1)$ for some integer n that is *less* than q-1. This integer n is called the **period** of h(x). It is conventional to refer to $(x^n - 1)/h(x)$ as $g(x) = (x^n - 1)/h(x)$.

Fact 10: The state vector period N of a Type B shift register generator equals the period n of h(x) . (2.2.32)

Proof: 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,

$$r_s(x) = \text{Rem}[\; x^s \, i_0(x) / h(x) \;] \; . \qquad (2.2.33)$$

Then we can evaluate

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

$$= \text{Rem}[\; (x^{s+n} - x^s) \, i_0(x) / h(x) \;]$$

$$= \text{Rem}[\; i_0(x) \, x^s \, (x^n - 1) / h(x) \;]$$

$$= \text{Rem}[\; i_0(x) \, x^s \, g(x) \;]$$

$$= 0 \; .$$

This shows that $r_{s+n}(x) = r_s(x)$, which says that the state vector period is N = n since we identify the sequence of state vectors with the sequence of Type B division remainders. 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. **QED**

Summary: This section has focused on the state vector as being an element of a Galois Field $GF(p^k)$. For the Type B divider circuit of Fig 1.5, this state vector can be identified with the remainder of polynomial division. We learned that the state vector period N of a Type B shift register generator is determined by the nature of h(x) -- N is equal to the period of h(x). The nature of the Type A divider's state vector sequence will be addressed in (2.3.20,21) below.

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 Fig 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 output period of any particular register of a Type A or Type B shift register generator circuit must integrally divide the state vector period N. (2.2.34)

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, one can imagine that a particular register might have a smaller period that divides evenly into the state vector period N.

Corollary: The *output period* of a Type A or Type B shift register generator must be a divisor of the state vector period N. (2.2.35)

This is so because the output of either generator type is equal to (or is a constant multiple of) one of the individual registers and so Fact 11 then applies to such an output.

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

(a) The connection between Output Sequences and Cyclic Codes

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

In this section, we concentrate on the *output sequence* of a Type A shift register generator, rather than on the state vector of its register set. The Type A state vector behavior can then be obtained by considering groupings of k consecutive output symbols, since that is what the contents of the Type A registers are (see Fig 1.1).

The fundamental equation of interest here is the time-domain description of the polynomial divider derived in Section 1.9 as (1.9.5),

$$i_n = \sum_{j=0}^{k} h_j \, o_{n+j} \qquad n = \text{any integer} \; . \qquad \textit{(1.9.5)} \qquad (2.3.1)$$

Equation (2.3.1) is valid for both Type A and Type B dividers since it corresponds in the z-domain to the statement that O(z) = I(z)/H(z) which we know is true for both divider types. Below we show that the solution of (2.3.1) is a function of a set of initial values $\{o_{k-1}, \ldots o_0\}$ which are the first k outputs of a shift register generator (Type A or Type B). It happens that for the Type A generator this set of initial values corresponds to the initial state vector. Except where stated otherwise, all the Facts which appear below in this section apply to the outputs of both Type A and Type B shift register generators.

If we set the input sequence to zero, $i_n = 0$ for $n \geq 0$, (2.3.1) becomes this time-domain equation which governs a shift register generator (Type A or Type B)

$$\sum_{j=0}^{k} h_j o_{n+j} = 0 \qquad n = 0,1,2.... \; . \qquad (2.3.2)$$

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

$$h_k \, o_k = -h_0 \, o_0 - h_1 \, o_1 - \ldots - h_{k-1} \, o_{k-1} \; . \qquad (2.3.3)$$

This last equation is valid for either Type A or Type B generators, but for a Type A generator it has a very visible interpretation. If the o bus holds sample o_0, then we know that the Type A registers hold the future outputs o_{k-1} through o_1, and it is then easy to interpret (2.3.3) in terms of Fig 1.1: The sum $h_k \, o_k$ is fed to the leftmost adder which then puts o_k at the D input of the leftmost register and that value is then destined to become the next output after o_{k-1}.

Equation (2.3.2) is a difference equation. If we assume some initial values for $\{o_{k-1}....o_0\}$, we can iteratively solve (2.32) for all remaining o_n. The first step is shown in (2.3.3), and the next step would be

$$h_k \, o_{k+1} = -h_0 \, o_1 - h_1 \, o_2 - \ldots - h_{k-1} \, o_k$$

$$= -h_0 \, o_1 - h_1 \, o_2 - \ldots - (h_{k-1}/h_k)\{ -h_0 \, o_0 - h_1 \, o_1 - \ldots - h_{k-1} \, o_{k-1}\} \; .$$

We shall solve (2.3.2) making use of the theory of cyclic codes. We start by making some observations about the properties that its solutions 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. (Type A or Type B) (2.3.4)

Proof: By "solution" we mean an infinite sequence

$$\text{solution} = \{o_j\} = \{ o_0, o_1, o_2, ... \}$$

Note: The curly brackets used here and below denote sets or sequences and have nothing to do with the special curly brackets of Section 2.2 (b) which denoted GF(q) polynomial remainders.

Consider Eq. (2.3.3) above. This shows that o_k is fully determined by the numbers $\{ o_{k-1}, ... o_0 \}$ which we call a "starting set". For the Type A circuit, this is the initial state vector, while for the Type B circuit it is just the first k outputs (which are of course some function of the Type B initial state vector and the h_j). If we iterate, we find that o_{k+1} is then determined by the set of numbers $\{ o_k, ... o_1 \}$. But we just noted that o_k is determined by $\{ o_{k-1}, ... o_0 \}$, therefore o_{k+1} is determined by $\{ o_{k-1}, ... o_0 \}$, as shown above (2.3.4). Similary, all subsequent o_{k+j} are determined by $\{ o_{k-1}, ... o_0 \}$. In other words, the entire solution is completely determined by the starting set $\{ o_{k-1}, ... o_0 \}$. Thus, the number of solutions is equal to the number of starting sets. Since each symbol o_i is a number in GF(p) = Mod(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 have different starting sets, and the starting set is the first k symbols of the solution sequence $\{o_j\}$. **QED**

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. (2.3.5)

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. We showed below (2.2.32) that if h(x) is a monic irreducible polynomial, then it is a minimal polynomial, and then n = q-1 is always a viable candidate for n.

Definition: Assuming that n exists, we define g(x) to be the quotient: $g(x) = (x^n - 1)/h(x)$. (2.3.6)

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). (2.3.7)

The subject of cyclic codes is elaborated in GA Chapter 8. We summarize some basic facts mostly from GA (8.1) :

- h(x) is of degree k, and g(x) is of degree n-k, and $g(x)h(x) = x^n - 1$. GA (8.1)

- 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).

- Since the d(x) have k coefficients in GF(p) (forming a data word), there are p^k possible data polynomials or data words. Therefore since c(x) = d(x)g(x) there are p^k possible code words.

- All cyclic permutations of a code word are code words. GA (8.17)

- The sum of any two code words is also a code word, because the sum of any two data words is a data word. That is, the mapping c(x) = d(x)g(x) is linear.

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 $\mathbf{c} = \{c_0, c_1, ...c_n\}$, 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

$$\{o_i\} = \{\mathbf{c},\mathbf{c},\mathbf{c} ...\}$$

is a solution of the difference equation (2.3.2). (2.3.8)

Proof: See Appendix B. This is the crucial Fact, so naturally the proof is more than a few lines. If n = 5, the meaning of the above sequence would be this

$$\{o_i\} = \{c_0, c_1, c_2, c_3, c_4, \quad c_0, c_1, c_2, c_3, c_4, \quad c_0, c_1, c_2, c_3, c_4, \quad .....\}$$

Fact 5: All p^k solutions of the difference equation (2.3.2) are of the form $\{o_i\} = \{\mathbf{c},\mathbf{c},\mathbf{c} ...\}$. (2.3.9)

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 (see early in Appendix B), so we have found p^k solutions of (2.3.2). According to Fact 1 above, (2.3.2) has exactly p^k solutions, so we have found them all, and they are therefore all of the form claimed.

Note that code word **c** = 0 gives the trivial solution $\{o_i\} = \{0,0,0.....\}$. We often ignore this trivial solution in our general discussion.

Fact 6: Any non-trivial 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. (2.3.10)

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

Example A for GF(16 = 2^4): h(x) not a primitive polynomial

According to the Peterson and Weldon [PW] Appendix C, here are some minimum polynomials for GF(2^4) where α is a primitive element:

1	23_8	$h(x) = (x - \alpha)(x - \alpha^2)(x - \alpha^4)(x - \alpha^8)$			α has order 15 ($\alpha^{15} = 1$)
		$23_8 = 010,011$	⇒	$h(x) = 1 + x + x^4$ = primitive	h(x) has period 15
3	37_8	$h(x) = (x - \alpha^3)(x - \alpha^6)(x - \alpha^{12})(x - \alpha^9)$			α^3 has order 5
		$37_8 = 011,111$	⇒	$h(x) = 1 + x + x^2 + x^3 + x^4$	h(x) has period 5
5	07_8	$h(x) = (x - \alpha^5)(x - \alpha^{10})$			α^5 has order 3
		$07_8 = 000,111$	⇒	$h(x) = 1 + x + x^2$	h(x) has period 3

(2.3.11)

These h(x) are candidates for defining the coefficients used in a shift register generator.

Digression: These are three of the four minimum polynomials of GF(2^4). For the interested reader, GA shows how these may be computed. Here are two quotes from the GA document. In the first, the display is the polynomial exponent set, the factored polynomial, and then the period of the polynomial which is the same as the order of any its roots. The second quote shows the results of multiplying out the factored forms and are seen to agree with (2.3.11) above.

```
p = 2, m = 4, q-1 = 15
Number of Primitive Polynomials is 2
Number of Minimum Polynomials is 4
```

$$[7, 11, 13, 14, Prim], (x - a^7)(x - a^{11})(x - a^{13})(x - a^{14}), 15$$
$$[5, 10], (x - a^5)(x - a^{10}), 3$$
$$[1, 2, 4, 8, Prim], (x - a)(x - a^2)(x - a^4)(x - a^8), 15$$
$$[3, 6, 9, 12], (x - a^3)(x - a^6)(x - a^9)(x - a^{12}), 5$$

GA (5.50)

$p_1(x)$	$= (x - \alpha)(x - \alpha^2)(x - \alpha^4)(x - \alpha^8)$	$= x^4 + x + 1$
$p_7(x)$	$= (x - \alpha^7)(x - \alpha^{14})(x - \alpha^{13})(x - \alpha^{11})$	$= x^4 + x^3 + 1$
$m_3(x)$	$= (x - \alpha^3)(x - \alpha^6)(x - \alpha^{12})(x - \alpha^9)$	$= x^4 + x^3 + x^2 + x + 1$
$m_5(x)$	$= (x - \alpha^5)(x - \alpha^{10})$	$= x^2 + x + 1$

GA (6.21)

For each of the h(x) candidates in (2.3.11), we indicate the order of the field element that appears in the first factor in that polynomial. Recall that $\alpha^{15} = 1$ since α is primitive, as shown for general GF(q) in (2.2.11).

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, but we shall use it anyway in this example.

Recall that the period of h(x) is the smallest integer such that $(x^n-1)/h(x) = g(x)$ is a polynomial. It is shown in GA Ch 5 (j) that this period is equal to the order of any of the roots of h(x), and it is shown in GA (4.32) that the order of a root α^k of h(x) is (q-1)/GCD(k,q-1). Thus for our selected h(x),

$$h(x) = 1 + x + x^2 + x^3 + x^4 = (x - \alpha^3)(x - \alpha^6)(x - \alpha^{12})(x - \alpha^9),$$

we find, using the root α^3, the period to be n = (q-1)/GCD(k,q-1) = (15)/GCD(3,15) = 15/3 = 5. This can also be determined by just trying all the values n = 1,3,5,15 which are divisors of 15 and finding the lowest one results in $\text{Rem}[(x^n-1)/h(x)] = 0$. Here Maple shows that 5 works :

```
h := 1 + x + x^2 + x^3 + x^4;
```

$$h := 1 + x + x^2 + x^3 + x^4$$

```
g := Quo(x^5-1,h,x) mod 2;
```

$$g := x + 1$$

```
Rem(x^5-1,h,x) mod 2;
```

$$0$$

Thus we find that g(x) = 1 + x.

Since n = 5, our code words have the form $\mathbf{c} = \{c_0, c_1, c_2, c_3, c_4\}$.

It is now an easy matter to list off all $p^k = 2^4 = 16$ words of the cyclic code generated by g(x). We start with the fact that in general c(x) = d(x)g(x) enumerates code words, and d(x) = 1 gives the particular code word c(x) = g(x) which we write out as

$$c_o + c_1x + c_2x^2 + c_3x^3 + c_4x^4 = 1 + x + 0x^2 + 0x^3 + 0x^4 \qquad (2.3.12)$$

or

$\mathbf{c} = \{ 1,1,0,0,0\}$.

Convention Note: There are two conventions for how to order coefficients. We are using the convention which puts the coefficient of the lowest power of the polynomial on the left. This is consistent with our statement for example that $\mathbf{c} = \{c_0, c_1, c_2, c_3, c_4\}$. This convention is the reverse of Peterson and Weldon as quoted in (2.2.22). In Ref. GA we use several notations to deal with this ambiguity,

$$f(x) = a + bx + kx^3 = \langle ab0k \rangle = [k0ba] = k0ba \qquad \text{// = a 4-tuple} \qquad \text{GA (4.6)}$$

but here we use {a,b,0,k} instead of <ab0k>.

In our search for code words, we show each newly discovered one in red. Of course {0,0,0,0,0} is always a code word. Right off the bat we can write 4 more code words by doing cyclic permutations of our first code word noted above,

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

Remembering that 1+1 = 0 in GF(2), we can form more code words by adding pairs of the above five, since the code words form a vector space. Adding the first above to the remaining four gives

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

We can then cyclically permute each of these in our search for new code words:

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

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

We can now stop since all $2^k = 16$ code words have been discovered. Notice that this is only half of the $2^5 = 32$ possible 5-tuples one can make with 0 and 1. For example, {1,1,1,1,1} is not a code word.

Looking back at Fact 5 above, we now know all 16 possible output sequences of a 4-register generator which has $h(x) = 1 + x + x^2 + x^3 + x^4$. For example, one is $\{o_i\}$ = {0,1,1,0,0, 0,1,1,0,0 ....}.

Example B for GF(16 = 2^4): h(x) a primitive polynomial

We now repeat the previous example this time selecting h(x) to be the primitive polynomial $1 + x + x^4$. The period of this new h(x) is n = q-1 = p^k-1= 2^4-1 = 15, as Maple verifies:

```
h := 1 + x + x^4:
g := Quo(x^15-1,h,x) mod 2;
```

$$g := x^{11} + x^8 + x^7 + x^5 + x^3 + x^2 + x + 1$$

```
Rem(x^15-1,h,x) mod 2;
```

$$0$$

Then as in (2.3.12) we identify the code word c(x) = g(x) to be

c = {1,1,1,1,0,1,0,1,1,0,0,1,0,0,0} // 15 bits

Here is how Maple can produce this sequence from its knowledge of g(x) shown above,

```
c := coeff(g,x,0):
for n from 1 to 14 do
   c := c,coeff(g,x,n);
od:
c;
```

1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0

According to GA Chap 8 (k), since h(x) is a primitive polynomial we can form all non-zero code words by doing rotations of the above code word. So here is the full set of 15 non-zero code words, as computed by some non-pretty Maple code, where each line is a rotation to the right of the previous line by 1 bit position.

```
for n from 16 to 2 by -1 do
    d := seq(c[i],i=n..15),seq(c[i],i=1..n-1):
    num := convert(sum((d[16-j])*2^(j-1),j=1..15),octal);
    print(d,"              ",num);
od:
```

```
1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, "       ", 75310
0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, "       ", 36544
0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, "       ", 17262
 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, "       ", 7531
1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, "       ", 43654
0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, "       ", 21726
0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, "       ", 10753
1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, "       ", 44365
1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, "       ", 62172
0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, "       ", 31075
1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, "       ", 54436
0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, "       ", 26217
1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, "       ", 53107
1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, "       ", 65443
1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, "       ", 72621
```

(2.3.13)

The bit pattern is shown in octal on the right, to make it easier to see that all the code words really are different. Looking back again at Fact 5 above, we now know all 16 possible output sequences of a 4-register generator which has $h(x) = 1 + x + x^4$. For example, one is

$\{o_i\} = \{0,1,1,1,1,0,1,0,1,1,0,0,1,0,0,\quad 0,1,1,1,1,0,1,0,1,1,0,0,1,0,0\ ....\}$.

(b) The Maximum Length Sequence (MLS)

Fact 7: If h(x) is a primitive polynomial of GF(q), then no code word c(x) has a period less than $n \equiv q-1$

(2.3.14)

In other words, since c(x) corresponds to **c** which has n components $c = \{c_1,c_2....c_n\}$, this Fact is saying that the set of components cannot be partitioned into a repeating set of subsequences such as **c** = {a,b,a,b....a,b}.

Proof: Suppose c(x) has a period m = n/M for some integer M > 1. In other words, suppose **c** consists of an integral number M>1 of repeating subsequences. From GA Chap 8 (k) we know that if h(x) is a primitive polynomial, then all non-zero code words c(x) are rotations of g(x) which is one of the code words. Suppose then that **c** had a repeating subsequence of components, for example

$c(x) \sim \{abcde\ abcde\ abcde\}$.

Suppose g(x) were a rotation of one symbol left relative to c(x). Then we would have

$g(x) \sim \{bcde\ abcde\ abcde\ a\} = \{bcdea\ bcdea\ bcdea\}$.

For a rotation of some general number of symbols we would get

$g(x) \sim \{ABCDE\ ABCDE\ ABCDE\}$.

Then g(x) must have a repeating subsequence of period m. If this is so, we can write

$$g(x) = f(x)\,[\ 1 + x^{m} + x^{2m} + x^{3m} + \ldots + x^{(M-1)m}\ ]$$

where f(x) is a polynomial of degree m representing the pattern that repeats M times. But we know that

$$[1 + y + y^{2} + \ldots + y^{s-1}\ ] = (y^{s} - 1)/(y-1)$$

so setting $y = x^{m}$ and s = M-1 we find that

$$[\ 1 + x^{m} + x^{2m} + x^{3m} + \ldots + x^{(M-1)m}\ ] = (x^{Mm} - 1)/(y-1) = (x^{n} - 1)/(x^{m}-1)$$

and therefore

$$g(x) = f(x)\,(x^{n} - 1)/(x^{m}-1)\ .$$

But since $(x^{n} - 1) = g(x)h(x)$ this says

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

or

$$1 = f(x)\,h(x)/(x^{m}-1)\ .$$

or

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

This says that h(x) integrally divides $(x^{m}-1)$ with m < n, but when h(x) is a primitive polynomial, we know that the lowest power m must be n. Therefore we arrive at a contradiction, so g(x) cannot have a repeating subsequence and since all other code words are rotations of g(x), they cannot have repeating subsequences either. **QED**

Example: In Example B above we found that **c** = {1,1,1,1,0,1,0,1,1,0,0,1,0,0,0} is a code word. Notice that this cannot be partitioned into a set of repeating subsequences. The same is then true for all the other code words shown in Example B, since they are rotations of **c**.

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 $P = p^k - 1$. (2.3.15)

Proof: We know from Fact 7 that, when h(x) is a primitive polynomial, the period of any code word c(x) is $n = q-1 = p^k-1$. From Fact 4, all p^k- 1 solution sequences $\{o_i\}$ have the form $\{o_i\}$ = {**c**,**c**,**c**,**c** ...}. Thus all these solutions 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 (Type A or Type B), and its period is $n = p^k - 1$, the longest any sequence can possibly be (see Section 2.1). This pattern is often called a **maximum length sequence** (MLS), sometimes referred to as a **characteristic sequence**. We shall somewhat redundantly refer to this as an **MLS sequence**. When referring to an MLS sequence, we imply that the h(x) polynomial causing this pattern is a primitive polynomial of the appropriate GF(p^k).

(2.3.16)

Proof: If h(x) is a primitive polynomial of GF($q=p^k$), we know that it has order $n = q-1 = p^k-1$, and we know that we then have $g(x) = (x^n-1)/h(x)$ as the generator of a cyclic code whose code words **c** have n components. We know from (2.3.9) that all $n = p^k-1$ possible non-zero output sequences of a k-register shift register generator have the form $\{o_i\}$= {**c**,**c**,**c**,**c** ...} where **c** is any of the p^k-1 non-zero code words generated by g(x) and where **c** has $n = p^k-1$ components. Since the code is cyclic, one can exhaust all these code words by starting with any of them and forming others one at a time by rotating the previous code word one position to the left (again, see GA Chapter 8 (k) for a proof of this fact). For example, if we start with sequence $\{o_i\}$= {**c**,**c**,**c**,**c** ...}, we can form the next **solution** as $\{o_i\}$= {$\mathbf{c}_1$,$\mathbf{c}_1$,$\mathbf{c}_1$,$\mathbf{c}_1$ ...} where $\mathbf{c}_1$ is **c** rotated one symbol to the left. If **c** = abcdefg, then $\mathbf{c}_1$ = bcdefga. Thus, if our first sequence is $\{o_i\}$ = { abcdefg abcdefg.....}, our second **solution** is { bcdefga bcdefga.....}. But this is just the first solution with the first symbol omitted. In other words, all other output sequences may be obtained from any given sequence by deleting one or more of the first symbols of the original sequence. This being the case, all solution sequences $\{o_i\}$ are the same apart from where in the sequence you start writing the symbols down. This is what we mean by saying that there is essentially only one output pattern, and that pattern is the MLS sequence.

For example,

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

Fig 2.3

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

Example C for GF(512 = 2^9): h(x) a primitive polynomial

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 MLS output sequence and it has period $n = 2^9 - 1 = 511$ bits ($p = 2$ for $GF(2^9)$ so symbols are bits).

Since $c(x) = d(x)g(x)$, if we set $d(x) = 1$ we see that $c(x) = g(x)$ is one of the non-zero cyclic code words, so if we can find the $n = p^k-1 = 2^9-1 = 511$ coefficients of c(x) then we can use these to form the MLS sequence $\{o_i\} = \{\mathbf{c},\mathbf{c},\mathbf{c}...\}$. We first obtain g(x) from $g(x) = (x^n - 1)/h(x) = (x^{511} - 1)/h(x)$. We then write out the equation $c(x) = g(x)$ as

$$\begin{aligned} & c_0 + c_1x + c_2x^2 + \ldots \quad c_kx^k + \ldots + c_{n-1}x^{n-1} \\ = \; & g_0 + g_1x + g_2x^2 + \ldots h_{n-k}x^{n-k} + 0 + \ldots + 0x^{n-1} \end{aligned} \qquad (2.3.17)$$

Then the first $n-k+1 = 511-9+1 = 503$ coefficients of c(x) are those of g(x) and the last $k-1 = 9-1 = 8$ coefficients of c(x) are 0.

Maple can be used to compute **c** as follows. We first obtain the polynomial g(x),

```
> h := 1 + x^4 + x^9:
> Rem(x^511-1,h,x) mod 2;
                                  0
> g1 := Quo(x^511-1,h,x) mod 2:
> g := sort(g1,x);
```

$$g := x^{502}+x^{497}+x^{493}+x^{492}+x^{487}+x^{484}+x^{483}+x^{482}+x^{479}+x^{477}+x^{475}+x^{473}+x^{472}$$
$$+x^{467}+x^{466}+x^{464}+x^{463}+x^{462}+x^{461}+x^{459}+x^{456}+x^{455}+x^{453}+x^{452}+x^{451}+x^{448}$$

.....

$$+x^{64}+x^{63}+x^{60}+x^{59}+x^{58}+x^{57}+x^{56}+x^{53}+x^{52}+x^{50}+x^{49}+x^{45}+x^{43}+x^{41}+x^{38}+x^{34}$$
$$+x^{33}+x^{32}+x^{28}+x^{27}+x^{25}+x^{24}+x^{22}+x^{20}+x^{18}+x^{17}+x^{16}+x^{12}+x^9+x^8+x^4+1 \qquad (2.3.18)$$

Then we can extract the coefficients to get our 511-bit long vector **c** from which we can construct the endless MLS sequence $\{o_i\} = \{\mathbf{c},\mathbf{c},\mathbf{c}...\}$. In Maple, the comma operator causes a new element to be appended to a sequence.

```
> c := coeff(g,x,0):
> for n from 1 to 510 do
      c := c,coeff(g,x,n);
  od:
> c;
```

```
1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1,
  0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0,
  0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0,
  1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1,
  0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1,
  0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0,
  0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1,
  1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
  1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1,
  1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,
  0, 0, 0, 0, 0, 0, 0
```

(2.3.19)

You can see how the first bits match $1 + x^4 + x^8 + x^9 + x^{12}$ and so on. According to Fact 7, the above sequence cannot be partitioned into a set of repeating subsequences, just another example of Fact 7.

(c) Type A generator periodicity and the Sift List

In Section 2.1 we found that in the Type B Fig 1.5 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 Type B Fig 1.5 circuit with primitive polynomial h(x), 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 Type A Fig 1.1, 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. If all registers have the same period P, then the state vector for all the registers must also have period P. Therefore:

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

(2.3.20)

Proof: For a Type A shift register, all registers have the same period since each is a time shift of any other. The state vector certainly cannot have a period smaller than the period of a register. Thus, the state vector period is the same as that of any register, since certainly in this period all registers return to their original state.

Corollary: If h(x) is a primitive polynomial of GF(p^k), then the state vector period of a Type A shift register generator is the maximal period p^k- 1. (2.3.21)

This corollary is just a combination of Fact 9 and Fact 10.

Definition: Sift List Assume k = 3 and the period of all (non-zero) solutions is n = 2^3 - 1 = 7. Here we obtain from the output sequence shown a **sift list** of k-symbol vectors. To create this list, we start at the beginning of the output sequence and pick off the first k symbols and that is the first vector on the sift list. We then step one symbol to the right in the output sequence and do the same thing. In this example, one period's worth of the output sequence is the 7-symbol sequence abcdefg. The sift list associated with this one period's worth of output sequence symbols contains the 7 vectors shown in red. Notice that each of these vectors starts somewhere in the first abcdefg sequence. We shall refer to such a sift list as the sift list associated with one period of the output sequence. (2.3.22)

```
output sequence=      abcdefgabcdefgabcdefg......
state vectors =       abc
(Fig 1.1 circuit)      bcd
                       cde
                        def
                         efg
                          fga
                           gab
                            abc
                             bcd
                               .....
```
(2.3.23)

Here we see the periodicity of the output sequence matching the periodicity of the sift list. For a Type A shift register, we interpret this as saying the period of the output sequence matches the period of the state vector sequence. This is always the case since the Type A generator is in the form of a shift register as shown in Fig 1.1. We just proved this in Fact 10.

Fact 11: Each symbol in one period of an output sequence of a k-register shift register generator appears k times in the associated sift list. (2.3.24)

Proof by example: Looking at the above figure with k = 3, we see that symbol c appears in three vectors of the sift list associated with one period of the output sequence (red). So does d, e, f and g. Looking at the start and finish of the sift list, we see that a and b also appear k = 3 times. Thus, every one of the 7 symbols in one period of the output sequence appears k=3 times in the associated sift list. We leave it to the reader to generalize this "proof" to general k.

2.4. Properties of MLS 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 (MLS). The circuit can be implemented as either Type A Fig 1.1 or Type B Fig 1.5.

Here we shall see what we can learn about such an MLS sequence.

Observation: Different primitive polynomials h(x) of degree k are likely to produce different MLS sequences. We do not want to leave the impression that there is only one such sequence. In our Section 2.2 example of GF(2^9=512), we noted that there were 48 distinct primitive polynomials (Fig 2.2). One usually selects one of the two possible primitive polynomials which have the minimum number of non-zero coefficients.

Fact 1: Any k-symbol sequence can only begin once in any period of an MLS sequence of a k-register shift register generator. In terms of the sift list defined above, Fact 1 states that any k-symbol sequence can occur only once in the sift list associated with a period of the MLS sequence. (2.4.1)

Before proving Fact 1, here are two examples of a 4-symbol sequence which starts twice within a period of an MLS sequence. In the first case all four symbols are the same as required to make the start of the two periods shown consistent.

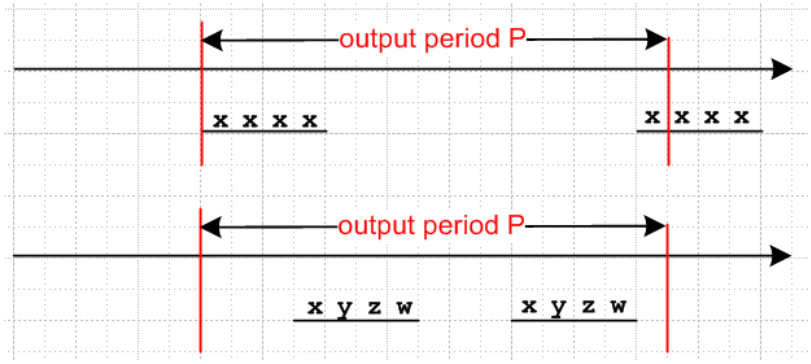


Fig 2.4

Neither of these situations can happen according to Fact 1. A weaker form of Fact 1 is to say that a given k-symbol sequence cannot occur twice within the period of an MLS sequence. This wording would seem to apply to the lower picture but not the upper, but Fact 1 applies to both cases.

Proof: If this fact is true for a Type A generator, it must also be true for a Type B generator since both have the same MLS sequence (see (2.3.1) and following paragraph). So we will prove the claim for Type A generator and then the claim will have been proven for both types.

Suppose within the period of the MLS sequence a certain k-symbol sequence were to start twice. One way this could happen is shown in the following figure, drawn for k = 4 and for a repeated sequence xxxx (the first example above)

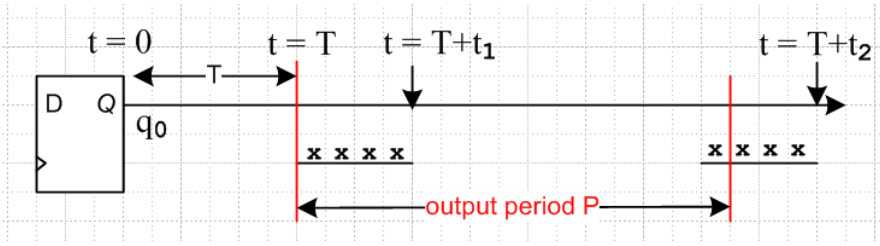


Fig 2.5

Since the Type A registers form a simple shift register, any k-symbol sequence in the output stream was at some earlier time the state vector of the Type A shift register generator. In the drawing, the first xxxx sequence was the state vector T + t_1 clocks earlier, while the second xxxx sequence was the state vector T + t_2 clocks earlier. Thus, the generator had the same state vector at two times separated by t_2 - t_1, so the state vector period must be t_2-t_1 (or less). Since the output period cannot exceed the state vector period of any generator, we conclude that the output period must be t_2-t_1 or perhaps even smaller, so $P \leq t_2$-t_1. But the picture shows that the output period $P > t_2$-t_1 even in the most extreme case of the largest possible value for t_2- t_1. In this extreme case, we have $P = t_2$-t_1+ 1. Since we then have $P \leq t_2$-t_1 and $P > t_2$-t_1, we have a contradiction. Therefore, the assumption that that same k-symbol sequence can appear twice within the period P is false. For any other positioning of the two xxxx sequences, the same conclusion is reached, although the value of t_2-t_1 will be smaller. The two sequences could even have partial overlap as shown here,

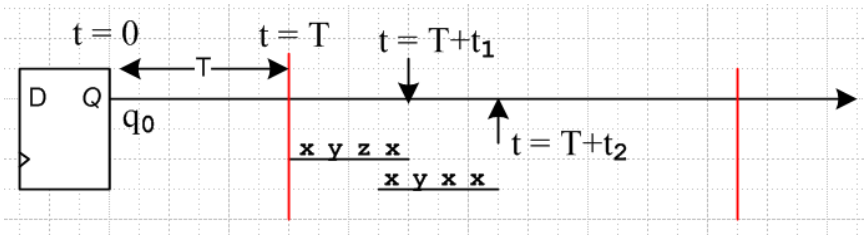


Fig 2.6

Fact 1 then provides a basis for the following observations:

Definition: A "**run** of some symbol of length N" refers to a consecutive string of all that symbol proceeded and followed by *some other* symbols. For GF(2) = {0,1} one can only have a run of 1's and a run of 0's. (2.4.2)

Here are two examples (where ***** = unspecified symbols)

*****011110***** a run of 4 ones

*****1001***** a run of 2 zeros

For GF(p) = {0,1,a,b,c....} we might have

*****abbbbc***** a run of 4 b's

where the terminating symbols can be the same or different.

Fact 2: In one period of an MLS sequence, the maximum run of zeros has length k-1. In the case of GF(2) symbols, this run occurs only once per period. (2.4.3)

Proof: First of all, suppose there were k or more consecutive 0's somewhere in period P. If that were possible, then a string of k zeros would have been in our Type A shift register at some earlier time. But that would have killed off the generator output forever, and thus we would not have an MLS sequence. This contradicts our assumption that we do have such a sequence in which these 0's are found. So even for general GF(p) symbols we cannot have a run of k or more zeros.

To show that there is always a string of k-1 zeros, we can think of the MLS sequence as belonging to code word c(x) = 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 exactly as illustrated in (2.3.12), so there we have a string of k-1 zeros for sure. Again, this is true for general GF(p) symbols.

For GF(2) only, we can show that this pattern of k-1 zeros cannot occur anywhere else in the period. If it did occur twice, and if we included an adjacent 1, we would have a k-bit state vector pattern repeating twice per period, which we showed in Fact 1 is impossible. As an example, for k = 4 we consider the possibility of a string of three 0's occurring twice within the period P:

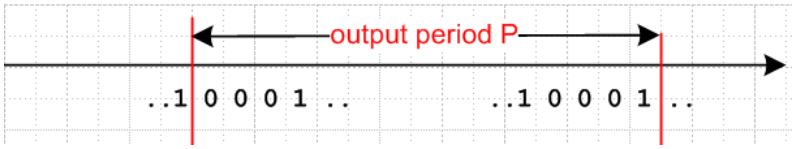


Fig 2.7

In this case (showing maximal separation) we have the k-length string 0001 starting twice within period P which Fact 1 says cannot happen. This proof fails for general GF(p) since we might have different symbols x and y on the two sides of the string of three zeros.

If we try move the right group one symbol to the right, we get a contradiction since the second period starts then with a 1 but the first starts with a 0, so the case of maximal separation is the one shown. As either or both groups are pulled toward the center of the drawing, the conclusion stays the same.

Fact 3: In one period of an MLS sequence, the maximum run of any non-zero symbol has length k, and such a run occurs exactly once per period for each non-zero symbol. (2.4.4)

Proof: The output period of an MLS sequence is $p^k - 1$. Since there are only this many non-zero possible state vectors for a Type A or Type B generator, and since the state vector period cannot be less than the output period, the state vector period is also $p^k - 1$. This means that during one period of an MLS sequence, every possible state vector is hit exactly once in the generator. This list of hit state vectors includes all sequences consisting of k identical non-zero symbols. Thinking momentarily of the Type A generator, this means that the MLS sequence therefore contains each such sequence of k identical non-zero symbols, since each Type A state vector flows into the output stream. We know that such a sequence must abut on each end with some different symbol. If that were not the case, we would have k+1 identical symbols in the MLS sequence period, but that is not allowed by Fact 1 since that would imply two strings of k such symbols starting in the same period. Thus, each string of k identical symbols occurs exactly once per output period.

Fact 4: In one period of an MLS 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 ,

$$(p^{k-1})\,(p-1) + (p^{k-1} - 1) = p^k - 1 = \text{total symbols in MLS sequence} \qquad (2.4.5)$$

Remember that the MLS sequence is associated with a primitive h(x) which is associated with GF(p^k).

Proof: Take all p^k state vectors and lay them side by side in a large 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 large set there must be $(kp^k) / p = kp^{k-1}$ occurrences of each symbol.

Now form a second set which includes all state vectors except the all-zero vector. All non-zero symbols occur in this second set kp^{k-1} times, just as in the first 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 second set.

Now create a sift list for one period of the MLS sequence as defined in (2.3.22). If the generator is Type A, then this sift list contains exactly the elements of the second set above. That is, our sift list contains all non-zero vectors exactly once each since the sift list in this case represents the sequence of state vectors of the Type A generator, see (2.3.23). Thus, there is a one-to-one correspondence between the symbols of our sift list and those of the second set. According to Fact 11 of (2.3.24), each symbol in the MLS sequence is repeated k times in the sift list. Thus we conclude from our analysis above of the second set that the MLS sequence contains $(kp^{k-1})/k = p^{k-1}$ copies of each non-zero symbol, and contains $(kp^{k-1} - k)/k = p^{k-1} - 1$ zero symbols. **QED**

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

We have seen that any period's worth of an MLS 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 4 (2.4.5) above, this weight is $(p-1)p^{k-1}$ since there are (p-1) types of non-zero symbols and each occurs p^{k-1} times in the MLS sequence. This is also the minimum weight of the code, which is the code *distance* as described in GA Chapter 7.

Fact 5: If we line up an MLS 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. (2.4.6)

Proof: An MLS 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). Assume that two code words differ in exactly N symbol places. When you subtract these two code words, you get a result which has N non-zero symbols. But this result is also a code word (the space of code words is a vector space), and it has weight N. But we know from the paragraph just above (2.4.5) that this weight is $(p-1)p^{k-1}$, so we have $N = (p-1)p^{k-1}$.

Binary MLS 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 MLS 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 MLS sequence. (2.4.1)'

Fact 2': In a binary MLS sequence, the maximum run of zeros has length k-1, and this sequence occurs only once per period. (2.4.3)'

Fact 3': In a binary MLS sequence, the maximum run of ones has length k, and this sequence occurs only once per period. (2.4.4)'

Fact 4': In a binary MLS sequence, 1 appears exactly 2^{k-1} times, and 0 appears exactly 2^{k-1} - 1 times. (2.4.5)'

Fact 5': If we line up a binary MLS 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. (2.4.6)'

Examples: Recall the GF(2^4) example above (k = 4) with

c,c = {1,1,1,1,0,1,0,1,1,0,0,1,0,0,0},{1,1,1,1,0,1,0,1,1,0,0,1,0,0,0}

Here we see the single run of k = 4 ones (Fact 3') , and the single run of k-1 = 3 zeros (Fact 2'), The total number of 1's is seen to be 8, which agrees with Fact 4' (2^{4-1}= 8), while the total number of 0's is seen to be 7, which agrees with Fact 4' as well (2^{4-1}- 1 = 7).

In (2.3.19) (k=9) one can locate the single run of k = 9 1's on the 3rd and 4th lines, and the single run of 8 0's which is at the end. We leave it to the reader to count the 1's and 0's.

The product of two shifted binary MLS 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 power density of a sequence.

Consider our two shifted sequences as row vectors one under the other, each containing n = 2^k - 1 components. The third row contains the bitwise sum of the two sequences while the fourth contains the bitwise product. Here is the Example C MLS sequence where Row 2 is shifted 2 bits to the right :

```
1  {1,1,1,1,0,1,0,1,1,0,0,1,0,0,0}          c  vector
2   0,0{1,1,1,1,0,1,0,1,1,0,0,1,0,0,0}      c vector shifted 2
3   1,1,0,0,1,0,0,0,1,1,1,1,0,1,0           bitwise sum
4   0,0,0,0,1,0,0,0,0,1,1,0,0,0,0           bitwise product
```

(2.4.7)

Looking at rows 1 and 2, we count the following cases of each bit alignment possibility:

$$n_1 = \text{number of} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = 3$$

$$n_2 = \text{number of} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 4$$

$$n_3 = \text{number of} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 4$$

$$n_4 = \text{number of} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 4$$

Now, having given this example, we *imagine* redrawing the four rows above as four columns and consider the first column to be some arbitrary MLS sequence, and the second column to be that same sequence shifted vertically by an arbitrary number of bits. The third column contains the sum bits and the fourth column the product bits, just as with the rows above. Call this the "uncompressed chart" which has $2^k - 1$ rows. In the above example, that would be 15 rows. We know that the first two columns of this uncompressed chart 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 columns of the imagined uncompressed chart, there are four possible combinations we can get at each row: (0,0), (0,1), (1,0), and (1,1). As before we let the n_i be the counts of each combination. We then draw the following "compressed chart" which has only four rows, and each row represents all uncompressed chart rows which have the indicated bit-pair (ignore the last two columns for the moment):

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}	(2.4.8)

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

$$\begin{aligned} n_2 + n_4 &= w \ = \text{number of 1's in the second column} \qquad w = 2^{k-1} \\ n_3 + n_4 &= w \ = \text{number of 1's in the first column} \\ n_2 + n_3 &= w \ = \text{number of 1's in the third column} \end{aligned} \tag{2.4.9}$$

We have three equations in three unknowns n_2, n_3 and n_4. Solving these equations is trivial and we find that $n_2 = n_3 = n_4 = w/2 = 2^{k-2}$.

Now the total number of all combinations is equal to the number of rows, so

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

so that

$$n_1 + 3\,(2^{k-2}) = 2^k - 1 \qquad \Rightarrow \qquad n_1 = (2^k - 1) - 3 \cdot 2^{k-2} \quad . \tag{2.4.10}$$

These four counts are then displayed in the second last column "Shifted n's".

Suppose now that there was no relative shift between the first two uncompressed chart columns. In that case, the counts n_2 and n_3 both vanish. Count n_1 is the number of zeros in the sequence and count n_4 is the number of ones, and these numbers were given in Fact 4'. So these four n_i appear in the last column "Unshifted n's". This column would also apply if the shift were an integral multiple of the period 2^k-1 of the sequence. The "compressed chart" is now fully explained.

Staring at the chart, we see that we have proven the following:

Fact 7: Exactly 2^{k-2} ones appear in (one period's worth of) the *product* of two shifted binary MLS 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. (2.4.11)

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

In the next section we shall argue that, for reasonably large k, any MLS sequence is very close to being a so-called **white sequence**. In such a sequence (as we use the term), the probability of a bit being a 1 is p = 1/2 and there is no correlation between different bit positions, each position is independent. Another name for such a sequence is a **random sequence**. For such a random sequence, we can arrive at certain conclusions which we then apply to the MLS sequence. One of these has to do with the probability of the occurrence of strings of repeated symbols.

Obviously the MLS sequence is not perfectly white since it never has a run of more than k ones or more than k-1 zeros (Fact 2' and 3'). 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 operate for a long time and we store all of the output data. Then later on, we scan through the data and count the runs 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 before the next 0 occurs.

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

$$p = 2^{k-1}/(2^k - 1) \qquad q = (2^{k-1} - 1)/(2^k - 1) \qquad p + q = 1 \tag{2.4.12}$$

For k = 9, p = 256/511 and q=255/511. In general, $p \approx q \approx 1/2$. We quietly assume that the probability of each bit having some value is independent of other bits in the sequence. This is true for a white sequence, and approximately true for a long MLS sequence.

Here is a picture outlining the possibilities for runs of 1's 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 run of 1's of length n is:

$$P(n) = q\, p^{n-1} = (q/p)\, p^{n} \quad . \tag{2.4.13}$$

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 runs 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 an MLS sequence. First consider the elementary series sum

$$\sum_{n=1}^{m} p^{n} = \frac{p - p^{m+1}}{1-p} = (p/q)\,(1 - p^{m}) \; . \tag{2.4.14}$$

Now consider the sum of our run-of-1's probabilities, making use of the above result:

$$\text{total probability of all 1-strings} = \sum_{n=1}^{m} P(n) = (q/p) \sum_{n=1}^{m} p^{n} = 1 - p^{m} \; . \tag{2.4.15}$$

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

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

$$\text{total probability of all 1-strings for char sequence} = 1 - p^{k} \tag{2.4.16}$$

Since p ≈ 1/2 , the error is $(1/2)^{k}$. For k = 9, this is 1/512 ≈ 0.2% . Because there are no runs of 1's 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 an MLS sequence to a white sequence.

As for runs 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 run is k-1 zeros, so our total probability sum stops one short of the 1's case. Adding up the total probability, we get:

$$\text{total probability of all 0-runs for char sequence} = 1 - q^{k-1} \qquad (2.4.17)$$

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 an MLS 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 run of exactly n identical bits is given by $P(n) \approx 1/2^{n}$. For k = 9, the error in this estimate is < 0.5%. (2.4.18)

2.5 The MLS Spectral Power Density and Autocorrelation Sequence

Before looking at the MLS spectra, it is useful to review the characteristics of a white power spectrum.

(a) Spectral Power Density of a White Sequence

The spectral power density of an infinite uncorrelated statistical pulse train which has amplitudes taken from the set {A,B} is shown in FT (35.37) to be

$$\mathcal{P}(\omega) = \mathcal{P}_{\mathbf{pulse}}(\omega) \left[\sigma^2 + \mu^2 \sum_{m=-\infty}^{\infty} 2\pi\, \delta(\omega T_1 - 2\pi m) \right] \tag{2.5.1}$$

where

$\mathcal{P}_{\mathbf{pulse}}(\omega)$ = the spectral power density of the underlying pulse train pulse $x_{\mathbf{pulse}}(t)$

$\sigma^2 = [\, p(1-p)] \, (A-B)^2$
$\mu = [p]A + [1-p]B$

Here p is the probability that y_n takes value A, and p-1 is the probability that y_n takes value B.

A white sequence is such a sequence in which p = 1/2, so that

$$\sigma^2 = (1/4)(A-B)^2$$
$$\mu^2 = (1/4)(A+B)^2 \tag{2.5.2}$$

so our general white sequence $\mathcal{P}(\omega)$ is given by

$$\mathcal{P}(\omega)_{\mathbf{white}} = \mathcal{P}_{\mathbf{pulse}}(\omega) \left[\frac{1}{4} (A-B)^2 + \frac{1}{4} (A+B)^2 \sum_{m=-\infty}^{\infty} 2\pi\, \delta(\omega T_1 - 2\pi m) \right] . \tag{2.5.3}$$

If {A,B} = {1,0} we find the following mixed spectrum (partially continuous, partially discrete),

$$\mathcal{P}(\omega)_{\mathbf{white}} = \mathcal{P}_{\mathbf{pulse}}(\omega) \left[\frac{1}{4} + \frac{1}{4} \sum_{m=-\infty}^{\infty} 2\pi\, \delta(\omega T_1 - 2\pi m) \right] \quad // \{1,0\} \tag{2.5.4}$$

and {A,B} = {1,-1} produces instead the following purely continuous spectrum,

$$\mathcal{P}(\omega)_{\mathbf{white}} = \mathcal{P}_{\mathbf{pulse}}(\omega) \ . \quad // \{1,-1\} \tag{2.5.5}$$

If the pulse is a box pulse of amplitude 1 and width T_1, then

$$\mathcal{P}_{\mathbf{pulse}}(\omega) = (1/\omega_1)\, \mathrm{sinc}^2(\omega T_1/2) \qquad \omega_1 = 2\pi/T_1 \qquad \text{FT (36.1)}$$

and one then finds for the {1,0} case that

$$\mathcal{P}(\omega)_{\mathtt{white}} = (1/\omega_1)\,\mathrm{sinc}^2(\pi\frac{\omega}{\omega_1})\,[\,\frac{1}{4} + \frac{1}{4}\sum_{m=-\infty}^{\infty}\delta(\frac{\omega}{\omega_1} - m)\,]$$

$$= (1/\omega_1)\,[\,\frac{1}{4}\,\mathrm{sinc}^2(\pi\frac{\omega}{\omega_1}) + \frac{1}{4}\,\delta(\frac{\omega}{\omega_1})\,] \qquad \text{// \{1,0\}} \qquad (2.5.4a)$$

since the sinc function kills off the lines for $m \neq 0$. For the {1,-1} case

$$\mathcal{P}(\omega)_{\mathtt{white}} = (1/\omega_1)\,\mathrm{sinc}^2(\pi\frac{\omega}{\omega_1}) \quad . \qquad \text{// \{1,-1\}} \qquad (2.5.5a)$$

The first spectrum has a DC line and a continuous sinc^2 spectrum as shown. The second spectrum has no DC line of course and has four times the continuous spectrum of the first since the peak-to-peak swing is double that of the {1,0} case.

(b) The general power formula for an infinite P-repeated sequence

Consider an infinite pulse train with underlying pulse $x_{\mathtt{pulse}}(t)$ whose amplitudes $y_{\mathtt{n}}$ are an MLS sequence of period P,

$$x(t) = \sum_{n=-\infty}^{\infty} y_{\mathtt{n}}\; x_{\mathtt{pulse}}(t - t_{\mathtt{n}}) \quad . \qquad \text{// infinite pulse train}$$

We are interested in determining the spectral power density $\mathcal{P}(\omega)$ of this pulse train.

First, we define the **autocorrelation sequence** $r_{\mathtt{s}}$ of a sequence $y_{\mathtt{n}}$ as the following "horizontal" average across the sequence:

$$r_{\mathtt{s}} \equiv \langle y_{\mathtt{n}}\, y_{\mathtt{n+s}} \rangle_{\mathtt{1}} \quad . \qquad (2.5.6)$$

For an infinite sequence this can be written

$$r_{\mathtt{s}} \equiv \lim_{\mathtt{N}\to\infty} [\frac{1}{(2N+1)} \sum_{n=-N}^{N} y_{\mathtt{n}}\, y_{\mathtt{n+s}}\,] \quad . \qquad \text{FT (F41.a)}$$

If the sequence $y_{\mathtt{n}}$ is periodic with period P, then it is shown in FT that

$$r_{\mathtt{s}} = \frac{1}{P}\sum_{n=0}^{P-1} y_{\mathtt{n}}\, y_{\mathtt{n+s}} \quad . \qquad \text{FT (F41.b)} \qquad (2.5.7)$$

Now, in FT Appendix F it is shown that for *any* pulse train constructed from a repeating subsequence of length P, the following theorem applies:

Theorem : For an infinite statistical sequence made of repeated subsequences of length P : FT (F.54)

If the following is found to be true concerning the autocorrelation sequence elements,

$$\langle y_m y_n \rangle_1 = \alpha \qquad \text{for } m \neq n + NP \qquad N = \text{any integer} \tag{2.5.8}$$
$$\langle y_m y_n \rangle_1 = \beta \qquad \text{for } m = n + NP$$

then the spectral power density is given by :

$$\mathcal{P}(\omega) = \mathcal{P}_{\mathtt{pulse}}(\omega)\ \omega_1 \sum_{m=-\infty}^{\infty} \left[(\beta-\alpha)\ \frac{1}{P}\,\delta(\omega - \omega_1 m/P) + \alpha\,\delta(\omega - \omega_1 m) \right] . \tag{2.5.9}$$

One can break out the DC term in (2.5.9) and write this alternative form (FT (F.52d))

$$\mathcal{P}(\omega) = \mathcal{P}_{\mathtt{pulse}}(\omega)\ \omega_1 \left[(\beta-\alpha)\frac{1}{P} \sum_{m\neq 0} \delta(\omega - \omega_1 m/P) + \alpha \sum_{m\neq 0} \delta(\omega - \omega_1 m) + \{(\beta-\alpha)/P + \alpha\}\ \delta(\omega) \right] . \tag{2.5.9a}$$

where m≠0 means that both positive and negative integers should be included in the sums. Since the sequence is periodic with period P, the spectrum is entirely discrete. The three terms are illustrated by delta function arrows in the following drawing, where the arbitrary red curve is $[\mathcal{P}_{\mathtt{pulse}}(\omega)\ \omega_1]$

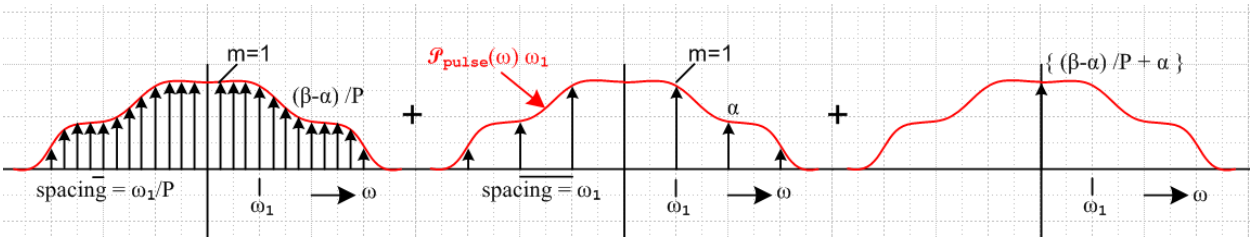


Fig 2.8

Each vertical arrow represents a spectral δ line. The height of the arrow is the value of the red envelope curve times the quantity shown. The angular frequency $\omega_1 = 2\pi/T_1$ is associated with the pulse duration T_1 of the pulse train. The spacing between the arrows in the first term is $\Delta\omega = \omega_1/P$. As P gets large, the density of "infinitely tall" delta arrows increases, while their "amplitude" decreases due to the 1/P factor. In effect, the first term becomes an "effective continuum" for large P, since the energy is spread out over so many closely spaced lines. In the complete limit P→∞, the delta arrows coalesce into a true continuum, due to this fact from FT,

$$\lim_{P\to\infty} \left[\frac{\omega_1}{P} \sum_{m=-\infty}^{\infty} \delta(\omega - \omega_1 m/P) \right] = 1 \qquad \text{FT (F.38)}$$

so the P→∞ limit of the above repeated-subsequence power spectrum becomes

$$\mathcal{P}(\omega) = \mathcal{P}_{\mathbf{pulse}}(\omega)\,[\,(\beta-\alpha) + \alpha\,\omega_1 \sum_{m\neq 0} \delta(\omega - \omega_1 m) + \alpha\,\omega_1\delta(\omega)\,]\,. \qquad P=\infty \qquad (2.5.9b)$$

The first term has now become (β-α) times the red envelope function $\mathcal{P}_{\mathbf{pulse}}(\omega)$. The second term is completely unchanged, while the DC line of the third term now has amplitude α.

We will now show that the above Theorem applies to any MLS sequence of period P.

(c) Case 1: The Spectral Power Density of an MLS Sequence with symbols in {1,0}.

Consider once again our "compressed" chart (2.4.8) which collected facts about the product of a specific MLS sequence and a shifted version of itself,

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}	(2.5.10)

where $P = 2^k - 1$. First of all, the mean value for an MLS sequence is given by (see (2.4.5)')

$$\langle y_n\rangle_1 = \text{Prob}(1)*1 + \text{Prob}(0)*0 = \text{Prob}(1) = [\text{number of 1's}]/P = 2^{k-1}/P = (1/2)(1+1/P)\,.$$

If the shifted version of our MLS sequence is shifted by s bits, then for $s \neq 0$,

$$\langle y_n y_{n+s}\rangle_1 = 1 * 1 * \text{Prob}(1,1) + 1*0*\text{Prob}(1,0) + \text{etc} = \text{Prob}(1,1) = n_4/P = 2^{k-2}/P.$$

Only facing bits of the pattern (1,1) contribute to $\langle y_n y_{n+s}\rangle_1$, and the count of those is $n_4 = 2^{k-2}$ as shown in the chart above (Shifted n's column), while the total number of bit pairs is P. Now

$$2^{k-2}/P = (1/4)2^k/P = (1/4)(P+1)/P = (1/4)(1 + 1/P)$$

so we have just shown that

$$\langle y_n y_{n+s}\rangle_1 = (1/4)(1 + 1/P) \qquad \text{for } s \neq 0\ . \qquad (2.5.11)$$

For s = 0, there is no shift and, since there are 2^{k-1} one's in any MLS sequence, this is also the number of (1,1) matching pairs and $n_4 = 2^{k-1}$ as shown in the last column of the chart, bottom row. Thus

$$\langle y_n y_{n+0}\rangle_1 = 1 * 1 * \text{Prob}(1,1) + \text{etc} = \text{Prob}(1,1) = 2^{k-1}/P = 2\,[2^{k-2}/P]$$
$$= (1/2)(1 + 1/P)\,. \qquad (2.5.12)$$

We have now established that

$$r_s = \langle y_n\, y_{n+s} \rangle_1 = \begin{cases} \alpha & s \neq NP \\ \beta & s = NP \end{cases} \qquad N = \text{ any integer}$$

where (2.5.13)

$$\alpha = (1/4)(1 + 1/P)$$
$$\beta = (1/2)(1 + 1/P) \qquad \Rightarrow \qquad (\beta-\alpha) = \alpha = (1/4)(1 + 1/P) \ .$$

Since this meets the condition (2.5.8), the spectral power density of a pulse train which uses this MLS sequence as its amplitudes is given by (2.5.9a)

$$\mathcal{P}(\omega) = \mathcal{P}_{pulse}(\omega)\, \omega_1 \left[(\beta-\alpha) \frac{1}{P} \sum_{m \neq 0} \delta(\omega - \omega_1 m/P) + \alpha \sum_{m \neq 0} \delta(\omega - \omega_1 m) + \{(\beta-\alpha)/P + \alpha\}\, \delta(\omega) \right]$$

$$= \mathcal{P}_{pulse}(\omega)\, \omega_1 \left[\alpha \frac{1}{P} \sum_{m \neq 0} \delta(\omega - \omega_1 m/P) + \alpha \sum_{m \neq 0} \delta(\omega - \omega_1 m) + \alpha\{1+1/P\}\, \delta(\omega) \right]$$

$$= \mathcal{P}_{pulse}(\omega)\, \omega_1 \left[(\beta/2) \frac{1}{P} \sum_{m \neq 0} \delta(\omega - \omega_1 m/P) + (\beta/2) \sum_{m \neq 0} \delta(\omega - \omega_1 m) + \beta^2\, \delta(\omega) \right] . \qquad (2.5.14)$$

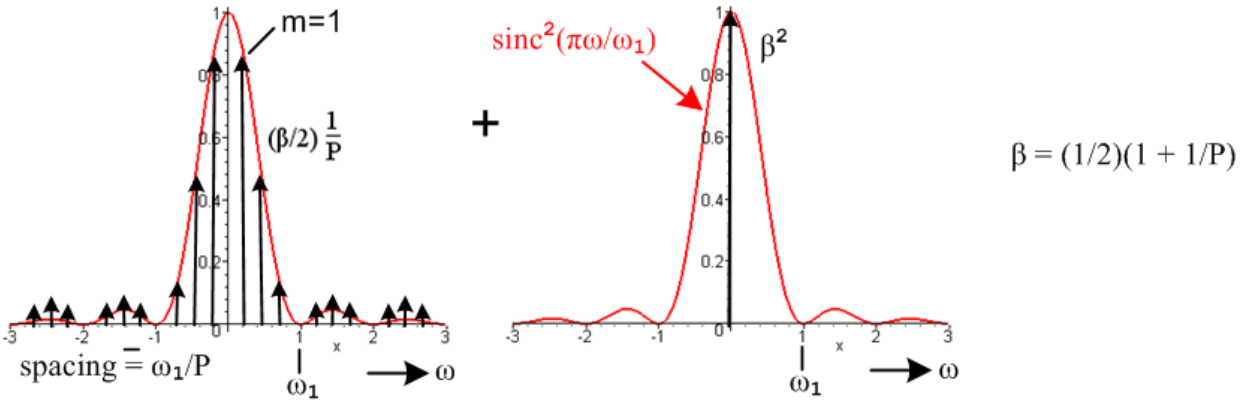
where $\beta = (1/2)(1+1/P)$. In the case of a box-shaped pulse $\mathcal{P}_{pulse}(\omega)\, \omega_1 = \text{sinc}^2(\pi \frac{\omega}{\omega_1})$, the entire second sum is killed off by the zeros of the sinc function leaving this result

$$\mathcal{P}(\omega) = (\beta/2)\, \text{sinc}^2(\pi \frac{\omega}{\omega_1})\, \frac{1}{P} \sum_{m \neq 0} \delta(\omega - \omega_1 m/P) + \beta^2\, \delta(\omega), \qquad \beta = (1/2)(1 + 1/P)$$

$$= (\beta/2)\, \text{sinc}^2(\pi \frac{\omega}{\omega_1})\, \frac{1}{P} \sum_{m \neq 0} \delta(\omega - \omega_1 m/P) + \beta^2\, \delta(\omega),$$

$$= \frac{1}{4}(1+\frac{1}{P})\, \text{sinc}^2(\pi \frac{\omega}{\omega_1})\, \frac{1}{P} \sum_{m \neq 0} \delta(\omega - \omega_1 m/P) + \frac{1}{4}(1+\frac{1}{P})^2\, \delta(\omega) \qquad (2.5.15)$$

Here is a graphical representation of this MLS $\mathcal{P}(\omega)$ drawn for P = 4 :



Spectrum of an MLS sequence with symbols in {1,0} and a box pulse shape. Fig 2.9

This is Fig 2.8 specialized to the box pulse spectrum shape. The little arrows are drawn too long so they can be seen. Using the same fact FT (F.38) quoted above ($\lim_{P\to\infty} [\frac{\omega_1}{P} \sum_{m=-\infty}^{\infty} \delta(\omega - \omega_1 m/P)] = 1$), we find this P→∞ limit of the above MLS power spectrum

$$\mathcal{P}(\omega) = (1/4)\,\mathrm{sinc}^2(\pi \frac{\omega}{\omega_1})\,(1/\omega_1) + (1/4)\,\delta(\omega), \qquad \beta = (1/2) \qquad (2.5.16)$$

which agrees with the white power spectrum (2.5.4a). Thus, the densely packed lines have coalesced into the continuous sinc^2 spectrum shown. For P finite but large, we have an "effective continuum" in the first term of (2.5.15) consisting a dense thicket of delta lines spaced by tiny amount $\Delta\omega = \omega_1/P$.

More generally, for any pulse shape it is easy to verify from our equations above that

$$\lim_{P\to\infty} [\mathcal{P}(\omega)_{\mathtt{MLS}}] = \mathcal{P}(\omega)_{\mathtt{white}}. \qquad // \{1,0\} \qquad (2.5.17)$$

(d) Case 2: The Spectral Power Density of an MLS Sequence with symbols in {1,-1}

Our starting point to carry out this program is to consider once again our "compressed" chart which collected facts about the product of a specific MLS sequence and a shifted version of itself. However, we now draw the chart for our new choice of symbols: (the right two columns are unchanged)

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

(2.5.10)'

First of all, the mean value for an MLS sequence of this type is given by (see (2.4.5)')

$$\langle y_n \rangle_1 = \text{Prob}(1)*1 + \text{Prob}(-1)*(-1) = [\text{number of 1's}]/P - [\text{number of -1's}]/P$$

$$= [2^{k-1}]/P - [(2^{k-1} - 1)]/P = 1/P \ .$$

Our new computation for $\langle y_n y_{n+s} \rangle_1$ goes like this, assuming $s \neq 0$,

$$\langle y_n y_{n+s} \rangle_1 = 1*1*\text{Prob}(1,1) + 1*(-1)*\text{Prob}(1,-1) + (-1)*1*\text{Prob}(-1, 1) + (-1)*(-1)\text{Prob}(-1,-1) \ .$$

For $s \neq 0$, the "shifted n's" column tells us that

$$\text{Prob}(1,1) = \text{Prob}(-1,1) = \text{Prob}(1,-1) = 2^{k-2}/P$$
$$\text{Prob}(-1.-1) = [(2^k - 1) - 3 \bullet 2^{k-2}]/P$$

Then for $s \neq 0$,

$$\langle y_n y_{n+s} \rangle_1 = 2^{k-2}/P - 2^{k-2}/P - 2^{k-2}/P + [(2^k - 1) - 3 \bullet 2^{k-2}]/P$$

$$= - 2^{k-2}/P + [(2^k - 1) - 3 \bullet 2^{k-2}]/P = [(2^k - 1) - 4 \bullet 2^{k-2}]/P$$

$$= [P - (4/4) \bullet 2^k]/P = [P - (P+1)]/P$$

$$= -1/P \ . \qquad (2.5.11)'$$

For $s = 0$ we get instead

$$\langle y_n y_{n+0} \rangle_1 = 1*1*\text{Prob}(1,1) + 1*(-1)*\text{Prob}(1,-1) + (-1)*1*\text{Prob}(-1, 1) + (-1)*(-1)\text{Prob}(-1,-1)$$

$$= \text{Prob}(1,1) + \text{Prob}(-1,-1) = 2^{k-1}/P + (2^{k-1}-1)/P = (2^k-1)/P = P/P = 1 \ . \qquad (2.5.12)'$$

This result is fairly obvious since $\langle y_n^2 \rangle = 1$ for both y_n in $\{1,-1\}$.

We have now established that

$$r_s = \langle y_n \, y_{n+s} \rangle_1 = \begin{cases} \alpha & s \neq NP \\ \beta & s = NP \end{cases} \qquad N = \text{ any integer}$$

where $\qquad (2.5.13)'$

$$\alpha = -1/P$$
$$\beta = 1 \qquad \Rightarrow \qquad (\beta-\alpha) = (1+1/P)$$

Since this meets the condition (2.5.8), the spectral power density of a pulse train which uses this MLS sequence as its amplitudes is given by (2.5.9a)

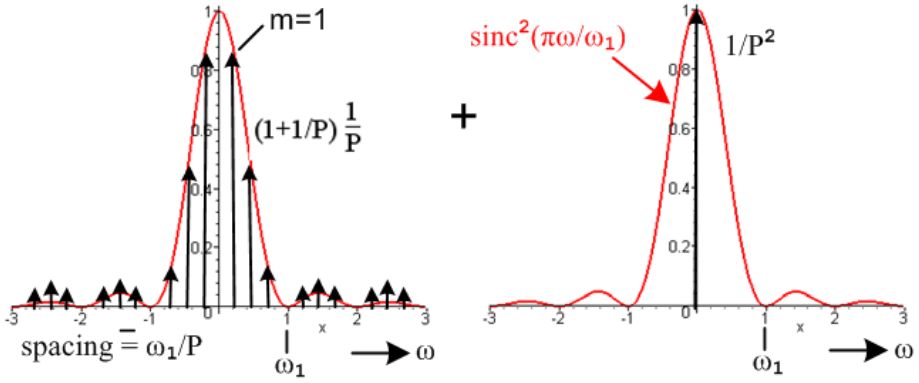
$$\mathcal{P}(\omega) = \mathcal{P}_{\mathtt{pulse}}(\omega)\,\omega_1 \left[(\beta-\alpha)\frac{1}{P} \sum_{m\neq 0} \delta(\omega - \omega_1 m/P) + \alpha \sum_{m\neq 0} \delta(\omega - \omega_1 m) + \{(\beta-\alpha)/P + \alpha\}\,\delta(\omega) \right]$$

$$= \mathcal{P}_{\mathtt{pulse}}(\omega)\,\omega_1 \left[(1+\frac{1}{P})\frac{1}{P} \sum_{m\neq 0} \delta(\omega - \omega_1 m/P) - \frac{1}{P} \sum_{m\neq 0} \delta(\omega - \omega_1 m) + \frac{1}{P^2}\,\delta(\omega) \right] \qquad (2.5.14)'$$

In the case of a box-shaped pulse $\mathcal{P}_{\mathtt{pulse}}(\omega)\,\omega_1 = \mathrm{sinc}^2(\pi \frac{\omega}{\omega_1})$, the entire second sum is again killed off by the zeros of the sinc function leaving this result

$$\mathcal{P}(\omega) = (1+\frac{1}{P})\ \mathrm{sinc}^2(\pi \frac{\omega}{\omega_1})\ \frac{1}{P} \sum_{m\neq 0} \delta(\omega - \omega_1 m/P) + \frac{1}{P^2}\,\delta(\omega) \qquad (2.5.15)'$$

Here is a graphical representation of this MLS $\mathcal{P}(\omega)$ drawn for P = 4 :



Spectrum of an MLS sequence with symbols in {1,-1} and a box pulse shape. Fig 2.10

Using the same FT (F.38) limit ($\lim_{P\to\infty} [\frac{\omega_1}{P} \sum_{m=-\infty}^{\infty} \delta(\omega - \omega_1 m/P)] = 1$), we find for P→∞

$$\mathcal{P}(\omega) = (1/\omega_1)\ \mathrm{sinc}^2(\pi \frac{\omega}{\omega_1}) \qquad (2.5.16)'$$

which agrees with the white power spectrum (2.5.5a). The DC line has gone away and the closely spaced lines on the left have again coalesced into the continuous sinc^2 spectrum shown.

More generally, for any pulse shape it is easy to verify from our equations above that

$$\lim_{P\to\infty} [\mathcal{P}(\omega)_{\mathtt{MLS}}] = \mathcal{P}(\omega)_{\mathtt{white}}. \qquad // \{1,-1\} \qquad (2.5.17)'$$

The two symbol cases can now be compared,

$$\mathcal{P}(\omega) = \frac{1}{4}(1+\frac{1}{P})\ \mathrm{sinc}^2(\pi\frac{\omega}{\omega_1})\ \frac{1}{P}\sum_{m\neq 0}\delta(\omega - \omega_1 m/P)\ + \frac{1}{4}(1+\frac{1}{P})^2\,\delta(\omega) \qquad \{1,0\} \qquad (2.5.15)$$

$$\mathcal{P}(\omega) = (1+\frac{1}{P})\ \mathrm{sinc}^2(\pi\frac{\omega}{\omega_1})\ \frac{1}{P}\sum_{m\neq 0}\delta(\omega - \omega_1 m/P)\ + \frac{1}{P^2}\,\delta(\omega) \qquad \{1,-1\} \qquad (2.5.15)'$$

The first terms describe the AC part of the spectrum, and since the second case has twice the peak-to-peak amplitude of the first case, it has four times the AC spectrum. This is what one would expect just doing an amplitude doubling and DC shift (of 1 unit) to get from Case 1 to Case 2. This same doubling and DC shift changes the mean from (1/2)(1+1/P) to 1/P resulting in the second terms shown.

The main point here is that for large P, an MLS sequence is essentially a white sequence. This is seen as well in the large P limits of the table in the next section.

(e) Statistics Summary, MLS correlation, and graphs of the autocorrelation sequences

We have seen from (2.5.6) and (2.5.7) that the autocorrelation sequence is given by

$$r_s = \frac{1}{P}\sum_{n=0}^{P-1} y_n\, y_{n+s} = \langle y_n\, y_{n+s}\rangle_1\ , \qquad y_{n+NP} = y_n$$

We may gather up our results of the last three sections to construct this table

Case 1 {1,0}	$\langle y_n\rangle = \mu$	$r_0 = \beta = \langle y_n^2\rangle$	$r_{m-n} = \alpha = \langle y_m y_n\rangle$		
uncorrelated	μ	μ	μ^2	$0 \le \mu \le 1$	
MLS	(1/2)(1+1/P)	(1/2)(1+1/P)	(1/4)(1+1/P)		
white	1/2	1/2	1/4		
Case 2 {1,-1}	$\langle y_n\rangle = \mu$	$r_0 = \beta = \langle y_n^2\rangle$	$r_{m-n} = \alpha = \langle y_m y_n\rangle$		
uncorrelated	μ	1	μ^2	$-1 \le \mu \le 1$	
MLS	1/P	1	-1/P		
white	0	1	0		(2.5.18)

Is the MLS sequence correlated or uncorrelated?

For {1,0} here is what we know (see FT Appendix G (c)) :

$$\sigma^2(Y_n) = E(Y_n^2) - E(Y_n)^2 = \langle y_n^2\rangle - \langle y_n\rangle^2 = (1/2)(1+1/P) - (1/2)^2(1+1/P)^2 \qquad \text{FT (G.30)}$$
$$= (1/4)(1-1/P^2)$$

$$\mathrm{cov}(Y_n,Y_m) = E(Y_nY_m) - E(Y_n)E(Y_m) = \langle y_n y_m\rangle - \langle y_n\rangle^2 = (1/4)(1+1/P) - (1/2)^2(1+1/P)^2$$
$$= -(1/4)(1/P)(1+1/P) \qquad \text{FT (G.24b)}$$

Therefore

$$\text{corr}(Y_n,Y_m) = \frac{\text{cov}(Y_n, Y_m)}{\sigma(Y_n)\,\sigma(Y_m)} = \frac{\text{cov}(Y_n, Y_m)}{\sigma^2(Y_n)} = \frac{-(1/4)\,(1/P)\,(1+1/P)}{(1/4)(1-1/P^2)} = \frac{-1/P}{1+1/P} \qquad \{1,0\}$$

Since the result is not zero, the {1,0} MLS sequence is correlated. As $P\to\infty$ it becomes uncorrelated.

For {1,-1} we find instead:

$$\sigma^2(Y_n) = E(Y_n^2) - \quad E(Y_n)^2 = \langle y_n^2\rangle - \langle y_n\rangle^2 \;= 1 \;- (1/P)^2$$

$$\text{cov}(Y_n,Y_m) = E(Y_nY_m) - E(Y_n)E(Y_m) = \langle y_ny_m\rangle - \langle y_n\rangle^2 = -1/P - \;(1/P)^2 = -(1/P)(1+1/P)$$

Therefore

$$\text{corr}(Y_n,Y_m) = \frac{\text{cov}(Y_n, Y_m)}{\sigma(Y_n)\,\sigma(Y_m)} = \frac{\text{cov}(Y_n, Y_m)}{\sigma^2(Y_n)} = \frac{-(1/P)(1+1/P)}{1\;-(1/P)^2} = \frac{-1/P}{1-1/P} \qquad \{1,-1\}$$

Since the result is not zero, the {1,-1} MLS sequence is correlated. As $P\to\infty$ it becomes uncorrelated.

We now plot the six autocorrelation sequences implied by (2.5.18). The sequence values are shown as black dots, and the values are linearly interpolated by a red line as a cosmetic guide. In these drawings it is assumed that P > 4 so we don't see the autocorrelation sequences peaking every P integers, but one should keep in mind that they do just that.

In the process of acquiring and tracking a spread spectrum signal, where one seeks to align the receiving clock with the transmitting clock, one reverts to a continuous (analog) autocorrelation function and then these red lines are more than cosmetic. See Comments at the start of Section 3.5 below.

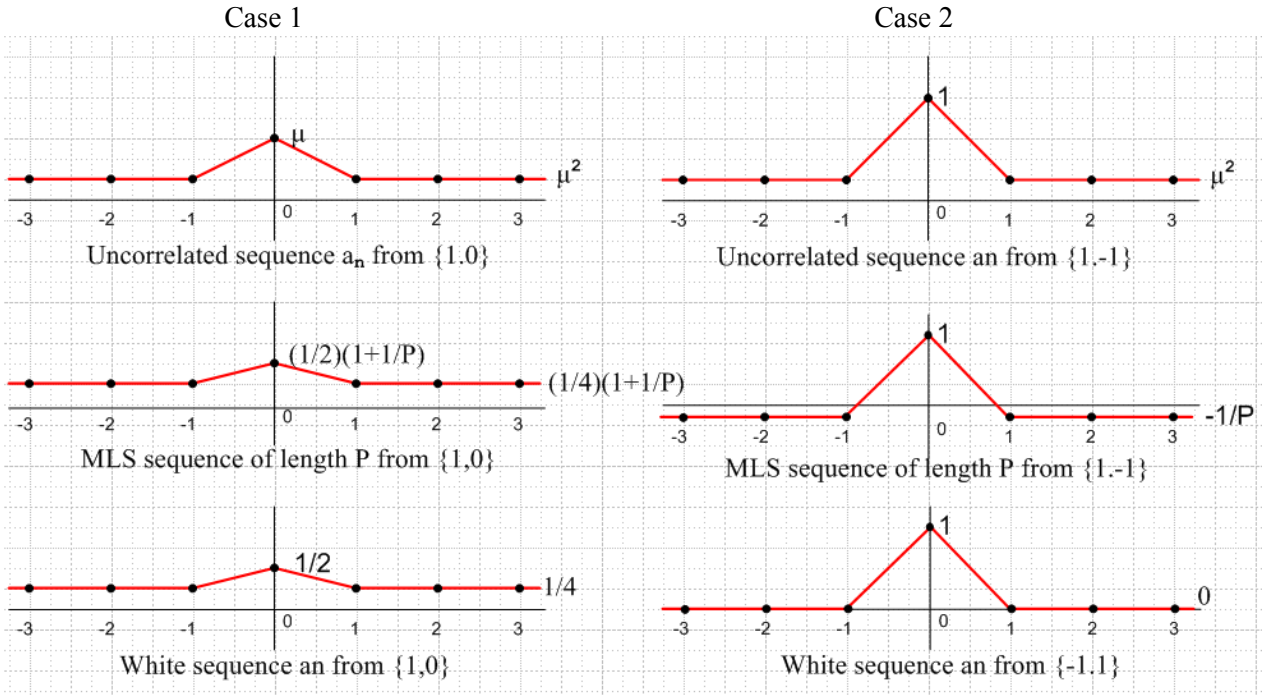


Fig 2.11

Chapter 3: The Matrix Approach to Scramblers

This Chapter will lean heavily on GA Chapter 10. There we show 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 GA Chapter 10. (A "scrambler" will be formally defined in Section 3.5.)

3.1 Review of earlier Chapters

Because this document is so lengthy, we feel it is always worthwhile to review our current status. The four key drawings of multipliers and dividers are shown together on the first page of Chapter 1.

In Chapter 1 we discussed the polynomial divider circuits of Fig 1.1 (Type A) and Fig 1.5 (Type B). 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 Fig 1.5 circuit was in some ways easier to understand since its registers always contain the current remainder, or current dividend. However, the Fig 1.1 circuit is normally used to implement a scrambler.

Polynomial multipliers were also treated in Chapter 1, and again two circuit forms were considered, Fig 1.8 and Fig 1.10. 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 (1.9.8) we presented 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, for the multipliers we get an explicit statement of the output sequence o_n in terms of the input sequence i_n. For the dividers we get instead a family of difference equations which must be solved for o_n in terms of the i_n, a task we shall undertake below.

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 equations which describe the output sequence o_n of a shift register generator, namely,

$$\sum_{j=0}^{k} h_j \, o_{n+j} = 0 \qquad n = \text{any integer} \quad . \qquad \textit{(2.3.2)}$$

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* MLS 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 an MLS sequence. Along the way, various Facts from GA 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 Fig 1.5 divider circuit could be described by a Galois Iterator equation, which we duplicate here:

$$\beta_s = \alpha^s \beta_0 \; + \; [\, \alpha^{s-1} i_0 + \alpha^{s-2} i_1 + ... + \alpha \; i_{s-2} \; + \; i_{s-1} 1 \;] 1 \; . \qquad \textit{(2.2.16)}$$

The symbol β_0 represented the starting state vector of the registers in Fig 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 Fig 1.5, but not to Fig 1.1. This certainly is unsatisfying, since both circuits do the same thing, and since Fig 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 the Type B Divider

The Type B divider was shown in Fig 1.5 which we replicate here,

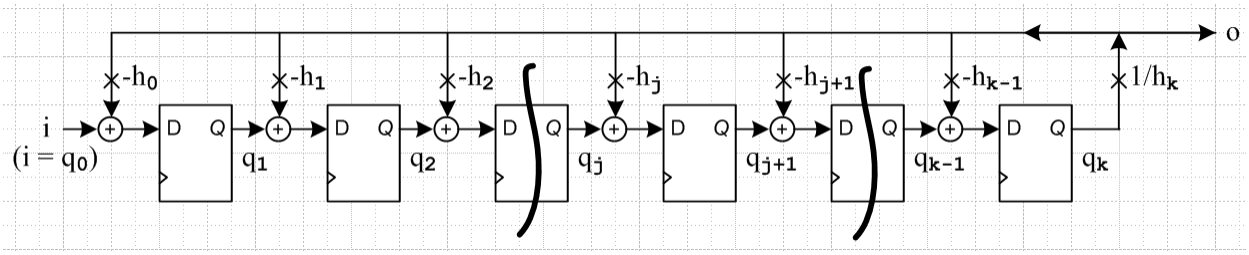


Fig 1.5. Type B polynomial divider.

We assume that h(x) is monic, so that $h_k = 1$. From (1.4.1) we describe the operation of the divider as

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

As earlier, we write f(n) = f, and f(n+1) = f '. Since $h_k = 1$, one has $o = q_k$. And $i = q_0$. Here then are the equations for all k registers,

$$\begin{aligned}
q'_1 &= i - h_0 \; q_k \qquad // \; j = 0 \\
q'_2 &= q_1 - h_1 \; q_k \qquad // \; j = 1 \\
q'_3 &= q_2 - h_2 \; q_k \\
q'_4 &= q_3 - h_3 \; q_k \\
&... \\
q'_{k-1} &= q_{k-2} - h_{k-2} \; q_k \\
q'_k &= q_{k-1} - h_{k-1} \; q_k \quad .
\end{aligned} \qquad (3.2.1)$$

These equations can be trivially re-expressed in matrix form as follows. (we apologize for the vertical bars in place of proper large parentheses)

$$\begin{vmatrix} q'_1 \\ q'_2 \\ q'_3 \\ q'_4 \\ ... \\ q'_{k-1} \\ q'_k \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & ... & 0 & -h_0 \\ 1 & 0 & 0 & ... & 0 & -h_1 \\ 0 & 1 & 0 & ... & 0 & -h_2 \\ 0 & 0 & 1 & ... & 0 & -h_3 \\ & & & ..... & & \\ 0 & 0 & 0 & ... & 0 & -h_{k-2} \\ 0 & 0 & 0 & ... & 1 & -h_{k-1} \end{vmatrix} \begin{vmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ ... \\ q_{k-1} \\ q_k \end{vmatrix} + \begin{vmatrix} i \\ 0 \\ 0 \\ 0 \\ .. \\ 0 \\ 0 \end{vmatrix} \qquad (3.2.2)$$

Now we can write this in compact vector/matrix notation as:

$$\mathbf{q'} = B\,\mathbf{q} + i\,\mathbf{1} \qquad (3.2.3)$$

Here **q** and **q'** are the obvious column vectors, **1** is a unit column vector $(1,0,0...)^{\mathbf{T}}$, and B is the matrix. Now put back the time indices n which we took out above, and get rid of the primes. Put the time index as a subscript, since this notational slot is again available,

$$\mathbf{q}_{n+1} = B\,\mathbf{q}_n + i_n\,\mathbf{1} \quad . \tag{3.2.4}$$

This is a vector difference equation for the state vector $\mathbf{q}_n$ of Fig 1.5. It is easy to solve (3.2.4) by iteration,

$$\mathbf{q}_1 = B\,\mathbf{q}_0 + i_0\,\mathbf{1}$$
$$\mathbf{q}_2 = B\,\mathbf{q}_1 + i_1\,\mathbf{1} = B\,[B\,\mathbf{q}_0 + i_0\,\mathbf{1}] + i_1\,\mathbf{1} = B^2\mathbf{q}_0 + [i_0B + i_1]\,\mathbf{1}$$
$$\mathbf{q}_3 = B\,\mathbf{q}_2 + i_2\,\mathbf{1} = B\,[B^2\mathbf{q}_0 + [i_0B + i_1]\,\mathbf{1}] + i_2\,\mathbf{1} = B^3\mathbf{q}_0 + [i_0B^2 + i_1B + i_2]\,\mathbf{1}$$
.....

and here is the result:

$$\mathbf{q}_n = B^n\,\mathbf{q}_0 + [\,i_0\,B^{n-1} + i_1\,B^{n-2} + ... + i_{n-2}\,B + i_{n-1}\,I\,]\,\mathbf{1} \tag{3.2.5}$$

Observations:

(1) The matrix B has exactly the "companion" form shown in GA (10.27) with $d_i \rightarrow h_i$. According to GA (10.30), this means that h(x) is the "characteristic polynomial" of B. This polynomial is called d(x) in GA, but we are calling it h(x) here.

(2) According to the Cayley-Hamilton Theorem GA (10.12), every square matrix solves its own "characteristic equation" which here means h(B) = 0, since h(x) is the characteristic equation of B. We can write out h(B) = 0 and move $h_kB^k = B^k$ to the left side to get,

$$B^k = -h_o\,I - h_1\,B - h_2\,B^2 - ... - h_{k-1}\,B^{k-1} \quad .$$

(3) Since B is a root of h(x), we know that B is some element of a *matrix representation* of the Galois Field $GF(p^k)$.

Why? The function h(x) which we use to construct $GF(p^k) = R/(\,h(x)\,)$ is of degree k, and is therefore (according to GA Fig 5.5 where k is called m) a minimum polynomial of $GF(p^k)$. All roots of any minimum polynomial of $GF(p^k)$ are elements of $GF(p^k)$ according to GA (5.17). This really follows from the definition GA (5.3) of a "minimum polynomial of $GF(p^k)$ ". In this case the root of h(x) is a matrix, so we have a matrix representing some element of $GF(p^k)$.

(4) It follows from the GA (10.10) that, if h(x) is a *primitive* polynomial of $GF(p^m)$, then the matrix B 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 B.

(5) The above solution (3.2.5) is a concrete realization of the abstract Galois Iterator obtained in Chapter 2, Eq. (2.2.16),

$$\beta_n = \alpha^n \beta_0 + [\, i_0 \alpha^{n-1} + i_1 \alpha^{n-2} + ... + i_{n-2} \alpha + i_{n-1} 1 \,] 1 \qquad (2.2.16)$$

$$\mathbf{q}_n = B^n \mathbf{q}_0 + [\, i_0 B^{n-1} + i_1 B^{n-2} + ... + i_{n-2} B + i_{n-1} I \,] \mathbf{1} \qquad (3.2.5)$$

In (2.2.16) the quantities β_0, β_n , α^k, and 1 are abstract field elements, and the i_k are numbers in GF(p). In Eq. (3.2.5) the field elements β_n, β_0, and the outside 1 are replaced with vectors $\mathbf{q}_n$, $\mathbf{q}_0$, **1**. The quantities α^k and the inside 1 are replaced with matrices B^k and I.

(6) Galois Interpretation. If we back up to the initial iteration step, we may compare these two equations, from which the above two equations were obtained :

$$\beta' = \alpha \beta + i\, 1 \qquad (2.2.13a)$$

$$\mathbf{q}' = B\, \mathbf{q} + i\, \mathbf{1} \qquad (3.2.3)$$

In the first equation, α,β,β',1 are all abstract elements of GF(p^k) while i lies in GF(p) = Z_p. The multiplication implied by α β is done using the abstract multiplication table for GF(p^k), and the subsequent addition of i 1 is done using the addition table for GF(p^k). If we were to create a matrix representation of GF(p^k) as in GA Chapter 10, then α,β,β',1 would all be kxk matrices and the product α β would be done using normal matrix multiplication and the subsequent addition using normal matrix addition with the usual restriction of elements to Z_p. In both the abstract and matrix representations of (2.2.13a), the four field elements α,β,β',1 are on an equal footing.

In contrast, in (3.2.3) B is the matrix representation of a (primitive) element of GF(p^k), while **q**,**q**',**1** are all k-tuple representations of elements of GF(p^k). Recall from GA that in the k-tuple representation, the k-tuple entries are coefficients of a remainder polynomial which represents an element of GF(p^k). So equation (3.2.3) is what (2.2.13a) looks like in this "mixed" representation. In the mixed representation, the product B **q** is the normal multiplication of a matrix times a vector, and the subsequent addition is the normal addition of two vectors, again subject to the Z_p restriction on all numbers. We always know how to convert between a matrix like B and a vector or k-tuple like **q**. For example, recalling (2.2.14),

$$\alpha = \{x\} \qquad \beta = \{q(x)\} \qquad \beta' = \{q'(x)\} \qquad \{i\} = i\{1\} = i\, 1 \ . \qquad (2.2.14)$$

we associate our matrix B with primitive GF(p^k) element α = {x} which corresponds to remainder polynomial x which in turn corresponds to the k-tuple <010...0> which we can interpret as a vector **a** = (0,1,0...). Similarly, the matrix 1 = B^0 corresponds to k-tuple <100...0> which we interpret as vector **1** = (1,0,0...). For powers of B higher than k-1, we have to use the GF(p^k) "enumeration table" as shown in GA which is based on item (2) above.

We have no conclusive statement to make about (3.2.3) other than to claim it is a strange representation of (2.2.13a) which involves a vector representation of the "k-tuple basis" elements **q**,**q**',**1** and a matrix representation of a "powers basis" element B. Recall that the powers of B enumerate all non-zero elements of GF(p^k) and in this powers basis multiplication is easy and addition is difficult, whereas

in the k-tuple basis addition is easy and multiplication is difficult. So in some sense, (3.2.3) involves two different bases and two different representations of field elements of GF(p^k). The good news for us is that we know all about matrix/vector math so this form is very convenient to use.

(7) Equation (3.2.5) is a vector equation of the form $\mathbf{v} = M\mathbf{u} + \mathbf{b}$ where M is a square matrix. The transformation from **u** to **v** is non-linear due to the presence of the term **b**. Nevertheless, if M^{-1} exists, the transformation can be inverted to give $\mathbf{u} = M^{-1}\mathbf{v} - M^{-1}\mathbf{b}$. Since $M = B^n$, $\det(M) = [\det(B)]^n = [(-1)^k h_0]^n$ according to GA (10.28), and since $h_0 \neq 0$, M is indeed invertible. The conclusion is that (3.2.5) is a non-linear but invertible transformation of the state vector from $\mathbf{q}_0$ to $\mathbf{q}_n$.

Before going on the draw more conclusions, we pause to derive a similar matrix equation for the Type A divider of Fig 1.1.

3.3 Matrix Solution of the Type A Divider

The Type A divider was shown in Fig 1.1 which we replicate here,

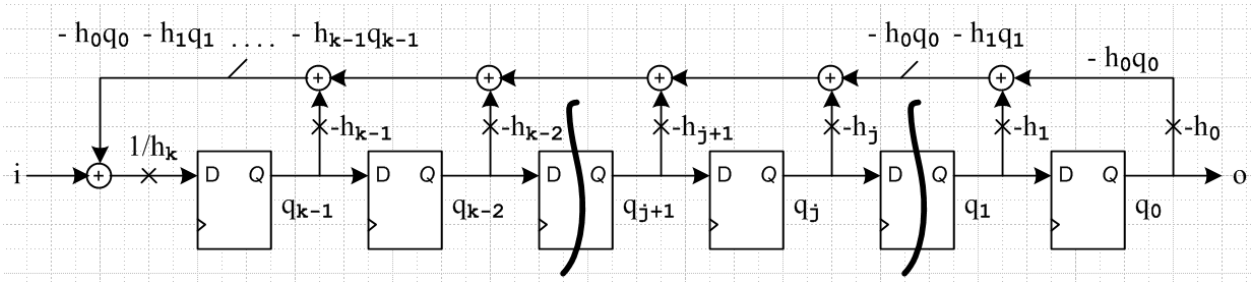


Fig 1.1: Type A polynomial divider.

Eq. (1.2.1) describes the operation of the leftmost register,

$$q_{k-1}(n+1) = i(n) - \sum_{j=0}^{k-1} h_j\, q_j(n) \quad . \qquad (1.2.1)$$

In notation similar to that of the previous section, here are the equations of motion for all the registers,

$$\begin{aligned}
q'_{k-1} &= i - h_0q_0 - h_1q_1 - h_2q_2 - \ldots - h_{k-1}q_{k-1} && \text{// from (1.2.1)}\\
q'_{k-2} &= q_{k-1}\\
q'_{k-3} &= q_{k-2} && \text{// from Fig 1.1 since registers form a simple shift register}\\
q'_{k-4} &= q_{k-3} && (3.3.1)\\
&\ldots\\
q'_1 &= q_2\\
q'_0 &= q_1 \ .
\end{aligned}$$

These equations can be trivially re-expressed in matrix form as follows,

$$\begin{vmatrix} q'_{k-1} \\ q'_{k-2} \\ q'_{k-3} \\ q'_{k-4} \\ \dots \\ q'_{1} \\ q'_{0} \end{vmatrix} = \begin{vmatrix} -h_{k-1} & -h_{k-2} & -h_{k-3} & \dots & -h_{1} & -h_{0} \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ & & & \dots & & \\ 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & 0 \end{vmatrix} \begin{vmatrix} q_{k-1} \\ q_{k-2} \\ q_{k-3} \\ q_{k-4} \\ \dots \\ q_{1} \\ q_{0} \end{vmatrix} + \begin{vmatrix} i \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \\ 0 \end{vmatrix} \tag{3.3.2}$$

As before, we write this in compact vector notation as,

$$\mathbf{q'} = A\,\mathbf{q} + i\,\mathbf{1} \quad . \tag{3.3.3}$$

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:

$$\mathbf{q}_{n+1} = A\,\mathbf{q}_n + i_n\,\mathbf{1} \tag{3.3.4}$$

$$\mathbf{q}_n = A^n\,\mathbf{q}_0 + [\, i_0\, A^{n-1} + i_1\, A^{n-2} + \dots + i_{n-2}\, A + i_{n-1}\, I\,]\,\mathbf{1} \tag{3.3.5}$$

The matrix A above is one of the alternate forms of a companion matrix (form X_1 shown in GA (10.35)). Thus, all observations made about matrix B in the previous section also apply to matrix A here, such as the fact that h(A) = 0.

This is a step forward from Chapter 2. We now see that the Galois Iterator type equation applies to Fig 1.1 as well as Fig 1.5. These iterator equations are (3.2.5) and (3.3.5) :

$$\mathbf{q}_n = B^n\,\mathbf{q}_0 + [\, i_0\, B^{n-1} + i_1\, B^{n-2} + \dots + i_{n-2}\, B + i_{n-1}\, I\,]\,\mathbf{1} \tag{3.2.5}$$

$$\mathbf{q}_n = A^n\,\mathbf{q}_0 + [\, i_0\, A^{n-1} + i_1\, A^{n-2} + \dots + i_{n-2}\, A + i_{n-1}\, I\,]\,\mathbf{1} \quad . \tag{3.3.5}$$

Remember that n on $\mathbf{q}_n$ and i_n is a time index, not a register label. Also, recall that the components of the two vectors are different due to the way we originally named our divider registers :

$\mathbf{q} = (q_1, q_2, q_3, \dots q_k)$ for Type B with matrix B as appears in (3.2.5), see (3.2.2)

$\mathbf{q} = (q_{k-1} \dots q_2, q_1, q_0)$ for Type A with matrix A as appears in (3.3.5), see (3.3.2)

However, in both cases the ordering of the registers is "left to right", see Fig 1.5 and Fig 1.1.

For either Type A or Type B, we can regard $\mathbf{q}_n$ as being the state vector at time n.

3.4 Shift Register Generators Revisited

We have learned that, whether implemented as Fig 1.1 or Fig 1.5, the "equation of motion" of the state vector of a polynomial divider 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 (3.2.5) or (3.3.5) we get,

$$\mathbf{q}_n = C^n \mathbf{q}_0 \qquad n = 0,1,2... \tag{3.4.1}$$

where C can be either matrix A (for Type A) or matrix B (for Type B). In particular we can write

$$C^j \mathbf{q}_n = C^{j+n} \mathbf{q}_0 = \mathbf{q}_{n+j} \quad . \tag{3.4.2}$$

Meanwhile, we know that matrix C is a root of the primitive polynomial h(x), h(C) = 0, so we can write:

$$\sum_{j=0}^{k} h_j C^j = 0 \quad . \tag{3.4.3}$$

Now apply both sides of matrix equation (3.4.3) onto the vector $\mathbf{q}_n$, and make use of (3.4.2) to get

$$\sum_{j=0}^{k} h_j \mathbf{q}_{n+j} = \mathbf{0} \qquad n = 0,1,2... \quad . \tag{3.4.4}$$

Each side of this equation is a k-component vector. We have now proved the following Fact, but first we recall equation (2.3.2),

$$\sum_{j=0}^{k} h_j o_{n+j} = 0 \qquad n = 0,1,2.... \quad . \tag{2.3.2}$$

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. This applies to Type A and Type B generators.

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 (2.3.8) and (2.3.9). The upshot is that any solution of (2.3.2) is an MLS sequence. Therefore we now know that:

Fact 2: The state of each individual register in a primitive polynomial shift register generator cycles through the MLS sequence. This applies to Type A and Type B generators.

Notice how this is an improvement over (2.2.34). For Type A the conclusion is fairly obvious since each register is a time advanced version of the output and the output is MLS, but for Type B it is less obvious.

Corollary 2: The output of a shift register generator cycles through the MLS sequence.

Proof: This is just because the output happens to be one of the registers.

3.5 The Output of a Scrambler

Definition: A **scrambler** is a polynomial divider whose k+1 coefficients h_i correspond to a primitive polynomial of GF(2^k). So a scrambler is just another name for such a divider. The term arises from the fact that an input data sequence i_n is "scrambled" in some sense (we shall see below) by the MLS sequence which would be the output of a shift register generator were there no data input. One might expect that the spectral power density of the input data sequence i_n could differ from that of the scrambler output sequence o_n, perhaps in a beneficial way. We already know that the effect of such a scrambler can be undone by a polynomial multiplier which in this context would be called a **descrambler**. (3.5.1)

Comments: One can regard the scrambling and descrambling processes as a form of signal modulation and demodulation. In some sense the scrambler adds white noise to the signal and the descrambler removes it. This is the basis of pseudo-noise **spread-spectrum** communication (as opposed to frequency-hopping spread spectrum). The MLS sequence is called a "spreading sequence" and it should be clear that the receiver needs to know this exact sequence in order to extract the transmitted signal from the noise in which it is intentionally embedded. Even if the receiver knows the right MLS sequence, the descrambler still has to lock up to the correct phase of the sequence in order to recover the transmitted data bits i_n. This process is called acquisition. Once a signal is acquired, it then has to be tracked and fine-tuned to compensate for effects such as Doppler shift between transmitter and receiver. In general, the "spreading sequence" might not be exactly a formal MLS sequence, but it will be something like it (a PN code) which has a very white spectrum. The longer the period of this sequence the whiter is its spectrum. One could imagine such a sequence being several thousand years in length which certainly suggests there could be some acquisition issues. We have already seen in (2.5.15)' that the differential output MLS sequence with large period P has essentially a continuous spectrum (perhaps a billion closely spaced lines on the left of Fig 2.9), so it is difficult for an "adversary" to know a signal is present, especially if the spread noisy signal is buried in some truly random noise. Conversely, such a spread-spectrum signal is fairly immune to noise at any particular frequency, such as contending non-spread signals or traditional "jamming". Finally, a data signal that has long runs of ones or zeros becomes much more "active" under scrambling, allowing better clock recovery at the receiving end of a transmission.

First, we need one more technical matter dealt with. Recall that C can be either of the two state-vector iterator matrices A or B of the previous section. Take (3.4.3) and multiply both sides by C^n to get,

$$\sum_{j=0}^{k} h_j \, C^{n+j} = 0 \quad . \tag{3.5.2}$$

Recall (2.3.2) which has the scalar solution o_n which for primitive h(x) is either 0 or the MLS sequence,

$$\sum_{j=0}^{k} h_j o_{n+j} = 0 \qquad n = 0,1,2.... \quad . \tag{2.3.2}$$

Equation (3.5.2) says that difference equation (2.3.2) is also obeyed by the following sequence of *matrices*: { I, C, C^2, ...}. If we examine any particular matrix element of this sequence of matrices, call it the r,s element, this sequence of numbers also satisfies (2.3.2), that is,

$$\sum_{j=0}^{k} h_j \, [C^{n+j}]_{rs} = 0 \quad . \tag{3.5.3}$$

Thinking of $[C^{n+j}]_{rs}$ as a scalar quantity like o_{n+j} with extra static labels, we now just proved:

Fact 1: The sequence of numbers $[C^n]_{rs}$ for any fixed r,s, and for n=0,1,2.... either cycles through the MLS sequence, or is identically 0. (3.5.4)

Exercise for the Reader: Is this second option possible for some r,s ? The answer is no, but a proof is not totally obvious. Consider $\mathbf{q}_n = C^n \mathbf{q}_0$ from (3.41). If $\mathbf{q}_0$ has all registers 0 except register s, then $[\mathbf{q}_n]_r = [C^n]_{rs} [\mathbf{q}_0]_s$. If it were possible that all the numbers $[C^n]_{rs}$ vanished for n = 1,2,3... then $[\mathbf{q}_n]_r$ is always zero which means the shift register has a dead register that is always in the 0 state. Show why this is not possible for both Type A and Type B shift register generators, and therefore any $[C^n]_{rs}$ must cycle through the MLS sequence (as in the following example). If possible, relate this to the fact that matrix C is a *primitive* element of GF(2^k) and thus no power of C can produce the all-zeros matrix.

Example: In GA Chap 10 (d) we generated a matrix representation for GF(2^3). Here it is:

$$X, \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad X^3, \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad X^5, \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad X^7, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$X^2, \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad X^4, \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad X^6, \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \text{"0"}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Notice that if we focus on any particular matrix element (for example, row 1, column 1) we see that the bit pattern cycles through 0010111 which is the MLS sequence for GF(2^3) corresponding to the primitive polynomial $h(x) = 1 + x + x^3$. There are no matrix elements which remain 0 for all powers of X.

Now consider the iteration equations (3.2.5) or (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:

$$\mathbf{q_n} = [\, i_0\, C^{n-1} + i_1\, C^{n-2} + ... + i_{n-2}\, C + i_{n-1}\, I\,]\,\mathbf{1} = \sum_{j=0}^{n-1} i_{n-1-j}\, [C^j]\, \mathbf{1} \quad . \qquad (3.5.5)$$

We can express the s^{th} component of this vector equation (corresponding to the state of some register s) as follows (since $(\mathbf{1})_t = \delta_{t,1}$)

$$[\mathbf{q_n}]_s = i_0\, [\, C^{n-1}]_{s,1} + i_1\, [C^{n-2}]_{s,1} + .. + i_{n-2}[C]_{s,1} + i_{n-1}\,[I]_{s,1} = \sum_{j=0}^{n-1} i_{n-1-j}\,[C^j]_{s,1}\,. \qquad (3.5.6)$$

According to Fact 1, the sequence of matrix elements $[C^j]_{s,1}$ in this sum *is* an MLS sequence. For convenience, we shall label this sequence as follows:

$$g_j \equiv [C^{j-1}]_{s,1} \quad . \qquad (3.5.7)$$

Then we can rewrite (3.5.4), reversing the order of the terms,

$$[\mathbf{q_n}]_s = i_{n-1}g_1 + i_{n-2}g_2 + i_{n-3}g_3 + .. i_1\, g_{n-1} + i_0\, g_n = \sum_{j=1}^{n} g_j\, i_{n-j}\,. \qquad (3.5.8)$$

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 an MLS sequence multiplied by the (time reversed) input sequence. (3.5.9)

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 Fig 1.1 or Fig 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 we 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

$$P_n = (1/2)\,[\,1 - (1-2\varepsilon)^n\,] = (1/2)\,(\,1 - \alpha^n) \qquad \alpha \equiv (1 - 2\varepsilon)\ . \qquad (3.5.10)$$

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

$$P_{n+1} = P_n(1- \varepsilon) + (1-P_n)\,\varepsilon = P_n\,(1 - 2\varepsilon) + \varepsilon = \alpha\, P_n + \varepsilon$$

where we define $\alpha = (1 - 2\varepsilon)$. The first term $P_n(1- \varepsilon)$ means the sum of n terms was 1, and to keep the sum one the next term has to be a zero, which means $(1-\varepsilon)$. If the sum of n terms was not a 1, probability $(1-P_n)$, then to make the sum be a 1 the next term has to be a 1, which means ε.

Iterate this difference equation starting with $P_1 = \varepsilon$ to get

$P_2 = \alpha P_1 + \varepsilon = \varepsilon(1+\alpha)$

$P_3 = \alpha P_2 + \varepsilon = \alpha[\varepsilon(1+\alpha)] + \varepsilon = \varepsilon(1 + \alpha + \alpha^2)$

.....

$\varepsilon = (1-\alpha)/2$

$P_n = \varepsilon (1 + \alpha + \alpha^2 + ... + \alpha^{n-1}) = \varepsilon (1 - \alpha^n)/(1-\alpha) = (1/2)(1 - \alpha^n)$. $\alpha \equiv (1 - 2\varepsilon)$ **QED**

Now consider the state of any scrambler register as being the (reversed) input sequence dotted with the MLS sequence, as shown in (3.5.8) 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 below (2.5.10) where $\langle y_n \rangle_1 = (1/2)(1+1/P)$ for Case 1 {0,1}] that the probability of an MLS sequence bit being 1 is $p' = \beta = (1/2)(1 + 1/P)$, where $P = 2^k - 1$ = the period of the MLS 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 $g_3 \; i_{n-3}$ is 1 is

$\varepsilon = p'p$ $\quad p' = (1/2)(1 + 1/P) \approx 1/2$, p = input sequence prob of 1, $P = 2^k - 1$. (3.5.11)

We can now use the above Lemma to obtain the probability that the n term sum shown in (3.5.8) is a 1:

{ Prob that $[q_n]_s = 1$ } $= (1/2)[1 - \alpha^n]$, where $\alpha = 1-2pp' \approx 1-p$ for long MLS sequences (3.5.12)

As the scrambler continues to run and n increases, we see that { Prob that $[q_n]_s = 1$ } *differs from 1/2* by the amount $(1/2)\alpha^n \approx (1/2)(1-p)^n$. If we assume that p > 0.1, say, for the input stream, then at worst case we have $(1/2)\alpha^n \leq (1/2)(0.9)^n$. After n = 1000 clocks, $(1/2)(0.9)^{1000} < 10^{-46}$. The point is that after a scrambler runs for some number of clocks with a random input stream, for any register in the scrambler we are going to have { Prob that $[q_n]_s = 1$ } $= 1/2$.

Our Lemma assumes that the terms being added ($r_i = g_i i_{n-j}$) are statistically independent. This is a reasonable assumption for a reasonable input stream, but one could of course concoct a "pathological" input stream which would disable this assumption, such as i_n being exactly the reverse MLS sequence. With these caveats, we claim to have proven the following fact :

Fact 3: For non-pathological input sequences, after some reasonable number of clocks, the probability of any scrambler register being a 1, { Prob that $[q_n]_s = 1$ }, is 1/2. (3.5.13)

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. (3.5.14)

Comment: If p = 0, then the input stream is all 0's, Since we started with $\mathbf{q_0} = 0$ in the ancient past, we expect { Prob that $[q_n]_s = 1$ } = 0 at any time n, which agrees with the above when $\alpha = 1\text{-}p = 1$.

Review of Leeper 1973 (Ref. LE)

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 a white sequence.

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 scrambler output sequence can be made arbitrarily close to that shown in our table (2.5.18) for "white" and also shown in the lower left graph of Fig 2.11. 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 large enough number of registers k, meaning a large enough P. 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. 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.

Conjecture: A scrambler with k = 9 stages with signals which come from a "live" video source will easily meet this technical qualification. See Fig 1.4.

3.6 The Spectral Power 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" implies in our current context.

We have already derived the spectral power density of such a signal with symbols in {1,-1}

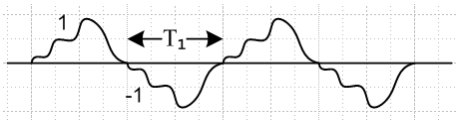
$$\mathcal{P}(\omega)_{\mathtt{white}} = (1/\omega_1)\,\mathrm{sinc}^2(\pi\frac{\omega}{\omega_1}) \quad . \qquad // \{1,-1\} \qquad (2.5.5a)$$

Here, T_1 is the width of a pulse and $\omega_1 = 2\pi/T_1$.

Interpretation of the Scrambler Output Spectrum

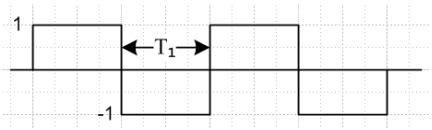
We start here considering the spectrum of a simple periodic signal, then see how this is spectrally related to the scrambler output signal.

The spectral power density for an infinite pulse train of arbitrary pulse shape and alternating amplitudes 1 and -1 was stated in FT (34.21) as



$$\mathcal{P}(\omega) = \mathcal{P}_{\mathtt{pulse}}(\omega) \sum_{m\,=\,\pm\mathrm{odd}} \delta(\frac{\omega}{\omega_1} - \frac{m}{2}) \quad . \qquad \textit{FT (34.21)} \qquad (3.6.1)$$

For a box pulse with $\mathcal{P}_{\mathtt{pulse}}(\omega) = (1/\omega_1)\,\mathrm{sinc}^2(\omega T_1/2)$ this becomes



square wave period = $2T_1$

$$\omega_1\mathcal{P}(\omega) = \mathrm{sinc}^2(\pi\frac{\omega}{\omega_1}) \sum_{m\,=\,\pm\mathrm{odd}} \delta(\frac{\omega}{\omega_1} - \frac{m}{2}) = \mathrm{sinc}^2(\pi x) \sum_{m\,=\,\pm\mathrm{odd}} \delta(x - \frac{m}{2}) \qquad x \equiv \frac{\omega}{\omega_1}$$

$$= \sum_{m\,=\,\pm\mathrm{odd}} (\pi x)^{-2}\ \delta(x - \frac{m}{2}) = \sum_{m\,=\,\pm\mathrm{odd}} (\frac{2}{\pi m})^2\ \delta(x - \frac{m}{2}) \quad . \qquad \textit{FT (34.23)} \qquad (3.6.2)$$

The discrete spectral power lines can be represented as vertical arrows in this plot, where we only display the positive frequency terms in the sum,

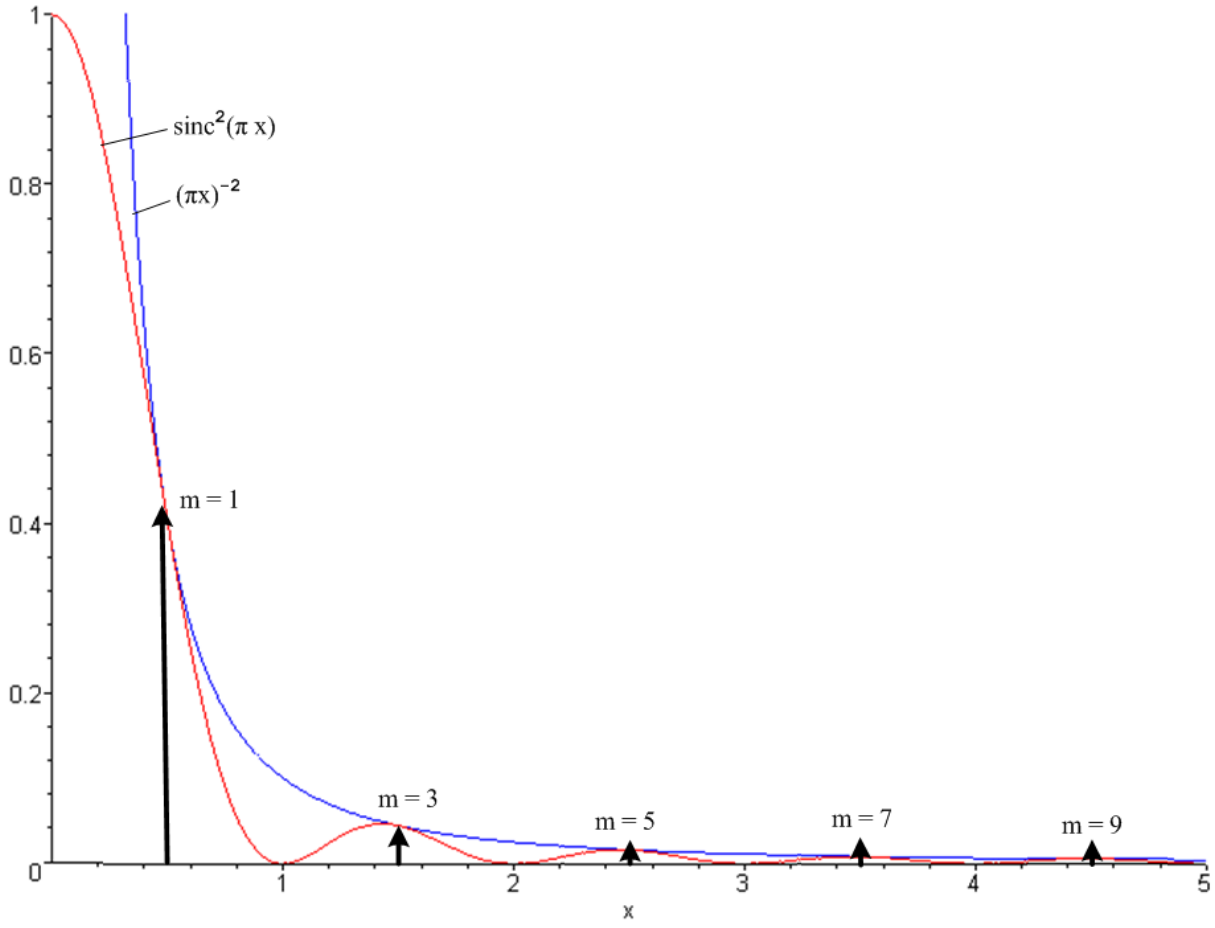


Fig 3.1

A square wave of period $T = 2T_1$ (as shown above) has a fundamental frequency $\omega = \omega_1/2$ or $f = f_1/2$ or $x = 1/2$, where $f_1 \equiv 1/T_1$ and $\omega_1 = 2\pi/T_1$. This is the m = 1 arrow in the graph. The higher arrows are the harmonics. The total power in the square wave is the sum of the arrows. Most of the power is in the fundamental line. First,

$$\Sigma_{m=1,3,5\ldots}^{\infty} (1/m^2) = \Sigma_{n=1,2,3}^{\infty} 1/(2n-1)^2 = \pi^2/8 \ . \qquad \text{// Gradshteyn Ryzhik 0.234 (2)}$$

So the fraction of power in the first few lines is

```
for m from 1 to 7 by 2 do
  line := (1/m^2)/(Pi^2/8):
  print(m,evalf(line));
od:
```

1, .8105694688
3, .09006327431
5, .03242277875
7, .01654223405

So the first line m = 1 has 81% of the power, the m = 3 line 9% and the m = 5 line 3% .

Our scrambler output signal is completely different, though the pulse width T_1 is the same as for the square wave studied above. For a scrambler with some reasonable large k (and thus very large P), we may regard the spectral power density as "essentially continuous" as discussed above, so

$$\omega_1 \mathcal{P}(\omega) = \text{sinc}^2(\pi \frac{\omega}{\omega_1}) = \text{sinc}^2(\pi x) \quad . \qquad x = \frac{\omega}{\omega_1} \qquad // \{1,-1\} \qquad (2.5.5a)$$

The total power ΔP in some frequency range is given by (x = ω/ω1 so dω = ω1dx)

$$\Delta P = \int_{\omega_a}^{\omega_b} \mathcal{P}(\omega) d\omega = \int_{x_a}^{x_b} \mathcal{P}(\omega)\omega_1 dx = \int_{x_a}^{x_b} \text{sinc}^2(\pi x) dx$$

which is the area under the red curve shown above. Regarding the first "hump" as the region from x = 0 to x = 1, and after that the humps really are humps, Maple tells us that

```
tot :=  int(sinc(Pi*x)^2,x = 0..infinity);
```

$$tot := \frac{1}{2}$$

```
for h from 0 to 4 do
  hump := int(sinc(Pi*x)^2,x = h..h+1)/tot:
  print(h,evalf(hump));
od:
```

0, .9028233334
1, .04711600652
2, .01647109466
3, .008338015422
4, .005027892454

Thus, 90% of the power lies to the left of x = 1 (ω = ω1), and the following two humps have 4.7% and 1.6%. The total power to the left of x = 1/2 (ω = ω1/2) is 77% ,

```
int(sinc(Pi*x)^2,x = 0..1/2)/tot: evalf(%);
```

.7736950098

so one would throw out 13% of the signal power running this signal through a brick wall filter with cutoff just above ω1/2.

3.7 The NRZI Mini-Scrambler

Typically, a scrambler circuit is followed by an output stage consisting of this Type A divider:

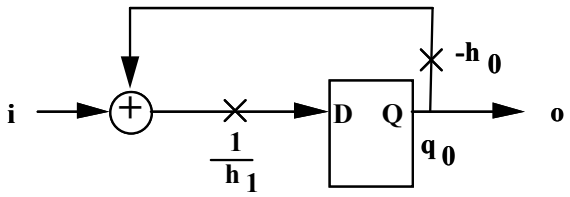


Fig 3.2: The NRZI mini-scrambler.

This is a k = 1 scrambler (see Fig 1.1), 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 A 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".

A = 1 h(A) = 0 = the characteristic polynomial = A + 1 .

The state vector for this scrambler is the state of its one register. Using this fact, and A = 1, we can write down equation (3.3.5),

$$\mathbf{q_n} = A^n \mathbf{q_0} + [\ i_0\ A^{n-1} + i_1\ A^{n-2} + ... + i_{n-2}\ A + i_{n-1}\ I\ ]\ \mathbf{1} \qquad (3.3.5)$$

as follows

$$o_n = o_0 + [\ i_0 + i_1 + i_2 + ... + i_{n-1}\ ] . \qquad (3.7.1)$$

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 :

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. (3.7.2)

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 one inputs an NRZ linecode. (3.7.3)

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, that is, 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 fact 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 one has a differential NRZI signal carried on a twisted pair, it does not matter which way the wires are hooked up 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.

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.

$$o_n = o_0 + [\, i_0 + i_1 + i_2 + ... + i_{n-1}] \,. \qquad (3.7.1)$$

This should be compared with Equation (3.5.8) which shows what a "real" scrambler does to its input stream.

$$[q_n\,]_s = i_{n-1}g_1 + i_{n-2}g_2 + i_{n-3}g_3 + .. i_1\, g_{n-1} + i_0\, g_n = \sum_{j=1}^{n} g_j\, i_{n-j} \,. \qquad (3.5.8)$$

The stream is multiplied term by term with the (reversed) MLS sequence of the scrambler. For reasonable k, this MLS sequence $\{g_i\}$ is "pseudorandom" and "induces" onto the output stream its random nature.

For the mini-scrambler, the length of the MLS sequence is $2^1 - 1 = 1$, so the MLS sequence is simply { 1,1,1,1,1 ...}. One can imagine in Eq. (3.7.1) just above that the input sequence is multiplied by this light-weight MLS sequence. Obviously, the sequence { 1,1,1,1,1 ...} does not carry a lot of whitener.

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 (3.5.10) with $\varepsilon = 1/2$ for the input sequence, so $\alpha = 1-2(1/2) = 0$ and

output stream probability of 1 $= P_n = (1/2)\,[\, 1 - 0^n\,] = 1/2$

In other words, white-in gives white-out. If the input stream to the mini-scrambler is slightly off white, perhaps $p = \varepsilon = 1/2 + \delta$ say, then we get $\alpha = 1 - 2(1/2+\delta) = -2\delta$ so

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 (in n) around the $P_n = 1/2$, always getting closer to 1/2 as time goes by (as n increases). 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 -- the "higher order statistics" -- that one cannot see just looking at the first order statistical quantity P_n. It is the job of the main scrambler to generate this independence as Leeper discusses.

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 power spectral density of the output is given by (we have taken $P \to \infty$ for the scrambler),

$$\mathcal{P}(\omega) \approx \mathcal{P}_{pulse}(\omega)\,[\,\frac{1}{4} + \frac{1}{4}\sum_{m=-\infty}^{\infty} 2\pi\,\delta(\omega T_1 - 2\pi m)\,] \qquad // \{1,0\} \qquad (2.5.4)$$

$$\mathcal{P}(\omega) \approx \mathcal{P}_{pulse}(\omega)\ . \qquad // \{1,-1\} \qquad (2.5.5)$$

The units of $\mathcal{P}(\omega)$ are joules, so $\mathcal{P}(\omega)d\omega$ is then watts.

(b) If the main scrambler input is set to all zeros -- a highly non-random data stream -- the above expressions are still correct since in this case the scrambler output is its MLS sequence.

Exercise for the Reader: What is the effect of an input of all ones?

(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 $x_{pulse}(t)$ is selected to be an ideal box of width T_1 and height 1, the above densities become

$$\mathcal{P}(\omega)\,\omega_1 = \frac{1}{4}\,\mathrm{sinc}^2(\pi\frac{\omega}{\omega_1}) + \frac{1}{4}\,\delta(\frac{\omega}{\omega_1}) \qquad // \{1,0\} \qquad (2.5.4a)$$

$$\mathcal{P}(\omega)\,\omega_1 = \mathrm{sinc}^2(\pi\frac{\omega}{\omega_1}) \quad . \qquad // \{1,-1\} \qquad (2.5.5a)$$

(e) Other pulse shapes can be evaluated using the first expression pair above.

(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 ("spread spectrum") and there are no strong lines, crosstalk between cables and circuits carrying scrambled signals is minimized. RFI emissions are also held in check. The presence of a signal is not betrayed by strong lines to an "adversary" nor is traditional jamming effective. In terms of the notion of "essentially continuous", these statements are still true. In practice for finite P there are thousands of lines closely spaced (as suggested by Fig 2.8) and the power is distributed into these lines so that no individual line has any significant amount of power to do something bad, like rise above an RFI emission specification.

(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 MLS sequence, as noted above.

(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 $P(n) = 2^{-n}$ (see above (2.4.14)). When the source input stream is all zeros, the maximum length of a zero output string is k-1 as in (2.4.3). 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 solution of the Type A Multiplier

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 now analyze the multiplier circuits in matrix form. One may regard this and the following sections as "notional warm-ups" for Section 3.10.

For the Type A multiplier circuit shown in Fig 1.8 (replicated below), 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.

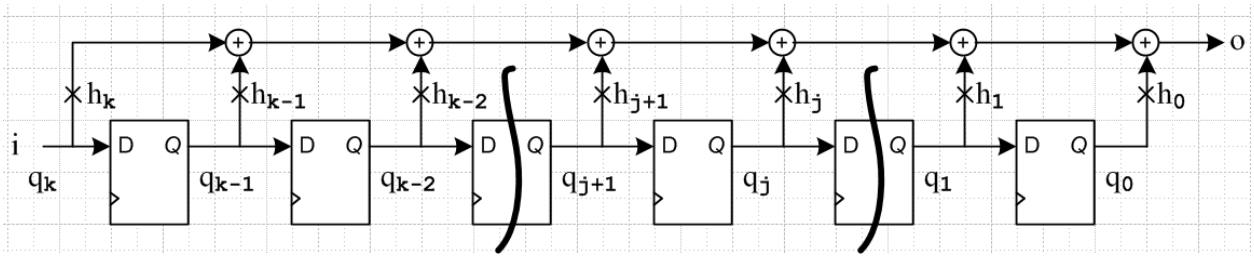


Fig 1.8: Type A polynomial multiplier.

The data simply 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,

$$\begin{vmatrix} q'_{k-1} \\ q'_{k-2} \\ q'_{k-3} \\ q'_{k-4} \\ \dots \\ q'_1 \\ q'_0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ & & & \dots & & \\ 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & 0 \end{vmatrix} \begin{vmatrix} q_{k-1} \\ q_{k-2} \\ q_{k-3} \\ q_{k-4} \\ \dots \\ q_1 \\ q_0 \end{vmatrix} + \begin{vmatrix} i \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \\ 0 \end{vmatrix} \qquad (3.8.1)$$

The matrix here has the same form as in (3.3.2) for the scrambler divider circuit of Fig 1.1, except here all the feedback entries in the first row are zero. As before, we use matrix/vector notation to rewrite the above as $\mathbf{q'} = D\mathbf{q} + i\mathbf{1}$ where we arbitrarily use the letter D for the matrix appearing above. We then re-install the time index n to get the vector difference equation and its solution (see (3.2.4) and (3.2.5)),

$$\mathbf{q}_{n+1} = D\,\mathbf{q}_n + i_n\mathbf{1} \qquad (3.8.2)$$

$$\mathbf{q}_n = D^n\,\mathbf{q}_0 \quad + [\, i_0\, D^{n-1} \;+\; i_1\, D^{n-2} + \dots + i_{n-2}\, D \;+\; i_{n-1} I \,]\,\mathbf{1} \quad . \qquad (3.8.3)$$

The column vector $\mathbf{q_n}$ has the following components, where we revert to a former notation of putting the time index n as an argument,

$$[\,\mathbf{q_n}\,]_s = q_{k-s}(n) \quad . \tag{3.8.4}$$

For example, with s=1 the first component of the column vector $\mathbf{q_n}$ refers to the contents of register q_{k-1} of Fig 1.8 at time n.

The matrix D has the following matrix elements, as can be seen from (3.8.1),

$$D_{ij} = \delta_{i,j+1}\,, \qquad [D^2]_{ij} = \delta_{i,j+2} \qquad , \ldots \qquad [D^n]_{ij} = \delta_{i,j+n} \tag{3.8.5}$$

For example,

$$[D^2]_{ij} = \Sigma_k\, D_{ik}\, D_{kj} = \Sigma_k\, \delta_{i,k+1}\, \delta_{k,j+1} = \delta_{i,j+2}\ .$$

The matrix D^2 has the ones slid down to the next lower subdiagonal, and so on for higher powers. Finally, the matrix D^{k-1} has a single non-vanishing element, a 1 in the lower left corner. For all higher powers $D^n = 0$ 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,

$$\mathbf{q_n} = D^n\,\mathbf{q_0} + [\; i_{n-k}\, D^{k-1} + i_{n-3}\, D^2 + i_{n-2}\, D + i_{n-1} I \;]\,\mathbf{1} \qquad I = D^0$$

$$= D^n\,\mathbf{q_0} + \sum_{j=1}^{k} i_{n-j}\,[D^{j-1}]\;\mathbf{1}\ . \tag{3.8.6}$$

Interpretation: If $n \geq k$, the current state vector of the multiplier is affected only by the most recent k input symbols and is independent of the starting vector $\mathbf{q_0}$. If $n < k$, the first term in (3.8.6) has not yet vanished, so some components of the starting state vector $\mathbf{q_0}$ still have an effect on $\mathbf{q_n}$.

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.

Comment: Unlike the matrices A and B associated with the polynomial dividers, this matrix D associated with the polynomial multipliers is not-invertible since clearly det(D) = 0 (see GA (10.28)). Thus, the transformation from $\mathbf{q_0}$ to $\mathbf{q_n}$ shown in (3.8.6) is not invertible since $\det(D^n) = 0^n = 0$.

Time-domain output sequence for a Type A multiplier

Consider the s^{th} component of the vector equation (3.8.6), where we use $[\,\mathbf{1}\,]_i = \delta_{i,1}$,

$$[q_n]_s = [D^n\,\mathbf{q}_0]_s + \sum_{j=1}^{k} i_{n-j} \sum_{i=1}^{k} [D^{j-1}]_{s,i}\,[\,\mathbf{1}\,]_i$$

or

$$[q_n]_s = \sum_{j=1}^{k} [\,D^n\,]_{s,j}\,[\mathbf{q}_0]_j \;+\; \sum_{j=1}^{k} i_{n-j}\,[D^{j-1}]_{s,1}\;. \qquad (3.8.7)$$

According to (3.8.5), $[\,D^n\,]_{s,j} = \delta_{s,n+j}$ and $[D^{j-1}]_{s,1} = \delta_{s,j}$ so both j sums go away giving

$$[\mathbf{q}_n]_s = [\mathbf{q}_0]_{s-n} + i_{n-s} = [\mathbf{q}_0]_{s-n}\;\theta(n<s) + i_{n-s}\,\theta(n\geq s) \quad . \qquad (3.8.8)$$

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 ≥ 1 ⇒ n<s for the first term. And the input sequence is assumed to begin with i_0 , hence n-s ≥ 0 in the second term.

Now use (3.8.4) to rewrite (3.8.8) as

$$[\mathbf{q}_n]_s = q_{k-s}(n) = q_{k-s+n}(0)\,\theta(n<s) + i_{n-s}\,\theta(n\geq s) \quad . \qquad (3.8.9)$$

This is an elaborate statement of a trivial fact, see Fig 1.8 (replicated below). Consider register q_{k-s} (think of s = k so this is register q_0). 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 places to its left. If n > s, register q_{k-s} contains i_{n-s}.

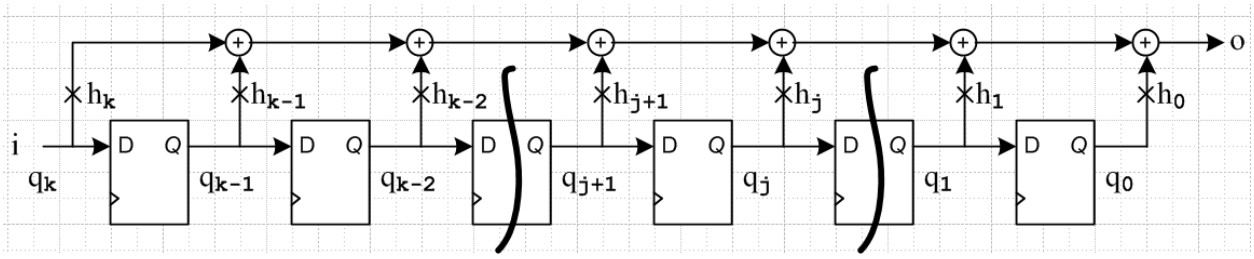


Fig 1.8: Type A polynomial multiplier.

Fig 1.8 shows that the output of the polynomial multiplier can be written in terms of the components of vector $\mathbf{q}_n$:

$$o_n = \sum_{s=0}^{k-1} h_s\; q_s(n) + h_k\, i_n = \sum_{s=0}^{k-1} h_s\; [\mathbf{q}_n]_{k-s} + h_k\, i_n\,. \qquad (3.8.10)$$

We now rewrite (3.8.9) taking s → k-s (which implies k-s → s),

$$[\mathbf{q_n}]_{\mathbf{k-s}} = q_{\mathbf{s}}(n) = q_{\mathbf{s+n}}(0)\,\theta(n<k-s) + i_{\mathbf{n-k+s}}\,\theta(n\geq k-s)$$

$$= q_{\mathbf{s+n}}(0)\,\theta(s\leq k-n-1) + i_{\mathbf{n-k+s}}\,\theta(s\geq k-n) \quad , \qquad (3.8.11)$$

and use this expression for $[\mathbf{q_n}]_{\mathbf{k-s}}$ in (3.8.10) to get

$$o_{\mathbf{n}} = \sum_{s=0}^{k-1} h_{\mathbf{s}}\,[\,q_{\mathbf{s+n}}(0)\,\theta(s\leq k-n-1) + i_{\mathbf{n-k+s}}\,\theta(s\geq k-n)\,] + h_{\mathbf{k}}\,i_{\mathbf{n}} \qquad // \; h_{\mathbf{k}} = 1$$

or

$$o_{\mathbf{n}} = \theta(n<k)\sum_{s=0}^{k-n-1} h_{\mathbf{s}}\,q_{\mathbf{n+s}}(0) + \theta(n\geq 0)\sum_{s=\max(0,k-n)}^{k} h_{\mathbf{s}}\,i_{\mathbf{n+s-k}} \qquad // \text{ Type A multiplier} \qquad (3.8.12)$$

where we add θ functions showing the range of n for which each term is active. For n = 1,2..k-1 both terms contribute, but for n ≥ k only the second term contributes and the initial register values play no role whatsoever. In this case that n ≥ k, max(0,k-n) = 0, so the second sum starts at s = 0.

As a test of this result, lets examine the first two terms in this output sequence,

$$o_0 = \sum_{s=0}^{k-1} h_{\mathbf{s}}\,q_{\mathbf{s}}(0) + h_{\mathbf{k}}i_0 \qquad // \text{ agrees with Fig 1.8}$$

$$o_1 = \sum_{s=0}^{k-2} h_{\mathbf{s}}\,q_{\mathbf{s+1}}(0) + h_{\mathbf{k-1}}i_0 + h_{\mathbf{k}}\,i_1 \; . \qquad // \text{ agrees with Fig 1.8}$$

For n ≥ k, as just noted only the second term in (3.8.12) contributes, so

$$o_{\mathbf{n}} = \sum_{s=0}^{k} h_{\mathbf{s}}\,i_{\mathbf{n+s-k}} \qquad n\geq k\,. \qquad (3.8.13)$$

Setting n → n+k we get

$$o_{\mathbf{n+k}} = \sum_{s=0}^{k} h_{\mathbf{s}}\,i_{\mathbf{n+s}} \qquad n\geq 0\;. \qquad (3.8.14)$$

This is in agreement with the multiplier time domain result shown in box (1.9.8), and this concludes our first notational exercise.

3.9 Matrix solution of the Type B Multiplier

Looking at Fig 1.10,

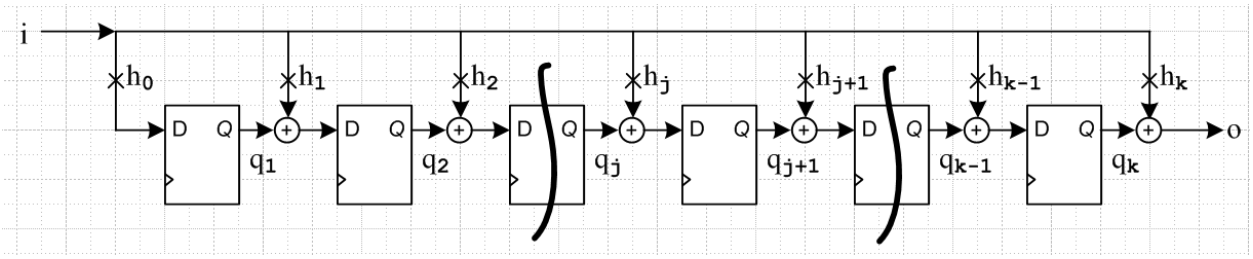


Fig 1.10: Type B polynomial multiplier.

one can write down the following series of equations,

$$
\begin{aligned}
q'_1 &= h_0\, i \\
q'_2 &= q_1 + h_1\, i \\
q'_3 &= q_2 + h_2\, i \\
&\ldots \\
q'_{k-1} &= q_{k-2} + h_{k-2}\, i \quad .
\end{aligned}
\tag{3.9.1}
$$

These equations may be combined into a single matrix equation,

$$
\begin{vmatrix} q'_1 \\ q'_2 \\ q'_3 \\ q'_4 \\ \ldots \\ q'_{k-1} \\ q'_k \end{vmatrix}
=
\begin{vmatrix} 0 & 0 & 0 & \ldots & 0 & 0 \\ 1 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ & & & \ldots & & \\ 0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & \ldots & 1 & 0 \end{vmatrix}
\begin{vmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ \ldots \\ q_{k-1} \\ q_k \end{vmatrix}
+ i
\begin{vmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ \ldots \\ h_{k-2} \\ h_{k-1} \end{vmatrix}
\tag{3.9.2}
$$

We recognize the matrix above as the same matrix D which appeared in the Type A multiplier analysis (3.8.1). 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**. Also, the ordering of the registers q_i differs from the ordering of (3.8.1).

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 $\mathbf{q_n}$:

$$\mathbf{q}_{n+1} = D\, \mathbf{q}_n + i_n \mathbf{h} \tag{3.9.3}$$

$$\mathbf{q}_n = D^n\, \mathbf{q}_0 \quad + [\, i_0\, D^{n-1} \;+\; i_1\, D^{n-2} + \ldots + i_{n-2}\, D \;+\; i_{n-1} I \,]\, \mathbf{h} \tag{3.9.4}$$

$$[\,\mathbf{q_n}\,]_s = q_s(n)\ . \tag{3.9.5}$$

The comments made about the matrix D all apply here as well, so we can rewrite (3.9.4) as,

$$\mathbf{q_n} = D^n\,\mathbf{q_0} + [\ i_{n-k}\ D^{k-1} + \ i_{n-3}\ D^2 + i_{n-2}\ D + i_{n-1} I\ ]\,\mathbf{h} \qquad I = D^0$$

$$= D^n\,\mathbf{q_0} + \sum_{j=1}^{k} i_{n-j}\ [D^{j-1}]\ \mathbf{h} \tag{3.9.6}$$

Time-domain output sequence for a Type B multiplier

In Fig 1.8, the output o_n was a sum of many terms. For Fig 1.10, the output is the sum of only two terms,

$$o_n = q_k(n) + h_k\, i_n = [\mathbf{q_n}]_k + h_k\, i_n$$

$$= \sum_{j=1}^{k} [\,D^n\,]_{k,j}\,[\mathbf{q_0}]_j + \sum_{j=1}^{k}\sum_{s=0}^{k-1} i_{n-j}\ [D^{j-1}]_{k,s}\,h_s + h_k\, i_n \qquad . \tag{3.9.7}$$

As in (3.8.5) we have $[\,D^n\,]_{k,j} = \delta_{k,n+j}$ and $[D^{j-1}]_{k,s} = \delta_{k,j+s}$ so both j sums go away:

$$o_n = \sum_{j=1}^{k} \delta_{k,n+j}\,[\mathbf{q_0}]_j + \sum_{j=1}^{k}\sum_{s=0}^{k-1} i_{n-j}\ \delta_{k,j+s}\,h_s + h_k\, i_n$$

$$= [\mathbf{q_0}]_{k-n} + \sum_{s=0}^{k-1} i_{n-k+s}\,h_s + h_k\, i_n = q_{k-n}(0) + \sum_{s=0}^{k} h_s\, i_{n-k+s} \qquad // \ h_k = 1$$

$$= q_{k-n}(0) + \sum_{s=0}^{k} h_s\, i_{n-k+s}\,\theta(n-k+s \ge 0) = q_{k-n}(0) + \sum_{s=0}^{k} h_s\, i_{n-k+s}\,\theta(s \ge k-n)$$

or

$$o_n = \theta(n<k)\ q_{k-n}(0) + \theta(n\ge 0) \sum_{s=\max(0,k-n)}^{k} h_s\, i_{n+s-k} \qquad // \text{ Type B multiplier} \tag{3.9.8}$$

where again we assume the first non-zero input is i_0. For n = 1,2,3..k-1 both terms are active, but for $n \ge k$ only the second term is active and the initial register values no longer influence the output. In this case that $n \ge k$, max(0,k-n) = 0, so the second sum starts at s = 0.

We can compare this Type B multiplier result to the Type A multiplier result found in the previous section:

$$o_n = \theta(n<k) \sum_{s=0}^{k-n-1} h_s\, q_{n+s}(0) + \theta(n\ge 0) \sum_{s=\max(0,k-n)}^{k} h_s\, i_{n+s-k} \qquad // \text{ Type A multiplier} \qquad \textit{(3.8.12)}$$

During the early clocks n < k the two o_n sequences differ, but for clocks n ≥ k only the second terms contribute and these second terms are *the same* for Type A and Type B multipliers. Furthermore, for n ≥k the second term sum is $\Sigma_{s=0}^{k}$.

As a check on (3.9.8), we write out the output o_n :

$o_0 = q_k(0) + h_k\, i_0$ // agrees with Fig 1.10

$o_1 = q_{k-1}(0) + h_{k-1} i_0 + h_k i_1$ // agrees with Fig 1.10

This concludes our matrix-method exercises for the two 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).

$$o_n = \sum_{s=0}^{k} h_s\, i_{n+s-k} \qquad n \geq k\ . \qquad (3.8.13)$$

3.10. Proof that a Descrambler really descrambles the output of a Scrambler.

A proposed (now in use) serial digital video standard involves the following scrambler - descrambler structure:

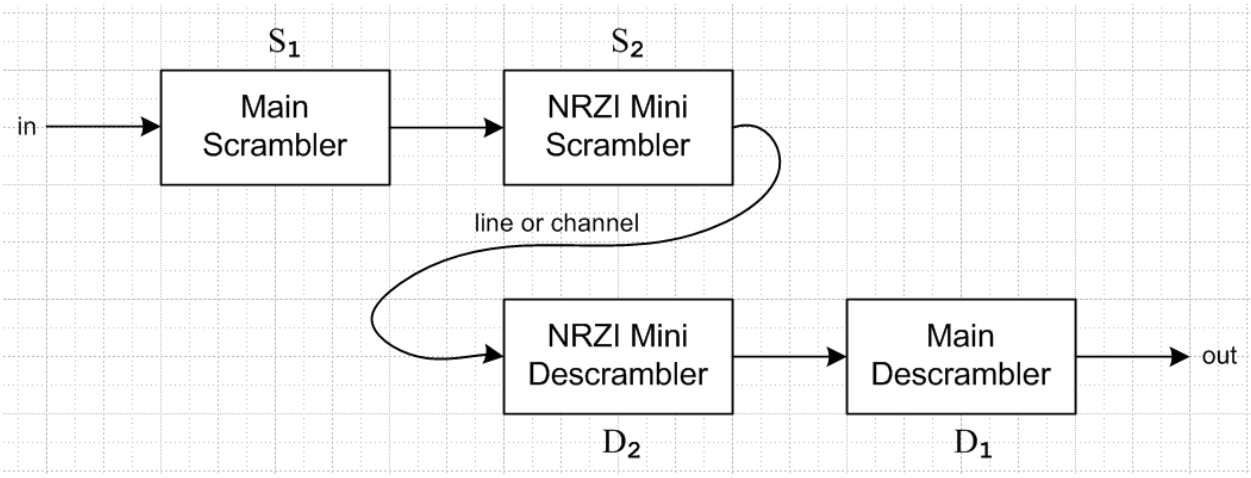


Fig 3.3: Scrambler structure for serial digital communication .

One can think of this as a set of four transformations acting to the right:

[output] = $D_1\, D_2\, S_2\, S_1$ [input] .

We want to prove that a "descrambler" really descrambles the output of its corresponding scrambler. In terms of transformations, this means that DS = 1. The "1" means there is no change to the signal after passing through a scrambler followed by the corresponding descrambler (ignoring a possible delay). "Corresponding" means the descrambler uses the same primitive polynomial h(x) as its partner scrambler.

We will show this for an arbitrary scrambler. Then, with regard to the above picture, we will have shown that

$$[\text{output}] = D_1 D_2 S_2 S_1 [\text{input}] = D_1 (D_2 S_2) S_1 [\text{input}] = (D_1 S_1) [\text{input}] = [\text{input}]$$

Proof in the Z Domain

In box (1.9.8) 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)
(Fig 1.1 or Fig 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)
(Fig 1.8 or Fig 1.10)			

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. In terms of the DS = 1 idea noted above, we can write

$$S(z) = \frac{1}{H(z)} \qquad \text{and } O(z) = S(z) I(z) \qquad \text{or} \qquad [\text{output}] = S [\text{input}] \quad \text{// scrambler}$$

$$D(z) = z^{-k} H(z) \qquad D(z)S(z) = [z^{-k} H(z)] \left[\frac{1}{H(z)}\right] = z^{-k} = \text{delay of k clocks} \sim 1 .$$

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 arbitrarily 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 Type A scrambler, using a matrix formalism. Equation (3.3.5) shows the status of the state vector **q** at time t = n clocks:

$$\mathbf{q_n} = A^n \mathbf{q_0} + \sum_{j=0}^{n-1} i_{n-1-j} A^j \mathbf{1} \quad . \tag{3.10.3}$$

Here $\mathbf{q_0}$ is the starting state vector, **1** is the unit column vector $(1,0.0...)^T$, and A is the companion matrix shown in (3.3.2). According to (3.3.2), the state of register k-s is given by the s^{th} component of the vector $\mathbf{q_n}$. Thus we can write (redefining o_n as noted below and using $(\mathbf{1})_r = \delta_{1,r}$) ,

$$o_n = q_{k-s}(n) = [\mathbf{q_n}]_s = [A^n \mathbf{q_0}]_s + \sum_{j=0}^{n-1} i_{n-1-j} [A^j]_{s,1} \quad . \tag{3.10.4}$$

Notice in Fig 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 $\mathbf{q_0}$. It is therefore the unique solution to the family of time-domain difference equations shown on the right of (3.10.1) above.

During the first k clocks of operation, we know that the Type A *descrambler* (Fig 1.8) 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:

$$o'_{k+n} = \sum_{r=0}^{k} h_r o_{n+r} \qquad n \geq 0 \quad . \tag{3.10.5}$$

This equation is valid for either a Type A or Type B descrambler.

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 ugly as we proceed, but in the end, the desired result will be obtained, and some insight will be gained along the way. Doing the indicated insertion, we get

$$o'_{k+n} = \sum_{r=0}^{k} h_r \{ [A^{n+r} \mathbf{q}_0]_s + \sum_{j=0}^{n+r-1} i_{n+r-1-j} [A^j]_{s,1} \} . \quad (3.10.6)$$

Consider the first term above, which can be expanded as

$$\sum_{r=0}^{k} h_r [A^{n+r} \mathbf{q}_0]_s = \sum_{i=0}^{k} [\mathbf{q}_0]_i \{ \sum_{r=0}^{k} h_r [A^{r+n}]_{s,i} \} = \sum_{i=0}^{k} [\mathbf{q}_0]_i \{ 0 \} = 0 . \quad (3.10.7)$$

According to Eq. (3.5.2), the sum over r inside {...} vanishes. This may be traced back to the fact that h(A) = 0. The reason this is true is that the matrix A has the "companion" form X_1 shown in GA (10.35) with $f_i = h_i$. The characteristic polynomial of $A = X_1$ is then h(x), and according to GA (10.38) it follows that h(A) = 0. This is the Cayley-Hamilton Theorem.

Since the term containing $\mathbf{q}_0$ completely vanishes in (3.10.6), we conclude that the descrambler 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:

$$o'_{k+n} = \sum_{r=0}^{k} h_r \sum_{j=0}^{n+r-1} i_{n+r-1-j} [A^j]_{s,1} . \quad (3.10.8)$$

Replace the j summation variable with a new summation variable m = n+r-1-j :

$$o'_{k+n} = \sum_{r=0}^{k} h_r \sum_{m=0}^{n+r-1} i_m [A^{n+r-1-m}]_{s,1} . \quad (3.10.9)$$

Consider now the shape of the region of the double summation in (3.10.9). Here is a picture:

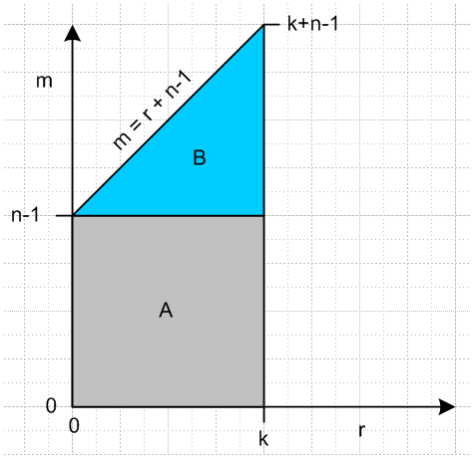


Fig 3.4

Consider first the summation over the rectangular region A excluding its top edge. Here, r goes 0 to k, and m goes 0 to n-2. The segment m = n-1 will be considered part of the upper triangle B. We get,

$$o'_{k+n} \text{ (over region A)} = \sum_{r=0}^{k} h_r \sum_{m=0}^{n-2} i_m \, [A^{n+r-1-m}]_{s,1} \quad . \tag{3.10.10}$$

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 A matrix to give zero just as happened in (3.10.7) based on (3.5.3),

$$o'_{k+n} \text{ (over region A)} = \sum_{m=0}^{n-2} i_m \, \{ \sum_{r=0}^{k} h_r \, [A^{r+n-m-1}]_{s,1} \} = 0 \quad . \tag{3.10.11}$$

Now we have only to worry about the triangular region B. So,

$$o'_{k+n} = o'_{k+n} \text{ (over region B)} = \sum_{r=0}^{k} h_r \sum_{m=n-1}^{r+n-1} i_m \, [A^{n+r-1-m}]_{s,1} \quad . \tag{3.10.12}$$

The next step is to replace index m with j = m-n+1 which yields,

$$o'_{k+n} = \sum_{r=0}^{k} h_r \sum_{j=0}^{r} i_{j+n-1} \, [A^{r-j}]_{s,1} \quad . \tag{3.10.13}$$

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:

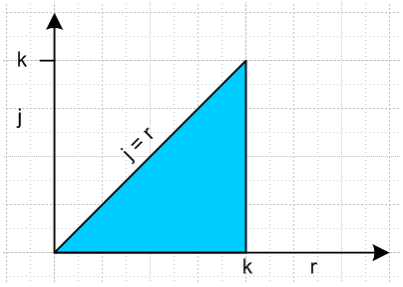


Fig 3.5

Based on this picture, we can reorder the double summation as follows:

$$o'_{k+n} = \sum_{j=0}^{k} i_{j+n-1} \sum_{r=j}^{k} h_r \, [A^{r-j}]_{s,1} \quad . \tag{3.10.14}$$

In Appendix D we prove the following non-obvious fact about the r sum appearing above,

$$\sum_{r=j}^{k} h_r \left[A^{r-j} \right]_{s,1} = \delta_{s,k-j+1} \quad . \tag{3.10.15}$$

Basically this is a property of the A matrices. Inserting (3.10.15) into (3.10.14) gives

$$o'_{k+n} = \sum_{j=0}^{k} i_{j+n-1} \{ \delta_{s,k-j+1} \} = i_{k+n-s} \; . \tag{3.10.16}$$

Recall that we took our scrambler output from register q_{k-s} of the scrambler of Fig 1.1. If we now select s=k, the output comes from register q_0 exactly as shown in Fig 1.1. In this case, the result is,

$$o'_{k+n} = i_n \qquad n = 0,1,2... \tag{3.10.17}$$

Conclusions:

(a) Eq. (3.10.17) says that if one runs a sequence i_n through a scrambler and then through the corresponding descrambler, one gets out exactly what was 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 one decides to take the scrambler output from an earlier register of Fig 1.1, then one 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 Fig 1.1, 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 "channel" of Fig 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 reason is that at a higher stack level the receiver is looking for a unique sync pattern in the data stream as in the examples of the next section. The only possible concern is in the overall latency delay which is N-(k-s). One could take the scrambler output a few registers back from the end (advanced) and attempt to cancel out any line delay N. That is, one could perhaps tune so that N - (k-s) = 0 for small N.

(e) Notice that in the above proof we assumed use of the Fig 1.1 Type A scrambler with its A matrix, but we made no assumption about whether Fig 1.8 or Fig 1.10 was used as the descrambler.

Exercise for the Reader: Carry out the above analysis for a Type B scrambler.

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 Fig 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 input 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 $\mathbf{q_n}$. 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:

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...000000000001xxxxxx...
      ↑
```

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 N zeros shown is at the scrambler input, the scrambler will contain k zeros. 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.

Here are a few informal examples showing some values of N encountered in digital video.

Example 1: The so-called "601" digital video standard has the following scan line format [see FI] :

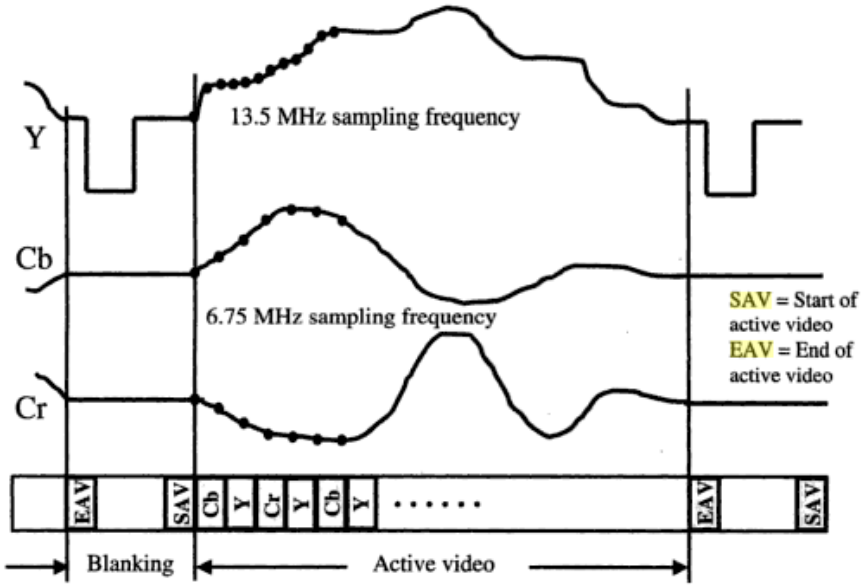


Fig. 4.2. Sampling of the Components in Accordance with ITU-BT.R.601

Fig 3.6

Here the Y are luminance samples while Cr and Cb are red and blue chrominance samples. Just before the start of the "active video" portion of the scan line there exists a sync pattern SAV (Start of Active Video) which has the following form (each grouping is 10 bits, 3FF is hex notation),

3FF, 000, 000, *** = 1111111111,0000000000,0000000000,1*********

The fourth 10-bit sample always begins with a 1 as shown, thus limiting the number of sequential zeros in the sync pattern to N = 20 bits. The other bits *** indicate information like field number and whether the scanline is in vertical blanking. There are also a few error protection bits. One other bit distinguishes SAV from EAV (End of Active Video). For 8-bit video, one has instead

FF, 00, 00, ** = 11111111,00000000,00000000,1*******

so there are now N = 16 zeros in a row. The SAV and EAV patterns are known as timing reference signals (TRS). In 8-bit video, the normal Blanking pattern is YC = 0x1080, so the longest zero strings there are N=10 bits:

0x10801080 = 0001 0000 1000 0000 0001 0000 1000 0000

In 10-bit video, we presume this extends to N = 12 bits.

Example 2: Suppose in some video specification, one were 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 N = 720*20 = 14,400 zeros. In a certain HDTV standard one gets N = 1920*20 = 38,400 zeros.

We now prove the claim made 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 Fig 1.1. 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. As usual, h(x) is assumed monic so $h_k = 1$.

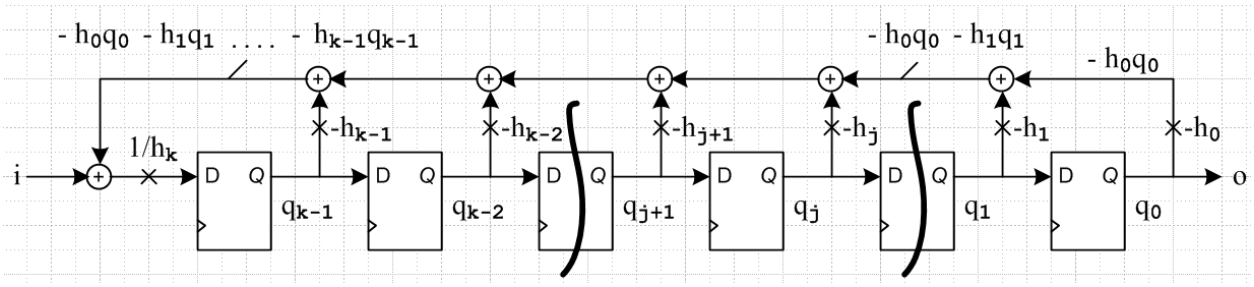


Fig 1.1: Type A polynomial divider.

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:

$$\begin{vmatrix} i_0 \\ i_1 \\ i_2 \\ i_3 \\ i_4 \\ .. \\ i_{k-2} \\ i_{k-1} \end{vmatrix} = \begin{vmatrix} a_0 & a_1 & a_2 & a_3 & .. & a_{k-3} & a_{k-2} & a_{k-1} \\ a_1 & a_2 & a_3 & a_4 & .. & a_{k-2} & a_{k-1} & b_0 \\ a_2 & a_3 & a_4 & a_5 & .. & a_{k-1} & b_0 & b_1 \\ a_3 & a_4 & a_5 & a_6 & .. & b_0 & b_1 & b_2 \\ a_4 & a_5 & a_6 & a_7 & .. & b_1 & b_2 & b_3 \\ .. & .. & .. & .. & .. & .. & .. & .. \\ a_{k-2} & a_{k-1} & b_0 & b_1 & .. & b_{k-5} & b_{k-4} & b_{k-3} \\ a_{k-1} & b_0 & b_1 & b_2 & .. & b_{k-4} & b_{k-3} & b_{k-2} \end{vmatrix} \begin{vmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ .. \\ h_{k-2} \\ h_{k-1} \end{vmatrix} + \begin{vmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ .. \\ b_{k-2} \\ b_{k-1} \end{vmatrix} \qquad (3.10.18)$$

Notice that all elements on any / diagonal are the same. We can write this in vector form as:

$$\mathbf{i} = X(\mathbf{a},\mathbf{b})\,\mathbf{h} + \mathbf{b} \qquad (3.10.19)$$

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 ($h_k = h_9 = 1$ is not part of vector **h**). In this case, each equation has only three terms. Here they are:

$i_0 = a_0 + a_4 + b_0$ $\qquad$ $i_5 = a_5 + b_0 + b_5$
$i_1 = a_1 + a_5 + b_1$ $\qquad$ $i_6 = a_6 + b_1 + b_6$
$i_2 = a_2 + a_6 + b_2$ $\qquad$ $i_7 = a_7 + b_2 + b_7$
$i_3 = a_3 + a_7 + b_3$ $\qquad$ $i_8 = a_8 + b_3 + b_8$
$i_4 = a_4 + a_8 + b_4$

This is the general result for going from **a** to **b**. To find the *kill vector*, set $b_i = 0$ to get:

$i_0 = a_0 + a_4$ $\qquad$ $i_5 = a_5$
$i_1 = a_1 + a_5$ $\qquad$ $i_6 = a_6$
$i_2 = a_2 + a_6$ $\qquad$ $i_7 = a_7$
$i_3 = a_3 + a_7$ $\qquad$ $i_8 = a_8$
$i_4 = a_4 + a_8$

So if vector **a** is sitting in the registers, the above vector **i** will result in all registers being 0.

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:

$\{ a_8, a_7, a_5. a_4, a_3, a_2. a_1, a_0 \} = \{ 1,1,1,1,0,0,0,0,1\} = 0x1E1$

Comments of the zeros problem

In general, for each video standard, one must find the specification, 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 audio. 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 clock recovery 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 one 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 sections:

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, such a scrambler with all ones as input behaves as a shift register generator with an inverter tacked onto the output. This output will therefore cycle through an inverted MLS 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 MLS sequence theorems quoted earlier then need to be adjusted to account to the effective inversion.

Proof: The GF(2) adder is just an XOR gate. It is easy to show that an inverter on one input of such an adder is equivalent to having the inverter be on the output (just make a truth table)

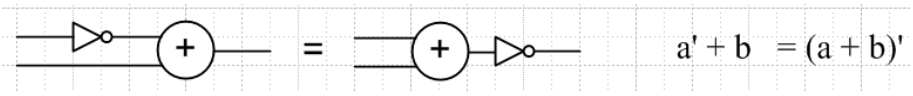


Fig 3.7

So assume the input to Fig 1.1 is all ones. We can add an input inverter and at the same time replace the input stream with all zeros, making no change in behavior. We then slide the inverter through the adder and it ends up at the D input of the q_{k-1} register. We next move the inverter to the Q output. We can then mentally move this inverter to the right in Fig 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 coming to rest at the upper input of the input XOR gate. 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 since they are then all in series at the upper input of the input XOR gate.

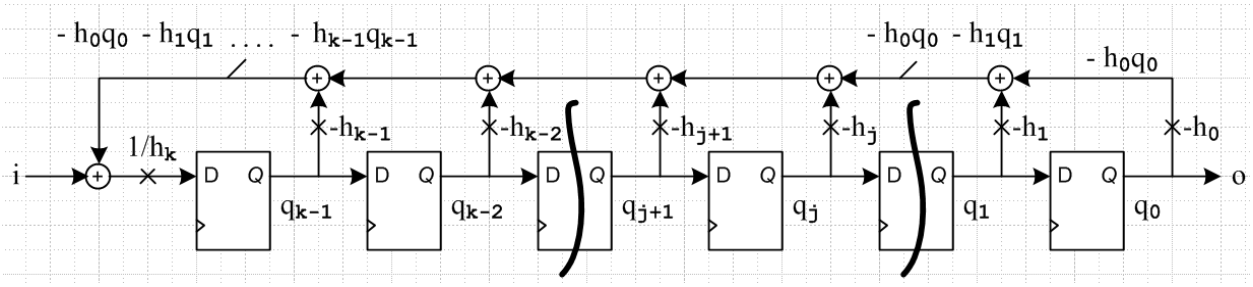


Fig 1.1: Type A polynomial divider.

Final comments:

(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 error propagation problem to speak of with a 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.

(4) Video Tilt. If the input to a scrambler is periodic with period N, the output will also be periodic with the same period if and only if the scrambler starts up with a certain specific "seed" state vector $\mathbf{q}_0$ given by (reader exercise; C is the companion matrix A or B for the Type A or Type B divider)

$$\mathbf{q}_0 = - [C^N - 1]^{-1} \sum_{j=0}^{N-1} i_{N-1-j} [C^j] \mathbf{1} \quad . \tag{3.10.20}$$

Note that since C is a Galois Field element, so is C^N and so is $C^N - 1$. Since all field elements have multiplicative inverses, the first factor is well-defined. In this case of a periodic scrambler output sequence, if the scrambler output stream has a large fraction of zeros (going across a video scanline, say), one can move far from having a 50% count of 1's and 0's (appearing as -1's in a transmitted bipolar data stream) for perhaps a long time, so a non-zero DC offset builds up during a scanline time. This problem is known as "video tilt" and can cause certain components like fiber optic converter modules to malfunction. Of particular interest are constant colors which result in the video stream having a period which is a multiple of 10 or 20 (assuming 10-bit video samples). Situations can arise where 19 out of every 20 bits in the scrambled signal are zeros. With a 9-register scrambler, one might expect the bad seed to be present on one of 511 scanlines on the average, which is on the order of one scanline per frame.

Appendix A. Tale of Two Rings

Here we ask the question: Is there some kind of analogy between real numbers and polynomials when the real numbers are written in usual decimal notation and when the polynomials have coefficients which lie in Mod(10) so they can also be written in a decimal notation. We pursue this question first in terms of multiplication and then later division.

1. Multiplication Example

Define an ∞-tuple of Mod(10) elements as something having this form

.....xxxxxxx.xxxxxx.....

For particular ∞-tuples, we need only show the required digits, as for example for 2242.457. That is, we ignore the infinite set of leading zeros and the infinite set of trailing zeros.

(a) We can represent a base-10 number (like 2242.457) as an ∞-tuple of elements each of which lie in field Z_{10} = Mod(10). Such ∞-tuples themselves form a ring which has a 1-to-1 correspondence with the ring of real numbers. Let's call this ring R_1. We have a certain familiar rule for the multiplication of such ∞-tuples in R_1. The rule is that we convert each ∞-tuple to a real number, get the product of the two real numbers, and then convert the result back to an ∞-tuple. This rule is very effectively implemented by a calculator. For example

22.16 * 3.42 = 75.7872

Since the real numbers form a field, R_1 is also a field, but we are mostly concerned with its ring sense.

(b) On the other hand, we can represent a polynomial f(z) having coefficients in Mod(10) as a an ∞-tuple of elements each in Z_{10} = Mod(10). For example,

22.16 = the ∞-tuple for polynomial f(z) = $2z^1 + 2z^0 + 1z^{-1} + 6z^{-2}$

The "decimal point" is placed just before any negative powers which might be present.

Such ∞-tuples form a ring we shall call R_2. We have a rule for multiplying these ∞-tuples; it is just "what we do" when we multiply polynomials whose coefficients lie in Mod(10). The result can also be represented as an ∞-tuple. For example (notice for example that $12z^{-4}$ is replaced by $2z^{-4}$)

$(2z + 2 + 1z^{-1} + 6z^{-2}) \bullet (3 + 4z^{-1} + 2z^{-2}) = (6z + 4 + 5z^{-1} + 6z^{-2} + 6z^{-3} + 2z^{-4})$.

It is easy to have Maple multiply polynomials in this manner, so we can then verify the above hand calculation:

```
f1 := 2*z + 2 + 1/z + 6/z^2:
f2 := 3 + 4/z + 2/z^2:
f := expand(f1*f2) mod 10;
```

$$f := 6z + 4 + 5\frac{1}{z} + 6\frac{1}{z^2} + 6\frac{1}{z^3} + 2\frac{1}{z^4}$$

The upshot is that, in ring R_2 we have

22.16 • 3.42 = 64.5662

The ∞-tuple product here is not the real number 64.5622, it is the ∞-tuple which represents the polynomial shown above in R_2.

The point so far is that the product of two ∞-tuples depends on which ring these ∞-tuples belong to.

In GA Section 3 (a) this ring R_2 is simply called R, and it is shown that it really is a ring and has a multiplicative identity. This ring can be defined just for "proper" polynomials (non-negative powers only), or for general polynomials (all powers allowed).

(c) The real number multiplication shown in (1) above can be done as follows:

```
 22.16
  3.42
   .4432
  8.864
 66.48
 75.7872
```

In this process, carries are maintained both in computing partial product rows and then in adding up those rows.

(d) The polynomial multiplication shown in (2) above can be done as follows:

```
 22.16
  3.42
   .4422
  8.844
 66.38
 64.5662
```

In this process, carries are discarded both in computing partial product rows and then in adding up those rows. This is the way Mod(10) math works.

2. Division Example 1

(a) We continue to represent a base-10 number (like 2242.457) as an ∞-tuple of elements each of which lie in field Z_{10} = Mod(10). As noted above, such ∞-tuples themselves form a ring R_1 which has a 1-to-1 correspondence with the ring of real numbers. We have a certain familiar rule for the *division* of such ∞-tuples in R_1. The rule is that you convert each ∞-tuple to a real number, get the quotient of the two real numbers, and then convert the result back to an ∞-tuple. This rule is very effectively implemented by a calculator (such as Maple). For example,

22.16 / 3.42 = 6.479532164.....

Since this example is a rational number, the decimals go into a repeating pattern. For 70 digits,

```
Digits := 70:
22.16/3.42;
      6.47953216374269005847953216374269005847953216374269005847953216374269
```

and one sees that

6.479532163742690058 479532163742690058 479532163742690058 479532163742690...

(b) On the other hand, we can represent a polynomial f(z) as a an ∞-tuple of elements each in Z_{10} = Mod(10). For example

22.16 = the ∞-tuple for polynomial f(z) = $2z^1 + 2z^0 + 1z^{-1} + 6z^{-2}$

Such ∞-tuples form a ring R_2. We have a rule for dividing these ∞-tuples; it is what we do when we do long division of polynomials whose coefficients lie in Mod(10). The result can also be represented as an ∞-tuple.

For example

$(2z + 2 + 1z^{-1} + 6z^{-2}) / (3 + 4z^{-1} + 2z^{-2}) = 4z + 2 + 5z^{-1} + 4z^{-2} + 8z^{-3} + 0z^{-4} + 8z^{-5} + ...$

22.16/3.42 = 42.54808....

(c) To find the ratio of two ∞-tuples in R_1, we can do long division by the usual method. For example,

```
          6.479.....
342 | 2216.0000
      2054      6
       1640
       1368     4
        2720
        2394    7
         3260
         3078   9
                      .......
```

Each time the divisor is multiplied by a quotient integer, there is *carry*. And each time there is a subtraction to obtain a new current dividend, there is *borrow*.

(d) To carry out a polynomial division in R_2, one method is to carry out the long division algorithm, armed with a multiplication and addition table for Mod(10). For example, things start off this way:

```
          42.54808 .........
342  |  2216.0000
       268              4
        636
        684             2
         520
         500            5
          200
          268           4
           420
           426          8
            40
            00          0
            400
            426         8
                              .......
```

At each stage of the process, there is *no carry* when the divisor is multiplier by a quotient integer, and there is *no borrow* when the subtraction is done at each stage to get the new current divisor.

Our conclusion about carries and borrows for division is thus very similar to that for multiplication found earlier.

The above manual method for polynomial division in R_2 is quite tedious but could of course be automated. For illustration, here is another method which allows a simple Maple evaluation of the quotient coefficients.

We know that the division O(z) = I(z)/H(z) (with all positive and negative powers allowed) results in the following time-domain equation for the polynomial coefficients,

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

Multiplying top and bottom by z^2, our division example above becomes (we redefine I and H),

$$(2z^3 + 2z^2 + 1z + 6) / (3z^2 + 4z + 2) = I(z) / H(z)$$

giving us a proper polynomial for H(z). In our ∞-tuple notation, this says

22.16/3.42 = 2216/342 .

This last line is true *not* because we multiply top and bottom by 100, because the ratio does not refer to the division of real numbers. It is true because of the way polynomial division is defined:

$$n(z)/d(z) = q(z) \Leftrightarrow n(z) = q(z)d(z) \Leftrightarrow [z^s n(z)] = q(z)\,[z^s\, d(z)] \Leftrightarrow [z^s n(z)]/\,[z^s\, d(z)] = q(z)\ .$$

In general, we can therefore always slide the "decimal point" in numerator and denominator by the same number of positions in either direction, just as we can when dividing ∞-tuples in the real field R_1.

As a counterexample, suppose we were to multiply our division ratio above top and bottom by 5. We would get (everything is Mod(10)) :

$$(2z^3 + 2z^2 + 1z + 6) / \ (3z^2 + 4z + 2) \ = (10z^3 + 10z^2 + 5z + 30) / \ (15z^2 + 20z + 10)$$

$$= (5z) / (5z^2) \ = z^{-1} \qquad \text{// not true !}$$

Now, given our new H(z) shown above, the time-domain equation simplifies to,

$$i_n \ = \ \sum_{j=0}^{2} h_j\, o_{n+j} \ = h_0 o_n + h_1 o_{n+1} + h_2 o_{n+2}\ . \qquad (1.9.5)$$

This can be regarded as the following recursion relation,

$$h_2 o_{n+2} \ = i_n - h_0 o_n - \ h_1 o_{n+1}$$

or

$$o_{n+2} \ = (i_n/h_2) - (h_0/h_2) o_n - \ (h_1/h_2) o_{n+1}$$

or

$$o_{n+2} \ = a\, i_n - b o_n - \ c o_{n+1} \qquad a \equiv (1/h_2) \qquad b \equiv \ (h_0/h_2) \qquad c \equiv \ (h_1/h_2)$$

in our example: $\qquad a \equiv (1/3) = 7 \qquad b \equiv \ (2/3) = 4 \qquad c \equiv \ (4/3) \ = 8$

The integer results for a,b,c are those of Mod(10). For example

$a = 3^{-1} = 7$, since $3*7 = 21 = 1$
$b = 2*3^{-1} = 2*7 = 14 = 4$
$c = 4*3^{-1} = 4*7 = 28 = 8$.

Using our usual manner of denoting the coefficients of I(z) and O(z) we have

$I(z) = i_{-3}\, z^3 + i_{-2}z^2 + i_{-1}z + i_0$ // in our example $= 2z^3 + 2z^2 + 1z + 6$

$O(z) = o_{-1}z + o_0 + o_1z^{-1} + o_2z^{-2} + o_3z^{-3} + ...$

We know that the leading coefficient of O(z) is o_{-1} as shown just looking at our polynomial ratio. All o_j prior to this vanish.

Here then is Maple code to compute the o_j coefficients for $(2z^3 + 2z^2 + 1z + 6) / (3z^2 + 4z + 2)$:

```
for j from -10 to 100 do i[j] := 0 od:          # clear out all i values
i[-3] := 2: i[-2] := 2: i[-1] := 1: i[0] := 6:  # set non-zero i values
a := 7: b := 4: c := 8:                         # set parameters
o(-3) := 0: o(-2) := 0:                         # initial values
for n from -3 to 40 do    # use recursion relation to compute the o values
     o(n+2) := a*i[n] - b*o(n) - c*o(n+1) mod 10;
od:
x := seq(o(n),n=-1..40);  # display these values
```

$$x := 4, 2, 5, 4, 8, 0, 8, 6, 0, 6, 2, 0, 2, 4, 0, 4, 8, 0, 8, 6, 0, 6, 2, 0, 2, 4, 0, 4, 8, 0, 8, 6, 0, 6, 2, 0, 2, 4, 0, 4, 8, 0$$

The first 20 terms of the quotient polynomial O(z) are then

```
unprotect(O): O := 0:   # it happens that O is the Order-Of function in Maple
for n from -1 to 20 do  # construct the polynomial O(z) from the o(n)
  O := O + o(n)*z^(-n);
  od:
O;
```

$$4z + 2 + 5\frac{1}{z} + 4\frac{1}{z^2} + 8\frac{1}{z^3} + 8\frac{1}{z^5} + 6\frac{1}{z^6} + 6\frac{1}{z^8} + 2\frac{1}{z^9} + 2\frac{1}{z^{11}} + 4\frac{1}{z^{12}} + 4\frac{1}{z^{14}} + 8\frac{1}{z^{15}} + 8\frac{1}{z^{17}} + 6\frac{1}{z^{18}} + 6\frac{1}{z^{20}}$$

We can verify our truncated O(z) result by multiplying it times H(z) to see how close we come to I(z):

```
H := 3*z^2+4*z+2:
I_ := 2*z^3+2*z^2+1*z+6:
expand(O*H - I_) mod 10: sort(%);
```

$$4\frac{1}{z^{19}} + 2\frac{1}{z^{20}}$$

We have now shown that our example polynomial division has this result,

$(2z + 2 + 1z^{-1} + 6z^{-2}) / (3 + 4z^{-1} + 2z^{-2}) = 4z + 2 + 5z^{-1} + 4z^{-2} + 8z^{-3} + 0z^{-4} + ...$

We can write this in terms of R_2 ring ∞-tuples in this manner

221.6 / 34.2 = 42.5480....

The ∞-tuple product here is not the real number 42.5488...., it is the ∞-tuple which represents the polynomial shown above in R_2.

From the Maple output we in fact have

22.16 / 3.42 = 42.5 480860620240 480860620240 480860620240 480 ....

and we see that the sequence 480860620240 repeats forever.

We conclude then by comparing the R_1 and R_2 ∞-tuple division notations:

R_1 22.16 / 3.42 = 6.479532163742690058 479532163742690058 479532163742690058 ....

R_2 22.16 / 3.42 = 42.5 480860620240 480860620240 480860620240 ....

3. Division Example 2

What happens now if we shuffle a pair of coefficients in the previous division example? Suppose we have this polynomial division of interest, where the 4 and 3 have been swapped,

$(2z + 2 + 1z^{-1} + 6z^{-2}) / (4 + 3z^{-1} + 2z^{-2}) = ???$

As before multiply top and bottom by z^2 to get

$(2z^3 + 2z^2 + 1z + 6) / (4z^2 + 3z + 2) \equiv I(z) / H(z)$.

In the ∞-tuple world of the real number ring R_1 we can write

22.16/4.32 = 2216/432 = 5.1 296 296 296 .....

with no complications. But something new appears in our R_2 ring polynomial division. The numerator is unchanged, and the output sequence o_j still begins with o_{-1} . The recursion relation is the same but with these new parameters

$o_{n+2} = a\, i_n - b o_n - c o_{n+1}$ $a \equiv (1/h_2)$ $b \equiv (h_0/h_2)$ $c \equiv (h_1/h_2)$

in our example: $a \equiv (1/4)$ $b \equiv (2/4)$ $c \equiv (3/4)$

Maple can do the polynomial division in terms of real polynomial coefficients,

```
for j from -10 to 100 do i[j] := 0 od:          # clear out all i values
i[-3] := 2: i[-2] := 2: i[-1] := 1: i[0] := 6:  # set non-zero i values
a := (1/4): b := (2/4): c := (3/4):             # set parameters
o(-3) := 0: o(-2) := 0:                         # initial values
for n from -3 to 40 do    #use recursion relation to compute the o values
     o(n+2) := a*i[n] - b*o(n) - c*o(n+1);
od:
x := seq(o(n),n=-1..8);  # display these values
```

$$x := \frac{1}{2}, \frac{1}{8}, \frac{-3}{32}, \frac{193}{128}, \frac{-555}{512}, \frac{121}{2048}, \frac{4077}{8192}, \frac{-13199}{32768}, \frac{6981}{131072}, \frac{84649}{524288}$$

```
unprotect(O): O := 0:   # it happens that O is the Order-Of function in Maple
for n from -1 to 8 do  # compute the polynomial O(z) = I(z)/H(z)
  O := O + o(n)*z^(-n);
  od:
O;
```

$$\frac{1}{2}z + \frac{1}{8} - \frac{3}{32}\frac{1}{z} + \frac{193}{128}\frac{1}{z^2} - \frac{555}{512}\frac{1}{z^3} + \frac{121}{2048}\frac{1}{z^4} + \frac{4077}{8192}\frac{1}{z^5} - \frac{13199}{32768}\frac{1}{z^6} + \frac{6981}{131072}\frac{1}{z^7} + \frac{84649}{524288}\frac{1}{z^8}$$

```
H := 4*z^2+3*z+2:
I_ := 2*z^3+2*z^2+1*z+6:
expand(O*H - I_): sort(%);
```

$$\frac{309795}{524288}\frac{1}{z^7} + \frac{84649}{262144}\frac{1}{z^8}$$

where in the last line we again verify the truncated quotient so obtained.

The *problem* is that in our R_2 ∞-tuple ring world we need a quotient polynomial which has coefficients in the ring Mod(10). It is easy to show that, for general Mod(2n), no even element has an inverse. For example, 4 does not have an inverse in Mod(10) since

4*0 = 0
4*1 = 4
4*2 = 8
4*3 = 12 = 2
4*4 = 16 = 6
4*5 = 20 = 0
4*6 = 24 = 4
4*7 = 28 = 8
4*8 = 32 = 2
4*9 = 36 = 6

Notice that 1 is missing from the list. One reason it is missing is that all the numbers in the right column must be even!

Therefore, there is no way to write our quotient,

$(2z^3 + 2z^2 + 1z + 6) / (4z^2 + 3z + 2) = (1/2)z + (1/8) - (3/32)z^{-1} + ...$

such that the quotient has coefficients in Mod(10). This says that in the ring R_2 the quotient of the two polynomials does not exist. Thus we end up with the following conclusions for Example 2

R_1 22.16 / 3.42 = 2216 / 342 = 6.479532163742690058 479532163742690058 ....

R_2 22.16 / 3.42 = 2216 / 342 = does not exist

If instead of Mod(10) we were to use symbols from Mod(p) where p is a prime number, then all elements have inverses and this problem cannot arise. An example would be Mod(7) with digits 0,1,2,3,4,5,6. Of course in this case, the ratios for R_1 would be written in base-7 math.

Appendix B: A Proof of Fact 4 (2.3.8)

Here is what we want to prove:

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

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

is a solution of the difference equation (2.3.2),

$$\sum_{j=0}^{k} h_j \, o_{m+j} = 0 \qquad m = 0,1,2.... \quad . \qquad (2.3.2)$$

The g(x) appearing in (2) below is what one gets doing the division $g(x) = (x^n-1)/h(x)$, and we assume that such an n exists such that such an even division exists (this assumption is discussed below).

Our proof is broken down into six short parts (1) through (6).

(1) GA Chapter 7 discusses the notion of a linear block code associated with a matrix G. One applies the non-square matrix G to a data vector **d** having k components and obtains thereby a code vector **c** having n components with n > k. If the vectors are row vectors, one writes **c** = **d**G. Since the data components lie in GF(p), there are p^k possible data words. Since each data word maps into one code word, there are p^k code words in the code.

Associated with matrix G is another matrix H called the parity check matrix for code G and one finds that, for any code word **c,** $\mathbf{c}H^T = \mathbf{0}$. One can use matrix H then to "check" to see whether **c** really is a code word of code G by seeing if $\mathbf{c}H^T = \mathbf{0}$ or not. One can show that the matrices G and H satisfy

$$GH^T = 0 \qquad \text{and} \qquad HG^T = 0 \qquad (B.1)$$

where each 0 is an all-zeros matrix of an appropriate size.

It turns out that the matrix H can itself be used to define another linear block code which is known as the dual code to code G. One applies the matrix H to a data vector **d'** having n-k components and obtains thereby a code vector **c'** having n components. If the vectors are row vectors, one writes **c'** = **d'**H.

It then turns out that the $\mathbf{c}' \bullet \mathbf{c} = 0$ for any code word **c** of code G and any code word **c'** of code H. The proof is simple:

$$\mathbf{c}' \bullet \mathbf{c} = \mathbf{c}'\mathbf{c}^T = (\mathbf{d}'H)(\mathbf{d}G)^T = (\mathbf{d}'H)G^T\mathbf{d}^T = \mathbf{d}'(HG^T)\mathbf{d}^T = \mathbf{d}'(0)\mathbf{d}^T = 0 \quad . \qquad (B.2)$$

(2) GA Chapter 8 then discusses cyclic codes which have a close relationship to the block codes. The k components of the data vector **d** are now the coefficients of a polynomial d(x) of degree k-1. The code words are generated now as the coefficients of polynomial c(x) of degree n-1 which is obtained from polynomial multiplication,

$$c(x) = d(x)\, g(x). \qquad (B.3)$$

Here g(x) is a polynomial of degree n-k which is called the generator of the cyclic code. Associated with this cyclic code is a dual cyclic code where we have

$$c'(x) = d'(x)\, h(x) \,, \tag{B.4}$$

where the n-k components of the data polynomial d'(x) are the components of the data vector **d'** mentioned above. Polynomial h(x) is the generator of the dual code and has degree k. The relationship between g(x) and h(x) is this

$$g(x)h(x) = (x^{n} - 1) \ . \tag{B.5}$$

Now the coefficients of c(x) which we call c_i are not necessarily the same as the c_i in part (1) above, but they are nonetheless the coefficients of some code word in the space of code words. The same comment applies to c'(x) and the c'_i.

(3) In (2.3.6) we *assumed* that there existed an integer n such that $g(x)h(x) = (x^{n} - 1)$ where h(x) was the polynomial used to construct the polynomial representation $GF(q=p^{k}) = R / (h(x))$ in (2.2.6). If h(x) is monic and irreducible with respect to GF(p) (sometimes we say "under GF(p)"), we know from (2.2.10) that h(x) is a minimum polynomial for some element α of GF(q). From the discussion above (2.2.32), we know that such an h(x) divides evenly into $(x^{q-1} - 1)$. We noted that there might be some n smaller than q-1 for which h(x) divides evenly into $(x^{n}-1)$. Whatever the smallest n is, we just call it n (known as the period of h(x)). It might be q-1 if nothing smaller works. So, given our irreducible monic h(x) of Chapter 2, we know for sure that there exists a g(x) such that $g(x)h(x) = (x^{n} - 1)$. So we use that exact g(x) to define the cyclic code described in (2) above, and then h(x) is our h(x) of interest in Chapter 2, it is not just some h(x) pulled out of a hat.

(4) Certainly d'(x) = 1 is a legal data word for the cyclic dual code and if we use it in (B.4) we obtain the legal dual code word c'(x) = h(x) . We can then write this equation c'(x) = h(x) as

$$c'_0 + c'_1 x + c'_2 x^2 + \ldots c'_k x^k + \ldots + c'_{n-1} x^{n-1} = h_0 + h_1 x + h_2 x^2 + \ldots h_k x^k + 0 + \ldots + 0 x^{n-1}$$

We then write a vector equality for the coefficients,

$$\mathbf{c'} = \mathbf{h} \qquad \text{where} \qquad \mathbf{h} \equiv (h_0, h_1, \ldots\ldots h_k, 0, 0 \ldots 0) \tag{B.6}$$

where we fill out the end of vector **h** with a set of n-k-1 zeros so it then has n components to match **c'**. We then apply (B.2) to conclude that, for any code word **c** of the cyclic code generated by g(x),

$$\mathbf{h} \bullet \mathbf{c} = 0 \ . \tag{B.7}$$

Writing this out as a summation gives,

$$\sum_{j=0}^{k} h_j c_j = 0 \qquad \text{(B.8)}$$

where the numbers $\{c_0, c_1, c_2, \ldots c_k\}$ are the first k+1 coefficients of **c**.

We have just proved in (B.8) this fact:

Fact : The dot product of the coefficients of h(x) with the first k+1 coefficients of any code word is 0.

(5) 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 permutations (rotations) of code words are also code words. Thus we have proved that :

Fact: The dot product of the coefficients of h(x) with *any* consecutive k+1 coefficients of any code word gives 0.

For example if k = 3 we might have this situation where (B.8) includes the products shown here as pairs of numbers one above the other :

c_0	c_1	c_2	c_3	c_4	c_5	.....	c_{n-1}
		h_0	h_1	h_2	h_3		

But (B.8) will also be true for any alignment which wraps the last coefficient like this one.

c_0	c_1	c_2	c_3	c_4	c_5	.....	c_{n-1}
h_1	h_2	h_3					h_0

Thus, when we say "consecutive coefficients" we would include a sequence like this

.... c_{n-2} c_{n-1} c_0 c_1 c_2.....

(6) Now consider the difference equation (2.3.2),

$$\sum_{j=0}^{k} h_j \; o_{m+j} = 0 \qquad m = 0,1,2.... \qquad \textit{(2.3.2)} \qquad \text{(B.9)}$$

Picture the sequence $\{o_i\}$ as an infinite row of symbols. Picture the sequence $\{h_j\}$ as a finite row of symbols. Place the row $\{h_j\}$ anywhere under the infinite row $\{o_i\}$, then take the sum of the product of the abutting symbols as done above. This is then the left side of (B.9), and the result is supposed to be zero. Varying m means varying the position of the row $\{h_j\}$. Here is a picture drawn for k=3 and m=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:

$$\{o_i\} = \{\mathbf{c}, \mathbf{c}, \mathbf{c}, \mathbf{c} \ldots\} \tag{B.10}$$

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**. Suppose k = 3 and n = 5 (the size of the code words). The for (B.9) with m = 4 we would have

c_0	c_1	c_2	c_3	c_4	c_0	c_1	c_2	c_3	c_4	c_0	c_1 ...
				h_0	h_1	h_2	h_3				

No matter where we put our h_i row (that is, for any m > 0), equation (B.9) is satisfied. Therefore, the sequence of (B.10) is a solution of the difference equation (B.9) and that is what we set out to prove.

Appendix C: A List of Primitive Polynomials for GF(2^k)

The list below was compiled by E.J. Watson in 1962 [WA].

This list gives a primitive polynomial of GF(2^k) 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.

Examples: 9 4 0 $\Rightarrow$ $h(x) = x^9 + x^4 + 1$ 52 3 0 $\Rightarrow$ $h(x) = x^{52} + x^3 + 1$

(table is on next page)

Primitive Polynomials (mod 2)

1	0						51	6	3	1	0			
2	1	0					52	3	0					
3	1	0					53	6	2	1	0			
4	1	0					54	6	5	4	3	2	0	
5	2	0					55	6	2	1	0			
6	1	0					56	7	4	2	0			
7	1	0					57	5	3	2	0			
8	4	3	2	0			58	6	5	1	0			
9	4	0					59	6	5	4	3	1	0	
10	3	0					60	1	0					
11	2	0					61	5	2	1	0			
12	6	4	1	0			62	6	5	3	0			
13	4	3	1	0			63	1	0					
14	5	3	1	0			64	4	3	1	0			
15	1	0					65	4	3	1	0			
16	5	3	2	0			66	8	6	5	3	2	0	
17	3	0					67	5	2	1	0			
18	5	2	1	0			68	7	5	1	0			
19	5	2	1	0			69	6	5	2	0			
20	3	0					70	5	3	1	0			
21	2	0					71	5	3	1	0			
22	1	0					72	6	4	3	2	1	0	
23	5	0					73	4	3	2	0			
24	4	3	1	0			74	7	4	3	0			
25	3	0					75	6	3	1	0			
26	6	2	1	0			76	5	4	2	0			
27	5	2	1	0			77	6	5	2	0			
28	3	0					78	7	2	1	0			
29	2	0					79	4	3	2	0			
30	6	4	1	0			80	7	5	3	2	1	0	
31	3	0					81	4	0					
32	7	5	3	2	1	0	82	8	7	6	4	1	0	
33	6	4	1	0			83	7	4	2	0			
34	7	6	5	2	1	0	84	8	7	5	3	1	0	
35	2	0					85	8	2	1	0			
36	6	5	4	2	1	0	86	6	5	2	0			
37	5	4	3	2	1	0	87	7	5	1	0			
38	6	5	1	0			88	8	5	4	3	1	0	
39	4	0					89	6	5	3	0			
40	5	4	3	0			90	5	3	2	0			
41	3	0					91	7	6	5	3	2	0	
42	5	4	3	2	1	0	92	6	5	2	0			
43	6	4	3	0			93	2	0					
44	6	5	2	0			94	6	5	1	0			
45	4	3	1	0			95	6	5	4	2	1	0	
46	8	5	3	2	1	0	96	7	6	4	3	2	0	
47	5	0					97	6	0					
48	7	5	4	2	1	0	98	7	4	3	2	1	0	
49	6	5	4	0			99	7	5	4	0			
50	4	3	2	0			100	8	7	2	0			
107	7	5	3	2	1	0	127	1	0					

On-line data is also available, see Olofsson [OL] for an extensive list. Some sample Olofsson data:

```
Primitive polynomials over GF( 2 )
==========================================================
The following is a list of primitive polynomials over GF( 2 )
of degrees from 2 to 100 .

x^2 + x + 1
x^3 + x + 1
x^4 + x + 1
x^5 + x^2 + 1
x^6 + x^4 + x^3 + x + 1
x^7 + x + 1
x^8 + x^4 + x^3 + x^2 + 1
x^9 + x^4 + 1
x^10 + x^6 + x^5 + x^3 + x^2 + x + 1
x^11 + x^2 + 1
x^12 + x^7 + x^6 + x^5 + x^3 + x + 1
x^13 + x^4 + x^3 + x + 1
x^14 + x^7 + x^5 + x^3 + 1
x^15 + x^5 + x^4 + x^2 + 1
x^16 + x^5 + x^3 + x^2 + 1
x^17 + x^3 + 1
x^18 + x^12 + x^10 + x + 1
x^19 + x^5 + x^2 + x + 1
x^20 + x^10 + x^9 + x^7 + x^6 + x^5 + x^4 + x + 1
x^21 + x^6 + x^5 + x^2 + 1
x^22 + x^12 + x^11 + x^10 + x^9 + x^8 + x^6 + x^5 + 1
x^23 + x^5 + 1
x^24 + x^16 + x^15 + x^14 + x^13 + x^10 + x^9 + x^7 + x^5 + x^3 + 1
x^25 + x^8 + x^6 + x^2 + 1
x^26 + x^14 + x^10 + x^8 + x^7 + x^6 + x^4 + x + 1
```

The number of primitive polynomials for GF(2^k) is given by $\varphi(2^k-1)/k$, Euler's totient function. In general, this is a large number and the above lists give just one sample primitive polynomial.

GA Chap 5 (k) gives a Maple program which computes all minimum polynomials of any GF(p^m). This program is not so useful for large m since in its current form it insists on displaying *all* minimum polynomials. For GF(2^{26}), for example, there are $\varphi(2^{26}-1)/26 =$ 1,719,900 primitive polynomials which include the 26 6 2 1 0 one shown in Watson's table and the one 26 14 10 8 7 6 4 1 1 above.

Appendix D: Proof of (3.10.15)

Here is the fact to be proved:

$$\sum_{r=j}^{k} h_r \, [A^{r-j}]_{s,1} = \delta_{s,k-j+1} \qquad \text{for } j = k,k-1,k-2..... \text{ and for any } s \text{ in } (1,k) \tag{D.1}$$

where A is the kxk companion matrix shown in (3.3.2). We can write its elements as,

$$A_{ij} = \delta_{i,j+1} - \delta_{i,1} \, h_{k-j} \, . \tag{D.2}$$

We assume as usual that h(x) of degree k is monic so $h_k = 1$.

We will prove (D.1) by induction.

(a) To start, we show that (D.1) is true for j = k. In this case, the LHS of (D.1) has one term and (D.1) says,

$$h_k \, I_{s,1} = \delta_{s,1}$$

But this is trivially true since $h_k = 1$ and I = identity matrix = A^0. So, we are off and running.

(b) Now we show that if (D.1) is true for some j, then it is also true for j-1 . Start with (D.1), and apply the operation $\Sigma_{s=0}^{k} [A]_{n,s}$ to both sides:

$$\sum_{s=0}^{k} [A]_{n,s} \sum_{r=j}^{k} h_r \, [A^{r-j}]_{s,1} = \sum_{s=0}^{k} [A]_{n,s} \, \delta_{s,k-j+1}$$

so

$$\sum_{r=j}^{k} h_r \, [A^{r-j+1}]_{n,1} = [A]_{n,k-j+1}$$

Using (D.2) we find,

$$\sum_{r=j}^{k} h_r \, [A^{r-(j-1)}]_{n,1} = [A]_{n,k-j+1} = \delta_{n,(k-j+1)+1} - \delta_{n,1} \, h_{k-(k-j+1)}$$

so

$$\sum_{r=j}^{k} h_r \, [A^{r-(j-1)}]_{n,1} = \delta_{n,k-(j-1)+1} - \delta_{n,1} \, h_{j-1} \, . \tag{D.3}$$

Now, if the sum on the LHS of (D.3) *had* a term for r = j-1, here is what that term would be:

$h_{j-1}\,[\,A^{0}]_{n,1} = h_{j-1}\delta_{n,1}$.

But this is the second term on the right of (D.3). So (D.3) can be rewritten as

$$\sum_{r=j-1}^{k} h_r \quad [\,A^{r-(j-1)}\,]_{n,1} \;=\; \delta_{n,k-(j-1)+1} \quad . \tag{D.4}$$

If we now set n=s, (D.4) is the same as (D.1) with j replaced by j-1. Thus, assuming (D.1) was true for j, we have shown it is true for j-1. Since we also showed (D.1) to be true for j = k, our induction proof has shown that (D.1) must be true for j = k, k-1, k-2 ... **QED**

References

[AS] ANSI/SMPTE 259M-1997 Serial Digital Interface (SDI) standard for digital television signals passing through a coaxial cable (sdi.org.ru/smpte-259m.pdf).

[FI] W. Fischer, *Digital Video and Audio Broadcasting Technology*, *3rd Ed.* (Springer-Verlag, Berlin, 2010).

[LE] D.G. Leeper, "A Universal Digital Data Scrambler", Bell System Technical Journal (BSTJ), Vol. 52, Issue 10, Dec 1973, pp 1851-186. The famous BSTJ ran from 1922 to 1983 and its entire contents is on line free at http://www3.alcatel-lucent.com/bstj/. This is a stunning contribution of many people to easily and freely accessible scientific knowledge.

[GA] P. Lucht, *Galois Fields and Cyclic Codes* (2013), http://user.xmission.com/~rimrock. If not there, search on "Phil Lucht Documents".

[FT] P. Lucht, *Fourier Transforms and their Application to Pulse Amplitude Modulated Signals* (2013), http://user.xmission.com/~rimrock. If not there, search on "Phil Lucht Documents".

[OL] M. Olofsson (Linkoping University, Sweden), lists of primitive polynomials over GF(2).
http://www.commsys.isy.liu.se/en/staff/mikael/polynomials/primpoly

[PW] W.W. Peterson and E.J. Weldon, *Error Correcting Codes, 2nd Ed.* (The MIT Press, Cambridge, 1972).

[WA] E.J. Watson, "Primitive Polynomials (Mod 2)", Math. Comp. 16, pp 368-369 (1962).
Search on: E.J. Watson "primitive polynomials".