University of Alberta

A Combined Error Control and Constrained Sequence Code through Multimode Coding

Aaron Hughes

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 2005

To my wife Valerie, we did it baby!

Abstract

This thesis introduces a novel approach of integrating error control codes and constrained sequence codes into a single monolithic code. This is done in order to overcome drawbacks with the conventional manner in which these coding procedures are concatenated. The technique is based on the principle of multimode coding, where each source word is represented by a set of complementary error control code words. From this set the encoder selects the error control codeword that best meets the constrained sequence goals of the system. The decoding structure avoids the problem of error propagation during constrained sequence decoding by performing error correction before removing the effects of the constrained sequence code. Power spectra of encoded sequences generated by a hardware implementation have a null at 0 Hz, confirming that these coded sequences are dc-free. Bit error rate simulations demonstrate the superior performance of this combined error control and constrained sequence code on a dc-constrained noisy channel.

Acknowledgements

I would like to thank the University of Alberta and TRLabs for giving me an exciting place to study that I always found challenging.

Thanks to Pranavi Anand, Tony Rapley and Shreeram Sigdel for being there through all of our classes.

Thanks to Yan Xin and Fengqin Zhai for always being able to help me solve my toughest problems, and locate hard to find papers.

Thanks to Steve Drake and Rick McGregor for letting me use the VLSI labs, oscilloscopes, power supplies, logic analyzers and many other little parts to make my project work.

Thanks to Marco Castellon for your VHDL expertise. Without you I might still be in the lab to this day.

Thanks to Dr. Witold Krzymień. You became a good friend and mentor and it was nice to know you were in my corner.

Thanks to Dr. Wayne Grover. You may not know this but I learned a surprising amount of communication theory from you.

Thanks to my good friend Shreeram Sigdel. Thanks for all the Tim Horton's coffees, thanks for studying with me, and mostly thanks for preparing me for my Qualcomm interview.

Thanks to my brother Duane for flying into Edmonton on numerous occasions to catch Oiler's hockey games. We smashed many joysticks and didn't make it home until 3am, and I wouldn't have had it any other way. I will never forget those times. Thanks!

Thanks to my Mother and Father for staying supportive all these years I *wasn't working*! This is what I was doing with my time.

Thanks to my best friend and wife Valerie. We spent almost four years walking over the windy high level bridge (even in –40°C), and we continued to support each other every day. When I had some low times and lost my confidence, you were there to remind me that I am worth something. Thank you for all of your support.

My biggest and most sincere thanks goes out to my supervisor Dr. Ivan Fair. You took a chance on me as a graduate student when I am certain no one else would, and you never gave up on me. You gave me skills and tools that I use everyday, taught me new ways to envision problems, and had the patience to weather my sometimes relentless questions.

Everyone needs a supervisor like you!

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AddSW	Add Source Word
AddCW	Add Code Word
CS	Constrained Sequence
CW	Code Word
DO	Digital Oscilloscope
EC	Error Correction / Error Correcting
ESD	Energy Spectral Density
LB	Last Bit
LC	Line Coding
LSb	Least Significant bit
MSb	Most Significant bit
MSW	Minimum Squared Weight
PSD	Power Spectral Density
RDS	Running Digital Sum
SW	Source Word
SA	Spectrum Analyzer
WRDS	Word-end Running Digital Sum

1. Introduction

The objective of a communication system is to transfer information from a source to a destination with as much accuracy as possible. While errors will inevitably occur, modern digital communication systems typically package their data in an intelligent manner to reduce these errors. As such, both error control coding and constrained sequence coding are a form of intelligent packaging used to deliver information from a source to a destination with a minimum number of errors [1,2,3].

For numerous reasons errors can occur during transmission. Error control coding allows for the detection and often correction of these errors by encoding source data in a redundant manner. Constrained sequence coding, also known as line coding [4,5], is an alternate encoding method that pre-conditions source data before transmission in an attempt to prevent errors from happening in the first place.

Both of these coding techniques aim to achieve the same goal of error free transmission. As a result, a logical advancement is to try and combine the two approaches to extract the benefits from both coding techniques [6-11]. Simple configurations involve using both codes in a concatenated fashion where the original source data is encoded with an error control code first, called the outer code, followed by a constrained sequence code, called the inner code. This simple arrangement aspires to avoid errors as a result of the inner constrained sequence code, with the expectation that the outer error control code can correct any errors that do occur.

A problem known as error propagation occurs with this setup when the constrained sequence decoder encounters errors that inevitably occur during transmission. That is, random bit errors from the channel are increased during constrained sequence decoding since the constrained sequence decoder has no means of handling these errors. As a result, the outer error control code must be powerful enough to correct the original errors as well as the additional errors introduced by the constrained sequence decoder. If the number of errors exceed the error correcting ability of the error control code, then the error control decoder may also introduce additional errors into the decoded sequence.

Since this problem results from the fact that the constrained sequence decoder cannot adequately handle errors, common sense would mandate that any errors be corrected before removing the constrained sequence code. However this is not easily accomplished since the order of decoding operations would no longer be the inverse of the encoding operations. As a result this problem has lead many channel coding researchers to investigate ways to combine both coding techniques into one monolithic code. With a combined code data could be simultaneously pre-conditioned for the channel to try and prevent errors from occurring, and as well, when errors do occur on the channel, they could be error corrected first before constrained sequence decoding, thus avoiding error propagation. This thesis introduces a new combined error control and constrained sequence code based on linear block codes. The new code introduced is analyzed, simulated, and implemented in FPGA circuits for proof of concept. The coding technique presented in this thesis can be used in fiber optic back planes which traditionally rely on CS coding alone, or even in CD and DVD formats such as the new Blu-ray or HD-DVD standards, which employ both forward EC codes and dc-free CS codes when formatting data for storage on the disk.

1.1. Thesis overview

Chapter 2 presents concepts of error control coding and constrained sequence coding that are used in this thesis. Chapter 3 follows by discussing general approaches that can be used to create a combined error control and constrained sequence code. Following this is a gradually evolving example that clearly details the new approach introduced in this thesis. Chapter 4 outlines the mathematics required for analyzing code performance in the time domain. The results of a computer search are also presented and recommendations are made detailing how this coding technique can be applied to other linear block codes. Chapter 5 investigates the coding scheme in the frequency domain and presents the results and analysis of the power spectral density of various configurations of this coding technique. Chapter 6 presents the FPGA hardware implementation and compares the measured time domain and frequency domain results with analytical results, while Chapter 7 considers the bit error rate performance of this combined code and compares various configurations. Finally, Chapter 8 summarizes the concept, configurations and performance of the combined coding technique as well as presents suggestions for future work. The appendix serves as a reference for calculating power spectral density in general as well as for block codes.

2. Error Control Coding and Constrained Sequence Coding Background

2.1. Digital Communication System Basics

A digital communication systems requires a **source**, which for the purpose of this thesis is considered to only output the binary symbols 0 and 1. Binary symbols will also be referred to as bits or digits depending on the context. This binary information is transmitted over a **channel** and is received by a **receiver**. The channel introduces a number of effects such as attenuation, distortion, interference and noise, which inevitably results in the receiver making errors [4,5]. The channel however is most easily modeled as simply introducing errors in the form of flipping bits from either a 0 to a 1, or vice versa. The receiver then receives the incoming data, and makes bit by bit decisions on whether or not it received a 0 or a 1. This is shown in Figure 2-1.

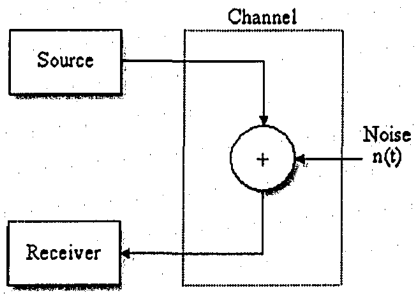


Figure 2-1: Simple digital communication system

2.2. Error Control Coding

Transmitting raw binary information over a channel provides no means of detecting errors which inevitably occur. The reason for this is because any combination of 0s and 1s on the channel is considered valid. Due to the addition of noise (that is modeled as simply flipping bits) the receiver can never really be certain if the bits it received were the ones that were sent.

A typical example would be copying a file from one computer to another. If the data was transmitted byte by byte, where one byte is 8-bits, all $2^8 = 256$ possible bit combinations of 0 and 1 would be considered valid. Thus if the source computer transmits 00000000 and the destination computer receives 00000001 (as a result of channel noise), the destination computer would have to assume that this was the data transmitted and the file would now contain errors.

The solution then is to reduce the number bytes considered valid. For instance, if out of the 256 possible bit combinations only 00000000 and 11111111 were considered valid, the receiver would now have a way of detecting errors on the channel. The destination computer would now know that byte 00000001 is invalid (**error detection**), and furthermore could infer that the intended byte was 00000000 (**error correction**). In fact, in this specific case, the channel can add as many as three bit errors per byte, such as 00000111, and the destination computer could still infer that this byte was 00000000. (Note that four bit errors such as 00001111 would result in the receiver being unable to determine the intended byte, and five or more bit errors such as 00011111 would result in the receiver mistakenly inferring that 11111111 was the intended byte). This ability to detect and correct errors comes at the expense of increased redundancy. With only two valid bytes available for transmission only $log_2(2) = 1$ bit of information is transmitted with each byte. Therefore in order to transmit the data byte 10101010, the source computer would need to transmit the eight bytes 11111111, 00000000, 11111111, 00000000, 11111111, 00000000, 11111111, 00000000. Thus the source computer must transmit eight times as much information, which results in a redundancy of 800% or an efficiency of 12.5%, to send one byte of data [1,2,3]. As a result this system can be considered as emitting 1-bit **source words** (SW) that are mapped to 8-bit **code words** (CW). This is an example of **error control** (EC) coding and is shown in Figure 2-2.

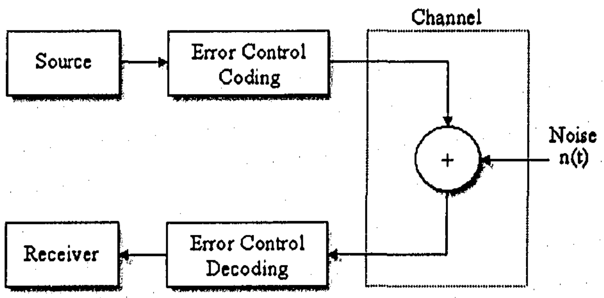


Figure 2-2: Digital communication system with EC coding

Redundancy is an essential part of EC coding since the more redundancy added the more errors that can be corrected. For example, if instead of a 1:8 mapping a 1:16 or 1:32 mapping was used, the destination computer could correct as many as seven or fifteen bit errors per CW respectively. Conversely, the more redundancy added the more inefficient the system becomes. This is known as the **coding tradeoff**.

There is also no restriction that requires SWs to be 1-bit. As will be seen in this thesis, mappings such as 4-bit SWs to 7-bit CWs, 11-bit SWs to 15-bit CWs and higher can be used. This mapping can be increased to any level desired, and as stated in Shannon's work, we can drive our error rates as low as we choose [12]. In either case, the fundamental concept of EC coding is that more bits are being sent on the channel then is needed to convey the original message, but this is being done in order to detect errors, and in many cases correct errors.

The previous example demonstrates the fundamental concepts and motivations of EC coding. The following sections introduce terms and equations required for analyzing and comparing various forms of EC coding.

2.3. Goals of Error Control Coding

Goals of EC coding include:

1. Detect and/or correct as many bit errors as possible
2. High efficiency
3. Simple scheme
4. Reduce error extension (to be defined)

Some of these goals are contradictory, since to improve efficiency usually means reducing the error controlling ability and vice versa. On the other hand, attempting to improve both would be at the cost of increased encoding and decoding complexity.

2.4. Block Codes

The example presented in Section 2.2 broadly illustrates a simple **block code.** In block codes the source data is grouped in sequential blocks of k bits defined as a **source word** (SW). Each k bit block is then encoded into an n bit block defined as a **code word** (CW), where n is larger than k [1,2,3]. This is often referred to as SW to CW mapping. The resultant code is called an (n,k) block code and thus in the example above, the code would be called an (8,1) block code.

The extra $(n-k)$ bits added are called **parity check bits.** This is the redundant information added to the source data that allows for error detection and/or correction. If the CW is constructed in such a manner that the SW appears at the beginning of the CW, it is said to be a **systematic code.** Figure 2-3 shows a systematic CW.

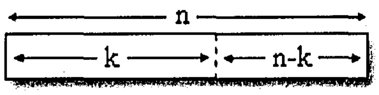


Figure 2-3: Systematic CW

As shown in Figure 2-4, the example in Section 2.2 used systematic CWs since the original SW appears at the beginning of the CW.

0	00000000
1	11111111

Figure 2-4: Mapping of the 1:8 code

2.5. Parity Check Codes

The redundant information that is added to the source data is usually done in a logical fashion. The goal is to have each CW unique, and different from all other CWs by the greatest amount possible. One way to do this is with **parity check codes.**

It is convenient to represent an (n,k) block code in matrix form as a row vector whose elements are the code symbols (bits). The original SW in vector form can be represented as d, and the CW in vector form as c as shown in Equation 2.1.

$$\begin{aligned} d &= [d_1, d_2, \ldots d_k] \\ c &= [c_1, c_2, \ldots c_k, \ldots c_n] \end{aligned} \tag{2.1}$$

As described in Section 2.4, in a systematic code the first k bits of c would equal the k bits of d. The remaining $n-k$ parity check bits are created through addition over a finite field of length 2, known as GF(2), which is also defined as modulo-2 addition, or XORing in programming notation. Let $m = n-k$ and let the systematic CW vector c be created as follows:

$$\begin{aligned} & c1 = d1 \\ & c2 = d2 \\ & \vdots \\ & ck = dk \\ & c_{k+1} = p_{11}d_1 + p_{12}d_2 + \ldots + p_{1k}d_k \\ & c_{k+2} = p_{21}d_1 + p_{22}d_2 + \ldots + p_{2k}d_k \\ & \vdots \\ & c_{k+m} = p_{m1}d_1 + p_{m2}d_2 + \ldots + p_{mk}d_k \end{aligned} \tag{2.2}$$

The coefficients p_{ij} can be collected into a **binary parity matrix** P that describes which SW bits are used to form the parity check bits. This logical mapping allows for 2^k unique SWs to be mapped to 2^k unique CWs out of a possible 2^n words. The SW to CW mapping can also be pre-computed and stored in a large **look-up table** (LUT) in computer memory, in which case each SW is treated as an index into the LUT, and the output is the CW.

2.6. Generator Matrix

Using a LUT to assign SWs to CWs becomes unmanageable as the CW size grows and the number of possible SWs increases. This is because memory size and access time start to become a factor. An alternative is to represent the above operation in matrix form as shown in Equation 2.3 and 2.4.

$$c = dG = [d_1, d_2, \ldots, d_k] \begin{bmatrix} 1 & 0 & \ldots & 0 & p_{11} & p_{21} & \cdots & p_{m1} \\ 0 & 1 & \ldots & 0 & p_{12} & p_{22} & \cdots & p_{m2} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & 1 & p_{1k} & p_{2k} & \cdots & p_{mk} \end{bmatrix} \tag{2.3}$$

$$c = dG \quad \text{where} \quad G = \begin{bmatrix} I_k & P^T \end{bmatrix} \tag{2.4}$$

Here G is known as the **generator matrix** and the CWs can now be generated on the fly using Equation 2.4. This provides a memory saving since only the generator matrix needs to be stored instead of all the CWs. This comes at the cost of added complexity in the form of matrix multiplications. For instance, the following could be a generator matrix with $k = 4$:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \quad \text{where} \quad I_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Using Equation 2.3 with generator matrix G from Equation 2.5, Table 2.1 shows all possible systematic CWs. Notice that the SW is left intact (bolded) inside the CW.

Table 2-1: Systematic CWs formed by Equations 2.3 to 2.5

Index	SW	CW	Index	SW	CW
0	0000	**0000**000	8	1000	**1000**101
1	0001	**0001**011	9	1001	**1001**110
2	0010	**0010**110	10	1010	**1010**011
3	0011	**0011**101	11	1011	**1011**000
4	0100	**0100**111	12	1100	**1100**010
5	0101	**0101**100	13	1101	**1101**001
6	0110	**0110**001	14	1110	**1110**100
7	0111	**0111**010	15	1111	**1111**111

2.7. Parity Check Matrix

Decoding is done by comparing the incoming CW (which possibly has errors) with all possible CWs. The CW that is the closest match is assumed to be the one that was transmitted. It is then simply a matter of removing the $(n-k)$ parity bits to extract the SW. This is known as **maximum likelihood decoding** [2]. This was seen in the example of Section 2.2, where receiving 00000111 would result in CW 00000000 being inferred and SW 0 being decoded. Conversely receiving 00011111 would result in CW 11111111 being inferred and SW 1 being decoded.

As the number of CWs increase however, this comparison operation can become too time consuming. A more elegant approach is to use the **Parity Check Matrix** H. Since G can be considered to create a k-dimensional sub-space of CW vectors, its **dual space** is of dimension $(n-k)$ and can be generated by $(n-k)$ linearly independent vectors that are all orthogonal to the subspace generated by the generator matrix. These vectors can be combined into an $(n-k)$ by n matrix H, such that

$$GH^T = 0 \tag{2.6}$$

The parity check matrix H can be formed in a manner similar to how G is formed using the parity matrix P.

$$G = [I_k \; P^T] \tag{2.7}$$

$$H = [P \; I_{n-k}] \tag{2.8}$$

such that

$$GH^T = \begin{bmatrix} I_k & P^T \end{bmatrix} \begin{bmatrix} P^T \\ I_{n-k} \end{bmatrix} = P^T \oplus P^T = 0 \tag{2.9}$$

2.8. Syndrome Decoding

Since every CW generated by *G* is orthogonal to all vectors in *H*, then $cH^T = 0$, i.e. the result is the all zero vector. Furthermore a received CW *r* that has been corrupted with errors such that it is no longer a valid CW will no longer be orthogonal and $rH^T \neq 0$, i.e. the result will not be the all zero vector.

Denoting *r* as the 1 by *n* received vector that results from sending the code vector *c* over a noisy channel, and considering the case of a single error in the i^{th} position, then

$$r = c + e \quad \text{where} \quad e = [\,0\;0\;0\;1\;0\;0\;0\,] \tag{2.10}$$

Since the channel is modeled as simply flipping a bit, this can be represented in matrix form as shown in Equation 2.10, where an error vector *e* is added to the original CW over GF(2). Next, evaluating rH^T yields

$$rH^T = (c+e)H^T = cH^T + eH^T = 0 + eH^T = eH^T = s \tag{2.11}$$

The 1 by *m* vector *s* is called the **syndrome** of the received vector *r*. A syndrome that is all zero indicates that *r* is a code vector and is presumably correct. A non-zero syndrome indicates that an error occurred.

Using the syndrome s and noting that eH^T is the i^{th} row of H^T [2], the error position can be identified by comparing s to the rows of H^T. Error correction can now be done simply by flipping the i^{th} bit in the received CW. Decoding by this simple comparison is called **syndrome decoding**.

2.9. Hamming bound

With syndrome decoding an (n,k) linear block code can correct up to t errors per CW if n and k satisfy the following Hamming bound.

$$2^{n-k} \geq \sum_{i=0}^{t} \binom{n}{i} \tag{2.12}$$

where

$$\binom{n}{i} = \frac{n!}{i!(n-i)!} \tag{2.13}$$

A block code for which the equality holds is known as a perfect code. Note that satisfying the Hamming bound is necessary but not sufficient for the construction of a t error correcting parity check code.

2.10. Hamming Distance

The **Hamming distance** $d(c_i, c_j)$ between two CWs c_i and c_j from the same code is defined as the number of positions in which their elements differ. The **Hamming weight** $w(c_i)$ of a CW c_i is defined as the number of 1s in c_i. Thus the Hamming weight of c_i is the Hamming distance between c_i and 0 (the all-zero CW), that is

$$w(c_i) = d(c_i, 0) \tag{2.14}$$

Similarly the Hamming distance can be rewritten in terms of Hamming weight as

$$d(c_i, c_j) = w(c_i + c_j) \tag{2.15}$$

The **minimum distance** $d_{\min}$ of a linear block code is defined as the smallest Hamming distance between any pair of CWs in the code. An (n,k) linear block code can correct up to t errors if and only if

$$d_{\min} \geq 2t + 1 \tag{2.16}$$

2.11. Linear Code Properties

Almost all useful block codes possess the property of linearity. It is defined as

1. Multiplying a CW by a valid scalar produces another CW
2. Adding two CWs together also produces another CW.

When dealing with binary symbols a valid scalar is 0 or 1, and hence multiplication by 0 produces the all-zero CW, and multiplication by 1 leaves the CW unchanged. Addition with binary symbols over GF(2) means that adding a CW with itself will result in the all zero CW, and the addition of two different CWs will produce another valid CW. For instance if c_i and c_j are two CWs, then the addition $c_i + c_j$ must also be a CW. Since $cH^T = 0$ then $(c_i + c_j)H^T = 0$. As well since $c_i + c_j \neq 0$, $c_i H^T = 0$, and $c_j H^T = 0$, then $c_i + c_j$ must also be a valid CW.

2.12. Hamming Codes

Hamming codes were the first linear block codes published on error correction. They have a minimum distance $d_{\min} = 3$, and thus they have the ability to correct single bit errors per CW. They are also perfect codes by definition of the Hamming bound and they belong to a subset of linear block codes known as **cyclic codes**. This subset of codes is popular since encoding and decoding can be done in a simple fashion.

2.13. Cyclic Codes

Cyclic codes are a subset of linear block codes. In addition to the properties of linearity, they contain an additional property that if they are shifted cyclically (circularly shifted or rotated in programming notation), the result is also a CW. For instance, Table 2-1 above is showing a (7,4) Hamming code, where it can be seen that CW1 is $[0,0,0,1,0,1,1]$, which when shifted left forms CW2, CW5 and then CW11. Further cyclic shifting, which involves wrapping the most significant bits back to the least significant position, gives CW6, CW12 and then CW8 before forming CW1 again. This cyclic nature of the CWs can be exploited to form simpler encoding and decoding configurations versus the matrix approaches used to this point. This effect is also easier to demonstrate when error control codes are represented using polynomials. The following subsections demonstrate how binary digits can be represented as polynomials, and how simple algebraic operations such as addition, subtraction, multiplication and division can be performed. Note that since the binary alphabet only has two symbols, all algebraic operations are performed in GF(2). As a result addition is the same as subtraction, and this operation can be implemented with XOR gates in hardware. Similarly, multiplication and division can be accomplished through shifting, which is done using feed-back and feed-forward shift registers respectively.

2.13.1. Polynomial representation of codes

$$101101 \;=\; 1x^5+0x^4+1x^3+1x^2+0x^1+1x^0 \;=\; x^5+x^3+x^2+1 \;=\; d(x) \quad (2.17)$$

2.13.2. Addition and Subtraction

$$\begin{aligned}
r(x) &= c(x) + c(x) \\
r(x) &= \operatorname{mod}\left(\left(x^5 + x^3 + x^2 + 1\right) + \left(x^5 + x^3 + x^2 + 1\right), 2\right) \\
r(x) &= \operatorname{mod}\left(\left(2x^5 + 2x^3 + 2x^2 + 2\right), 2\right) = 0
\end{aligned} \tag{2.18}$$

2.13.3. Multiplication

$$\begin{aligned}
c(x) &= g(x)d(x) \\
c(x) &= x^2\left(x^5 + x^3 + x^2 + 1\right) = x^7 + x^5 + x^4 + x^2
\end{aligned} \tag{2.19}$$

2.13.4. Division

$$d(x)/g(x) \quad = \quad q(x) \quad \textit{and} \quad r(x) \tag{2.20}$$

$$\begin{array}{r l}
 & \phantom{x^6 + {}}x^3 + x \leftarrow q(x) \\ \cline{2-2}
x^3 + x + 1 \,\big) & x^6 \phantom{{}+ x^4} + x^3 \\
 & \underline{x^6 + x^4 + x^3} \\
 & \phantom{x^6 + {}}x^4 \\
 & \phantom{x^6 + {}}\underline{x^4 + x^2 + x} \\
 & \phantom{x^6 + x^4 + {}}x^2 + x \leftarrow r(x)
\end{array}
\qquad\qquad
\begin{array}{r l}
 & 1010 \leftarrow q(x) \\ \cline{2-2}
1011 \,\big) & 1001000 \\
 & \underline{1011} \\
 & 1000 \\
 & \underline{1011} \\
 & 110 \leftarrow r(x)
\end{array}$$

With division there is a quotient $q(x)$ and a remainder $r(x)$.

2.14. Encoding of Cyclic Codes

To encode cyclic codes first choose a **generator polynomial** [2] $g(x)$ such that

1. It is of degree $(n-k)$, in order to have $n-k$ redundant symbols.
2. It is a factor of $x^n + 1$, to ensure that the code is cyclic.

The simplest way to create CWs is to multiply the SW $i(x)$ of length k by the generator polynomial $g(x)$, i.e. $c(x)=i(x)g(x)$. To match the systematic form however the data bits must remain intact, and the parity bits must be simply concatenated. Thus the first step is to shift the data bits $i(x)$ up by $(n-k)$, i.e. $i(x)x^{n-k}$. Then since it still needs to be a multiple of $g(x)$, $i(x)x^{n-k}$ can be divided by $g(x)$ to see if the remainder $r(x)$ is zero. If it is not zero then $i(x)x^{n-k}$ must be modified so that it is a multiple of $g(x)$. The simplest way to do this is to subtract the non-zero remainder from the original dividend [2]. This operation is shown in Equation 2.21 noting that addition and subtraction are the same operation in GF(2).

This procedure works since this is an application of Euclidean division which states that the $dividend = (quotient)(divisor) + remainder$. Since $i(x)$ was pre-multiplied by x^{n-k}, the degree of the remainder will always be less than the degree of the divisor and the dividend terms will all be greater than or equal to $(n-k)$. Consequently adding the divisor to the quotient will not modify any terms in $i(x)$, maintaining systematic form. As well, choosing the divisor to be the generator polynomial $g(x)$ ensures that the result is always a CW. In other words, the polynomial representation of a CW can be represented by

$$c(x)=i(x)x^{n-k}+R_{g(x)}\left[i(x)x^{n-k}\right] \tag{2.21}$$

where $R_{g(x)}[]$ represents taking the remainder after division through $g(x)$. All of this work is neatly represented by the circuit in Figure 2-5 which consists only of shift registers and XOR gates.

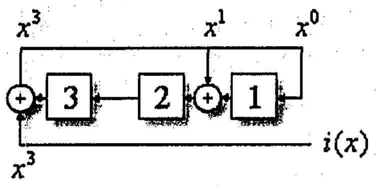


Figure 2-5: Cyclic code encoder

This circuit represents a simultaneous multiplication by x^3, and a division by x^3+x+1. In order to build a systematic CW the registers are initially cleared, then the $i(x)$ bits are shifted through one at a time. After all k bits have been shifted in, the registers will contain the remainder. Thus using the circuit shown in Figure 2-5 to encode $i(x)=x^3+1$, or in binary 1001, with the generator polynomial $g(x)=x^3+x+1$, is identical to the process shown above in Equation 2.20 under polynomial division. For example if $k=4$ and $n=7$, $i(x)$ is first multiplied by $x^{7\text{-}4}$ to yield x^6+x^3, and then this dividend is divided by $g(x)$. The final step is to append the remainder $r(x)=x^2+x$ to form the CW $c(x)=x^6+x^3+x^2+x$, or in binary 1001110.

Inspection of the CW $c(x)$ created above with the generator polynomial $g(x)=x^3+x+1$ shows that it is the same CW created with the generator matrix G in Equation 2.5, i.e. CW9 in Table 2-1. Furthermore this generator polynomial can be used to construct all the CWs shown in Table 2-1. Thus while the explanation of cyclic codes is more difficult, the implementation is simpler and these encoders can be easily implemented with digital circuits using very basic logic components.

2.15. Decoding of Cyclic Codes

The received signal from the channel is a linear combination of the CW $c(x)$ plus an error vector $e(x)$, denoted $y(x)=c(x)+e(x)$. Since all CWs are multiples of $g(x)$, testing if the received word is valid is as simple as dividing it by $g(x)$ and checking if the remainder $r(x)$ is zero. If it is, then it can be assumed that there are no errors. If $r(x)$ is not zero then there was at least one or more errors present. This remainder is once again called the syndrome. Equation 2.22 demonstrates how the received vector $y(x)$ is divided by the generator polynomial $g(x)$ and any non-zero remainder is a result of a non-zero error vector $e(x)$. If the error vector $e(x)$ was zero then the syndrome $s(x)$ will also be zero, and it can be assumed that there were no errors.

$$s(x) = R_{g(x)}[y(x)] = R_{g(x)}[c(x) + e(x)] = 0 + R_{g(x)}[e(x)] \tag{2.22}$$

This syndrome can be found through division with the encoder circuit shown in Figure 2-5. A benefit of cyclic codes is that it is not necessary to keep track of all possible syndromes as it was with syndrome decoding. In fact, only the syndrome that represents an error vector in the most significant bit (MSb) position needs to be tracked. This is because a left cyclic shift of the error pattern results in a syndrome that changes by the amount equivalent to $R_{g(x)}\left[x \bullet s(x)\right]$ [2]. What this means is, if a non-zero syndrome is present in the shift registers, each clock of the division circuit is equivalent to a left cyclic shift of the error pattern. For instance, assume a corrupted word $y(x)$ has an error in its 5th MSb, i.e. $e(x) = [0,0,1,0,0,0,0]$. When it is clocked into the divisor circuit a syndrome will be formed that indicates this error. Now a further clocking of the divisor circuit will result in a left cyclic shift of the error vector to $e(x) = [0,1,0,0,0,0,0]$, and thus, a syndrome will be formed indicating an error in the 6th MSb. One further clocking of this circuit then will produce a syndrome indicating an error in the 7th MSb, i.e. $e(x) = [1,0,0,0,0,0,0]$. Thus, as long as the circuit is monitoring for *this* syndrome it can correct the flipped bit. To check for an error in the MSb, the syndrome to monitor for when $g(x) = [1,0,1,1]$ is shown in Equation 2.23.

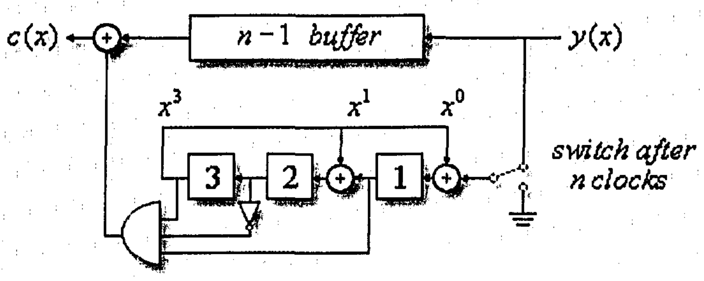
$$s(x) = R_{g(x)}\left[1,0,0,0,0,0,0\right] = \left[1,0,1\right] \tag{2.23}$$

Therefore it will take exactly n clocks to form the original syndrome, and then $n-1$ more clocks to cycle through all the remaining $n-1$ possible error vectors [2].

The practical significance of this result is that the circuit only needs to look for a single error pattern and continue clocking the division circuit until the syndrome for this error pattern is matched. Once this happens it simply corrects the erroneous bit by inverting it. On the other hand if the syndrome is never matched it can be assumed that there were no errors. Decoders using this principle are called **Meggitt Decoders** [2].

The circuit in Figure 2-6 shows how the above process can be achieved using very simple digital circuits. For example, taking the CW $c(x) = [1,0,0,1,1,1,0]$, and flipping the 5th MSb as described gives $y(x) = [1,0,1,1,1,1,0]$. Figure 2-6 shows that for the first $n = 7$ clocks the corrupted $y(x)$ CW is fed in, which forms the original syndrome $s(x) = [1,1,0]$ as shown at $clk = 7$. As the last bit of $y(x)$ is being fed in the first bit of $y(x)$ is leaving the buffer of length $n-1$, where it is XORed with the syndrome $s(x)$ through the boolean circuit. Had there been an error in the MSb of $y(x)$ the syndrome would have been 101 causing the output of the AND gate to be high, thus flipping or correcting this erroneous bit.

Hence it can be seen from Figure 2-6 that on the 9th clock $(clk = 9)$, the syndrome $s(x) = [1,0,1]$ has been matched and the erroneous bit has been corrected. The original CW $c(x)$ is then formed by ignoring the first n-1 bits that leave the buffer and keeping the last n bits. Furthermore since this is a systematic code the original SW $d(x)$ is recovered by removing the last n-k bits. Clearly this circuit is simpler than decoding with the parity check matrix.



clk	i(x)	3	2	1	c(x)
		0	0	0	
1	1	0	0	1	
2	0	0	1	0	
3	1	1	0	1	
4	1	0	0	0	
5	1	0	0	1	
6	1	0	1	1	
7	0	1	1	0	1
8		1	1	1	0
9		1	0	1	0
10		0	0	1	1
11		0	1	0	1
12		1	0	0	1
13		0	1	1	0

Figure 2-6: Meggitt decoder and operation correcting the 5th MSb

2.16. Advantages of Error Control Coding

Before the use of error control coding the only way to improve the performance of a communication system was to increase the transmitted power, or increase the amount of bandwidth used, both of which come with a never ending financial penalty. However EC coding allows for the clever encoding of data which results in an improvement in performance using the current allotted bandwidth and transmit power, at the cost of added complexity. Thus EC codes do an excellent job of bringing the bit error rate down to any level desired, as long as the system can tolerate the added redundancy.

2.17. Drawbacks of Error Control Coding

The drawback of EC coding is that it is impossible to detect and correct all errors. Since this is a stochastic problem there is always some chance that errors will slip through the system undetected. In addition, when the number of errors in a received word exceed the error correcting abilities of the EC code, the error correction process may in fact introduce additional errors into the decoded sequence. This is called **error extension** and it occurs because the EC code is fooled by the excessive errors and incorrectly flips bits that did not have errors to begin with, consequently introducing additional errors into the system.

2.18. Constrained Sequence Coding

When digital pulse signals are conveyed over a practical transmission medium, certain bit patterns can be more prone to errors than others [13]. Knowing some characteristics about the channel and the type of corruption the signals will undergo allows for the information to be encoded prior to transmission to avoid transmitting those sequences that are likely to be decoded in error. These types of encoded sequences are being constrained and thus this type of coding is called **Constrained Sequence** (CS) coding. It is also known as **Line Coding** (LC) in digital transmission systems [4] and **Recording Coding** (RC) in digital storage systems. These terms will be used interchangeably.

2.19. Line Coding Basics

When a binary data stream is transmitted over a channel it is first converted into an electrical or light waveform through **transmission encoding** [4]. One of the simplest formats used is **on-off coding** also known as **unipolar coding** shown in Figure 2-7. Here a logic 1 is represented by a positive voltage and a logic 0 is represented by no voltage. This can also be interpreted as assigning a positive square pulse shape $p(t)$ to a logic 1, and assigning no pulse to a logic 0. This is the most prevalent signaling arrangement used in fiber optics as the laser is either on or off [14,15].

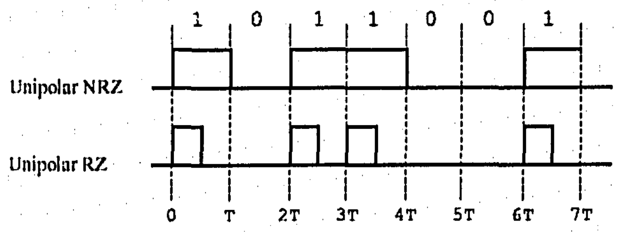


Figure 2-7: On-Off Signaling Formats

If the voltage is constant over the symbol period T, then the format is called **Non-Return Zero** (NRZ). If however there are too many like valued symbols in a row it is difficult to determine the end of one symbol and the start of the next. Thus some systems use **Return Zero** (RZ) signaling format to force the voltage to return to zero before the end of each symbol period. This comes at the cost of less energy per symbol, and thus a decreased signal to noise ratio. Both formats are shown above in Figure 2-7.

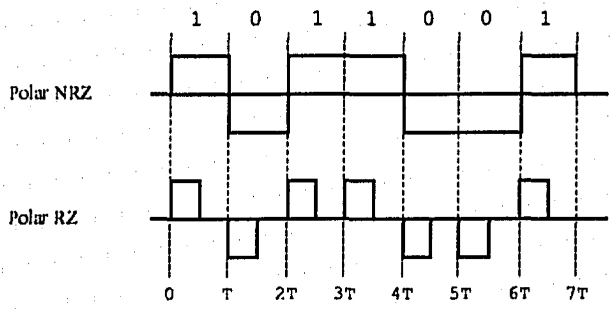


Figure 2-8: Polar Signaling Formats

Another commonly used format is **polar coding**, also known as **bipolar coding** shown in Figure 2-8. Following from the last example, a logic 1 is transmitted by the square pulse shape $p(t)$ and a logic 0 is transmitted by the inverted square pulse shape $-p(t)$. This is the most power efficient scheme since it requires the least power to achieve a given error probability [4].

Regardless of the transmission encoding used the serial bit stream requires periodic maintenance as it travels on the channel in order to combat the accumulation of noise and signal distortion. This requires that the signal be received, decoded, and regenerated for further transmission. Most often the receivers used are AC coupled as they are easier to design and capable of better performance [16,15]. The high-impedance amplifier in Figure 2-9 is often used in fiber optic transmission systems because it offers the lowest noise level and hence the highest detection sensitivity [16]. However, because of the high load impedance the frequency response of the amplifier is limited by the RC time constant at the input [15].

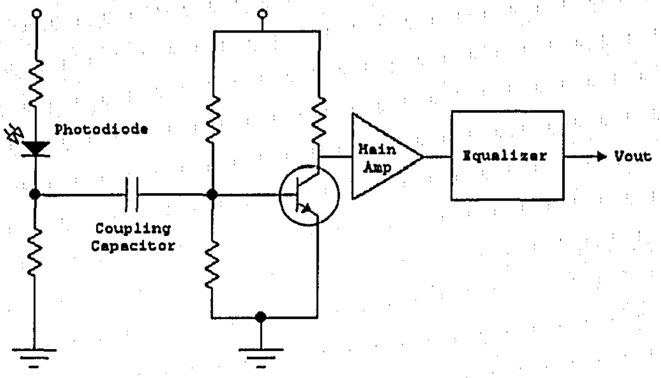


Figure 2-9: Simple high-impedance preamplifier design using a bipolar transistor

AC coupling in these receivers has the effect of blocking the average dc value of the signal. As a result a 50% duty cycle unipolar square wave will appear as a 50% duty cycle polar square wave on the other side of the capacitor as shown in Figure 2-10.

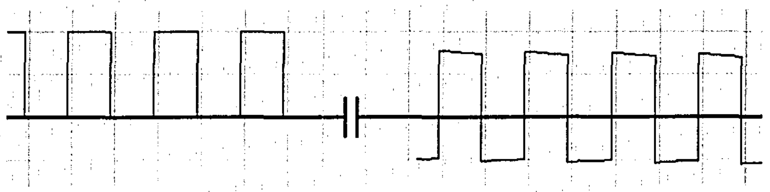


Figure 2-10: 50% duty cycle square wave through a dc blocking capacitor

There is an additional effect caused by the large RC time constant. The high impedance receiver tends to integrate the detected signal as shown in Figure 2-11. This leads to a variation of the midpoint known as **baseline wander** [16].

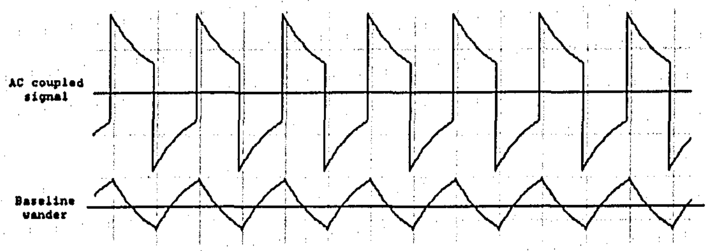


Figure 2-11: AC coupled signal and baseline wander

When a series of like valued symbols arrive in succession, such as more 1s than 0s, the transmission is said to be unbalanced. This causes the integration effect to become more severe as shown in Figure 2-12 because the signal decays towards zero [15,16,17]. Note that the integration effect has been exaggerated for demonstration purposes. Nevertheless it is clear that the AC coupled signal in Figure 2-12 would have an increased probability of error in the presence of noise. Thus in order to minimize baseline wander the output needs to be balanced. That is, there must be an equal number of 1s and 0s on average in the output sequence.

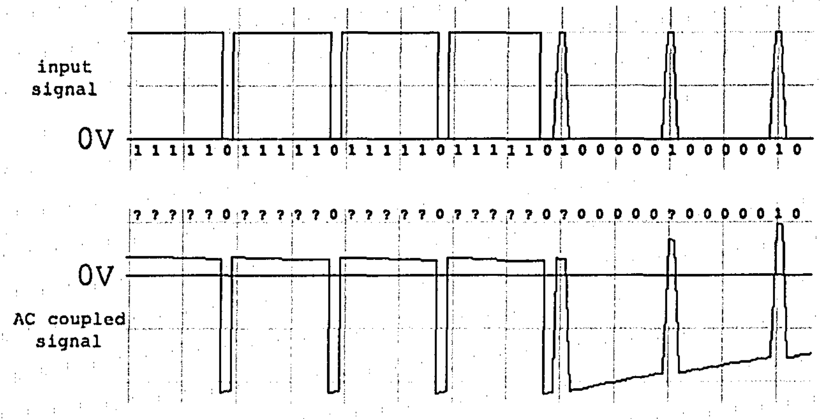


Figure 2-12: AC coupling of an unbalanced sequence

The receiver also must know the duration of a symbol interval (length of a bit), in order to determine when to make a decision regarding the value of each symbol. To avoid transmission of a separate clock signal receivers typically derive a clock from transitions in the received data stream. This is done by having the oscillator in the receiver lock on to level shifts in the received signal. If there is a long series of logic 1s or 0s in a row the receiver's oscillator frequency may drift and become unsynchronized. When this occurs the receiver can lose track of where it is supposed to sample the transmitted data, as shown in Figure 2-13. This is a more serious problem than bits simply being corrupted by the channel [1] as now the system is completely missing data.

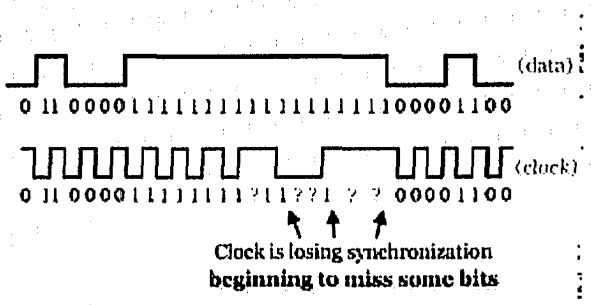


Figure 2-13: Clock recovery circuit losing synchronization

Another reason to require adequate transitions is to reduce **timing jitter** [5]. Random deviations of the incoming pulses from their ideal locations are always present even in the most sophisticated systems [5]. The tuned circuits used for timing extraction are sensitive to the pulse patterns, and long strings of 0s or 1s introduce additional jitter as the oscillation slows [4,5]. As well, most data transmitted is packaged in a regular fashion known as framing. This typically consists in a set number of bits for header, payload and tail information. The framing patterns result in a periodic frame structure and these regular patterns can also cause pattern dependent jitter [4].

All of these factors collectively contribute to errors in the received signal. However when CS coding is introduced the transmission signal characteristics are improved in such a way that the regular patterns get randomized, and the long strings get broken up as the transition density is increased. Consequently the clock recovery in the receiving terminal is stabilized and jitter is reduced, decreasing the inter symbol interference (ISI) and improving the accuracy of data reception.

2.20. Line Coding Goals

Goals of line coding include:

1. Adequate transitions for timing recovery
2. Balanced transmission to reduce dc drift
3. Low jitter
4. High efficiency, i.e. low redundancy

5. Simple implementation
6. Low error extension
7. Good performance regardless of the source statistics

The unpredictable nature of the digital source makes achieving the above goals challenging. For instance, if the probability of the source outputting a 0 is very high, the outgoing sequence would naturally have long runs of 0s. As a consequence the receiver would have very few transitions to lock onto. The outgoing sequence would also be heavily unbalanced which would lead to drift, jitter and ultimately errors. Thus it would be up to the CS code to modify this bit stream into a more suitable sequence for the channel, removing the long runs of like valued bits. A block diagram of a system incorporating CS coding, also known as line coding, is shown in Figure 2-14.

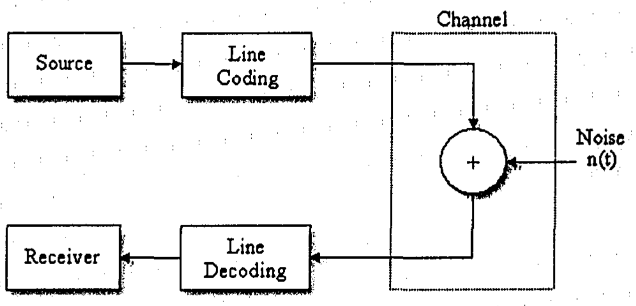


Figure 2-14: Digital communication system with CS coding

2.21. Line Coding approaches

Manchester Encoding is an almost ideal CS coding scheme in that it meets nearly all line coding goals. In this system a logic 1 is mapped to the two bit pattern 01, and a logic 0 is mapped to the complement two bit pattern 10. This guarantees at least one transition per bit as well as a completely balanced transmission [5]. Even if a source constantly emitted logic 0s the transmitted sequence would simply appear like a 50% duty cycle square wave. The drawback with this approach however is that with this 1:2 mapping the output sequence is 50% redundant.

4B/5B encoding is a line coding technique sometimes called block coding [13]. Here every four bits from the source are encoded into five bits for the channel. With $2^4 = 16$ possible SWs mapped to 16 of the $2^5 = 32$ possible CWs, a straightforward implementation results in the 5-bit CWs always having two 1s even if the source data is all 0s. This allows for clock synchronization and is more efficient than Manchester Encoding. A drawback however is that it is no longer balanced since CWs have an odd length.

5B/6B uses the same idea as 4B/5B with the improvement that there are now three 0 bits and three 1 bits in each 6-bit CW. As a result the output is dc balanced. This coding scheme also provides an added error checking capability since invalid data patterns, such as more than three 0s or three 1s in a word can be detected.

There are many similar codes such as **7B/8B, 10B/12B** and so on which map m information bits into n transmission bits and they are commonly referred to as ***mBnB*** codes where the B denotes binary signaling. They are also known as **Alphabetic Codes** where the pre-selected set of n bit CWs are called an alphabet [1,4, 5]. The mapping is typically done using look-up tables stored in ROMs. The physical limitations of ROMs impose CW length restrictions and speed (access time) restrictions. Because of this they have not gained popularity in high bit rate systems.

Scrambling is a bit-level (or pulse-level) processing applied to a digital signal just prior to transmission on the channel. The objective of scrambling is to improve clock recovery as well as randomize the bit sequence as much as possible to remove long strings of 1s or 0s. Scrambling is designed to work well regardless of source statistics and also does not affect the bandwidth requirements of the system [14,18]. Therefore the bit rate of the system is the same before and after scrambling and thus it has found great popularity in fiber optic systems.

Scrambling can be described with the same polynomial representation used to describe cyclic EC codes. For example, to scramble a stream of source data $s(x)$ with the **scrambling polynomial** $d(x)$ using **Self Synchronous Scrambling** (SSS), the data stream is multiplied by the degree of the scrambling polynomial and divided by $d(x)$. The scrambled sequence is the quotient [18] as shown in Equation 2.24.

$$c(x) = Q_{d(x)}\left[s(x)x^d\right] \quad (2.24)$$

The scrambled sequence is formed using the same polynomial operations as were used with EC coding. For example, consider a sequence that contains a long run of zeros, i.e. $s_0(x) = 01000000001$, and scrambling polynomial $d(x) = 100101$. The scrambled sequence would be

$$c_0(x) = Q_{d(x)}\left[s_0(x)x^d\right] = 00010010111 \quad (2.25)$$

Clearly the scrambled sequence $c_0(x)$ meets our line coding goals far better than the unscrambled sequence. While scrambling can greatly increase the probability of avoiding long runs of like valued bits it does not provide the same guarantee that many block coding techniques such as *mBnB* codes provide. It can be seen that there is still situations that can cause the transmitted sequence to contain long run of 1s or 0s. For instance, if $s_1(x) = 0100101000100101010100101$, the output sequence would be

$$c_1(x) = Q_{d(x)}\left[s_1(x)x^d\right] = 010000000010000000100000 \quad (2.26)$$

Here the unscrambled sequence appears better suited to meet our constrained sequence goals then the scrambled sequence. To account for this **guided scrambling** [14] is an extension and improvement of SSS which augments the SW prior to scrambling. As a result it is capable of guiding the outcome of an otherwise normal scrambling process to produce a highly efficient balanced bit stream [14].

2.22. Advantages of Constrained Sequence Coding

CS codes do an excellent job of ensuring sufficient transitions for timing recovery and encoding the output sequence to match channel constraints [18]. In this way they can be thought of as pre-conditioning the data in a suitable fashion for transmission on a real channel. This new data sequence will then suffer less corruption than had the data been sent uncoded [14,18,19].

CS coding is often thought of as spectral shaping [20] since the channel constraints are being met by modifying the frequency domain characteristics of the transmitted signal. In essence, CS coding is a proactive approach to preventing errors from occurring in the first place.

2.23. Disadvantages of Constrained Sequence Coding

The major drawback of CS coding comes from their inability to effectively handle errors that inevitably occur. When errors are present in the received sequence error propagation results as the CS decoder incorrectly decodes the message. In many cases this error extension can affect numerous subsequent CWs [18].

2.24. Combining Error Control Coding and Constrained Sequence Coding

Since EC coding and CS coding both aim to improve the accuracy of digital communication systems, many systems utilize both approaches to extract the benefits from the two coding techniques [6]. Figure 2-15 shows the configuration that is normally used. The idea is to apply an EC code first and then use CS coding to condition the information for the channel. The hope is that the overall number of errors will be reduced, and any errors that do occur can be corrected by the EC code.

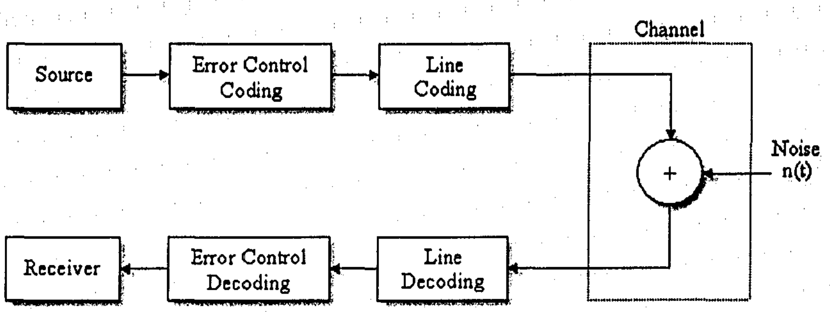


Figure 2-15: Typical configuration that incorporates EC coding and CS coding

The difficulty with this approach arises during demodulation/decoding. When errors inevitably occur the CS decoder has no means of dealing with them. As a result it incorrectly decodes the incoming sequence, in effect increasing or multiplying the number of errors [6,7,8]. This error multiplication can be thought of as turning a single error into a burst (multiple bit errors) [7]. Hence the error correcting code must be a burst correcting code even when noise in the channel is dominated by random errors [7]. In general more redundancy is needed to correct t error bursts than to correct t random errors [2]. This means that there is a greater cost in efficiency and complexity with the above approach. Therefore a logical conclusion is that errors should be corrected prior to CS decoding to avoid the error multiplication as shown in Figure 2-16.

However straightforward exchange of the decoders is impossible since the required order of operations is no longer followed. The EC decoder would not recognize any of the words arriving off of the channel since the CS encoder would have modified the EC CWs for the channel before they were transmitted. The EC decoder would then mistakenly try to correct these words unintentionally introducing more errors. Finally, since the CS decoder is now last, it would receive words that it would also not recognize (as a result of the EC decoder) and most likely introduce additional errors when it attempts to remove the effect of the line code.

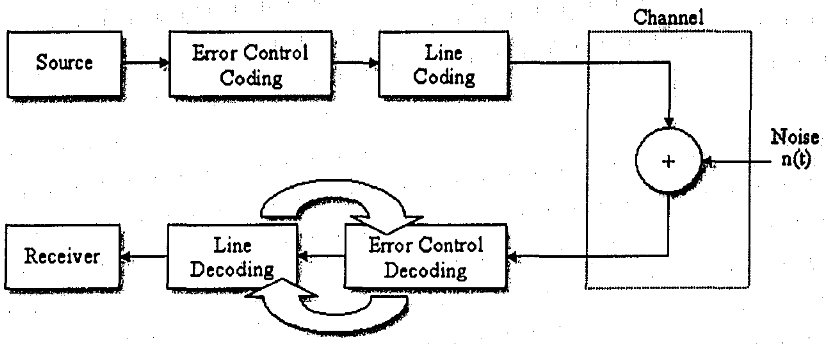


Figure 2-16: Desired configuration that incorporates EC coding and CS coding

Channel coding researchers are now investigating ways of combining the above codes into one monolithic code to avoid this error multiplication and extract the benefits of both. Several methods are similar in their approach of adding additional redundancy in the form of multimode coding, where one SW is mapped to a selection set of CWs [7], or through appending additional CS bits to an existing EC code [10], or through augmentation of EC and CS code bits [6]. These codes are sometimes called combi-codes [21] or CC-EC codes [11] and are depicted in Figure 2-17.

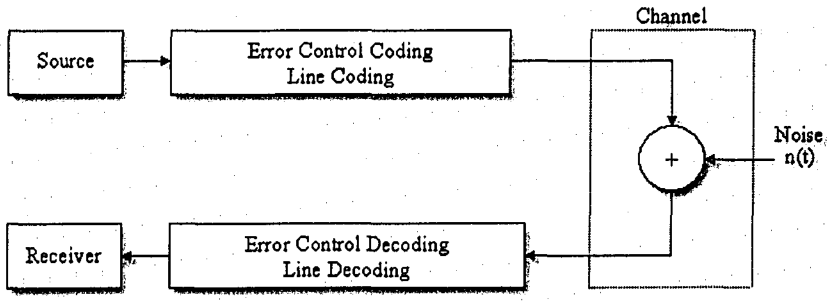


Figure 2-17: Combined approach that incorporates EC coding and CS coding

This thesis introduces a simple approach to achieving the above goal of integrating EC coding with CS coding that is easily implemented and has good performance tradeoffs. The approach uses linear cyclic block codes and the simple family of Hamming codes for proof of concept. CS coding is achieved through multimode coding and this new technique is thoroughly analyzed, simulated, and implemented in hardware using field programmable gate arrays (FPGAs).

3. Integrating EC coding and CS coding

Creating a combined code is not straightforward since the methodologies behind EC coding and CS coding are different. Error control CWs are designed to be as different from one another as possible in order to increase the probability of detecting and/or correcting the most errors. Therefore CWs such as the all-zero CW $c(x) = 0000000$ or the all-one CW $c(x) = 1111111$ are often present. These CWs have no transitions and they are completely unbalanced, but clearly it would take multiple flipped bits (many errors) to mistake one for the other. Constrained sequence CWs on the other hand are designed to give balanced output and ensure an adequate number of transitions. Thus with properties like these CWs often exist that only differ by a single bit. Consequently single bit errors can often make one CW appear as another.

In order to create a combined code one of the following two approaches can be employed. More redundancy can be added to an existing CS code, to increase the distance between the CWs in an attempt to turn it into an EC code. Alternatively, more redundancy could be added to an existing EC code in the form of restricting the CWs that are used to only those that meet the systems CS coding requirements in an attempt to turn it into a CS code.

If the latter approach is taken and a typical (n,k) error correcting block code is used, inspection of the 2^k possible CWs will show that many of them meet the CS coding goals outlined in Section 2.20. Therefore if transmission is restricted to this subset of EC CWs, and since these CWs can natively correct errors, a combined EC and CS code is obtained. This is the novel approach introduced in this thesis, which uses the EC CWs themselves as the CS code.

While the above approach qualifies as a combined EC and CS code, straightforward implementation presents some difficulties. For example, the mapping of SWs to the restricted set of CWs can be accomplished through look-up tables. However this would result in the encoder being subject to all of the same ROM limitations as *mBnB* codes, preventing its use in high bit rate systems. As well, many EC codes have

CWs that contain an odd number of bits. In these cases any individual CW will have more 1s than 0s and vice versa. Thus in a worst case situation where a source continually emits the same SW, the encoder would continually transmit the same odd length CW, resulting in a transmission that grows increasingly unbalanced. These problems need to be addressed, and the complete details for the combined EC and CS code presented in this thesis are presented through example in the next Section.

3.1. A Simple Scheme

The major flaw in the 1:1 mapping of SWs to CWs described above is that the system cannot guarantee balanced output. If however there existed the flexibility to send the CW, or the complement of the CW, this guarantee could be made. For example, examining the CWs of the (7,4) Hamming code shown in Table 2-1 shows that $c_7(x) = 0111010$ and $c_8(x) = 1000101$, $c_6(x) = 0110001$ and $c_9(x) = 1001110$, all the way to $c_0(x) = 0000000$ and $c_{15}(x) = 1111111$ are all CW complements of one other. Thus in a worst case situation where a source continually emits the same SW repeatedly, this system would remain balanced since the encoder could alternately transmit the original CW followed by its complement. That is $c_7(x) = 0111010$ which has four 1s and three 0s could be transmitted, followed by $c_8(x) = 1000101$ which has three 1s and four 0s, thus balancing the transmission. An encoder that maps one SW to a choice of more than one CW is classified as a **multimode** encoder, which was first discussed in Section 2.24. Specifically, this encoder is further classified as a **bimode** encoder since each SW is mapped to a choice of only two CWs.

This multimode encoder could be implemented without look-up tables and using the original digital logic gate encoders of Figure 2-5 by taking advantage of linear code properties of block codes. For example, from Section 2.11, any CW added to the all-zero CW will result in the original CW unchanged. However any CW added to the all-one CW will result in the CWs complement. For example, $c_7(x) = 0111010$ added with $c_0(x) = 0000000$ is $c_7(x) = 0111010$ (no change), but $c_7(x) = 0111010$ added with

$c_{15}(x) = 1111111$ is $c_8(x) = 1000101$ (every bit inverted). Thus based on this idea of CW addition, Figure 3-1 shows a possible setup of a bimode coding scheme where the encoder can choose to send the original CW or its complement.

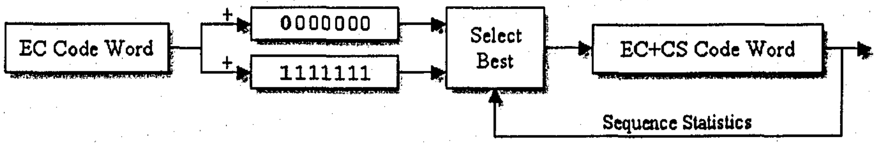


Figure 3-1: A simple EC + CS encoder using bimode coding

From the discussion above SWs are still converted to CWs in the usual fashion. However with the above system, these CWs are then added to both the all-zero and the all-one CW. This has the effect of presenting the original CW and its complement to the Select Best decision block. The decision of selecting which CW to transmit is based on feedback from CWs already transmitted on the channel. Depending on these sequence statistics, the CW that keeps the overall number of 1s and 0s sent on the channel approximately equal will be chosen. Therefore with this setup, in a worst case situation where a source continually emits the same SW, such as the all-zero SW, this system can ensure balanced transmission. These output sequences are contrasted in Figure 3-2, which shows the original all-zero sequence listed as $before_{(7,4)}$, versus the improved sequence listed as $after_{(7,3)}$. Clearly this is an improved sequence since it is balanced and there is a transition once every CW.

$before_{(7,4)}$	0000000	0000000	0000000	0000000
$after_{(7,3)}$	0000000	1111111	0000000	1111111

Figure 3-2: The effect of the bimode coding: balanced output

As shown below in Table 3-1 this setup has SW 0 mapped to two CWs 0 and 15. Likewise, SW 1 is mapped to CWs 1 and 14 and so on. Hence SWs 8 to 15 can no longer be used to represent data, or in other words, the ability to achieve balanced transmission has resulted in an additional bit of redundancy, thus modifying the (7,4) Hamming code to a (7,3) Hamming code (i.e. there are now $2^3 = 8$ possible SWs, 0 through 7 as listed in Table 3-1). Since the overall length of the code stayed the same, this is known as expurgating the code [2], i.e. one of the source bits is now acting like a parity bit. This extra parity bit is the MSb of each SW listed in Table 3-1.

Table 3-1: SW to CW mapping of this bimode (7,3) setup. Compare with Table 2-1

Index	SW	CW_A	$\overline{CW_A}$ (complement)
0	0000	$0000000_{(0)}$	$1111111_{(15)}$
1	0001	$0001011_{(1)}$	$1110100_{(14)}$
2	0010	$0010110_{(2)}$	$1101001_{(13)}$
3	0011	$0011101_{(3)}$	$1100010_{(12)}$
4	0100	$0100111_{(4)}$	$1011000_{(11)}$
5	0101	$0101100_{(5)}$	$1010011_{(10)}$
6	0110	$0110001_{(6)}$	$1001110_{(9)}$
7	0111	$0111010_{(7)}$	$1000101_{(8)}$

Decoding of these CWs is simple. Examination of Table 3-1 shows that the MSb of every CW is either a 0 or a 1 (shown in bold). This MSb is the information bit that is now acting as a parity bit. Thus the decoder can determine whether or not the received CW is the original or the complement, simply by inspecting the MSb. Thus as shown in Figure 3-3 the decoder first error corrects the received word, and then based on the value of the MSb, removes the CS coding by adding either the all-zero or all-one CW. Recovering the SW is then the simple operation of removing the parity check bits.

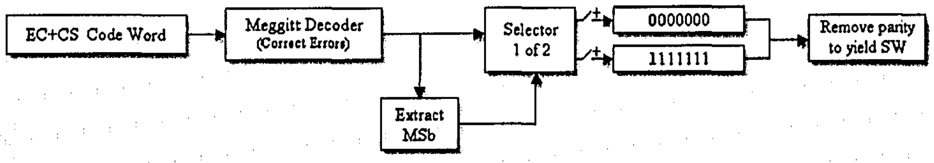


Figure 3-3: A simple EC + CS decoder using two AddCWs

There are two main advantages of this scheme. First of all the original encoder and decoder circuits which consist of simple boolean logic can still be used with only a slight modification for CS coding. Secondly, received words arriving off the channel avoid error propagation since they are error corrected first before removing the CS coding through CW addition. Thus the encoder of Figure 3-1 and decoder of Figure 3-3 demonstrate a combined EC and CS code that incorporates feedback to maintain balanced output. In this coding scheme the all-zero and all-one CWs used for CW addition are defined as **Add Code Words** (AddCWs). Therefore the encoder of Figure 3-1 and decoder of Figure 3-3 use two AddCWs.

3.2. Improved scheme

Looking once again at the output data stream $after_{(7,3)}$ in Figure 3-2 demonstrates that this setup can guarantee balanced transmission. However it also demonstrates that it cannot guarantee numerous transitions. For example, this bimode system presented above turns the all-zero sequence into an alternating sequence of seven zeros followed by seven ones. Therefore in this (7,3) example there is a transition every seven bits since $n=7$. However if the system used a larger code such as the (63,57) code expurgated to (63,56), there would only be a transition every sixty-three bits since $n=63$. That is, the output sequence would alternately consist of sixty-three zeros followed by sixty-three ones. While the output would still be balanced, this situation is not desirable since so few transitions could pose problems for clock recovery. Therefore a way to introduce more transitions is required.

This can be accomplished using the same AddCW technique if the encoder has more CW choices per SW. This is because the encoder could choose to transmit the CW that not only balances the transmission but also contains the most transitions. Based on the CW addition technique above, this can be achieved by doubling the number of AddCWs. For example, consider the same worst case situation where the all-zero CW $c_0(x)=0000000$ is continually emitted. If the encoder has two additional AddCWs available for CW addition, such as $c_4(x)=0100111$ and $c_{11}(x)=1011000$ as shown in

Figure 3-4, then the Select Best decision block would have four choices for transmission that are: $c_{result0}(x) = 0000000$, $c_{result1}(x) = 0100111$, $c_{result2}(x) = 1011000$ and $c_{result4}(x) = 1111111$. This is also a multimode scheme where one SW is mapped to a choice of four CWs.

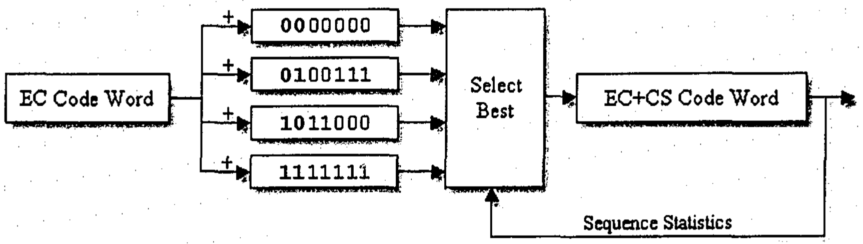


Figure 3-4: A simple EC + CS encoder using four AddCWs

In this situation it is clear that $c_{result1}(x)$ and $c_{result2}(x)$ are better choices for transmission than $c_{result0}(x)$ and $c_{result3}(x)$ simply because they contain transitions within each CW. The encoder can now alternately choose $c_{result1}(x)$ and $c_{result2}(x)$ for transmission resulting in the improved output sequence listed as $after_{(7,2)}$ in Figure 3-5. Comparison with the original all-zero sequence $original_{(7,4)}$ and the simple bimode scheme $before_{(7,3)}$ demonstrate how this multimode setup with double the number of AddCWs can introduce transitions and still maintain balanced output.

$original_{(7,4)}$	0000000	0000000	0000000	0000000
$before_{(7,3)}$	0000000	1111111	0000000	1111111
$after_{(7,2)}$	0100111	1011000	0100111	1011000

Figure 3-5: The effect of the (7,2) scheme is balanced output and numerous transitions

This CW addition technique has taken the all-zero CW with zero transitions and a 7:0 ratio of 0s to 1s per CW, and produced a sequence of alternating CWs that have three transitions and a 3:4 ratio and 4:3 ratio of 0s to 1s respectively. This technique is successful since the addition of a CW that has poor statistics (i.e. unbalanced with few transitions) to a CW that has better statistics (i.e. balanced with numerous transitions), results in a new CW whose statistics are an average of the two, upper bounded by the statistics of the better CW.

As show in Table 3-2, SW 0 is now mapped to four CWs, 0, 4, 11 and 15. Likewise, SW 1 is now mapped to CWs 1, 5, 10 and 14 and so on. Hence SWs 4 to 15 can no longer be used to represent data, or in other words, the ability to add transitions has resulted in an additional bit of redundancy over the previous bimode system. Now the first two bits of each SW can be viewed as redundant bits. Thus the (7,4) Hamming code has now been expurgated to a (7,2) Hamming code. Also note that these CWs still maintain the complementary nature of the system since cw_A and $\overline{cw_A}$ as well as cw_B and $\overline{cw_B}$ are complements.

Table 3-2: SW to CW mapping of this multimode (7,2) setup

Index	SW	CW_A	CW_B	$\overline{CW_B}$	$\overline{CW_A}$
0	0000	**00**00000$_{(0)}$	**01**00111$_{(4)}$	**10**11000$_{(11)}$	**11**11111$_{(15)}$
1	0001	**00**01011$_{(1)}$	**01**01100$_{(5)}$	**10**10011$_{(10)}$	**11**10100$_{(14)}$
2	0010	**00**10110$_{(2)}$	**01**10001$_{(6)}$	**10**01110$_{(9)}$	**11**01001$_{(13)}$
3	0011	**00**11101$_{(3)}$	**01**11010$_{(7)}$	**10**00101$_{(8)}$	**11**00010$_{(12)}$

Decoding of these CWs does not change significantly. Examination of Table 3-2 shows that the two MSbs of each CW are either 00, 01, 10 or 11 (shown in bold), or in decimal notation this is 0, 1, 2 or 3. Thus the decoder now inspects the two MSbs of the error corrected words to determine which of the four AddCWs was added. Thus as shown in Figure 3-6, the decoder first error corrects the received CW, then based on the value of the two MSbs, removes the CS coding by adding the correct AddCW. Recovering the SW is then the simple operation of removing the parity check bits.

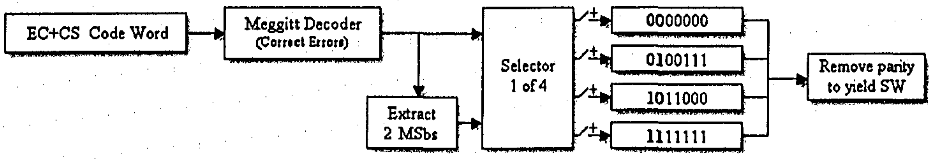


Figure 3-6: A simple EC + CS decoder using four AddCWs

Looking at the output data stream $after_{(7,2)}$ in Figure 3-5, it is clear that this system can guarantee balanced transmission and definitely improves the number of transitions. However there is another factor that needs to be considered which is the run of like valued bits across CW boundaries. For instance, when the concatenation of CWs is considered a run of four 1s and four 0s in still present in the output sequence as shown in Figure 3-7.

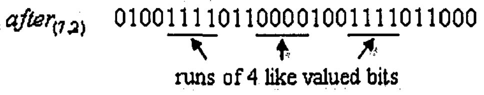

$after_{(7,2)}$ 0100111101100001001111011000
runs of 4 like valued bits

Figure 3-7: Concatenation of CWs shows runs of four like valued bits

While this may not seem significant, consider that this is a small code of $n=7$, and this represents a run of like valued bits more than half the CW length. Thus in a longer code with $n=63$, this could mean a run of more than thirty-three like valued bits in a row, which could cause problems for clock recovery. Therefore a way to break up these runs of like valued bits is required.

This could be accomplished once again using the same AddCW techniques if the encoder had more CW choices per SW. This is because the encoder could choose to transmit the CW that not only balances the transmission and contains the most transitions, but also limits the runlengths of like valued bits. That is, building again on the previous example consider the same situation where the all-zero CW $c_0(x)=0000000$ is continually emitted. If four additional AddCWs were available for CW addition, such as $c_2(x)=0010110$, $c_6(x)=0110001$, $c_9(x)=1001110$ and $c_{13}(x)=1101001$ as shown in

Figure 3-8, then the Select Best decision block would have eight choices for transmission that are: $c_{result0}(x) = 0000000$, $c_{result1}(x) = 0010110$, $c_{result2}(x) = 0100111$, $c_{result3}(x) = 0110001$, $c_{result4}(x) = 1001110$, $c_{result5}(x) = 1011000$, $c_{result6}(x) = 1101001$ and $c_{result7}(x) = 1111111$. This is also a multimode scheme where one SW is mapped to a choice of eight CWs.

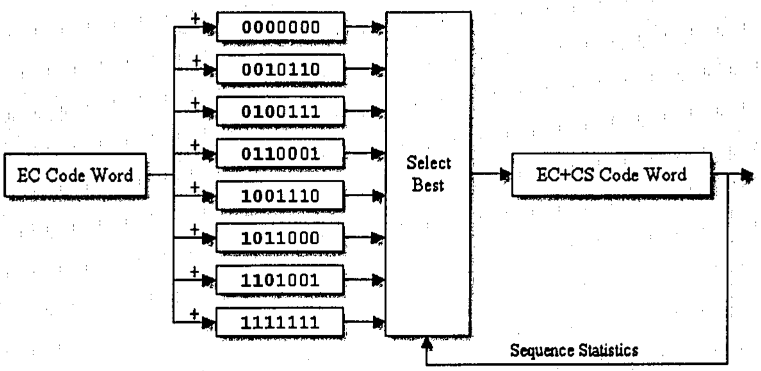


Figure 3-8: A simple EC + CS encoder using eight AddCWs

In this situation the Select Best decision block would select $c_{result1}(x) = 0010110$ and $c_{result6}(x) = 1101001$ for transmission since they not only ensure balanced transmission and contain numerous transitions , but a maximum runlength of two like valued bits is achieved. This is shown as $after_{(7,1)}$ in Figure 3-9.

Comparison with the original (7,4) all-zero sequence $original_{(7,4)}$, simple bimode (7,3) scheme $before_{(7,3)}$ and the four AddCW (7,2) system $before_{(7,2)}$ demonstrate how this multimode setup with eight AddCWs can introduce transitions, maintain balanced output and constrain the output to a maximum runlength of two like valued bits.

$original_{(7,4)}$	0000000	0000000	0000000	0000000
$before_{(7,3)}$	0000000	1111111	0000000	1111111
$before_{(7,2)}$	0100111	1011000	0100111	1011000
$after_{(7,1)}$	0010110	1101001	0010110	1101001

Figure 3-9: SW to CW mapping of this (7,1) setup

As shown in Table 3-3, SW 0 is now mapped to eight CWs, 0, 2, 4, 6, 9, 11, 13 and 15. Likewise, SW 1 is now mapped to CWs 1, 3, 5, 7, 8, 10, 12 and 14. Hence SWs 2 to 15 can no longer be used to represent data, or in other words, the ability to decrease the runlengths has resulted in an additional bit of redundancy over the previous setup. Thus the (7,4) Hamming code has now been expurgated to a (7,1) Hamming code. Once again the CWs have still maintained the complementary nature of the system since cw_A and $\overline{cw_A}$, as well as cw_B and $\overline{cw_B}$, cw_C and $\overline{cw_C}$ and cw_D and $\overline{cw_D}$ are complements.

Table 3-3: SW to CW mapping of this multimode (7,1) setup

I	SW	CW_A	CW_B	CW_C	CW_D	$\overline{CW_D}$	$\overline{CW_C}$	$\overline{CW_B}$	$\overline{CW_A}$
0	0000	$0000000_{(0)}$	$0010110_{(2)}$	$0100111_{(4)}$	$0110001_{(6)}$	$1001110_{(9)}$	$1011000_{(11)}$	$1101001_{(13)}$	$1111111_{(15)}$
1	0001	$0001011_{(1)}$	$0011101_{(3)}$	$0101100_{(5)}$	$0111010_{(7)}$	$1000101_{(8)}$	$1010011_{(10)}$	$1100010_{(12)}$	$1110100_{(14)}$

Decoding of these CWs is identical to the previous decoders. Examination of Table 3-3 shows that the three MSbs of each CW are either 000, 001, 010 011, 100, 101, 110 or 111. In decimal notation, this is 0, 1, 2, 3, 4, 5, 6, or 7. The decoder now checks the three MSbs to determine which of the eight AddCWs was added. Thus as shown in Figure 3-10, the decoder first error corrects the received CW, then based on the value of the three MSbs, removes the CS coding by adding the correct AddCW. Recovering the SW is then the simple operation of removing the parity check bits.

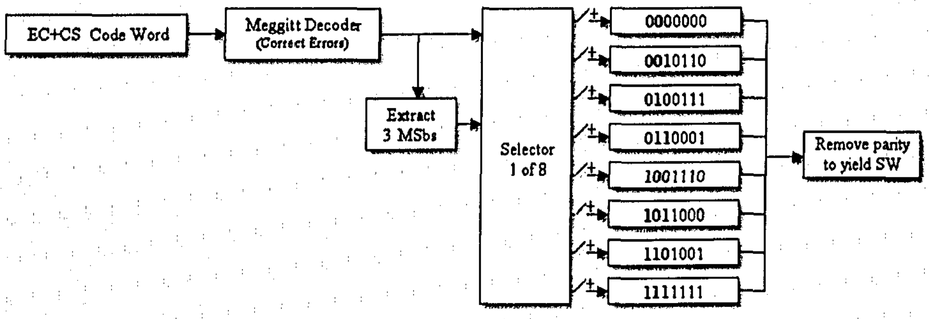


Figure 3-10: A simple EC + CS decoder using eight AddCWs

3.3. Tradeoffs

All three schemes presented yield balanced outputs, and when more AddCWs are available, increase the transition density. This can also be viewed as limiting the runlengths of like valued bits. These benefits however come at the cost of added redundancy. A quick comparison of the coding schemes is shown in Table 3-4, assuming equiprobable source statistics.

With 1 bit of redundancy for CS coding the (7,3) code is balanced and has on average an 80% chance of runlengths that are three or less. However it still has a 20% chance of runlengths that are higher than this, with a 0.147% chance of a run of twelve (not shown in the table). With two bits of redundancy for CS coding, the (7,2) code has a 96.41% chance of runlengths of three or less, and no possibility of a run longer than four. Finally with three bits of redundancy for CS coding, the (7,1) code can guarantee runlengths of two or less with a probability of 100%. Note that the (7,4) code is completely unbounded in terms of runlengths, i.e. it is possible to have runlengths of any length, whose likelihood is dependent on the statistics of the source data stream.

Table 3-4: (7,x) code performance with equiprobable source statistics

RunLengths	1	2	3	4	5	6	7	8	9
(7,3) Code	**0.2719**	**0.2609**	**0.2683**	**0.0364**	**0.0151**	**0.0049**	**0.0562**	**0.0378**	**0.0285**
(7,2) Code	**0.3794**	**0.3571**	**0.2276**	**0.0357**	0.0000	0.0000	0.0000	0.0000	0.0000
(7,1) Code	**0.4285**	**0.5714**	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Clearly with more AddCWs available the CS properties are improved. This however comes at a tradeoff of adding more redundancy.

3.4. Goals of the combined EC and CS system

The above examples demonstrate the combined EC and CS concept, whose multimode approach is made possible through CW addition. The following sections reiterate the most important goals, outline common pitfalls, and demonstrate the details of this approach and how it can be implemented with any linear block code.

The main goals of the system are:

1. Simple implementation;
2. Guarantee of the desired performance regardless of the incoming sequence statistics;
3. Addition of minimal redundancy into the encoded sequence.

Due to the cyclic nature of the simple Hamming EC code, encoding and decoding can be achieved using the simple logic circuits of Figure 2-5 and 2-6, i.e. look-up tables in ROMs are not required ensuring the use in high speed systems. The introduction of CS coding through CW addition can also be easily implemented using simple logic circuits, i.e. XOR gates since the addition is over GF(2).

Guaranteeing the desired performance of balanced transmission, numerous transitions and limiting the runlengths of like valued bits is achieved through feedback in the system. As discussed the system monitors the overall number of 0s and 1s sent on the channel to record transitions and runlengths, and then uses this information to determine the best CW for transmission. This feedback guarantees the system's CS goals will be maintained regardless of source statistics.

Finally, the (7,4) code discussed so far was used because it is small and presentable, but it suffers from too much redundancy with this scheme. However this redundancy is less of a factor when larger codes are used. For instance with three additional bits of redundancy (eight AddCWs), the (15,11) code expurgated to (15,8) is 46% redundant, but the (31,26) expurgated to (31,23) is only 25% redundant, and the (63,57) expurgated to (63,54) is only 14% redundant and so on. Therefore, the redundancy can be minimized to any desired level as larger and different codes are used.

3.5. Generic Systems

The following systems presented can be used with any linear block code and can be incorporated into existing digital circuits with a minimum of additional complexity.

Balanced transmission: As discussed through example in Section 3-1, balanced transmission can be achieved with a single additional bit of redundancy. The generic structure of the complementary bimode encoder is shown in Figure 3-11, and the corresponding decoder is shown in Figure 3-12.

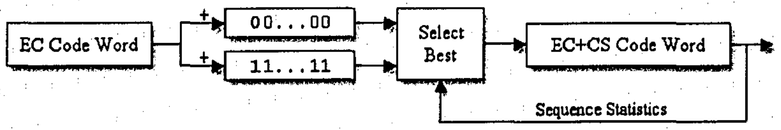


Figure 3-11: Generic bimode complementary encoder using two AddCWs

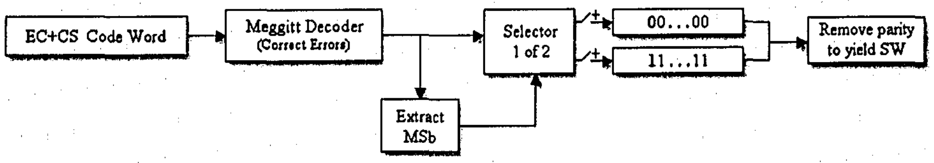


Figure 3-12: Generic bimode complementary decoder using two AddCWs

With this setup the two AddCWs are recommended to be the all-zero and all-one CWs as shown in Figure 3-13. This gives the Select Best decision block the ability to transmit the original CW or its complement. (Note that other CWs could be used here and this will be discussed below). Figure 3-13 also displays these CWs as 00...00 and 11...11 since their length is dependent on the (n,k) block code used. Therefore if the (7,4) Hamming code is used like in the examples of Section 3.1 and 3.2, then their length is $n=7$ bits. However if the (1023,1014) Hamming code was used, then their length would be $n=1023$ bits.

00...00   **1**1...11

Figure 3-13: AddCW format for a system using two AddCWs

In Figure 3-13 the MSbs of the two AddCWs (shown in bold) are 0 and 1 respectively, which as discussed in Section 3-1, is the extra bit of redundancy that allows the decoder to determine whether the received CW is the original or the complement. Strictly speaking then, for the decoder to function properly these two AddCWs do not necessarily have to be the all-zero and all-one CW. All that is required is that the two AddCWs used have their MSbs as 0 and 1 respectively. As long as this criteria is met, the decoder will be able to determine which AddCW needs to be added at the receiver in order to recover the original CW (i.e. remove the CS coding). Therefore Figure 3-14 shows the specific format that these AddCWs must take, where the x's represent any available CW that has the MSbs in this fashion.

0x...xx   **1**x...xx

Figure 3-14: Format of the two AddCWs for proper decoder operation

Note that since the code is cyclic, not using the all-zero and all-one CWs as AddCWs only has the effect of changing the overall mapping of SWs to CW. For example, if the two AddCWs used were $c_6(x)=0110001$ and $c_9(x)=1001110$, the SW to CW mapping would be that shown in Table 3-5. This table can be compared with Table 3-1 where the AddCWs were the all-zero and all-one CW.

Table 3-5: Using AddCWs $c_6(x)$=0110001 and $c_9(x)$=1001110

Index	SW	CW_A	$\overline{CW_A}$ (complement)
0	0000	$0110001_{(6)}$	$1001110_{(9)}$
1	0001	$0111010_{(7)}$	$1000101_{(8)}$
2	0010	$0100111_{(4)}$	$1011000_{(11)}$
3	0011	$0101100_{(5)}$	$1010011_{(10)}$
4	0100	$0010110_{(2)}$	$1101001_{(13)}$
5	0101	$0011101_{(3)}$	$1100010_{(12)}$
6	0110	$0000000_{(0)}$	$1111111_{(15)}$
7	0111	$0001011_{(1)}$	$1110100_{(14)}$

It is clear that with this new SW to CW mapping the output transmission will still be balanced, since in a worst case situation where a source continually emits the same CW such as the all-zero CW $c_0(x) = 0000000$, it would be mapped alternately to $c_6(x) = 0110001$ followed by $c_9(x) = 1001110$ and so on.

Note that if the two AddCWs were not chosen to be complements then balanced output can no longer be guaranteed. For example Table 3-6 shows the SW to CW mapping if the two AddCWs were $c_6(x) = 0110001$ and $c_8(x) = 1000101$.

Table 3-6: Using AddCWs $c_6(x)$=0110001 and $c_8(x)$=1000101

Index	SW	CW_A	$\overline{CW_A}$ (complement)
0	0000	$0110001_{(6)}$	$1000101_{(8)}$
1	0001	$0111010_{(7)}$	$1001110_{(9)}$
2	0010	$0100111_{(4)}$	$1010011_{(10)}$
3	0011	$0101100_{(5)}$	$1011000_{(11)}$
4	0100	$0010110_{(2)}$	$1100010_{(12)}$
5	0101	$0011101_{(3)}$	$1101001_{(13)}$
6	0110	$0000000_{(0)}$	$1110100_{(14)}$
7	0111	$0001011_{(1)}$	$1111111_{(15)}$

In this case if the all-zero CW $c_0(x) = 0000000$ is continually emitted it would be mapped alternately to $c_6(x) = 0110001$ followed by $c_8(x) = 1000101$ and so on, which are both unbalanced with a 0 to 1 ratio of 4:3. Hence this system would continually grow unbalanced. As a result, the AddCWs used in this bimode system must be chosen such that their MSbs are 0 and 1 respectively, and that they are CW complements. Therefore, since inspection of Table 3-1 and Table 3-5 shows that the only difference is the SW to CW mapping, for simplicity it is recommended that the AddCWs are always chosen to be the all-zero and all-one CW. In general the only requirement that this places on an EC code is that it contains the all-zero and all-one words as CWs.

Balanced transmission, Transitions: As discussed in Section 3-2, balanced transmission can be guaranteed and transition density can be improved with two additional bits of redundancy. Figure 3-15 shows the generic structure of the multimode complementary encoder that uses four AddCWs, with the corresponding decoder shown in Figure 3-16.

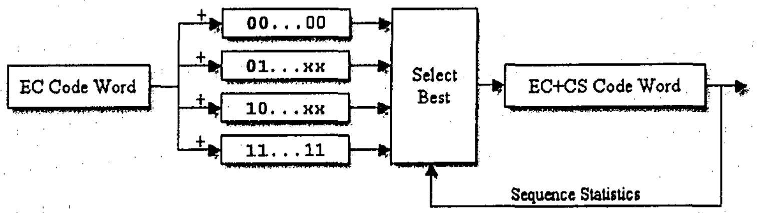


Figure 3-15: Generic multimode complementary encoder using four AddCWs

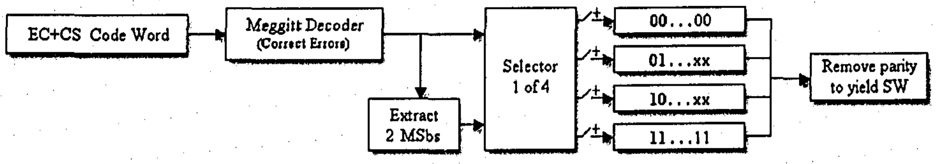


Figure 3-16: Generic multimode complementary decoder using four AddCWs

As with the previous setup in order for the decoder to know which AddCW was added at the transmitter, the AddCWs must follow the format shown in Figure 3-17. Since this system uses four AddCWs, two MSbs are used and they must be in the form 00, 01, 10 and 11, which in decimal is 0, 1, 2 and 3. Therefore strictly speaking, each AddCW could be any CW that meets this criteria.

`00...xx    01...xx    10...xx    11...xx`

Figure 3-17: Choosing an AddCW set with four AddCWs

However, from the previous discussion in order to guarantee balanced transmission the encoder needs the ability to send complementary CWs. Therefore the AddCWs should maintain the format of Figure 3-17, and they should be chosen to be complements as shown in Figure 3-18. Hence the all-zero and all-one CWs are chosen, and CW a can be chosen in the form 01...xx, and $\overline{a}$ is simply its complement.

00...00 a $\overline{a}$ 11...11

Figure 3-18: Strict format of the four AddCWs

Balanced transmission, Transitions, limit Runlengths: As discussed in Section 3-2, balanced transmission can be guaranteed, transition density can be improved, and runlengths can be greatly controlled with three additional bits of redundancy. Figure 3-19 shows the generic structure of the multimode complementary encoder that uses eight AddCWs, with the corresponding decoder shown in Figure 3-20.

As with the previous setup in order for the decoder to know which AddCW was added at the transmitter, the AddCWs must follow the format shown in Figure 3-21. Since this system uses eight AddCWs, three MSbs are used and they must be in the form 000, 001, 010, 011, 100, 101, 110 and 111, which in decimal is 0, 1, 2, 3, 4, 5, 6 and 7. Therefore strictly speaking each AddCW could be any CW that meets this criteria.

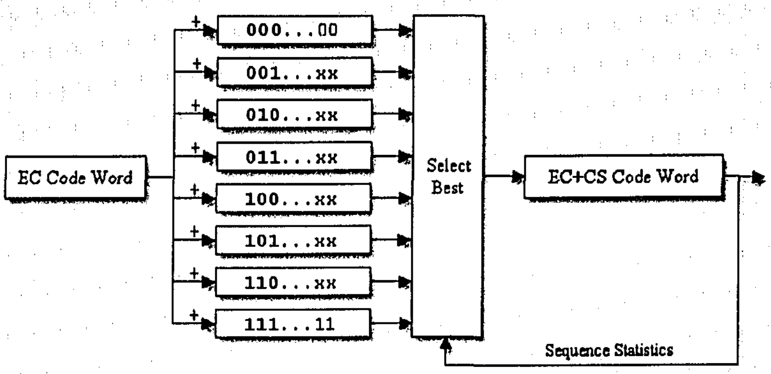


Figure 3-19: Generic multimode complementary encoder using eight AddCWs

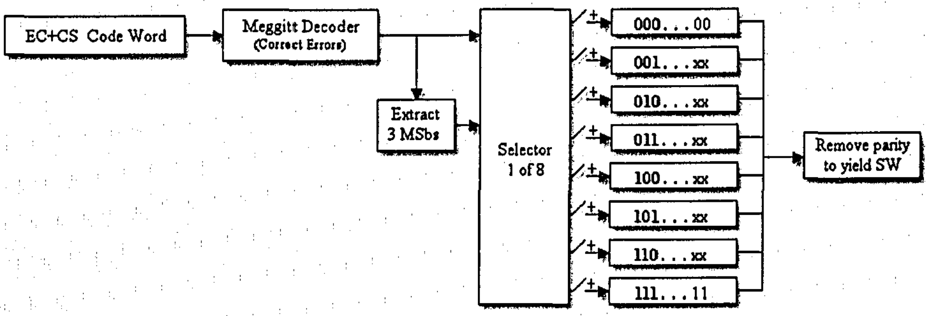


Figure 3-20: Generic multimode complementary decoder using eight AddCWs

`000...xx   001...xx   010...xx   011...xx   100...xx   101...xx   110...xx   111...xx`

Figure 3-21: Choosing an AddCW set with eight AddCWs

However, from previous discussions in order to guarantee balanced transmission the encoder needs the ability to send complementary CWs. Therefore the AddCWs should maintain the format of Figure 3-21, and they should be chosen to be complements as shown in Figure 3-22. Hence the all-zero and all-one CWs are chosen, and CW a can be chosen in the form 001...xx, CW b can be chosen in the form 010...xx, and CW c can be chosen in the form 011...xx. Then CWs $\bar{a}$, $\bar{b}$ and $\bar{c}$ are simply chosen as the complements of a, b and c respectively.

00...00 a b c $\overline{c}$ $\overline{b}$ $\overline{a}$ 11...11

Figure 3-22: Strict format of the eight AddCWs

3.6. General notes on AddCWs

3.6.1. AddSWs and hexadecimal notation

From the previous example in Figure 3-4, the four AddCWs used were $c_0(x) = 0000000$, $c_4(x) = 0100111$, $c_{11}(x) = 1011000$ and $c_{15}(x) = 1111111$, which follow the above guidelines since $c_0(x)$ and $c_{15}(x)$ as well as $c_4(x)$ and $c_{11}(x)$ are CW complements. However these could have also been listed in terms of SWs. For instance, inspection of Table 2-1 confirms that the SWs $s_0(x) = 0000$, $s_4(x) = 0100$, $s_{11}(x) = 1011$ and $s_{15}(x) = 1111$ are the SWs that form the CWs $c_0(x)$, $c_4(x)$, $c_{11}(x)$ and $c_{15}(x)$ respectively. Therefore these SWs are defined as **Add Source Words** (AddSWs). Thus using a compact hexadecimal notation this AddCW set can be represented as the AddCWs $\{00h, 27h, 58h, 7Fh\}$, or the AddSWs $\{0h, 4h, Bh, Fh\}$. These two representations will be used interchangeably since AddCWs are simply AddSWs that have been encoded.

3.6.2. Number of AddCWs can be any power of two

Analysis in this chapter has shown that the number of AddCWs can be increased to any power of two. While the generic systems listed in Section 3.5 have only used $2^1 = 2$, $2^2 = 4$ and $2^3 = 8$ AddCWs, more could be utilized such as $2^4 = 16$, $2^5 = 32$ or higher. These AddCWs would need to be chosen in the same complementary fashion, and the decoder would need to extract the appropriate number of MSbs in order to add the correct AddCW at the receiver. Note that as discussed in Section 3-4, the use of more AddCWs increases the transition density and decreases the runlengths of like valued bits. However these CS coding benefits come at an efficiency cost in terms of code rate.

3.6.3. MSbs and identification bits

Up to this point the encoders and decoders have used the MSbs to identify which AddCW has been used. However these identification bits can be located anywhere in the CW and are located in the MSb position simply due to simplicity.

3.6.4. Good AddCW set

An aspect not covered in this chapter is what makes a good AddCW set. For example, in the (7,2) Hamming code example from Figure 3-4, the four AddCWs used were arbitrarily chosen as $\{00h,27h,58h,7Fh\}$. However based on the AddCW set format outlined above, three more AddCW sets could have been used: $\{00h,2Ch,53h,7Fh\}$, $\{00h,31h,4Eh,7Fh\}$ or $\{00h,3Ah,45hh,7Fh\}$. The effects of using these different AddCW sets is thoroughly examined in Chapter 4, as well as the criteria regarding what makes a good AddCW.

3.6.5. Number of AddCW sets

To the casual observer it is often surprising to see that only four AddCW sets exist with the (7,2) Hamming code introduced in this chapter. It is unexpected since the format from Figure 3-17 may lead one to believe that there are $(4)(4)(4)(4)=16$ AddCW sets. However, this (7,2) code only has four possible AddCW sets which results from the AddCW recommendations and the size of this code. Recall from Figure 3-18 that two AddCWs are recommended to be the all-zero and all-one CW. Furthermore only CW a can be freely chosen to meet the format 01...xx, and $\bar{a}$ is simply chosen to be its complement. Therefore the correct number of possible AddCW sets is $(1)(4)(1)(1)=4$, and they were listed above in Section 3.6.4. These four AddCW sets can also be written in compact hexadecimal notation in terms of AddSWs as $\{0h,4h,Bh,Fh\}$, $\{0h,5h,Ah,Fh\}$, $\{0h,6h,9h,Fh\}$ and $\{0h,7h,8h,Fh\}$ respectively.

This small number of AddCW sets simply results from the small size of the code. If a larger code such as the (31,26) Hamming code expurgated to a (31,24) code was used, the number of AddCW sets increases to $(1)(2^{24})(1)(1)=16777216$.

Likewise in an eight AddCW system, a (7,4) Hamming code expurgated to a (7,1) code will only have eight possible AddCW sets. This is because Figure 3-22 shows that two of the AddCWs are recommended to be the all-zero and all-one CW. Furthermore only CWs *a*, *b* and *c* can be freely chosen to meet the format 001...xx, 010...xx, and 011...xx respectively, and $\bar{a}$, $\bar{b}$ and $\bar{c}$ must be chosen as their CW complements.

In general for any linear block code using this multimode coding approach the number of AddCW sets will be

$$\left(2^k\right)^{\frac{z-2}{2}} \tag{3.1}$$

where *k* is from the codes expurgated size (*n*,*k*) and *z* is the number of AddCWs.

Therefore the number of possible AddCW sets for the (7,4) code, expurgated to a (7,1) code using eight AddCWs is $\left(2^1\right)^{\frac{8-2}{2}}=\left(2^1\right)^3=(1)(2)(2)(2)(1)(1)(1)(1)=8$. In terms of AddSWs these sets can be written in hexadecimal form as {0h,2h,4h,6h,9h,Bh,Dh,Fh}, {0h,2h,4h,7h,8h,Bh,Dh,Fh}, {0h,2h,5h,6h,9h,Ah,Dh,Fh}, {0h,2h,5h,7h,8h,Ah,Dh,Fh}, {0h,3h,4h,6h,9h,Bh,Ch,Fh}, {0h,3h,4h,7h,8h,Bh,Ch,Fh}, {0h,3h,5h,6h,9h,Ah,Ch,Fh} and {0h,3h,5h,7h,8h,Ah,Ch,Fh}. Once again this small number of AddCW sets simply results from the restrictions placed on AddCW selection and the small size of the code. If a larger code such as the (31,26) Hamming code expurgated to a (31,23) code using eight AddCWs was used, the number of AddCW sets increases to $\left(2^{23}\right)^3=(1)\left(2^{23}\right)\left(2^{23}\right)\left(2^{23}\right)(1)(1)(1)(1)=2^{69}$. Note that Appendix B lists all AddSW sets used in this thesis.

3.7. Combined EC and CS code summary

This chapter introduced a combined EC and CS code that takes advantage of CW addition through linear code properties to achieve a multimode coding system. This combined code can be incorporated into existing systems with minimal complexity, avoiding the use of look-up tables in ROMs and making it applicable to high bit rate systems. Furthermore this system uses feedback to achieve its CS goals regardless of source statistics and avoids CS error propagation by error correcting CWs from the channel before removing the effect of the CS coding.

The real challenge of this system then comes in deciding which CWs to use as AddCWs. For instance, from Section 3.6.4 with the (7,2) Hamming code there are only four AddCW sets and therefore they can be directly compared against one another. However when dealing with larger codes there could be as many as 2^{24}, 2^{69} or more possible AddCW sets to choose from. As a result, directly comparing each AddCW set is not feasible. In general a good AddCW set should lead to a transmitted sequence that is balanced, contains numerous transitions, and has a high probability of low runlengths. Chapter 4 will examine the use of different AddCW sets and their effect on the output sequence in the time domain. Furthermore Chapter 5 will examine the use of different AddCW sets and their effect on the power spectral density of the transmitted sequence in the frequency domain.

4. Code Word Search and Analytical Results

The multimode coding system introduced in Chapter 3 involves generating an original CW and adding it to a specific set of CWs called AddCWs. This produces a new set of CWs from which the *best* is chosen for transmission. This chapter defines what is considered to be a *best* CW, as well as the metrics used for finding good AddCW sets.

4.1. Evaluating CW Statistics

The objective of the encoder is to transmit balanced sequences that contain numerous transitions. The encoder obtains this goal by transmitting CWs from the selection set that best matches these objectives. The encoder is able to make this decision by comparing the following CW statistics.

Transitions: A transition in a sequence occurs when there is a low to high or high to low level shift. Counting the number of transitions in a CW requires examination of the sequence of bits in the word and also requires keeping track of the last bit transmitted on the channel. For a simple example consider the case when the CW {0111010} is transmitted, represented in hexadecimal notation as 3Ah. As shown in Figure 4-1 if the last bit transmitted was a 0 then the number of transitions would be four. However if the last bit transmitted was a 1, then the number of transitions would have been five. Thus a particular CW will have a different number of transitions depending on the value of the last bit (LB) transmitted.

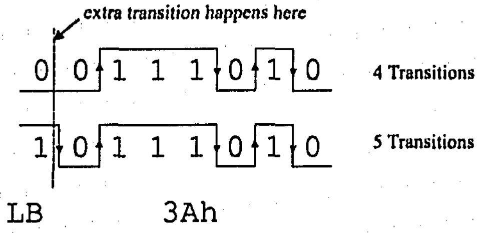


Figure 4-1: Counting Transitions

Running Digital Sum: In order to achieve a balanced output the number of 1s and 0s must be equal over the long term. These 1s and 0s will sometimes be referred to as logic 1s and logic 0s to avoid confusion when the transmitted sequence uses bipolar transmission encoding of Section 2.19. That is, during transmission logic 0s are mapped to the value -1 and logic 1s are mapped to the value +1. Therefore a binary sequence x of logic 1s and 0s during transmission is represented by Equation 4.1.

$$\{x\} = \{\ldots, x_{-1}, x_0, \ldots, x_i, \ldots\} \quad x_i = \{-1, 1\} \tag{4.1}$$

This mapping allows for the testing of a balanced transmission by simply summing up all of the bits in the sequence and testing if the result is zero. This process is known as evaluation of the **Running Digital Sum** (RDS) [19] and it is defined as

$$z_i = \sum_{j=-\infty}^{i} x_j \tag{4.2}$$

The RDS is calculated and updated with each bit transmitted. The RDS value at the end of the CW is defined as the **word end running digital sum** (WRDS). To demonstrate this, assume that the system begins with an RDS of 0, then continuing with the previous example, CW 3Ah would transition through various RDS values and ultimately result in a WRDS of +1 as shown in Figure 4-2. This updates the systems RDS to +1. If CW 3Ah was transmitted once again, its WRDS would be +2 updating the overall systems RDS to +2 and so on.

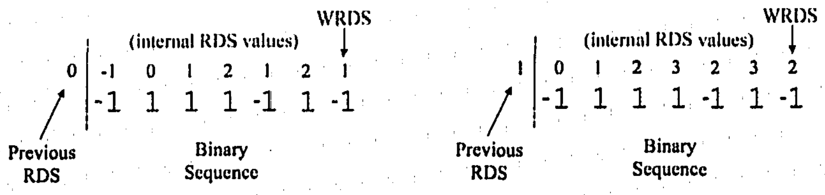


Figure 4-2: Running Digital Sum of the code word 3Ah

Having an RDS of +2 indicates that the system has currently had two more logic 1 bits than logic 0 bits and thus the transmission is slightly unbalanced. The next CW would need to have two more logic 0 bits than logic 1 bits to bring the system RDS back down to 0. This however represents an ideal situation that might not always be possible. It may take one or more additional CWs to bring the RDS back to 0. However it has been proven that as long as the RDS is bounded the transmission will be balanced [19].

Minimum Squared Weight: The WRDS is a useful metric that gives the overall balance of a CW. However this metric only tracks the RDS at the end of a CW and does not give any indication of the range of RDS values assumed within the CW. For instance an RDS value that steadily grows and falls within a CW indicates long runs of like valued bits. Inspecting only the number of transitions and WRDS of a CW may not indicate these runs. For example, consider the case when the two CWs 2F0h and 0E6h are being considered for transmission as shown in Figure 4-3.

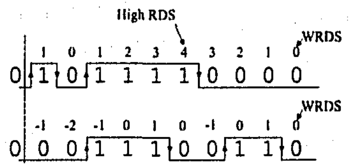


Figure 4-3: Minimum squared weight can take the internal RDS into account

It can be seen that both CWs have four transitions and a WRDS of 0. Hence the encoder would consider this a tie since both CWs are equally suitable for transmission. However it can be seen that the first CW has a run of four like valued bits, while the second CW only has runs of three like valued bits. The encoder would need an additional metric in order to monitor these runs and select the second CW for transmission which is a better choice in terms of limiting the runlengths. As a result the **squared weight** metric takes into account the RDS within a CW. It is defined as the square of the RDS values summed across the sequence as shown in Equation 4.3.

$$z_i = \sum_{j=-\infty}^{i} RDS_j^2 \tag{4.3}$$

Figure 4-4 shows the same CWs as Figure 4-3 with the squared weight calculated. Due to the long runs of like valued bits the squared weight of the first sequence is much higher at forty-five than the squared weight of the second sequence at nine. Therefore by using the **minimum squared weight** (MSW) criteria the second CW would be chosen for transmission.

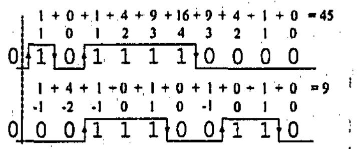


Figure 4-4: Minimum squared weight criteria chooses the second CW

4.2. Choosing the Best Code Word

The multimode coding system is presented with a choice of CWs for each SW, and must select only one CW for transmission. The CW chosen will be the one that best meets the system's requirements. Based on the above discussion there is some flexibility when choosing the best CW.

1. **Choose the CW that has the most transitions:** This will ensure that there is adequate timing information and will therefore satisfy the clock recovery objectives and reduce jitter. However this alone cannot ensure balanced transmission. For example the waveform in Figure 4-5 has numerous transitions but is clearly unbalanced.

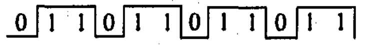


Figure 4-5: Sequence has many transitions but it is unbalanced

2. **Choose the CW that has the smallest abs(WRDS):** This by itself will guarantee balanced transmission, but it cannot ensure a large number of transitions. For example the waveform in Figure 4-6 is clearly balanced but has very few transitions.

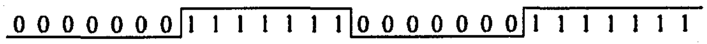


Figure 4-6: Sequence is balanced but it contains few transitions

3. **Choose the CW that has the smallest squared weight**: The MSW criteria works best as a tie breaking metric and does not perform well alone. The problem is that the MSW criteria squares the RDS values in an attempt to penalize long runs of like valued bits. If the run of like valued bits passes through an RDS of 0, the squared weight will be small and the MSW criteria can be fooled into believing this CW is the best. For example consider the case when the current RDS value is +3, the LB is 1 and the following CWs are considered for transmission.

Table 4-1: With the MSW criteria the all-zero CW would be transmitted

	WRDS	TRAN	SqW
00h	- 4.0	1.0	35.0
27h	+ 4.0	4.0	47.0
58h	+ 2.0	3.0	95.0
7Fh	+10.0	0.0	371.0

From Table 4-1 it can be seen that based on the lowest RDS criteria, CW 58h would be chosen as the best with a WRDS of +2. With the most transitions criteria, CW 27h would be chosen as the best with 4 transitions. However, with the MSW criteria, CW 00h would be chosen to be the best, even though it is the all-zero CW which this multimode system is trying to avoid. This demonstrates that the MSW criterion does not work well alone and is most useful as a tie breaking mechanism.

In general the CW decision process depends on the system requirements. If system performance is highly dependent on clock recovery, then the best CW to transmit will be the one with the most transitions. If system performance depends to a large extent on having a balanced transmission, then the best CW will be the one that minimizes the RDS. Therefore what is considered to be the best CW is subjective.

Furthermore, as already mentioned it is possible for more than one CW to satisfy the best CW requirements. In other words ties are possible. In these situations either CW is equally suitable for transmission. That is, the system can always choose to transmit the first CW that meets the systems goals, even if alternatives exist. This will still result in a superior output that is balanced and contains numerous transitions.

However an undesirable problem can arise in this situation. If ties occur in a regular fashion and the encoder continually chooses the first CW that meets the CS goals of the system, the encoder may be selecting this CW in a periodic fashion. This can result in discrete like components in the power spectrum of the transmitted waveform, possibly leading to cross-talk and possibly violating any spectral masks that must be followed. As a result it is useful to have enough tie breaking mechanisms to break any and all ties such that there is always a single clear best CW. Otherwise it is recommended that when one or more CWs are equally suitable for transmission, that the best CW is chosen randomly from the set of words considered equally good.

4.3. Selection process used in this thesis

In this thesis the analysis was done based on selecting the CW that minimized the RDS. This is because it is the only selection criteria that can guarantee balanced transmission. Other criteria such as the number of transitions and MSW are still employed in order to break ties.

4.4. Example of the CW selection process based on minimizing the RDS

Consider the following (7,2) system using the AddSWs {0h,7h,8h,Fh} encoded to the AddCWs {00h,3Ah,45h,7Fh} as shown in Figure 4-7.

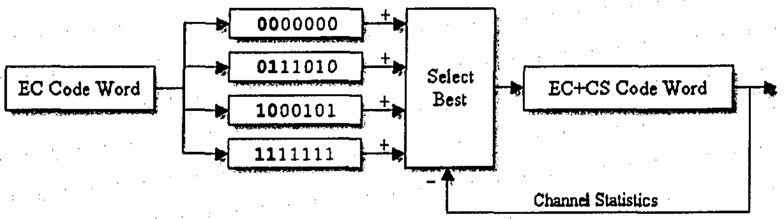
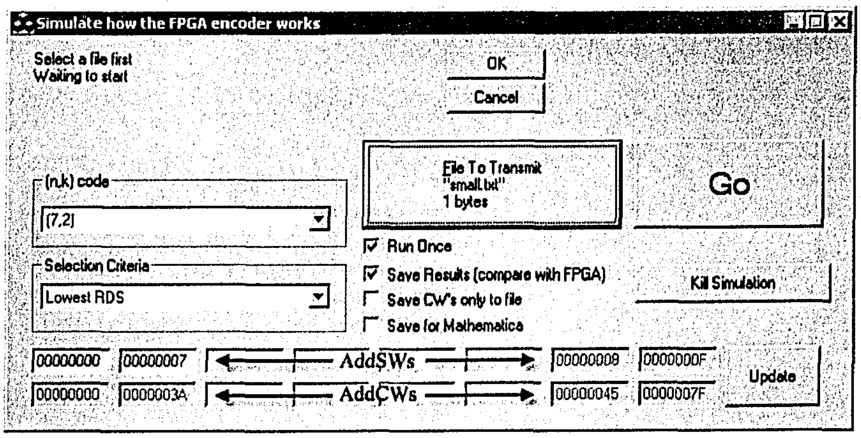


Figure 4-7: (7,2) encoder with AddCWs { 00h, 3Ah, 45h, 7Fh }

In this system there are $2^2 = 4$ possible SWs. Therefore in each encoding interval one of the four possible SWs is encoded to a CW, added to all four AddCWs listed above, and then the best CW is chosen from this set. In order to verify this process a simulation called **SimFPGA** was written and its interface is shown in Figure 4-8. Note that the name of this program comes from the fact that this system was implemented on two FPGA boards as will be discussed in Chapter 6. This program writes every step of the best CW decision process to a file.



Select a file first
Waiting to start

File To Transmit
"small.txt"
1 bytes

(n,k) code: (7,2)

Selection Criteria: Lowest RDS

[✓] Run Once
[✓] Save Results (compare with FPGA)
[ ] Save CW's only to file
[ ] Save for Mathematica

AddSWs: 00000000 00000007 00000008 0000000F
AddCWs: 00000000 0000003A 00000045 0000007F

Figure 4-8: SimFPGA - Program for verifying the CW selection process

Consider a simple example using the (7,2) code where there are four 2-bit SWs for transmission, 00, 10, 11, and 11, or in decimal 0, 2, 3 and 3. As shown in Figure 4-8 the two AddSWs $\{0h, 7h\}$ are entered into the program in hexadecimal; their complements $\{8h, Fh\}$ are automatically generated. These four AddSWs are then converted into the AddCWs $\{00h, 3Ah, 45h, 7Fh\}$ respectively. The output from this program is written to a file as shown in Table 4-2 below.

Table 4-2: Decision process from SimFPGA for the first SW

```
SW = 00          WRDS    TRAN     SqW     1stTran
00000000h        -7.0     0.0     140.0    0.0
0000003Ah        +1.0     4.0     12.0     2.0
00000045h        -1.0     5.0     12.0     2.0****
0000007Fh        +7.0     1.0     140.0    0.0

WinningLoc = 2 BitsLeft = 6     FilePointer = 0        CW's sent 1
Tie = 00000005 NumTies = 0      state 1 of numstates 2
((((( 00000045h                 RDS = -1.0    LB = 1 )))))
```

The first 2-bit SW 00 results in CW 00h which is added to all AddCWs generating the four CWs $\{00h,3Ah,45h,7Fh\}$. From these four CW choices the encoder must now choose the best one for transmission. In order to do this the encoder arbitrarily assumes the initial RDS of the system is 0 and the initial LB is 0. Table 4-2 lists the metrics of each CW choice. It is clear that the all-zero and all-one CW perform poorly with respect to all metrics since they have high WRDS values of -7 and +7, low transitions of 0 and 1, and a squared weight of 140. Therefore the choice comes down to either CW 3Ah or 45h. Since these CWs are complements the abs(RDS) and squared weight (SqW) are the same at 1 and 12 respectively. The only difference between them is the number of transitions. Since the LB was initially chosen to be 0 the winning CW selected is 45h and it is considered to be the best.

Since CW 45h was transmitted and it has a WRDS of -1 and a LB of 1, the system RDS is updated to -1 and the LB is updated to 1. Now the system can move onto the next SW. Continuing in the same fashion the next 2-bit SW 01 is encoded to the CW 16h and added to the 4 AddCWs to generate the CWs {16h,2Ch,53h,69h}. Table 4-3 shows that two CWs will bring the system RDS back to 0. Thus in terms of smallest abs(WRDS) either of these CWs would be considered the best and the encoder has encountered a tie.

Table 4-3: Decision process from SimFPGA for the second SW

```
SW = 10          WRDS   TRAN    SqW     1stTran
00000016h        -2.0   5.0     35.0    3.0
0000002Ch        -2.0   5.0     15.0    2.0
00000053h        +0.0   4.0     7.0     2.0
00000069h        +0.0   4.0     3.0     3.0****

WinningLoc = 3 BitsLeft = 4   FilePointer = 0        CW's sent 2
Tie = 00000007 NumTies = 0    state 2 of numstates 3
((((( 00000069h               RDS = 0.0   LB = 1 )))))
```

Since the main requirement of the system has been met the encoder can now choose the CW that meets the secondary goals of maximizing transitions for clock recovery, i.e. removing long runs of like valued bits. The first tie breaking mechanism considered is the number of transitions. However both “good” CWs have 4 transitions and this tie persists. Only as a result of the MSW criteria has the tie been broken and CW 69h is chosen to be the best since its squared weight is smaller than CW 53h. This demonstrates how the number of transitions and MSW criteria can still be used even though the overall goal is only to transmit the CW with the lowest abs(RDS).

Finally consider when the source outputs the same SW 11 twice. As shown in Table 4-4 the first SW 11 causes the CW 27h to be transmitted and the second SW 11 causes the CW 62h to be transmitted. This demonstrates that the same SW will often be represented with different CWs as a result of multimode coding. It also shows that the best CW will not always be the same for the same SW. The best CW will often change depending on the current RDS and LB value. Therefore these two values are formally defined as a **state** of the system and will be represented as an (RDS,LB) state.

Table 4-4: Decision process from SimFPGA for the third and fourth SW

```
SW = 11         WRDS   TRAN    SqW    1stTran
0000001Dh       +1.0    4.0    8.0     3.0
00000027h       +1.0    4.0    8.0     2.0****
00000058h       -1.0    3.0    8.0     2.0
00000062h       -1.0    3.0    8.0     3.0

WinningLoc = 1 BitsLeft = 2   FilePointer = 0        CW's sent 3
Tie = 0000000F NumTies = 0
        state 3 of numstates 4
((((( 00000027h               RDS = 1.0   LB = 1 )))))

SW = 11         WRDS   TRAN    SqW    1stTran
0000001Dh       +2.0    4.0    11.0    3.0
00000027h       +2.0    4.0    7.0     2.0
00000058h       +0.0    3.0    23.0    2.0
00000062h       +0.0    3.0    19.0    3.0****

WinningLoc = 3 BitsLeft = 0   FilePointer = 0        CW's sent 4
Tie = 0000000F NumTies = 0
        state 0 of numstates 4
((((( 00000062h               RDS = 0.0   LB = 0 )))))
```

4.5. Comparing unconstrained and constrained transmission

If no CS coding had been used in the above example then only the regular (7,4) Hamming code would had have been used. As a result the transmitted sequence would have been {00h,16h,1Dh,1Dh}. This sequence would have had 11 transitions, been unbalanced with an RDS of -6, and it would have contained a run of nine like valued bits as shown in Figure 4-9.

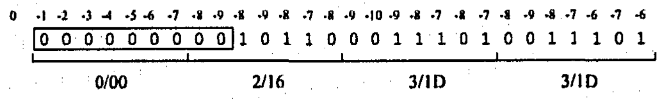


Figure 4-9: Uncoded sequence is unbalanced and has a run of 9

Conversely, the multimode coding system that incorporates CS coding has resulted in an output sequence that is completely balanced with an RDS of 0 and containing 16 transitions.

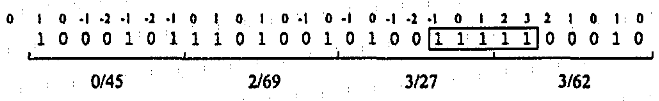


Figure 4-10: Coded sequence is balanced but still has a run of 5 like valued bits

While this is a definite improvement inspection of Figure 4-10 shows that this constrained sequence still has a run of five like valued bits. The question that arises then is how likely is this run of length five? Furthermore, could this run be removed if different AddCWs were used?

4.6. Effect of Different AddCWs

Analysis in Sections 3.6.4 and 3.6.5 reported that there are only four possible AddCW sets in this (7,2) system. They are {00h,27h,58h,7Fh}, {00h,2Ch,53h,7Fh}, {00h,31h,4Eh,7Fh} and {00h,3Ah,45h,7Fh}, or in terms of AddSWs are {0h,4h,Bh,Fh}, {0h,5h,Ah,Fh}, {0h,6h,9h,Fh} and {0h,7h,8h,Fh} respectively. Using the same sequence of four SWs, 00, 10, 11 and 11 with the AddSW set {0h,4h,Bh,Fh} (versus {0h,7h,8h,Fh} previously) an entirely different sequence will be transmitted as shown in Tables 4-5 and 4-6.

Here it can be seen that the first 2-bit SW 00 is now encoded to the CW 58h instead of 45h as it was previously. In addition, this CW has caused the RDS to be -1 and the LB to be 0. Comparison with the last AddCW set shows that this is an (RDS,LB) state that was not reached before. As a result it is possible that this AddCW set will cause the encoder to experience different RDS and LB combinations and therefore have a different number of states than the last example. Table 4-6 continues this example by showing the decision process for the next three SWs.

Table 4-5: Decision process from SimFPGA for the 1st SW

```
SW = 00          WRDS    TRAN     SqW      1stTran
00000000h        -7.0    0.0      140.0    0.0
00000027h        +1.0    3.0      8.0      2.0
00000058h        -1.0    4.0      8.0      2.0****
0000007Fh        +7.0    1.0      140.0    0.0

WinningLoc = 2 BitsLeft = 6   FilePointer = 0        CW's sent 1
Tie = 00000005 NumTies = 0    state 1 of numstates 2
((((( 00000058h               RDS = -1.0    LB = 0 )))))
```

Table 4-6: Decision process from SimFPGA for the second, third and fourth SW

```
SW = 10          WRDS    TRAN     SqW      1stTran
00000016h        -2.0    4.0      35.0     3.0
00000031h        -2.0    3.0      23.0     2.0
0000004Eh        +0.0    4.0      7.0      2.0
00000069h        +0.0    5.0      3.0      3.0****

WinningLoc = 3 BitsLeft = 4   FilePointer = 0        CW's sent 2
Tie = 00000005 NumTies = 0    state 2 of numstates 3
((((( 00000069h               RDS = 0.0    LB = 1 )))))

SW = 11           WRDS    TRAN     SqW      1stTran
0000001Dh        +1.0    4.0      8.0      3.0
0000003Ah        +1.0    5.0      12.0     2.0****
00000045h        -1.0    4.0      12.0     2.0
00000062h        -1.0    3.0      8.0      3.0

WinningLoc = 1 BitsLeft = 2   FilePointer = 0        CW's sent 3
Tie = 00000005 NumTies = 0    state 3 of numstates 4
((((( 0000003Ah               RDS = 1.0    LB = 0 )))))

SW = 11          WRDS    TRAN     SqW      1stTran
0000001Dh        +2.0    3.0      11.0     3.0
0000003Ah        +2.0    4.0      31.0     2.0
00000045h        +0.0    5.0      7.0      2.0****
00000062h        +0.0    4.0      19.0     3.0

WinningLoc = 2 BitsLeft = 0   FilePointer = 0        CW's sent 4
Tie = 00000005 NumTies = 0    state 2 of numstates 4
((((( 00000045h               RDS = 0.0    LB = 1 )))))
```

In this case the transmitted CWs are {58h,69h,3Ah,45h}. This sequence has 19 transitions, is completely balanced and now has a maximum run of only three like valued bits as shown in Figure 4-11.

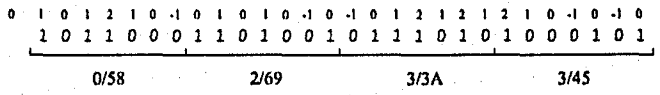


Figure 4-11: Coded sequence is balanced and now has a run of only three like valued bits

Based on this result it would appear that this is a better AddCW set. However making this assumption would be premature since only four SWs were tested. A complete analysis will look at the probability of various runlengths occurring and can only be performed once all SW/CW combinations have been tested. In order to conduct this analysis and evaluate different AddCW sets, it is helpful to model the encoder as a **finite state machine** (FSM).

4.7. Finite State Machines

Any system that operates at discrete instants of time and takes on a finite number of configurations can be represented as a finite state machine (FSM) [22]. A FSM is classified as either a **Moore machine**, where the output is a function of the internal state only, or as a **Mealy machine** where the output is a function of the internal state and the present input [22]. The encoder presented in this thesis is classified as a Mealy machine since the output CW and next state are both dependent on the present SW and present (RDS,LB) state.

A FSM can also be represented graphically by a **state diagram** which shows the progression of states through which the system operates and the resulting outputs based on specific inputs.

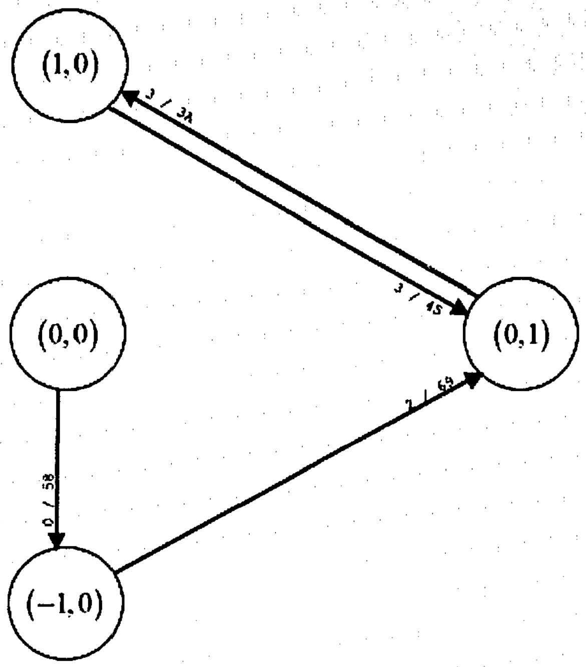


Figure 4-12: Partial state diagram for the (7,2) encoder transmitting the four SWs 0, 2, 3 and 3. This causes transitions from state (0,0), to state (-1,0), to state (0,1), to state (1,0), and back to state (0,1)

For instance, in the second (7,2) example above with AddSWs {0h,4h,Bh,Fh}, the sequence of four SWs transmitted caused the system to move into four (RDS,LB) states as shown in Figure 4-12. The state diagram shows how each SW is mapped to each CW depending on which state the system is presently in. For instance the system was assumed to start with an RDS of 0 and a LB of 0. As a result the first state is denoted (0,0). The first SW emitted was 0h and this caused the CW 58h to be transmitted and it moved the system into state $(-1,0)$, i.e. an RDS of -1 and a LB of 0. This SW/CW interaction is shown as 0/58 on the trace that moves from state (0,0) to state $(-1,0)$. Likewise the next SW was 2h with corresponding CW 69h, shown as 2/69 and this moved the system into state (0,1). Finally the next two SWs were both 3h, which caused the CWs 3Ah and 45h to be transmitted, shown as 3/3A and 3/45 respectively, which moved the system to state (1,0) and then back again to state (0,1).

Figure 4-12 only shows a partial state diagram after transmitting four CWs. In order to see the **complete state diagram** all SW/CW combinations must be explored from every state. Figure 4-13 shows the complete state diagram for this (7,2) encoder using the AddSW set {0h,7h,8h,Fh} encoded to the AddCW set {00h,3Ah,45hh,7Fh} where CW selection is based on minimizing the RDS. It consists of six (RDS,LB) states numbered 0 through 5 for convenience. Inspection of this state diagram shows that each state will be left the interval after it is entered. However the probability of entering each state is not equally probable. Furthermore since the goal of the encoder is to minimize the RDS, the states with the most entry points are states 2 and 3 which both represent an RDS of 0, and a LB of 0 and 1 respectively.

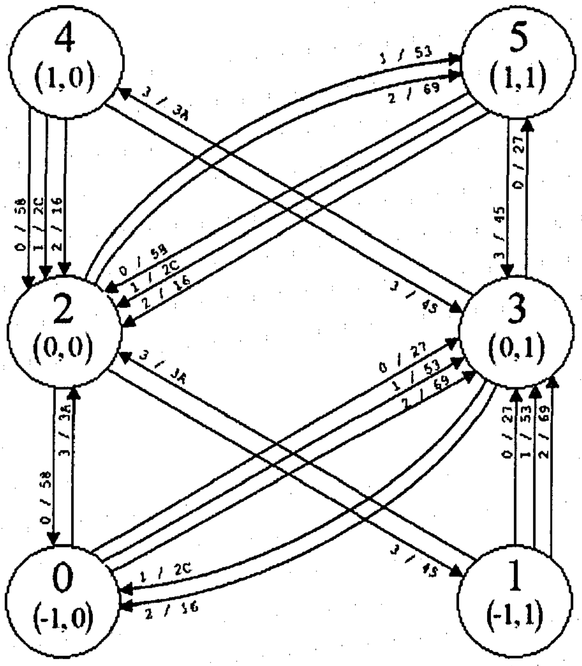


Figure 4-13: State Diagram for (7,2) code with AddSWs={0h,4h,Bh,Fh}

Focusing on the left half of the state diagram, i.e. the three states 0, 2 and 4 that each have a LB of 0, it can be seen that there are eight different CWs that take the system into state 2, three different CWs that take the system into state 0, but only one CW that can take the system into state 4. Clearly state 2 will be visited more often than states 0 and 4. This is an important distinction since even though all exit paths from a particular state are equally likely with equiprobable SWs, the probability of being in each state is not. This means that the CWs that leave state 2 will be seen on the channel more often than the CWs that leave states 0 and 4. Hence the probability of a CW being transmitted is weighted by the probability of being in a particular state. When attempting to calculate runlength probabilities these state probabilities need to be taken into consideration. This form of analysis can be performed by interpreting the FSM as a **Markov Chain**.

4.8. Markov Processes and Markov Chains

4.8.1. Markov Chains

A random process *X*(*t*) is a **Markov Process** if the future value of the process is dependent only on its immediate present value. In other words the process is independent of the past; its future value is dependent only on its present value. This is expressed concisely in Equation 4.4, which states that the probability of $X(t)$ assuming a new value given all the previous values for all time is equal to the probability of $X(t)$ assuming a new value if only the immediate value x_k is considered.

$$P[X(t_{k+1}) = x_{k+1} \mid X(t_k) = x_k, \ldots, X(t_1) = x_1] = P[X(t_{k+1}) = x_{k+1} \mid X(t_k) = x_k] \qquad (4.4)$$

In Markov chains the **probability distribution functions** (PDFs) that are conditioned on several time instants always reduce to one PDF that is conditioned only on the most recent time instant. This is known as the **Markov property** [23].

An integer valued Markov Process is a discrete time random process called a **Markov Chain**. Here the random variable X_n takes on a countable number of values at discrete moments in time, where *T* is the interval between discrete time events. The value of X_n at the discrete time n is referred to as the **state of the process** at time n.

$$X_n = X(nT) \qquad (4.5)$$

4.8.2. Markov Chain Properties

A system can be modeled by a Markov chain if the sequence of trials satisfies the following properties:

1. Each outcome belongs to a finite set of outcomes $\{a_1, a_2, ..., a_m\}$ called the **state space** of the system. If the outcome of the n^{th} trial is a_i, then the system is defined to be in state a_i at time *n*, or the system is in state a_i at the n^{th} step.
2. The outcome of any trial depends at most upon the outcome of the immediately preceding trial and not upon any other previous outcome.
3. There is a given probability p_{ij} that state a_j occurs immediately after a_i occurs. This can also be interpreted as follows: if the process is currently in state a_i, then p_{ij} is the probability of it moving to state a_j.

The numbers p_{ij} are called the transition probabilities and they can be arranged in a matrix called the **transition matrix** as shown in Equation 4.6.

$$P = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix} \tag{4.6}$$

The transition matrix *P* is a **stochastic matrix** since each row is a **probability vector** that sums to 1. Additionally a Markov chain has an **initial probability vector** *p* which indicates the starting state. For instance, calculation of the probability of the system starting in state 0 and then moving from state 0 to 1, and then from state 1 to 2 and so on would be done as shown in Equation 4.7.

$$p(X_0 = i_0, X_1 = i_1, X_2 = i_2, ..., X_n = i_n) = p(i_0)P(i_0, i_1)P(i_1, i_2)...P(i_{n-1}, i_n) \tag{4.7}$$

For example, let P be a transition matrix with the initial probability vector p as shown in Equation 4.8 below. Since $p = [1 \quad 0]$, the system begins in state 0 with 100% probability. The probability $P(0001110)$ denotes the probability of starting in state 0, staying in state 0 for two more time steps, moving to state 1 on the 3rd time step, staying in state 1 for the next two time steps, and finally ending in state 0 on the 6th time step. As shown in Equation 4.9 this probability is approximately 0.01627. Likewise, the probability $P(0011110) = 0.01808$ as shown in Equation 4.10.

$$P = \begin{bmatrix} 3/4 & 1/4 \\ 1/6 & 5/6 \end{bmatrix} \qquad p = [1 \quad 0] \tag{4.8}$$

$$P(0001110) = (1)\left(\frac{3}{4}\right)\left(\frac{3}{4}\right)\left(\frac{1}{4}\right)\left(\frac{5}{6}\right)\left(\frac{5}{6}\right)\left(\frac{1}{6}\right) = 0.01627 \tag{4.9}$$

$$P(0011110) = (1)\left(\frac{3}{4}\right)\left(\frac{1}{4}\right)\left(\frac{5}{6}\right)\left(\frac{5}{6}\right)\left(\frac{5}{6}\right)\left(\frac{1}{6}\right) = 0.01808 \tag{4.10}$$

Equation 4.9 and 4.10 show two cases of the system starting in state 0 and after six time steps finishing in state 0. If we were interested in all possible cases of starting in state 0 and after six time steps ending in state 0, we would need to find all possible **chains** of the type 0xxxxx0, where x indicates a don't care state, and add together the probabilities of these chains for the total probability which would be denoted as $P(X_6 = 0 \mid X_0 = 0)$. There would be $2^5 = 32$ possible chains, and clearly this operation would become cumbersome, especially for a large number of time steps.

The probability $P(X_n = j \mid X_0 = i)$ can be obtained however from the nth power of the transition matrix P^n [23]. This is due to a nice link between linear algebra and probability. The transition matrix P can be interpreted as P^1, which is the probability that the system changes from the state a_i to the state a_j in one time step. Likewise P^2 is the probability that the system changes from the state a_i to the state a_j in two time steps and continuing in this fashion P^n would represent the probability that the system changes from state a_i to the state a_j in n time steps. Thus the probability $P(X_6 = 0 | X_0 = 0)$ can be obtained from the 6th power of the transition matrix *P*.

As shown in Equation 4.11 below, $P(X_6 = 0 | X_0 = 0) = 0.424$, which means that if the starting state is state 0, the probability of it ending in state 0 six time steps later is 42.4%. Likewise the probability of starting in state 0 and ending in state 1 is 57.6% and so on.

$$P^6 = \begin{bmatrix} .424 & .576 \\ .384 & .616 \end{bmatrix} \quad \text{so} \quad P(X_6 = 0 | X_0 = 0) = 0.424 \tag{4.11}$$

In general a stochastic matrix is said to be **regular** if all the entries of P^n for $n > 1$ are positive (non-zero) [23]. In Equation 4.8 the stochastic matrix *P* is considered to be a regular stochastic matrix by this definition.

4.8.3. Long Term Behavior of Markov Chains

Understanding the long term behavior of a Markov chain simplifies to understanding P^n for large n. Each P^n preserves the property of rows summing to one and having non-negative entries. Thus all P^n matrices are still transition matrices. For example, *P* after 20 time steps is shown in Equation 4.12.

$$P^{20} = \begin{bmatrix} .400 & .599 \\ .399 & .600 \end{bmatrix} \tag{4.12}$$

It appears to be converging and one can conjecture that as n approaches infinity

$$P^n = \begin{bmatrix} .4 & .6 \\ .4 & .6 \end{bmatrix} \tag{4.13}$$

Each row (which is a probability vector) of P^n for large n is identical. This probability vector is of special interest and is called the **invariant probability vector** π or λ_∞ [23]. The probability vector π is invariant since

$$\pi = \pi P = \pi P^2 = \pi P^3 = \ldots = \pi P^n \quad \text{as} \quad n \to \infty \tag{4.14}$$

This expression demonstrates that with distribution π, the chain is in equilibrium. That is, if the system starts with this probability distribution π, then this distribution will be maintained for all time. The π vector is also a limiting probability vector since

$$\pi = \lim_{n\to\infty} vP^n \tag{4.15}$$

where v is any initial probability vector. Equation 4.15 states that for any initial probability vector v, vP^n approaches the invariant probability vector π for large n. Every Markov chain will have an invariant probability vector π [23] and this vector gives the long term probability of being in any particular state. That is, if the system was running for some period of time, and it was suddenly stopped, π would give the probability of being in each state. In general if P is a regular stochastic matrix then the subsequent powers of P approach a matrix Π that has rows that consist of the fixed probability vector π [23].

However π is not always unique [23]. If π is a unique vector and the rows of the transition matrix P raised to large n converge to π, then the chain/system is said to be **ergodic**. This comes from the **Perron-Frobenius theorem** [23]. This theorem states that if there exists an n such that $P^n > 0$ for $n > 1$ then P is ergodic [23]. **Ergodicity** of Markov chains is a desired property since it guarantees that no matter which state you start in, if you run the system long enough the influence of the initial probability vector p vanishes and the system will approach the invariant distribution π. This result has practical applications such as with Monte Carlo Markov Chain (MCMC) algorithms and simulations [23].

4.8.4. Non-Regular Markov Chains

It is possible to have a Markov chain that is ergodic and has a unique invariant probability vector that does not satisfy the Perron-Frobenius theorem. For example consider this trivial case

$$P=\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \quad P^2=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \quad P^3=\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \quad P^4=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \tag{4.16}$$

This chain will never converge and the powers of P will alternate between the two matrices in Equation 4.16. This is because the chain has a **period** of size $d=2$. For chains with a period of d the subsequent powers of P cycle between d different matrices [23]. In this case P^n will not converge and thus vP^n will continue to alternate between $d=2$ vectors unless $v=\pi$. Therefore this chain is **not regular.**

However since the transition matrix states that when the system is in state 0 it must move to state 1, and on the next time instant it must move back to state 0 and so on, one can intuitively conclude that 50% of the time must be spent in each state. Therefore even though there is no convergence it can be expected that $\pi=\left[.5 \quad .5\right]$. In systems like this that are not regular the **average** of the transition matrices raised to large n **will always converge** to a matrix M [23]. Furthermore, if all of the rows of M are identical then the invariant probability vector π is unique [23].

$$\frac{1}{n}\sum_{k=1}^{n} P^k \rightarrow M \tag{4.17}$$

Using Equation 4.17 the invariant probability vector can be found from averaging the transition matrices as shown in Equation 4.18 to find $\pi = [.5 \quad .5]$.

$$M = \frac{1}{2}\left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \begin{bmatrix} .5 & .5 \\ .5 & .5 \end{bmatrix} \quad \text{thus} \quad \pi = [.5 \quad .5] \tag{4.18}$$

The above example demonstrates a case where π is unique and yet P does not converge, hence they do not go hand in hand. As well this chain does not satisfy the Perron-Frobenius theorem and yet it is ergodic. This is an important distinction since many of the combined EC and CS codes presented in this thesis, when analyzed as Markov chains are also ergodic but do not satisfy the Perron-Frobenius system and their transition matrices never converge.

4.8.5. State Diagrams

A Markov chain can also be represented graphically by a state diagram just like a FSM. For example the state diagram for the Mealy FSM state graph of the (7,2) encoder using the AddSWs {0h,4h,Bh,Fh} from Figure 4-13 can be redrawn using the probability of taking a particular trace, instead of the SW/CW relationship, as shown in Figure 4-14. With this representation it is easy to see the probability of each CW being transmitted from each state. Furthermore, assuming that the source symbols are independent and equally likely the transition matrix can be constructed by inspection.

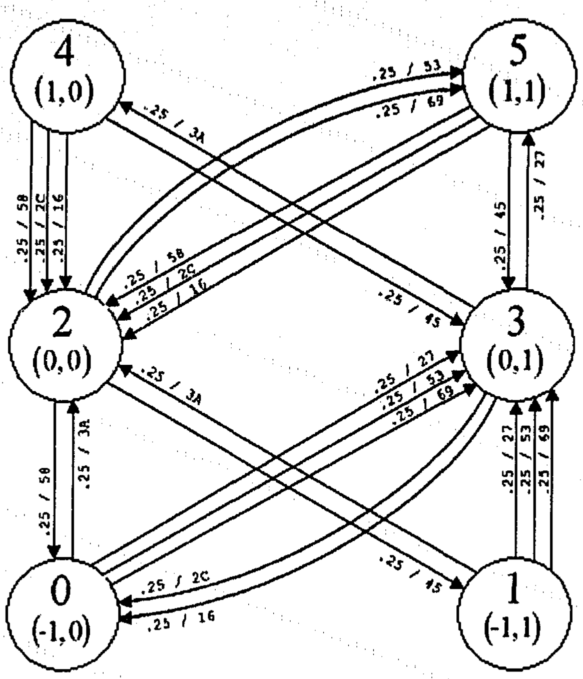


Figure 4-14: Markov Chain State Diagram

4.9. Analyzing the Encoder as a Markov Chain

Analyzing Figure 4-13 or Figure 4-14 shows that if the system is in state 0 it will move to state 2 with probability $1/4$ and move to state 3 with probability $3/4$. Likewise if the system is in state 2 it will move to state 0 with probability 1/4, move to state 1 with probability 1/4 and move to state 5 with probability 1/2. Continuing in this fashion all possible transition probabilities can be found and they are shown in Equation 4.19.

$$P = \begin{bmatrix} 0 & 0 & 1/4 & 3/4 & 0 & 0 \\ 0 & 0 & 1/4 & 3/4 & 0 & 0 \\ 1/4 & 1/4 & 0 & 0 & 0 & 1/2 \\ 1/2 & 0 & 0 & 0 & 1/4 & 1/4 \\ 0 & 0 & 3/4 & 1/4 & 0 & 0 \\ 0 & 0 & 3/4 & 1/4 & 0 & 0 \end{bmatrix} \qquad (4.19)$$

Another approach to finding this transition matrix is to use the definitions first presented by Cariolaro and Tronca [24]. Define the **input probability matrix** θ_u for $u = 1, 2, \ldots, S$, where S is the number of SWs as

$$P_u(SW \mid l_i)_{diag} = \theta_u(i,j) = \begin{cases} P_u(SW \mid l_i), & i = j \\ 0 & otherwise \end{cases} \qquad (4.20)$$

where l_i denotes the i^{th} state. Equation 4.20 is the probability of a particular SW occurring given the current state. Since there are 4 possible SWs and they are equally likely then there is a ¼ probability of any one of them being chosen regardless of the present state.

$$\theta_0 = \theta_1 = \theta_2 = \theta_3 = \begin{bmatrix} .25 & 0 & 0 & 0 & 0 & 0 \\ 0 & .25 & 0 & 0 & 0 & 0 \\ 0 & 0 & .25 & 0 & 0 & 0 \\ 0 & 0 & 0 & .25 & 0 & 0 \\ 0 & 0 & 0 & 0 & .25 & 0 \\ 0 & 0 & 0 & 0 & 0 & .25 \end{bmatrix} \qquad (4.21)$$

Next define E_u for $u = 1, 2, \ldots, S$ as the **next-state matrices,** where $E_u(i,j) = 1$ if and only if state l_j is entered from state l_i given input S_u. For each possible SW the system will move to only one state and therefore the E_u matrices will have a single 1 on each row. This is shown in Equation 4.22.

$$E_0 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad E_1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$E_2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{4.22}$$

By considering all the ways in which these state transitions can happen, and the probability of their occurrence, it is straightforward to verify that

$$P = \sum_{u=1}^{S} \theta_u E_u \tag{4.23}$$

Using Equation 4.23 the transition matrix for the encoder is found to be

$$P = \begin{bmatrix} .25 & 0 & 0 & 0 & 0 & 0 \\ 0 & .25 & 0 & 0 & 0 & 0 \\ 0 & 0 & .25 & 0 & 0 & 0 \\ 0 & 0 & 0 & .25 & 0 & 0 \\ 0 & 0 & 0 & 0 & .25 & 0 \\ 0 & 0 & 0 & 0 & 0 & .25 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 3 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 2 \\ 2 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 3 & 1 & 0 & 0 \\ 0 & 0 & 3 & 1 & 0 & 0 \end{bmatrix}$$

$$P = \begin{bmatrix} 0 & 0 & .25 & .75 & 0 & 0 \\ 0 & 0 & .25 & .75 & 0 & 0 \\ .25 & .25 & 0 & 0 & 0 & .5 \\ .5 & 0 & 0 & 0 & .25 & .25 \\ 0 & 0 & .75 & .25 & 0 & 0 \\ 0 & 0 & .75 & .25 & 0 & 0 \end{bmatrix} \tag{4.24}$$

Next define C_u for $u = 1, 2, \ldots, S$ as matrices, where the row $C_u(i)$ is the CW generated when state l_i is entered due to a given input S_u. For the example above:

$$C_0 = \begin{bmatrix} 0100111 \\ 0100111 \\ 1011000 \\ 0100111 \\ 1011000 \\ 1011000 \end{bmatrix} \quad C_1 = \begin{bmatrix} 1010011 \\ 1010011 \\ 1010011 \\ 0101100 \\ 0101100 \\ 0101100 \end{bmatrix}$$

$$C_2 = \begin{bmatrix} 1101001 \\ 1101001 \\ 1101001 \\ 0010110 \\ 0010110 \\ 0010110 \end{bmatrix} \quad C_3 = \begin{bmatrix} 0111010 \\ 0111010 \\ 1000101 \\ 0111010 \\ 1000101 \\ 1000101 \end{bmatrix} \tag{4.25}$$

With these definitions all information regarding the likelihood of entering a state and the CW emitted from each state is neatly formatted into a matrix representation. Note that the transition matrix P found in Equation 4.24 is the same as the transition matrix found by inspection in Equation 4.19. P can now be used to find the invariant probability vector π.

4.9.1. Finding the invariant probability vector of the encoder

From Section 4.8.3 and Equation 4.15 the invariant probability vector π can be found for any initial starting distribution v by evaluating vP^n as n approaches infinity. The rows of P^n approach the invariant distribution for large n for regular stochastic matrices satisfying the property $\pi = \pi P$. However for the encoder described above, taking P^n to large powers of n shows that it alternates between the two matrices below for odd and even powers of n.

$$P^{30} = \begin{bmatrix} \frac{3}{8} & \frac{1}{8} & 0 & 0 & \frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{1}{8} & 0 & 0 & \frac{1}{8} & \frac{3}{8} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{3}{8} & \frac{1}{8} & 0 & 0 & \frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{1}{8} & 0 & 0 & \frac{1}{8} & \frac{3}{8} \end{bmatrix} \qquad P^{31} = \begin{bmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{3}{8} & \frac{1}{8} & 0 & 0 & \frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{1}{8} & 0 & 0 & \frac{1}{8} & \frac{3}{8} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{bmatrix} \qquad (4.26)$$

This system has a chain of $d = 2$ and will cycle between these d matrices. This is because the Markov chain is not regular, and it cannot be expected to converge. The invariant distribution as found by Equation 4.17 is

$$M = \left(\frac{1}{2}\right) \begin{bmatrix} \frac{3}{8} & \frac{1}{8} & \frac{1}{2} & \frac{1}{2} & \frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{1}{8} & \frac{1}{2} & \frac{1}{2} & \frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{1}{8} & \frac{1}{2} & \frac{1}{2} & \frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{1}{8} & \frac{1}{2} & \frac{1}{2} & \frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{1}{8} & \frac{1}{2} & \frac{1}{2} & \frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & \frac{1}{8} & \frac{1}{2} & \frac{1}{2} & \frac{1}{8} & \frac{3}{8} \end{bmatrix} = \begin{bmatrix} 3/16 & 1/16 & 1/4 & 1/4 & 1/16 & 3/16 \\ 3/16 & 1/16 & 1/4 & 1/4 & 1/16 & 3/16 \\ 3/16 & 1/16 & 1/4 & 1/4 & 1/16 & 3/16 \\ 3/16 & 1/16 & 1/4 & 1/4 & 1/16 & 3/16 \\ 3/16 & 1/16 & 1/4 & 1/4 & 1/16 & 3/16 \\ 3/16 & 1/16 & 1/4 & 1/4 & 1/16 & 3/16 \end{bmatrix}$$

$$\pi = \begin{bmatrix} 3/16 & 1/16 & 1/4 & 1/4 & 1/16 & 3/16 \end{bmatrix} \qquad (4.27)$$

This means that if the system was running for some amount of time and the number of times each state was visited was recorded, it would be found that the invariant distribution is $\pi = \begin{bmatrix} 3/16 & 1/16 & 1/4 & 1/4 & 1/16 & 3/16 \end{bmatrix} = \begin{bmatrix} .1875 & .0625 & .25 & .25 & .0625 & .1875 \end{bmatrix}$, which is the probability of being in any particular state. Combining this information with the state diagram shown in Figure 4-15, it can be seen that the system is in the two most popular states, states 2 and 3 that correspond to an RDS of 0, 50% of the time.

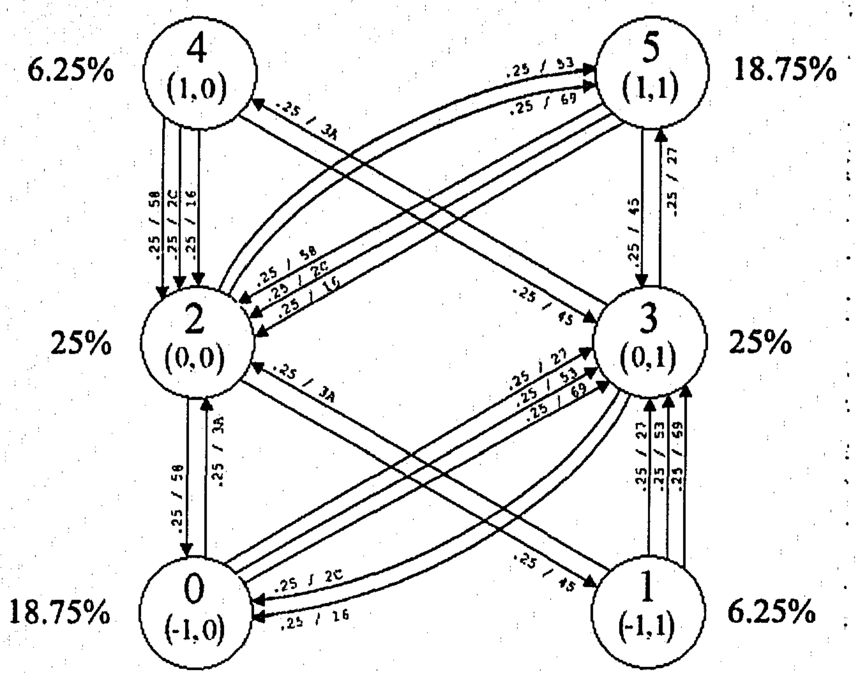


Figure 4-15: Markov Chain State Diagram with State Probabilities

Using Markov chain analysis it is now possible to evaluate the likelihood of a particular CW being sent. On average the probability of a CW being transmitted from any state is ¼, weighted by the probability of being in the state that emitted the CW.

For example, the probability of being in state 0 is 18.75%. Therefore the CWs emitted from this state {27h,53h,69h,3Ah} would be seen on the channel (18.75%)(25%) or 4.68% of the time. Likewise the probability of being in state 1 is 6.25%. Therefore even though the CWs emitted from this state are the same{27h,53h,69h,3Ah}, they would only be seen on the channel (6.25%)(25%) or 1.56% of the time. Continuing in this manner the likelihood of all possible CWs being transmitted can be calculated. With this information it is now possible to characterize the encoder in terms of runlength probabilities.

4.10. Finding Runlength Probabilities

To find the runlength probabilities it is convenient to decompose the state diagram into the format shown in Figure 4-16. From this representation it easy to see the probability of specific CWs as the encoder enters and leaves states.

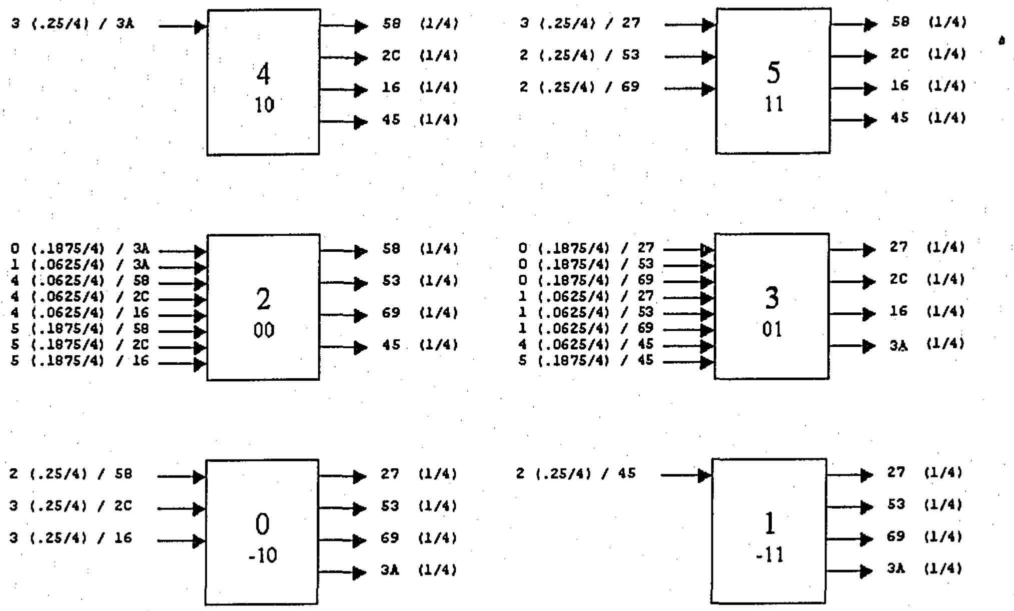


Figure 4-16: Decomposed state diagram illustrating the probability of a SW/CW entering each state, and probability of each CW leaving each state, allowing straightforward calculation of runlengths

For example, Figure 4-16 shows that CW 58h enters state 0 with probability $(.25)(.25)=.0625$. This is because CW 58h is one of four possible CWs that can be emitted from state 2, which the system visits on average 25% of the time. Once the system is in state 0 there are 4 possible CWs $\{27h,53h,69h,3Ah\}$ that can be emitted all with probability ¼ . Hence the probability of the particular sequence 58h followed by 27h is $(.25)(.25)(.25)=.015625$.

In order to accurately evaluate runlength probabilities the concatenation of CWs must be taken into account. For example Figure 4-17 shows the sequence formed through the concatenation of the two CWs 58h and 27h.

$$10110000100111 \longleftarrow 5827$$

$$1011\,\boxed{0000100}\,111 \longleftarrow \text{Portion of interest}$$

Figure 4-17: Concatenation of two CWs and the Portion of Interest

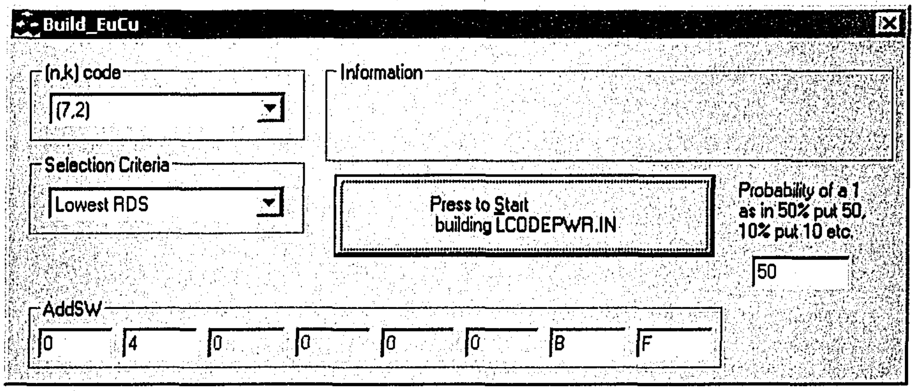
Note that CWs 58h and 27h alone only have a worst case runlength of 3 like valued bits. However once they are concatenated a runlength of 4 like valued bits appears. Therefore the string of three 0s at the end of CW 58h cannot be counted simply as a run of three, since it may lead to a longer run of 0s when concatenated with the next CW. Likewise the final string of three 1s in CW 27h is not counted as a run of three either since it may lead to a longer run of 1s when it is concatenated to the next CW. As shown in Figure 4-17 the middle portion of the concatenated CWs is called the **portion of interest**.

4.11. Analysis of runlength probabilities

As previously discussed the probability of this particular two CW sequence 58h followed by 27h is $(.25)(.25)(.25)=.015625$ or (.25)*¼*¼. The probability of being within the seven bit **portion of interest** out of these fourteen bits is 7/14. Inspection shows that there is one run of 1, one run of 2 and one run of 4 that will occur with probability 1/7, 2/7 and 4/7 respectively. Finally since the portion of interest spans two CWs, and the particular run of interest could commence at any encoding interval, including the n^{th} and $(n+1)^{th}$ intervals, then the above sequence probability must be multiplied by two. For this example then,

$$\begin{aligned}
\text{Run of } 1 &= .25\left(\frac{1}{4}\right)\left(\frac{1}{4}\right)2\left(\frac{7}{14}\right)\frac{1}{7} = \frac{1}{448} = .002232 \\
\text{Run of } 2 &= .25\left(\frac{1}{4}\right)\left(\frac{1}{4}\right)2\left(\frac{7}{14}\right)\frac{2}{7} = \frac{2}{448} = .004464 \\
\text{Run of } 4 &= .25\left(\frac{1}{4}\right)\left(\frac{1}{4}\right)2\left(\frac{7}{14}\right)\frac{4}{7} = \frac{4}{448} = .008929
\end{aligned} \tag{4.28}$$

Enumeration like this is required for all possible two CW combinations, and even for this simple system there are 96 possible two CW combinations. Since this calculation is best performed by a computer, two programs were written.


Build_EuCu

(n,k) code: (7,2)

Selection Criteria: Lowest RDS

Information

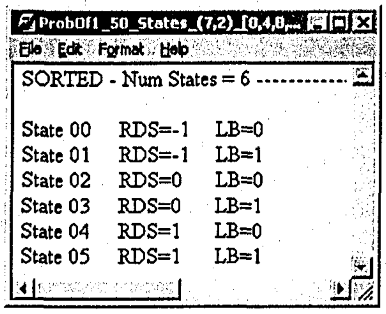
Press to Start building LCODEPWR.IN

Probability of a 1 as in 50% put 50, 10% put 10 etc.: 50

AddSW: 0 | 4 | 0 | 0 | 0 | 0 | B | F

Figure 4-18: Build_EuCu.exe Program that builds 0_u, E_u and C_u matrices

The first program **Build_EuCu.exe** is shown in Figure 4-18 and it allows the user to select the expurgated code, selection criteria, source statistics and AddSWs. It then generates three files. The first file generated is shown below in Figure 4-19 called **ProbOf1_50_States_(7,2)_[0,4,B,F]_Lowest_RDS.txt**, and it shows the RDS and LB states generated by the code. These states are found by testing all SW/CW combinations and they are sorted from lowest to highest to show the symmetry in the system. Inspection shows that these are the same states from Figure 4-15.



```
SORTED - Num States = 6 -----------

State 00   RDS=-1   LB=0
State 01   RDS=-1   LB=1
State 02   RDS=0    LB=0
State 03   RDS=0    LB=1
State 04   RDS=1    LB=0
State 05   RDS=1    LB=1
```

Figure 4-19: Output states from Build_EuCu.exe

Two other files are generated by this program. One is used in the evaluation of the power spectral density that will not be considered until Chapter 5, and the other contains the E_u and C_u matrices shown in Figure 4-20.

SW 0		SW 1		SW 2		SW 3	
3	27	3	53	3	69	2	3A
3	27	3	53	3	69	2	3A
0	58	5	53	5	69	1	45
5	27	0	2C	0	16	4	3A
2	58	2	2C	2	16	3	45
2	58	2	2C	2	16	3	45

Figure 4-20: Output E_u and C_u matrices from Build_EuCu.exe

This file lists the $E_0, C_0, E_1, C_1, E_2, C_2, E_3, C_3$ matrices as shown in Equations 4.22 and 4.25 respectively. The difference is that instead of writing the information in binary matrix form, it is written in a compact hexadecimal form. The reason for this compact notation is because this data is read into the second program **CalcPIMandProbs.exe** which is shown in Figure 4-21. This program parses the E_u and C_u information to calculate the transition matrix, invariant probability vector and runlength probabilities. The compact form may not seem necessary for this (7,2) code which only had six states and four possible SWs. However large codes like the (15,x) and (31,x) codes could have thousands of CWs and hundreds of states and hence this file can grow to hundreds of megabytes. Therefore the most compact form possible was adopted.

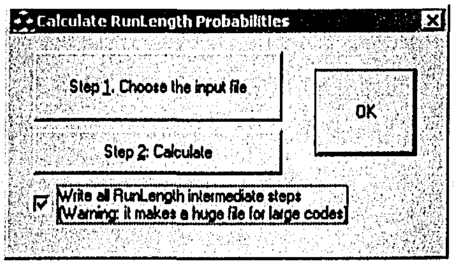
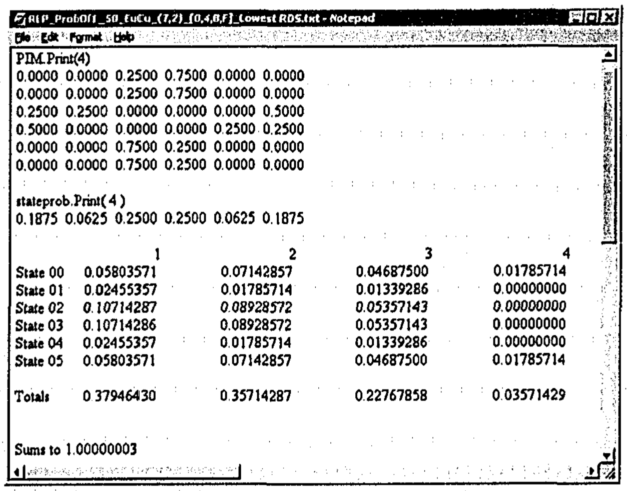

Calculate RunLength Probabilities
Step 1. Choose the input file
Step 2: Calculate
OK
Write all RunLength intermediate steps
Warning: it makes a huge file for large codes

Figure 4-21: CalcPIMandProbs.exe

The information in this file is read in a top to bottom fashion as follows. Looking at the first line in Figure 4-20, when SW 0 is chosen and the system is in state 0, it moves to state 3 and emits CW 27h. Likewise on the next line down, when SW 0 is chosen and the system is in state 1, it moves to state 3 and also emits CW 27h and so on.

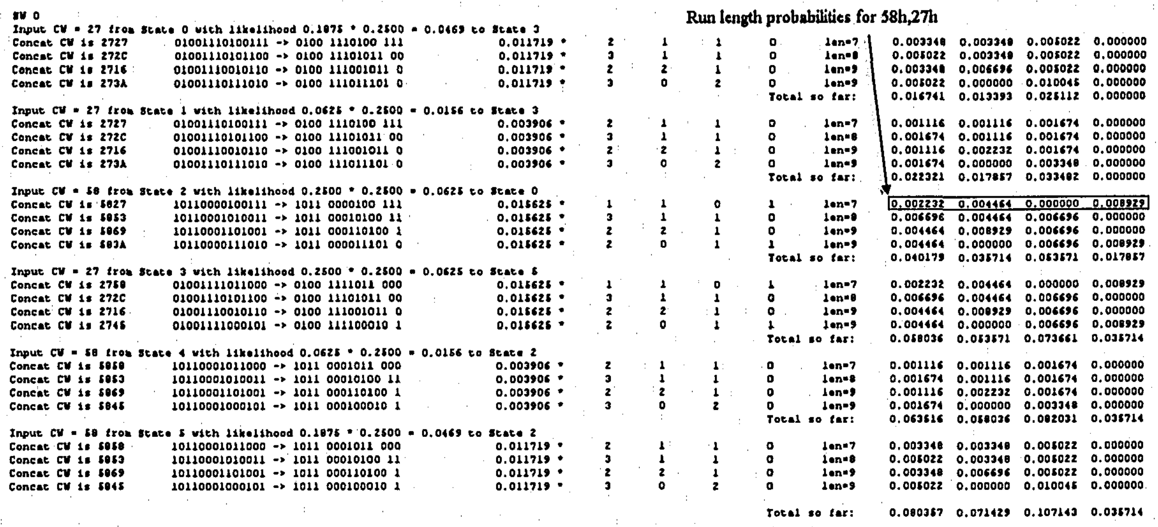
CalcPIMandProbs.exe takes this information and generates a single file that is shown in Figure 4-22. Inspection shows that it has calculated the same transition matrix as found in Equation 4-24. Furthermore it has calculated the same invariant vector as found in Equation 4-27 and finally it has calculated all the runlength probabilities. It shows that this system with AddSWs {0h,4h,Bh,Fh} using the lowest RDS selection criteria has on average a 37.94% chance of a run of one, 35.71% chance of a run of two, 22.76% chance of a run of three and a 3.57% chance of a run of four. Runlengths of five or more will never occur.


RLP_ProbOff_50_EuCu_(7,2)_[0,4,B,F]_Lowest RDS.txt - Notepad
File Edit Format Help

```
PIM.Print(4)
0.0000 0.0000 0.2500 0.7500 0.0000 0.0000
0.0000 0.0000 0.2500 0.7500 0.0000 0.0000
0.2500 0.2500 0.0000 0.0000 0.0000 0.5000
0.5000 0.0000 0.0000 0.0000 0.2500 0.2500
0.0000 0.0000 0.7500 0.2500 0.0000 0.0000
0.0000 0.0000 0.7500 0.2500 0.0000 0.0000

stateprob.Print( 4 )
0.1875 0.0625 0.2500 0.2500 0.0625 0.1875

                     1                2                3                4
State 00   0.05803571       0.07142857       0.04687500       0.01785714
State 01   0.02455357       0.01785714       0.01339286       0.00000000
State 02   0.10714287       0.08928572       0.05357143       0.00000000
State 03   0.10714286       0.08928572       0.05357143       0.00000000
State 04   0.02455357       0.01785714       0.01339286       0.00000000
State 05   0.05803571       0.07142857       0.04687500       0.01785714

Totals     0.37946430       0.35714287       0.22767858       0.03571429


Sums to 1.00000003
```

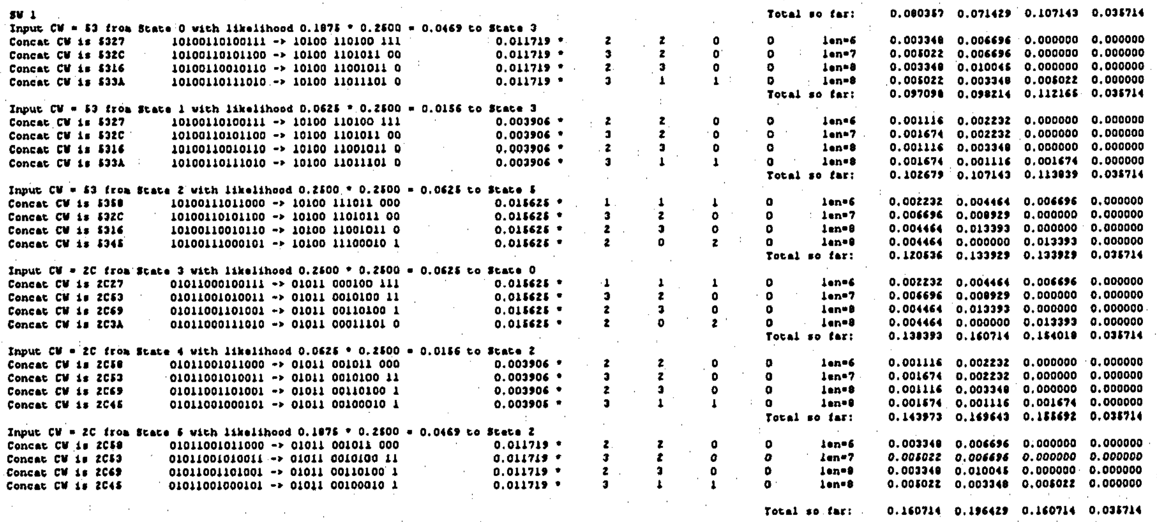
Figure 4-22: Output from CalcPIMandProbs.exe

An additional feature of CalcPIMandProbs is that it can also show the intermediate steps of how the runlengths were calculated. This feature is only feasible for small codes because the file size grows very quickly. Nevertheless the intermediate steps are shown in Figures 4-23 to 4-26 where it can be seen that the same runlength probabilities are calculated as shown in Equation 4.28.



```
                                                                            Run length probabilities for 58h,27h
SW 0
Input CW = 27 from State 0 with likelihood 0.1875 * 0.2500 = 0.0469 to State 3
Concat CW is 2727     0100111010011l -> 0100 1110100 111     0.011719 *   2   1   1   0   len=7    0.003348  0.003348  0.005022  0.000000
Concat CW is 272C     0100111010110 0 -> 0100 11101011 00    0.011719 *   3   1   1   0   len=8    0.005022  0.003348  0.005022  0.000000
Concat CW is 2716     01001110010110 -> 0100 111001011 0     0.011719 *   2   2   1   0   len=9    0.003348  0.006696  0.005022  0.000000
Concat CW is 273A     01001110111010 -> 0100 111011101 0     0.011719 *   3   0   2   0   len=9    0.005022  0.000000  0.010045  0.000000
                                                                                      Total so far:    0.016741  0.013393  0.025112  0.000000

Input CW = 27 from State 1 with likelihood 0.0625 * 0.2500 = 0.0156 to State 3
Concat CW is 2727     01001110100111 -> 0100 1110100 111     0.003906 *   2   1   1   0   len=7    0.001116  0.001116  0.001674  0.000000
Concat CW is 272C     01001110101100 -> 0100 11101011 00     0.003906 *   3   1   1   0   len=8    0.001674  0.001116  0.001674  0.000000
Concat CW is 2716     01001110010110 -> 0100 111001011 0     0.003906 *   2   2   1   0   len=9    0.001116  0.002232  0.001674  0.000000
Concat CW is 273A     01001110111010 -> 0100 111011101 0     0.003906 *   3   0   2   0   len=9    0.001674  0.000000  0.003348  0.000000
                                                                                      Total so far:    0.022321  0.017857  0.033482  0.000000

Input CW = 58 from State 2 with likelihood 0.2500 * 0.2500 = 0.0625 to State 0
Concat CW is 5827     10110000100111 -> 1011 0000100 111     0.015625 *   1   1   0   1   len=7    0.002232  0.004464  0.000000  0.008929
Concat CW is 5853     10110001010011 -> 1011 00010100 11     0.015625 *   3   1   1   0   len=8    0.006696  0.004464  0.006696  0.000000
Concat CW is 5869     10110001101001 -> 1011 000110100 1     0.015625 *   2   2   1   0   len=9    0.004464  0.008929  0.006696  0.000000
Concat CW is 583A     10110000111010 -> 1011 000011101 0     0.015625 *   2   0   1   1   len=9    0.004464  0.000000  0.006696  0.008929
                                                                                      Total so far:    0.040179  0.035714  0.053571  0.017857

Input CW = 27 from State 3 with likelihood 0.2500 * 0.2500 = 0.0625 to State 5
Concat CW is 2758     01001111011000 -> 0100 1111011 000     0.015625 *   1   1   0   1   len=7    0.002232  0.004464  0.000000  0.008929
Concat CW is 272C     01001110101100 -> 0100 11101011 00     0.015625 *   3   1   1   0   len=8    0.006696  0.004464  0.006696  0.000000
Concat CW is 2716     01001110010110 -> 0100 111001011 0     0.015625 *   2   2   1   0   len=9    0.004464  0.008929  0.006696  0.000000
Concat CW is 2745     01001111000101 -> 0100 111100010 1     0.015625 *   2   0   1   1   len=9    0.004464  0.000000  0.006696  0.008929
                                                                                      Total so far:    0.058036  0.053571  0.073661  0.035714

Input CW = 58 from State 4 with likelihood 0.0625 * 0.2500 = 0.0156 to State 2
Concat CW is 5858     10110001011000 -> 1011 0001011 000     0.003906 *   2   1   1   0   len=7    0.001116  0.001116  0.001674  0.000000
Concat CW is 5853     10110001010011 -> 1011 00010100 11     0.003906 *   3   1   1   0   len=8    0.001674  0.001116  0.001674  0.000000
Concat CW is 5869     10110001101001 -> 1011 000110100 1     0.003906 *   2   2   1   0   len=9    0.001116  0.002232  0.001674  0.000000
Concat CW is 5845     10110001000101 -> 1011 000100010 1     0.003906 *   3   0   2   0   len=9    0.001674  0.000000  0.003348  0.000000
                                                                                      Total so far:    0.063616  0.058036  0.082031  0.035714

Input CW = 58 from State 5 with likelihood 0.1875 * 0.2500 = 0.0469 to State 2
Concat CW is 5858     10110001011000 -> 1011 0001011 000     0.011719 *   2   1   1   0   len=7    0.003348  0.003348  0.005022  0.000000
Concat CW is 5853     10110001010011 -> 1011 00010100 11     0.011719 *   3   1   1   0   len=8    0.005022  0.003348  0.005022  0.000000
Concat CW is 5869     10110001101001 -> 1011 000110100 1     0.011719 *   2   2   1   0   len=9    0.003348  0.006696  0.005022  0.000000
Concat CW is 5845     10110001000101 -> 1011 000100010 1     0.011719 *   3   0   2   0   len=9    0.005022  0.000000  0.010045  0.000000

                                                                                      Total so far:    0.080357  0.071429  0.107143  0.035714
```

Figure 4-23: Runlength probabilities with SW 0



```
SW 1                                                                                  Total so far:    0.080357  0.071429  0.107143  0.035714
Input CW = 53 from State 0 with likelihood 0.1875 * 0.2500 = 0.0469 to State 3
Concat CW is 5327     10100110100111 -> 10100 110100 111     0.011719 *   2   2   0   0   len=6    0.003348  0.006696  0.000000  0.000000
Concat CW is 532C     10100110101100 -> 10100 1101011 00    0.011719 *   3   2   0   0   len=7    0.005022  0.006696  0.000000  0.000000
Concat CW is 5316     10100110010110 -> 10100 11001011 0     0.011719 *   2   3   0   0   len=8    0.003348  0.010045  0.000000  0.000000
Concat CW is 533A     10100110111010 -> 10100 11011101 0     0.011719 *   3   1   1   0   len=8    0.005022  0.003348  0.005022  0.000000
                                                                                      Total so far:    0.097098  0.098214  0.112165  0.035714

Input CW = 53 from State 1 with likelihood 0.0625 * 0.2500 = 0.0156 to State 3
Concat CW is 5327     10100110100111 -> 10100 110100 111     0.003906 *   2   2   0   0   len=6    0.001116  0.002232  0.000000  0.000000
Concat CW is 532C     10100110101100 -> 10100 1101011 00    0.003906 *   3   2   0   0   len=7    0.001674  0.002232  0.000000  0.000000
Concat CW is 5316     10100110010110 -> 10100 11001011 0     0.003906 *   2   3   0   0   len=8    0.001116  0.003348  0.000000  0.000000
Concat CW is 533A     10100110111010 -> 10100 11011101 0     0.003906 *   3   1   1   0   len=8    0.001674  0.001116  0.001674  0.000000
                                                                                      Total so far:    0.102679  0.107143  0.113839  0.035714

Input CW = 53 from State 2 with likelihood 0.2500 * 0.2500 = 0.0625 to State 5
Concat CW is 5358     10100111011000 -> 10100 111011 000     0.015625 *   1   1   1   0   len=6    0.002232  0.004464  0.006696  0.000000
Concat CW is 532C     10100110101100 -> 10100 1101011 00    0.015625 *   3   2   0   0   len=7    0.006696  0.008929  0.000000  0.000000
Concat CW is 5316     10100110010110 -> 10100 11001011 0     0.015625 *   2   3   0   0   len=8    0.004464  0.013393  0.000000  0.000000
Concat CW is 5345     10100111000101 -> 10100 11100010 1     0.015625 *   2   0   2   0   len=8    0.004464  0.000000  0.013393  0.000000
                                                                                      Total so far:    0.120536  0.133929  0.133929  0.035714

Input CW = 2C from State 3 with likelihood 0.2500 * 0.2500 = 0.0625 to State 0
Concat CW is 2C27     01011000100111 -> 01011 000100 111     0.015625 *   1   1   1   0   len=6    0.002232  0.004464  0.006696  0.000000
Concat CW is 2C53     01011001010011 -> 01011 0010100 11    0.015625 *   3   2   0   0   len=7    0.006696  0.008929  0.000000  0.000000
Concat CW is 2C69     01011001101001 -> 01011 00110100 1     0.015625 *   2   3   0   0   len=8    0.004464  0.013393  0.000000  0.000000
Concat CW is 2C3A     01011000111010 -> 01011 00011101 0     0.015625 *   2   0   2   0   len=8    0.004464  0.000000  0.013393  0.000000
                                                                                      Total so far:    0.138393  0.160714  0.154018  0.035714

Input CW = 2C from State 4 with likelihood 0.0625 * 0.2500 = 0.0156 to State 2
Concat CW is 2C58     01011001011000 -> 01011 001011 000     0.003906 *   2   2   0   0   len=6    0.001116  0.002232  0.000000  0.000000
Concat CW is 2C53     01011001010011 -> 01011 0010100 11    0.003906 *   3   2   0   0   len=7    0.001674  0.002232  0.000000  0.000000
Concat CW is 2C69     01011001101001 -> 01011 00110100 1     0.003906 *   2   3   0   0   len=8    0.001116  0.003348  0.000000  0.000000
Concat CW is 2C45     01011001000101 -> 01011 00100010 1     0.003905 *   3   1   1   0   len=8    0.001574  0.001116  0.001674  0.000000
                                                                                      Total so far:    0.143973  0.169643  0.155692  0.035714

Input CW = 2C from State 5 with likelihood 0.1875 * 0.2500 = 0.0469 to State 2
Concat CW is 2C58     01011001011000 -> 01011 001011 000     0.011719 *   2   2   0   0   len=6    0.003348  0.006696  0.000000  0.000000
Concat CW is 2C53     01011001010011 -> 01011 0010100 11    0.011719 *   3   2   0   0   len=7    0.005022  0.006696  0.000000  0.000000
Concat CW is 2C69     01011001101001 -> 01011 00110100 1     0.011719 *   2   3   0   0   len=8    0.003348  0.010045  0.000000  0.000000
Concat CW is 2C45     01011001000101 -> 01011 00100010 1     0.011719 *   3   1   1   0   len=8    0.005022  0.003348  0.005022  0.000000

                                                                                      Total so far:    0.160714  0.196429  0.160714  0.035714
```

Figure 4-24: Runlength probabilities with SW 1

```
SW 2
Input CW = 69 from State 0 with likelihood 0.1875 * 0.2500 = 0.0469 to State 3
Concat CW is 6927      11010010100111 -> 110100 10100 111       0.011719 *   3   1   0   0     len=5     0.005022  0.003348  0.000000  0.000000
Concat CW is 692C      11010010101100 -> 110100 101011 00       0.011719 *   4   1   0   0     len=6     0.006696  0.003348  0.000000  0.000000
Concat CW is 6916      11010010010110 -> 110100 1001011 0       0.011719 *   3   2   0   0     len=7     0.005022  0.006696  0.000000  0.000000
Concat CW is 693A      11010010111010 -> 110100 1011101 0       0.011719 *   4   0   1   0     len=7     0.006696  0.000000  0.005022  0.000000
                                                                                        Total so far:   0.184152  0.209821  0.165737  0.035714

Input CW = 69 from State 1 with likelihood 0.0625 * 0.2500 = 0.0156 to State 3
Concat CW is 6927      11010010100111 -> 110100 10100 111       0.003906 *   3   1   0   0     len=5     0.001674  0.001116  0.000000  0.000000
Concat CW is 692C      11010010101100 -> 110100 101011 00       0.003906 *   4   1   0   0     len=6     0.002232  0.001116  0.000000  0.000000
Concat CW is 6916      11010010010110 -> 110100 1001011 0       0.003906 *   3   2   0   0     len=7     0.001674  0.002232  0.000000  0.000000
Concat CW is 693A      11010010111010 -> 110100 1011101 0       0.003906 *   4   0   1   0     len=7     0.002232  0.000000  0.001674  0.000000
                                                                                        Total so far:   0.191964  0.214285  0.167411  0.035714

Input CW = 69 from State 2 with likelihood 0.2500 * 0.2500 = 0.0625 to State 5
Concat CW is 6958      11010011011000 -> 110100 11011 000       0.015625 *   1   2   0   0     len=5     0.002232  0.008929  0.000000  0.000000
Concat CW is 692C      11010010101100 -> 110100 101011 00       0.015625 *   4   1   0   0     len=6     0.008929  0.004464  0.000000  0.000000
Concat CW is 6916      11010010010110 -> 110100 1001011 0       0.015625 *   3   2   0   0     len=7     0.006696  0.008929  0.000000  0.000000
Concat CW is 6945      11010011000101 -> 110100 1100010 1       0.015625 *   2   1   1   0     len=7     0.004464  0.004464  0.006696  0.000000
                                                                                        Total so far:   0.214286  0.241071  0.174107  0.035714

Input CW = 16 from State 3 with likelihood 0.2500 * 0.2500 = 0.0625 to State 0
Concat CW is 1627      00101100100111 -> 001011 00100 111       0.015625 *   1   2   0   0     len=5     0.002232  0.008929  0.000000  0.000000
Concat CW is 1653      00101101010011 -> 001011 010100 11       0.015625 *   4   1   0   0     len=6     0.008929  0.004464  0.000000  0.000000
Concat CW is 1669      00101101101001 -> 001011 0110100 1       0.015625 *   3   2   0   0     len=7     0.006696  0.008929  0.000000  0.000000
Concat CW is 163A      00101100111010 -> 001011 0011101 0       0.015625 *   2   1   1   0     len=7     0.004464  0.004464  0.006696  0.000000
                                                                                        Total so far:   0.236607  0.267857  0.180804  0.035714

Input CW = 16 from State 4 with likelihood 0.0625 * 0.2500 = 0.0156 to State 2
Concat CW is 1658      00101101011000 -> 001011 01011 000       0.003906 *   3   1   0   0     len=5     0.001674  0.001116  0.000000  0.000000
Concat CW is 1653      00101101010011 -> 001011 010100 11       0.003906 *   4   1   0   0     len=6     0.002232  0.001116  0.000000  0.000000
Concat CW is 1669      00101101101001 -> 001011 0110100 1       0.003906 *   3   2   0   0     len=7     0.001674  0.002232  0.000000  0.000000
Concat CW is 1645      00101101000101 -> 001011 0100010 1       0.003906 *   4   0   1   0     len=7     0.002232  0.000000  0.001674  0.000000
                                                                                        Total so far:   0.244420  0.272321  0.182478  0.035714

Input CW = 16 from State 5 with likelihood 0.1875 * 0.2500 = 0.0469 to State 2
Concat CW is 1658      00101101011000 -> 001011 01011 000       0.011719 *   3   1   0   0     len=5     0.005022  0.003348  0.000000  0.000000
Concat CW is 1653      00101101010011 -> 001011 010100 11       0.011719 *   4   1   0   0     len=6     0.006696  0.003348  0.000000  0.000000
Concat CW is 1669      00101101101001 -> 001011 0110100 1       0.011719 *   3   2   0   0     len=7     0.005022  0.006696  0.000000  0.000000
Concat CW is 1645      00101101000101 -> 001011 0100010 1       0.011719 *   4   0   1   0     len=7     0.006696  0.000000  0.005022  0.000000

                                                                                        Total so far:   0.267857  0.285714  0.187500  0.035714
```

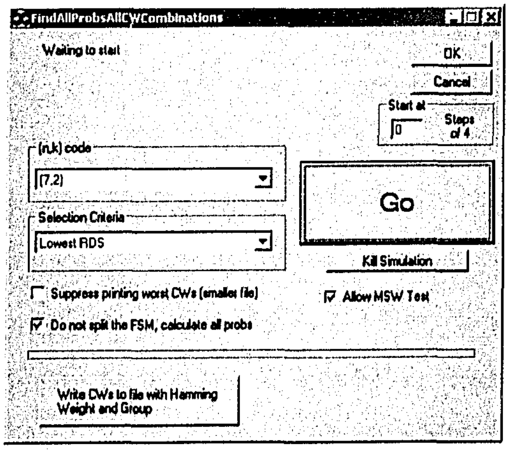
Figure 4-25: Runlength probabilities with SW 2

```
SW 3
Input CW = 3A from State 0 with likelihood 0.1875 * 0.2500 = 0.0469 to State 2
Concat CW is 3A58      01110101011000 -> 011101 01011 000       0.011719 *   3   1   0   0     len=5     0.005022  0.003348  0.000000  0.000000
Concat CW is 3A53      01110101010011 -> 011101 010100 11       0.011719 *   4   1   0   0     len=6     0.006696  0.003348  0.000000  0.000000
Concat CW is 3A69      01110101101001 -> 011101 0110100 1       0.011719 *   3   2   0   0     len=7     0.005022  0.006696  0.000000  0.000000
Concat CW is 3A45      01110101000101 -> 011101 0100010 1       0.011719 *   4   0   1   0     len=7     0.006696  0.000000  0.005022  0.000000
                                                                                        Total so far:   0.291295  0.299107  0.192522  0.035714

Input CW = 3A from State 1 with likelihood 0.0625 * 0.2500 = 0.0156 to State 2
Concat CW is 3A58      01110101011000 -> 011101 01011 000       0.003906 *   3   1   0   0     len=5     0.001674  0.001116  0.000000  0.000000
Concat CW is 3A53      01110101010011 -> 011101 010100 11       0.003906 *   4   1   0   0     len=6     0.002232  0.001116  0.000000  0.000000
Concat CK is 3A69      01110101101001 -> 011101 0110100 1       0.003906 *   3   2   0   0     len=7     0.001674  0.002232  0.000000  0.000000
Concat CW is 3A45      01110101000101 -> 011101 0100010 1       0.003906 *   4   0   1   0     len=7     0.002232  0.000000  0.001674  0.000000
                                                                                        Total so far:   0.299107  0.303571  0.194196  0.035714

Input CW = 45 from State 2 with likelihood 0.2500 * 0.2500 = 0.0625 to State 1
Concat CW is 4527      10001010100111 -> 100010 10100 111       0.015625 *   3   1   0   0     len=5     0.006696  0.004464  0.000000  0.000000
Concat CW is 4553      10001011010011 -> 100010 110100 11       0.015625 *   2   2   0   0     len=6     0.004464  0.008929  0.000000  0.000000
Concat CW is 4569      10001011101001 -> 100010 1110100 1       0.015625 *   2   1   1   0     len=7     0.004464  0.004464  0.006696  0.000000
Concat CW is 453A      10C01010111010 -> 100010 1011101 0       0.015625 *   4   0   1   0     len=7     0.008929  0.000000  0.006696  0.000000
                                                                                        Total so far:   0.323661  0.321429  0.207589  0.035714

Input CW = 3A from State 3 with likelihood 0.2500 * 0.2500 = 0.0625 to State 4
Concat CW is 3A58      01110101011000 -> 011101 01011 000       0.015625 *   3   1   0   0     len=5     0.006696  0.004464  0.000000  0.000000
Concat CW is 3A2C      01110100101100 -> 011101 001011 00       0.015625 *   2   2   0   0     len=6     0.004464  0.008929  0.000000  0.000000
Concat CW is 3A16      01110100010110 -> 011101 0001011 0       0.015625 *   2   1   1   0     len=7     0.004464  0.004464  0.006696  0.000000
Concat CW is 3A45      01110101000101 -> 011101 0100010 1       0.015625 *   4   0   1   0     len=7     0.008929  0.000000  0.006696  0.000000
                                                                                        Total so far:   0.348214  0.339286  0.220982  0.035714

Input CW = 45 from State 4 with likelihood 0.0625 * 0.2500 = 0.0156 to State 3
Concat CW is 4527      10001010100111 -> 100010 10100 111       0.003906 *   3   1   0   0     len=5     0.001674  0.001116  0.000000  0.000000
Concat CW is 452C      10001010101100 -> 100010 101011 00       0.003906 *   4   1   0   0     len=6     0.002232  0.001116  0.000000  0.000000
Concat CW is 4516      10001010010110 -> 100010 1001011 0       0.003906 *   3   2   0   0     len=7     0.001674  0.002232  0.000000  0.000000
Concat CW is 453A      10001010111010 -> 100010 1011101 0       0.003906 *   4   0   1   0     len=7     0.002232  0.000000  0.001674  0.000000
                                                                                        Total so far:   0.356027  0.343750  0.222656  0.035714

Input CW = 45 from State 5 with likelihood 0.1875 * 0.2500 = 0.0469 to State 3
Concat CW is 4527      10001010100111 -> 100010 10100 111       0.011719 *   3   1   0   0     len=5     0.005022  0.003348  0.000000  0.000000
Concat CW is 452C      10001010101100 -> 100010 101011 00       0.011719 *   4   1   0   0     len=6     0.006696  0.003348  0.000000  0.000000
Concat CW is 4516      10001010010110 -> 100010 1001011 0       0.011719 *   3   2   0   0     len=7     0.005022  0.006696  0.000000  0.000000
Concat CW is 453A      10001010111010 -> 100010 1011101 0       0.011719 *   4   0   1   0     len=7     0.006696  0.000000  0.005022  0.000000
                                                                                        Total so far:   0.379464  0.357143  0.227679  0.035714
```

Figure 4-26: Runlength probabilities with SW 3

4.12. Code Word Search

In order to determine the best AddCW sets, the effect that different AddCW sets have on the number of transitions and runlength probabilities must be evaluated. To do so involves testing each set of AddCWs using the above Markov chain analysis and searching for the set that has the best statistics. This was done through an exhaustive computer search. For example, for the (7,1) code the exhaustive search will begin with the AddSW set {0h,2h,4h,6h,9h,Bh,Dh,Fh}, since SWs 2h, 4h and 6h are the first SWs to meet the selection criteria from Section 3.5 and Figure 3-21. Furthermore SWs 9h, Bh and Dh are chosen because they are the SW complements of 2h, 4h and 6h, and SWs 0h and Fh are chosen since it was recommended that all AddCW sets include the all-zero and all-one CW. Note that this AddSW set must be encoded into its corresponding AddCW set. After gathering statistics on this AddSW set the exhaustive search will increment SW 6h to the next SW 7h, whose complement is 8h, and as a result the AddSW set {0h,2h,4h,7h,8h,Bh,Dh,Fh} will be tested. In this fashion the computer search would continue testing the following AddSW sets {0h,2h,5h,6h,9h,Ah,Dh,Fh}, {0h,2h,5h,7h,8h,Ah,Dh,Fh}, {0h,3h,4h,6h,9h,Bh,Ch,Fh}, {0h,3h,4h,7h,8h,Bh,Ch,Fh}, {0h,3h,5h,6h,9h,Ah,Ch,Fh} and {0h,3h,5h,7h,8h,Ah,Ch,Fh}, for testing a total of eight sets. Note that the total number of AddSW sets tested agrees with the number given by Equation 3.1.



FindAllProbsAllCWCombinations

Waiting to start

Start at: 0 Steps of 4

(n,k) code: (7,2)

Selection Criteria: Lowest RDS

[ ] Suppress printing worst CWs (smaller file)

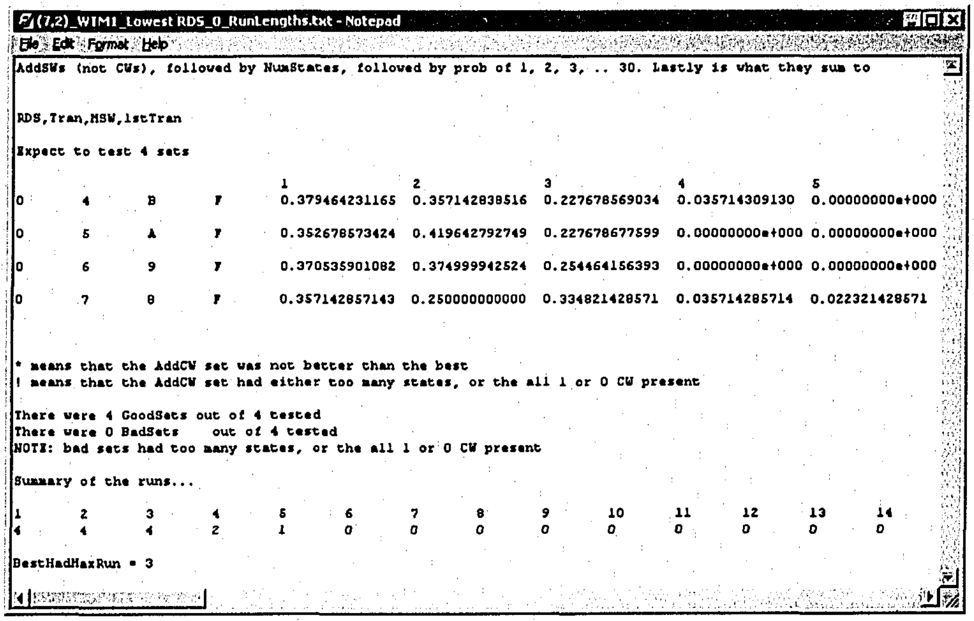
[✓] Allow MSW Test

[✓] Do not split the FSM, calculate all probs

Figure 4-27: CW search program

To perform this computer search the program **FindAllProbs.exe** in Figure 4-27 was created, which is a combination of the two programs **BuildEuCu.exe** and **CalcPIMandProbs.exe.**

Like the previous two programs **FindAllProbs** allows the user to choose the selection criteria and the expurgated code of interest. The major difference here is that the AddSWs are not specified. Instead this program runs through all possible AddSW sets (automatically converting them to AddCW sets) and tallies the statistics. For the (7,2) code considered so far, **FindAllProbs** will test the four AddSW sets {0h,4h,Bh,Fh}, {0h,5h,Ah,Fh}, {0h,6h,9h,Fh} and {0h,7h,8h,Fh}.



```
AddSWs (not CWs), followed by NumStates, followed by prob of 1, 2, 3, .. 30. Lastly is what they sum to


RDS,Tran,MSW,1stTran

Expect to test 4 sets

                                  1              2              3              4              5
0       4       B       F         0.379464231165 0.357142838516 0.227678569034 0.035714309130 0.00000000e+000

0       5       A       F         0.352678573424 0.419642792749 0.227678677599 0.00000000e+000 0.00000000e+000

0       6       9       F         0.370535901082 0.374999942524 0.254464156393 0.00000000e+000 0.00000000e+000

0       7       8       F         0.357142857143 0.250000000000 0.334821428571 0.035714285714 0.022321428571



* means that the AddCW set was not better than the best
! means that the AddCW set had either too many states, or the all 1 or 0 CW present

There were 4 GoodSets out of 4 tested
There were 0 BadSets    out of 4 tested
NOTE: bad sets had too many states, or the all 1 or 0 CW present

Summary of the runs...

1       2       3       4       5       6       7       8       9       10      11      12      13      14
4       4       4       2       1       0       0       0       0       0       0       0       0       0

BestHadMaxRun = 3
```

Figure 4-28: CW search results for the (7,2)

Figure 4-28 shows the results for the (7,2) code found by **FindAllProbs.** It can be seen that for the AddSWs {0h,4h,Bh,Fh}, the runlength probabilities match the results from the previous example in Figure 4-22. Furthermore, using the AddSWs {0h,7h,8h,Fh} leads to runs of length five 2.2% of the time, confirming the result shown in Figure 4-10.

Inspection of this output file shows that the two sets {0h,5h,Ah,Fh} and {0h,6h,9h,Fh} are the *best* since they only have runs up to length 3, i.e. a transition will occur in the output sequence at worst case every three bits. Deciding on a clear winner between these two sets however is subjective, since even though set {0h,6h,9h,Fh} has the highest probability of runs of length one with 37%, it also has a higher probability of runlengths of length three with 25.4%.

4.13. Analysis and CW Search on larger codes

Up to this point only the (7,*x*) codes have been discussed since their small size makes them easy to present and demonstrate. However, this multimode coding scheme with a length $n = 7$ code suffers greatly in terms of code rate from the increased redundancy. Therefore larger codes such as the (15,11) and (31,26) codes are now considered.

With eight AddCWs the (15,11) code is expurgated to a (15,8) code resulting in $2^8 = 256$ possible SWs. This means there will be $(1)(2^8)(2^8)(2^8)(1)(1)(1)(1) = 2^{24}$ AddCW sets to analyze based on the arguments from Section 3.6.5 and Equation 3.1. For each AddCW set the state diagram of the encoder must be constructed. This can only be found by testing all possible SWs, determining the CWs produced, and keeping track of the resulting states. For each newly discovered state however, the process must be repeated because each CW transmitted has the potential to cause a new state.

For instance, consider the case of transmitting the 15-bit all-one CW from the (RDS,LB) state of (0,0). This would put the system into a new state of (15,1). If the all-one CW is transmitted again it would put the system into another new state of (30,1) and so on. This scenario simply illustrates how each CW has the potential to cause new states. On the other hand, if lowest RDS is the primary metric (balanced transmission), many of the states discovered should be duplicates. That is, the number of states in the system should not grow unbounded since ideally the system is attempting to return to the states with an RDS of 0. However, if the maximum transitions or MSW selection criteria is used as the primary metric, then as successive words are encoded, it is possible for the encoder to have an infinite number of states. For example, many CWs contain numerous transitions but they are unbalanced. Consider the case when the system chooses one of these CWs that results in the (RDS,LB) state of (7,1). In the next encoding interval one of these CWs might be chosen again, resulting in the (RDS,LB) state of (14,1). Continuing in this fashion with maximizing transitions as the primary metric, it is entirely possible that the system will continue choosing these CWs causing the number of states to be (21,1), (28,1), (35,1) and so on, and therefore grow unbounded as encoding proceeds.

To limit the evaluation time a hard upper limit of 256 states was established for the number of states considered by **FindAllProbs** regardless of the primary decision metric. Any AddCW set that exceeds this limit is discarded.

Once the state diagram is found, the θ_u, E_u and C_u matrices must be constructed in order to build the transition matrix and solve for the invariant distribution. Once this information is complete the runlengths can be found using the analysis presented in Section 4.10. Recapping this information shows how this problem grows in complexity.

With this (15,8) code and 256 possible SWs, it can be estimated that each state can be entered and exited 256 ways. Assuming that there are only 32 states (well below the upper limit of 256), this will represent $\left(256^2\right)(32)$ iterations. This amount of work however is only for a single AddCW set and must be done once for each of the 2^{24} sets (from Section 3.6.5). Furthermore each iteration involves numerous matrix multiplications. As a result the computer search for the (15,8) code needed to be run over a three month period using sixteen Pentium 3 machines. This intensive task was split up such that each computer worked on sets of 262144 AddCWs at a time, with each set taking approximately one week to complete running twenty-four hours a day.

It is obvious then that conducting the same CW search for the (31,26) code is just not feasible since even when expurgated down to a (31,23) code it still has over 2^{69} possible AddCW sets to test (from Section 3.6.5). Therefore the results gathered for the (15,8) code will be used to make suggestions on what constitutes a good AddCW set for larger codes, which can be extrapolated to (31,x) codes, (63,x) codes and larger.

4.14. CW Search Results

Recall from Figure 3-22 that the AddCWs (or AddSWs since the codes considered are systematic) ***a***, ***b***, and ***c*** must have their MSbs in the binary format 001, 010 and 011. As a result the (15,8) CW search compared the 11-bit SWs ***a***, ***b***, and ***c*** in the range 100h to 1FFh, 200h to 2FFh and 300h to 3FFh respectively. Out of all the AddSW/AddCW sets tested only four had maximum runlengths of 7. As well each of these four AddCW sets had a maximum of 14 states which means the RDS values of the sequences only varied in the range ±3 (Note that this means RDS values of -3, -2, -1, 0, +1, +2, and +3 are possible, as well as LB values of 0 and 1). Therefore these four AddSW/AddCW sets were considered to be the best. They are shown in Table 4-7 and their runlength probabilities are shown in Table 4-8.

Table 4-7: Best AddSW sets from the (15,8) search

A	0h	107h	2C8h	323h	4DCh	537h	6F8h	7FFh
B	0h	14Dh	29Bh	370h	48Fh	564h	6B2h	7FFh
C	0h	1AAh	29Bh	323h	4DCh	564h	655h	7FFh
D	0h	1E7h	2C8h	323h	4DCh	537h	618h	7FFh

Table 4-8: Runlength probabilities for the Best AddCWs

	1	2	3	4	5	6	7	8
A	0.3515993125	0.3252365187	0.1917786803	0.0966918019	0.0288151237	0.0056458649	0.0002328572	0
B	0.3698323555	0.3238076524	0.2029532910	0.0765068932	0.0209136549	0.0059514779	0.0000347347	0
C	0.3582579240	0.3189974631	0.1980528103	0.0844292445	0.0340671827	0.0056263416	0.0005492588	0
D	0.3549359883	0.3238715642	0.1886010711	0.0963552075	0.0292343068	0.0067199212	0.0002818141	0

Inspection shows that these four best AddSW sets could guarantee runlengths of four or less 96.53% of the time. This means that the output sequence will have a transition every four bits (or less) 96.53% of the time. Furthermore these AddSW sets could guarantee runlengths of five or less 99.41% of the time, and six or less 99.97% of the time. Only 0.023% of the time would runlengths ever be higher than this to a maximum of seven. These are great results considering the CW length is fifteen.

In addition there were only four AddCW sets considered to be the worst. These sets had maximum runlengths of 15 and a maximum of 28 states representing system

RDS values that varied in the range ±7. These sets are shown in Table 4-9 with their runlength probabilities shown in Table 4-10.

Table 4-9: Worst AddSW sets from the (15,8) search

E	0h	108h	200h	300h	4FFh	5FFh	6F7h	7FFh
F	0h	10Ch	200h	300h	4FFh	5FFh	6F3h	7FFh
G	0h	120h	200h	300h	4FFh	5FFh	6DFh	7FFh
H	0h	130h	200h	300h	4FFh	5FFh	6CFh	7FFh

Table 4-10: Runlength probabilities for the Worst CWs

	1	2	3	4	5	6	7	8
E	0.328473565	0.304656190	0.189284263	0.106383216	0.044738961	0.015932939	0.004961433	0.003010110
F	0.337667123	0.309838523	0.182793727	0.117408239	0.036369763	0.011969801	0.003141914	0.000671532
G	0.328816060	0.306967874	0.194495328	0.103729269	0.040345690	0.019072758	0.003513444	0.002903702
H	0.330055271	0.306040341	0.196458392	0.110386161	0.036408923	0.015802792	0.001505133	0.000846763

	9	10	11	12	13	14	15	16
E	0.002508024	3.74071E-05	1.25344E-05	9.94439E-07	2.31383E-07	7.4575E-08	2.7276E-08	0
F	0.000107569	2.49273E-05	5.12467E-06	1.48213E-06	1.82689E-07	4.9901E-08	2.6733E-08	0
G	0.000132108	1.81316E-05	4.14227E-06	1.15889E-06	1.98428E-07	7.3293E-08	2.6913E-08	0
H	0.002445873	4.11733E-05	2.94851E-06	5.93432E-06	1.86449E-07	5.1507E-08	2.7593E-08	0

Even though these four AddSW sets had the worst statistics, they could still guarantee runlengths of four or less 92.87% of the time, runlengths of seven or less 99.44% of the time, and only 0.56% of the time would runlengths be higher than this to a maximum of fifteen. Note that the probability of a run of fifteen occurring is extremely small at 0.00000027% (i.e. 2.7276E-6%).

4.15. AddCW recommendations

No distinct pattern emerges from inspection of Tables 4-7 to 4-10. However some general conclusions can be drawn from these results and recommendations can be made.

Details of the worst AddCW sets from Table 4-9 are shown in Table 4-11 to Table 4-14. These tables show that on average the Hamming weight is 3.5, the average number of transitions is 4.9, and ratio of 0-bits to 1-bits is 11.5 to 3.5. Clearly these AddCWs are unbalanced with few transitions and on average will do very little to help the constrained sequence goals of the system.

Table 4-11: AddSW set {0h,108h,200h,300h,4FFh,5FFh,6F7h,7FFh}

SW	CW	Weight	Tran	RDS	0:1
108h	1084h	3	6	-9	12:3
200h	200Dh	4	5	-7	11:4
300h	3002h	3	4	-9	12:3

Table 4-12: AddSW set {0h,10Ch,200h,300h,4FFh,5FFh,6F3h,7FFh}

SW	CW	Weight	Tran	RDS	0:1
10Ch	10C8h	4	6	-7	11:4
200h	200Dh	4	5	-7	11:4
300h	3002h	3	4	-9	12:3

Table 4-13: AddSW set {0h,120h,200h,300h,4FFh,5FFh,6DFh,7FFh}

SW	CW	Weight	Tran	RDS	0:1
120h	1205h	4	7	-7	11:4
200h	200Dh	4	5	-7	11:4
300h	3002h	3	4	-9	12:3

Table 4-14: AddSW set {0h,130h,200h,300h,4FFh,5FFh,6CFh,7FFh}

SW	CW	Weight	Tran	RDS	0:1
130h	1300h	3	4	-9	12:3
200h	200Dh	4	5	-7	11:4
300h	3002h	3	4	-9	12:3

Therefore the first recommendation to make regarding the choice of a good AddCW set regardless of code size is to choose relatively balanced AddCWs that contain a significant number of transitions. This is validated by inspecting the statistics of the best AddCW sets as shown in Table 4-15 to Table 4-18. Here the average Hamming weight is 7.5, the average number of transitions is 7.5 and the average ratio of 0-bits to 1-bits is also 7.5.

Table 4-15: AddSW set {0h,107h,2C8h,323h,4DCh,537h,6F8h,7FFh}

SW	CW	Weight	Tran	RDS	0:1
107h	1076h	6	6	-2	9:6
2C8h	2C8Fh	8	7	+1	7:8
323h	323Dh	8	7	+1	7:8

Table 4-16: AddSW set {0h,14Dh,29Bh,370h,48Fh,564h,6B2h,7FFh}

SW	CW	Weight	Tran	RDS	0:1
14Dh	14DCh	7	8	-1	8:7
29Bh	29B8h	7	8	-1	8:7
370h	370Ah	7	8	-1	8:7

Table 4-17: AddSW set {0h,1AAh,29Bh,323h,4DCh,564h,655h,7FFh}

SW	CW	Weight	Tran	RDS	0:1
1AAh	1AA6h	7	10	-1	8:7
29Bh	29B8h	7	8	-1	8:7
323h	323Dh	8	7	+1	7:8

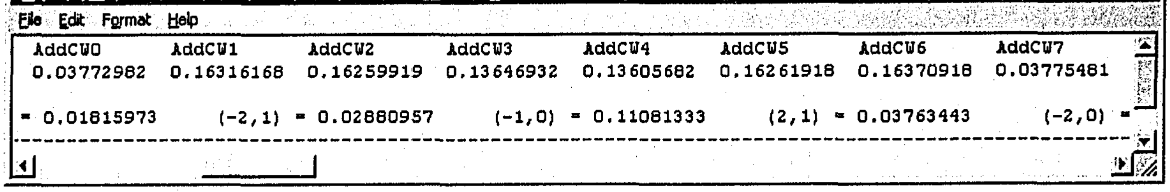
b

Table 4-18: AddSW set {0h,1E7h,2C8h,323h,4DCh,537h,618h,7FFh}

SW	CW	Weight	Tran	RDS	0:1
1E7h	1E75h	9	7	+3	6:9
2C8h	2C8Fh	8	7	+1	7:8
323h	323Dh	8	7	+1	7:8

Closer inspection of CWs 2C8Fh and 323Dh in Tables 4-15 and 4-18 show that they are cyclically shifted versions of one another. As well all CWs in Table 4-16 are cyclically shifted versions of one another. However the CWs in Table 4-17 show no such symmetry.

As a result it is difficult to make precise recommendations on what constitutes a good AddCW set. Furthermore it is tempting to assume that in this system all eight AddCWs are used equally. This however is not the case as can be seen in Figure 4-29. Here the SimFPGA output shows that AddCW1 (1076h), AddCW2 (2C8Fh) and their CW complements AddCW5 (5370h) and AddCW6 (6F89h), are each used approximately 16% of the time. On the other hand AddCW3 (323Dh) and its CW complement AddCW4 (4DC2h) are used less at approximately 13%. Finally the all-zero and all-one CWs (AddCW0 and AddCW7) are only used approximately 3.7% of the time. Thus, deciding on what constitutes a good AddCW is made more difficult by the fact that not all AddCWs are used with the same frequency. Note that this information is found simply by counting how many times each AddCW is used when building the state diagram.


(15,8)_[0,1076,2C8F,323D,4DC2,5370,6F89,7FFF]_WTM1_Stats.txt - Notepad
File Edit Format Help

```
 AddCW0     AddCW1     AddCW2     AddCW3     AddCW4     AddCW5     AddCW6     AddCW7
 0.03772982 0.16316168 0.16259919 0.13646932 0.13605682 0.16261918 0.16370918 0.03775481

= 0.01815973      (-2,1) = 0.02880957      (-1,0) = 0.11081333      (2,1) = 0.03763443      (-2,0) =
```

Figure 4-29: SimFPGA keeps track of percentage of time each AddCW is used

It can be stated that in general for codes with odd length, a good AddCW will belong to a set of CWs that are balanced in the fashion $(n-1)/2:(n+1)/2$ or $(n+1)/2:(n-1)/2$. From this set the best AddCWs will consist of those with as many transitions as possible. Furthermore if any of the AddCWs can be cyclically shifted to yield other AddCWs whose MSbs are in the form 001, 010, 011 (as shown in Figure 3-21), then these are most likely good AddCWs.

However since CWs such as 1076h exist in the best set, and such CWs are not balanced and have less than the average amount of transitions, this is a loose recommendation. Nevertheless when these recommendations are followed the maximum runlengths produced can be expected to be near the minimal values observed of $(n-1)/2$, where n is the CW length.

Further analysis on what constitutes good AddCWs can also be done in the frequency domain by inspecting the power spectral density of these codes. This is considered in Chapter 5.

5. Power Spectral Density Results

Chapter 3 introduced the multimode coding technique and Chapter 4 discussed how to model it as a finite state machine and how to analyze it using Markov chain theory. Chapter 4 then continued by presenting the results of the AddCW enumeration for the (7,x) and (15,x) codes. It was shown that a great deal of insight about what constitutes a good AddCW set can be found by analyzing these time domain statistics. However an equally suitable method of assessing this combined EC and CS code is to calculate its power spectral density (PSD). This is because the coding rules of this technique will directly influence the shape of the power spectrum. This chapter presents how to theoretically calculate the PSD of an encoded sequence and how to interpret the results.

5.1. Relationship between the time and frequency domain

As discussed in Section 2.19 a binary sequence using bipolar coding maps logic 0 values to a negative square pulse shape $-p(t)$, and logic 1 values to a positive square pulse shape $+p(t)$. When a transmitted sequence contains an overall equal number of logic 0s and 1s the transmission is said to be balanced. This is because the transmitted signal has a negative amplitude half of the time and an equal but opposite positive amplitude the other half of the time. As a result the dc average is 0 volts. Therefore inspecting this balanced signal in the frequency domain would reveal that the PSD would have a null at 0 Hz. For example Figure 5-1 shows the PSD of the Manchester code presented in Section 2.21. Recall that this line coding scheme maps logic 0 values to the 2-bit pattern 10, and maps logic 1 values to the complement 2-bit pattern 01. This guarantees at least one transition per bit as well as a completely balanced transmission [5]. As a result it can be seen that the PSD of this balanced coding technique has a null (zero power) at 0 Hz. This is shown on both the linear and logarithmic scales in Figure 5-1.

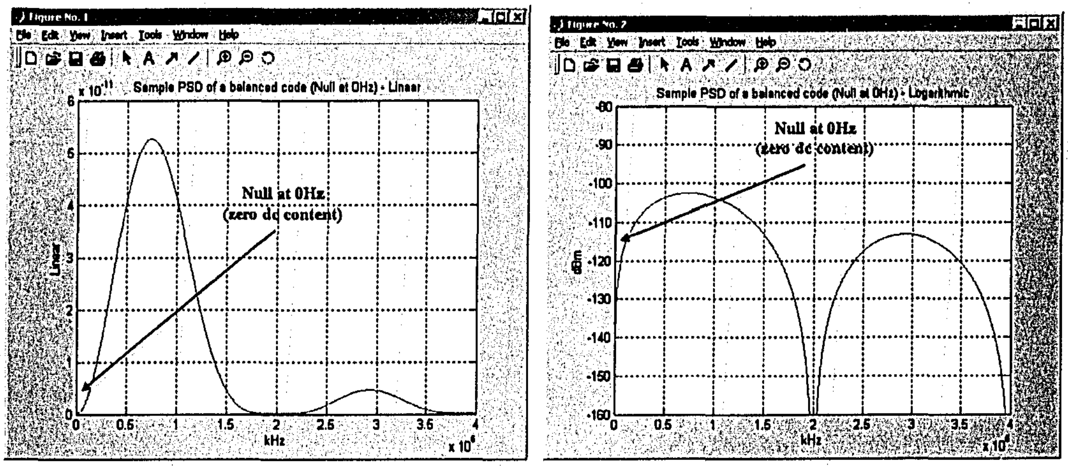


Figure 5-1: Demonstrating nulls at 0 Hz on both linear and logarithmic scales

To understand how coding can effect the shape of the PSD, consider Figure 5-2 which shows an output sequence with long runs of 0s and 1s. It is clear that a signal such as this will appear as a slowly varying square wave. As a result the PSD of this sequence would have the majority of its power at low frequencies.

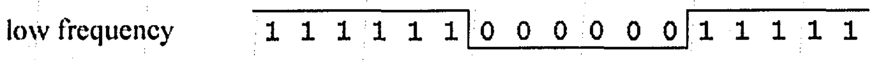


Figure 5-2: Low frequency content of a signal

On the other hand consider an output sequence that has numerous transitions as shown in Figure 5-3. A signal such as this will appear as a rapidly changing square wave. These rapid changes are indicative of high frequencies in the signal. As a result this sequence would have a significant amount of its power at high frequencies.

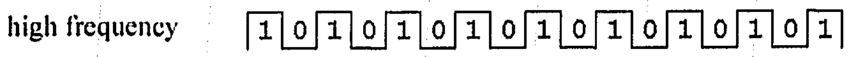


Figure 5-3: High frequency content of a signal

Since the multimode coding technique of Chapter 3 is designed to remove long runs of like valued bits (low frequencies) and introduce more transitions (high frequencies), the PSD of the encoded waveform is expected to have most of its power at high frequency and less power at low frequency. Thus the performance of the multimode coding system can be evaluated by analyzing the PSD of the output sequence.

5.2. Evaluating the Power Spectral Density

Communication signals are inherently random. This is because the receiver does not know beforehand the message that the transmitter sent. As a result a communication signal is usually treated as a **random process**, since the data is random with time. Each possible waveform is called a **sample function** and the collection of all possible waveforms is called the **ensemble** of the random process [23].

To completely characterize the random process a probability density function (PDF) or cumulative distribution function (CDF) would need to be known at every instant in time. Since this information is usually difficult to obtain, random processes are often characterized in terms of **average sequence statistics** [23]. There are generally two approaches used to evaluate the average statistics. A **time average** considers a single sample function over all time, and an **ensemble average** considers all possible sample functions and averages them at a single point in time. Processes where the time averages equal the ensemble averages are called **ergodic** [23]. If the random process has statistics that do not change with time it is also called **stationary** [23].

Useful information to obtain about a random process is the **mean** or **expected value**, as well as how quickly the process is expected to change which is given by its **autocorrelation function** [23]. The mean value of a random process has an intuitive meaning as the **dc content** of the signal and the autocorrelation function gives an indication of the **frequency content** of the random process. This is due to the fact that the correlation of a random process with itself at two different points in time depends on how rapidly the amplitude is expected to change with time [23]. All of these topics are investigated in detail with examples in Appendix A.

5.3. Evaluating the Power Spectral Density of a Coded System

As previously stated the coding rules of the multimode coding system introduced in Chapter 3 will have a direct influence on the shape of the PSD. Furthermore these coding rules make calculating the PSD slightly more complicated. As a result this section demonstrates through example the efficient procedure for calculating the PSD of a coded system as outlined by Cariolaro and Tronca [24]. Their method uses the finite state machine model of the encoder first presented in Section 4.7, and the Markov chain analysis presented in Section 4.9.

The multimode coding system is designed to transmit an output sequence with an equal number of 0s and 1s even if the source statistics are unbalanced. Thus if the logic values 0 and 1 are represented with voltage levels of 0V and 1V respectively, the average value transmitted is $(0+1)/2=0.5\text{V}$. However if a system used the polar NRZ format from Section 2.19 with 0s and 1s mapped to $\pm p(t)$, with $p(t)$ being a square pulse shape of amplitude 1V, then the average value transmitted would be 0 since $(-1+1)/2=0\text{V}$. Therefore in terms of output CWs one can intuitively expect that the average CW will be the all-zero sequence. The output is collectively known as the **mean sequence** and the average CW is called the **mean symbol vector** η_c [24].

Recall from the Markov chain analysis in Section 4.9 the **input probability matrix** θ_u, **code word matrix** C_u and **long term invariant distribution vector** π. Using these definitions it can be seen that each CW $c_u(i)$ will occur with probability $\pi(i)\theta_u(i,i)$. Therefore with mutually exclusive input symbols the mean symbol vector η_c or average CW is defined as

$$\eta_c = \pi \sum_{u=1}^{S} \theta_u c_u \tag{5.1}$$

Continuing with the example from Section 4.9 the mean symbol vector would be

$$\eta_c = \begin{bmatrix} .1875 & .0625 & .25 & .25 & .0625 & .1875 \end{bmatrix} \begin{bmatrix} .25 & 0 & 0 & 0 & 0 & 0 \\ 0 & .25 & 0 & 0 & 0 & 0 \\ 0 & 0 & .25 & 0 & 0 & 0 \\ 0 & 0 & 0 & .25 & 0 & 0 \\ 0 & 0 & 0 & 0 & .25 & 0 \\ 0 & 0 & 0 & 0 & 0 & .25 \end{bmatrix} \begin{bmatrix} 2 & 3 & 2 & 2 & 1 & 3 & 3 \\ 2 & 3 & 2 & 2 & 1 & 3 & 3 \\ 4 & 1 & 2 & 2 & 1 & 1 & 3 \\ 0 & 3 & 2 & 2 & 3 & 3 & 1 \\ 2 & 1 & 2 & 2 & 3 & 1 & 1 \\ 2 & 1 & 2 & 2 & 3 & 1 & 1 \end{bmatrix}$$

$$\eta_c = \begin{bmatrix} .5 & .5 & .5 & .5 & .5 & .5 & .5 \end{bmatrix} \tag{5.2}$$

The result in Equation 5.2 is intuitively pleasing since it demonstrates that the **average CW** (without bipolar NRZ mapping) is in-between the all-zero and all-one CW, i.e. a vector of all 0.5. However if the bipolar NRZ mapping had been used the mean symbol vector would have been the all-zero vector as shown in Equation 5.3.

$$\eta_c = \begin{bmatrix} .1875 & .0625 & .25 & .25 & .0625 & .1875 \end{bmatrix} \begin{bmatrix} .25 & 0 & 0 & 0 & 0 & 0 \\ 0 & .25 & 0 & 0 & 0 & 0 \\ 0 & 0 & .25 & 0 & 0 & 0 \\ 0 & 0 & 0 & .25 & 0 & 0 \\ 0 & 0 & 0 & 0 & .25 & 0 \\ 0 & 0 & 0 & 0 & 0 & .25 \end{bmatrix} \begin{bmatrix} 0 & 2 & 0 & 0 & -2 & 2 & 2 \\ 0 & 2 & 0 & 0 & -2 & 2 & 2 \\ 4 & -2 & 0 & 0 & -2 & -2 & 2 \\ -4 & 2 & 0 & 0 & 2 & 2 & -2 \\ 0 & -2 & 0 & 0 & 2 & -2 & -2 \\ 0 & -2 & 0 & 0 & 2 & -2 & -2 \end{bmatrix}$$

$$\eta_c = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.3}$$

This confirms that the transmitted sequence of this encoder has a dc average of zero volts, verifying that the goal of balanced transmission has been obtained.

From Appendix A the correlation of a random process with itself is called **autocorrelation** [4] and leads to spectral information of the random process. For comparison the **autocorrelation function** $R_{xx}(t_1,t_2)$ of a random process $X(t)$ is shown below in Equation 5.4.

$$\begin{aligned} R_{xx}(t_1,t_2) &\equiv \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x_1 x_2 f_x\left(x_1,x_2;t_1,t_2\right) dx_1 dx_2 \\ R_{xx}(t_1,t_2) &= E\left[X(t_1)X(t_1+\tau)\right] \end{aligned} \tag{5.4}$$

Since the autocorrelation function of a stationary process only depends on **time separation** $\tau = t_2 - t_1$, it is often represented in terms of the expectation operator [4] as shown in Equation 5.5.

$$R_x(\tau) = E\left[X(t)X(t+\tau)\right] \tag{5.5}$$

As a result Cariolaro and Tronca define the CW sequence autocorrelation $R_{C,k}$ as

$$R_{C,k} = E\left[c_n c_{n+k}\right] \tag{5.6}$$

Evaluation of $R_{C,k}$ at time separation $k = 0$ is $R_{C,0} = E\left[c_n c_n\right]$. Since CW $c_u(i)$ will occur with probability $\pi(i)\theta_u(i,i)$, its contribution to $R_{C,0}$ will be $R_{C,0} = c_u^2(i)\pi(i)\theta(i,i)$. To find the contribution from all CWs to $R_{C,0}$ it is helpful to define Λ as an L-square diagonal matrix such that

$$\Lambda(i,j) = \begin{cases} \pi(i) & i = j \\ 0 & otherwise \end{cases} \tag{5.7}$$

Then the autocorrelation with zero time separation can be obtained by summing over all CWs as shown in Equation 5.8.

$$R_{C,0} = \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u c_u \tag{5.8}$$

Continuing in this manner and using the definitions defined by Cariolaro and Tronca, when the input symbols are independent of state, the autocorrelation for all time separations k can be shown to be

$$R_{C,k} = \begin{cases} \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u c_u & k = 0 \\ \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u E_u \Pi^{k-1} \sum_{v=1}^{S} \theta_v c_v & k \geq 1 \\ R_{C,k}^{\mathrm{T}} & k \leq -1 \end{cases} \tag{5.9}$$

Then continuing with the example there are four SWs and as a result the autocorrelation at zero time separation $R_{C,0}$ is found by summing up the four matrices $c_0^{\mathrm{T}}\Lambda\theta_0 c_0$ + $c_1^{\mathrm{T}}\Lambda\theta_1 c_1$ + $c_2^{\mathrm{T}}\Lambda\theta_2 c_2$ + $c_3^{\mathrm{T}}\Lambda\theta_3 c_3$ shown in Equation 5.10. This was done by mapping the CW matrices in Equation 4.25 to $\pm A$ with $A = 1$.

$$c_0^{\mathrm{T}}\Lambda\theta_0 c_0 = \begin{bmatrix} +r & -r & +r & +r & -r & -r & -r \\ -r & +r & -r & -r & +r & +r & +r \\ +r & -r & +r & +r & -r & -r & -r \\ +r & -r & +r & +r & -r & -r & -r \\ -r & +r & -r & -r & +r & +r & +r \\ -r & +r & -r & -r & +r & +r & +r \\ -r & +r & -r & -r & +r & +r & +r \end{bmatrix} \quad c_1^{\mathrm{T}}\Lambda\theta_1 c_1 = \begin{bmatrix} +r & -r & +r & -r & -r & +r & +r \\ -r & +r & -r & +r & +r & -r & -r \\ +r & -r & +r & -r & -r & +r & +r \\ -r & +r & -r & +r & +r & -r & -r \\ -r & +r & -r & +r & +r & -r & -r \\ +r & -r & +r & -r & -r & +r & +r \\ +r & -r & +r & -r & -r & +r & +r \end{bmatrix}$$

$$c_2^{\mathrm{T}}\Lambda\theta_2 c_2 = \begin{bmatrix} +r & +r & -r & +r & -r & -r & +r \\ +r & +r & -r & +r & -r & -r & +r \\ -r & -r & +r & -r & +r & +r & -r \\ +r & +r & -r & +r & -r & -r & +r \\ -r & -r & +r & -r & +r & +r & -r \\ -r & -r & +r & -r & +r & +r & -r \\ +r & +r & -r & +r & -r & -r & +r \end{bmatrix} \quad c_3^{\mathrm{T}}\Lambda\theta_3 c_3 = \begin{bmatrix} +r & -r & -r & -r & +r & -r & +r \\ -r & +r & +r & +r & -r & +r & -r \\ -r & +r & +r & +r & -r & +r & -r \\ -r & +r & +r & +r & -r & +r & -r \\ +r & -r & -r & -r & +r & -r & +r \\ -r & +r & +r & +r & -r & +r & -r \\ +r & -r & -r & -r & +r & -r & +r \end{bmatrix} \tag{5.10}$$

where $r = 0.25$.

Using Equation 5.9 the autocorrelation obtained for all CWs at time separation 0 is shown in Equation 5.11.

$$R_{C,0}=\begin{bmatrix} 1 & -.5 & 0 & 0 & -.5 & -.5 & .5 \\ -.5 & 1 & -.5 & .5 & 0 & 0 & 0 \\ 0 & -.5 & 1 & 0 & -.5 & .5 & -.5 \\ 0 & .5 & 0 & 1 & -.5 & -.5 & -.5 \\ -.5 & 0 & -.5 & -.5 & 1 & 0 & 0 \\ -.5 & 0 & .5 & -.5 & 0 & 1 & 0 \\ .5 & 0 & -.5 & -.5 & 0 & 0 & 1 \end{bmatrix} \tag{5.11}$$

The interpretation of this autocorrelation matrix is as follows. Equation 5.11 is showing how correlated the individual bits (0 through 6) of any 7-bit CW are with themselves at time separation $k=0$. For instance the diagonal ones indicate that the 1st bit is 100% correlated with the 1st bit, the 2nd bit is 100% correlated with the 2nd bit and so on down to the 7th bit is 100% correlated with the 7th bit. Conversely, the correlation between the 1st bit with the 2nd bit is –.5. This means that the first two bit combinations of any CW are more likely to be 1,-1 or -1,1 then they are to be 1,1 or -1,-1. Continuing in this fashion the correlation between bit 0 and bit 2 is zero, which indicates that knowing the value of the 1st bit gives no indication of the value of the 3rd bit. Hence all four combinations -1,-1, -1,1, 1,-1 or 11 are equally likely. The rest of the matrix can be understood in this fashion.

Before evaluating the autocorrelation of the encoded sequence for time separation $k>0$, inspection of Equation 5.9 suggests that the second summation term can be computed separately as shown in Equation 5.12 since it does not change with *k*.

$$\beta=\sum_{v=1}^{S}\theta_v c_v=\begin{bmatrix} 0 & .5 & 0 & 0 & -.5 & .5 & .5 \\ 0 & .5 & 0 & 0 & -.5 & .5 & .5 \\ 1 & -.5 & 0 & 0 & -.5 & -.5 & .5 \\ -1 & .5 & 0 & 0 & .5 & .5 & -.5 \\ 0 & -.5 & 0 & 0 & .5 & -.5 & -.5 \\ 0 & -.5 & 0 & 0 & .5 & -.5 & -.5 \end{bmatrix} \tag{5.12}$$

Thus $R_{c,1}$ is evaluated as

$$R_{c,1} = \left(C_0^T \Lambda \theta_0 E_0 \Pi^0 + C_1^T \Lambda \theta_1 E_1 \Pi^0 + C_2^T \Lambda \theta_2 E_2 \Pi^0 + C_3^T \Lambda \theta_3 E_3 \Pi^0\right) \begin{bmatrix} 0 & .5 & 0 & 0 & -.5 & .5 & .5 \\ 0 & .5 & 0 & 0 & -.5 & .5 & .5 \\ 1 & -.5 & 0 & 0 & -.5 & -.5 & .5 \\ -1 & .5 & 0 & 0 & .5 & .5 & -.5 \\ 0 & -.5 & 0 & 0 & .5 & -.5 & -.5 \\ 0 & -.5 & 0 & 0 & .5 & -.5 & -.5 \end{bmatrix}$$

$$R_{c,1} = \begin{bmatrix} -.125 & .0625 & 0 & 0 & .0625 & .0625 & -.0625 \\ 0 & -.0625 & 0 & 0 & .0625 & -.0625 & -.0625 \\ .125 & -.0625 & 0 & 0 & -.0625 & -.0625 & .0625 \\ .125 & -.0625 & 0 & 0 & -.0625 & -.0625 & .0625 \\ 0 & .0625 & 0 & 0 & -.0625 & .0625 & .0625 \\ 0 & -.0625 & 0 & 0 & .0625 & -.0625 & -.0625 \\ -.25 & .0625 & 0 & 0 & .1875 & .0625 & -.0625 \end{bmatrix} \tag{5.13}$$

This autocorrelation matrix indicates how similar any CW is with the preceding CW on the channel. Clearly the correlation falls off rapidly. This is because in this example knowing the value of the 1st bit in the first CW only gives a marginal -.125 indication of the value of the 1st bit in the next CW and so on. Nevertheless the autocorrelation can be evaluated for greater time separation as shown in Equation 5.14.

$$R_{c,2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .0625 & -.03125 & 0 & 0 & -.03125 & -.03125 & -.03125 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -.0625 & .03125 & 0 & 0 & .03125 & .03125 & -.03125 \\ .0625 & -.03125 & 0 & 0 & -.03125 & -.03125 & .03125 \\ .0625 & -.03125 & 0 & 0 & -.03125 & -.03125 & .03125 \end{bmatrix} \quad R_{c,3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R_{c,100} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad R_{c,\infty} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.14}$$

Notice that by time separation $k = 3$ the autocorrelation matrices have converged to the all-zero matrix, which demonstrates that this random process loses correlation with itself very rapidly. Further notice that the autocorrelation matrix at $k = \infty$ can also be obtained by multiplying the transpose of the mean symbol vector with itself as shown in Equation 5.15.

$$R_{c,\infty} = \eta_c^{\mathrm{T}} \eta_c = \eta_c^2 \tag{5.15}$$

Autocovariance matrices can also be found which indicate how the random process varies with time separation. They are related to the autocorrelation matrices as shown in Equation 5.16.

$$\begin{aligned} K_{c,k} &= R_{c,k} - \eta_c^2 \\ K_{c,k} &= R_{c,k} - R_{c,\infty} \end{aligned} \tag{5.16}$$

It can be seen from this expression that if the mean symbol vector is zero, the autocovariance matrices are equal to the autocorrelation matrices. As noted previously this will occur when the output transmission is balanced and bipolar signaling is used. $K_{c,0}$ is given by $R_{c,0} - R_{c,\infty}$, while $K_{c,1} = R_{c,1} - R_{c,\infty}$ and $K_{c,2} = R_{c,2} - R_{c,\infty}$ and thus as k tends to infinity $R_{c,k}$ tends to $R_{c,\infty}$ and the all-zero matrix is reached as shown below in Equation 5.17

$$K_{c,0} = \begin{bmatrix} 1 & -.5 & 0 & 0 & -.5 & -.5 & .5 \\ -.5 & 1 & -.5 & .5 & 0 & 0 & 0 \\ 0 & -.5 & 1 & 0 & -.5 & .5 & -.5 \\ 0 & .5 & 0 & 1 & -.5 & -.5 & -.5 \\ -.5 & 0 & -.5 & -.5 & 1 & 0 & 0 \\ -.5 & 0 & .5 & -.5 & 0 & 1 & 0 \\ .5 & 0 & -.5 & -.5 & 0 & 0 & 1 \end{bmatrix} \quad K_{c,1} = \begin{bmatrix} -.125 & .0625 & 0 & 0 & .0625 & .0625 & -.0625 \\ 0 & -.0625 & 0 & 0 & .0625 & -.0625 & -.0625 \\ .125 & -.0625 & 0 & 0 & -.0625 & -.0625 & .0625 \\ .125 & -.0625 & 0 & 0 & -.0625 & -.0625 & .0625 \\ 0 & .0625 & 0 & 0 & -.0625 & .0625 & .0625 \\ 0 & -.0625 & 0 & 0 & .0625 & -.0625 & -.0625 \\ -.25 & .0625 & 0 & 0 & .1875 & .0625 & -.0625 \end{bmatrix}$$

$$K_{c,2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .0625 & -.03125 & 0 & 0 & -.03125 & -.03125 & -.03125 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -.0625 & .03125 & 0 & 0 & .03125 & .03125 & -.03125 \\ .0625 & -.03125 & 0 & 0 & -.03125 & -.03125 & .03125 \\ .0625 & -.03125 & 0 & 0 & -.03125 & -.03125 & .03125 \end{bmatrix} \quad K_{c,\infty} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.17}$$

The PSD is obtained using the autocorrelation and autocovariance matrices and it can be defined in terms of a continuous component $X_C(f)$, and a discrete component $X_D(f)$. These components are weighted by the Fourier Transform of the pulse shape $P(f)$.

$$W(f)=\frac{|P(f)|^2}{NT}\left(X_C(f)+\frac{X_D(f)}{NT}\sum_{m=-\infty}^{\infty}\delta\left(f-\frac{mf_0}{N}\right)\right) \tag{5.18}$$

where

$$X_C(f)=v\left[K_{C,0}+2\sum_{k=1}^{\infty}K_{C,k}e^{-j2\pi fkNT}\right]v^* \tag{5.19}$$

$$X_D(f)=vR_{C,\infty}v^* \tag{5.20}$$

$$v=\begin{bmatrix}1 & e^{j2\pi fT} & e^{j4\pi fT} & e^{j6\pi fT} & \ldots & e^{j2\pi f(N-1)T}\end{bmatrix} \tag{5.21}$$

In this balanced transmission example $R_{C,\infty}=0$ and by Equation 5.20 there is no discrete component $X_D(f)$. Therefore Equation 5.18 can be rewritten as

$$W(f)=\frac{|P(f)|^2}{NT}\left(X_C(f)+0\right) \tag{5.22}$$

Since the covariance matrices are only non-zero up to $k=2$, $X_C(f)$ is found to be

$$X_C(f)=v\left[\begin{bmatrix}1 & -.5 & 0 & 0 & -.5 & -.5 & .5\\ -.5 & 1 & -.5 & .5 & 0 & 0 & 0\\ 0 & -.5 & 1 & 0 & -.5 & .5 & -.5\\ 0 & .5 & 0 & 1 & -.5 & -.5 & -.5\\ -.5 & 0 & -.5 & -.5 & 1 & 0 & 0\\ -.5 & 0 & .5 & -.5 & 0 & 1 & 0\\ .5 & 0 & -.5 & -.5 & 0 & 0 & 1\end{bmatrix}+2\sum_{k=1}^{2}K_{C,k}e^{-j2\pi fkNT}\right]v^* \tag{5.23}$$

where

$$v = \begin{bmatrix} 1 & e^{j2\pi fT} & e^{j4\pi fT} & e^{j6\pi fT} & e^{j8\pi fT} & e^{j10\pi fT} & e^{j12\pi fT} \end{bmatrix} \tag{5.24}$$

The result is a continuous function of frequency f that is weighted by the pulse shape. If a square pulse shape is used and normalized to $A = 1\text{V}$ with duration $T = 10\mu\text{s}$, Equation 5.25 shows its time domain representation and Fourier transform. The PSD is then this function (pulse shape) squared, and plotted on either a linear or logarithmic scale as shown in Figure 5-4.

$$p(t) = A\text{rect}\left(\frac{t}{T}\right) \quad \leftrightarrow \quad P(\omega) = AT\text{sinc}\left(\frac{\omega T}{2}\right)$$

$$P(f) = (10\mu\text{s})\,\text{sinc}(\pi f 10\mu\text{s}) \tag{5.25}$$

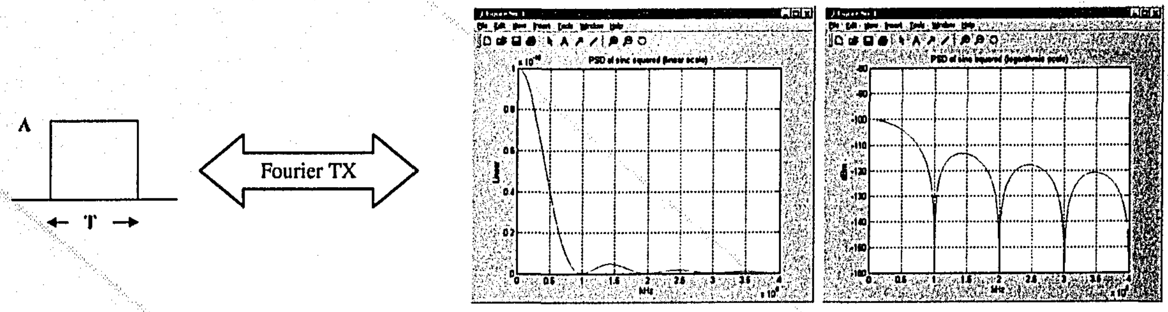


Figure 5-4: PSD of a square pulse shape plotted on linear and logarithmic scales

Therefore using Equation 5.22 and evaluating $X_C(f)$ at $f = 500Hz$, for example, gives

$$X_C(500) = .0543 + .215j = .2216e^{1.3233j} \tag{5.26}$$

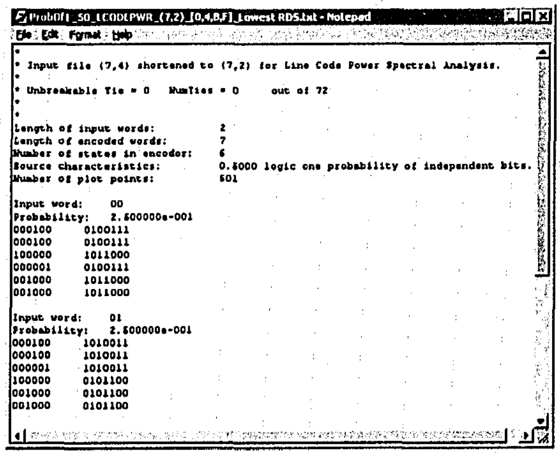
$$W(500) = \frac{\left|(10\mu s)\,\text{sinc}(\pi 500 * 10\mu s)\right|^2}{7 * 10\mu s}(.0543 + .215j)$$
$$W(500) = 7.75E-8 + j3.07E-7$$
$$|W(500)| = 3.167E-7$$

This is the PSD calculated at a single frequency. This evaluation however needs to be done as a continuous function of frequency since by definition the power contribution of a single point on a density curve is zero [4]. This time consuming operation is best performed using mathematical software such as Matlab.

5.4. Evaluating the Power Spectral Density using Matlab

Recall from Section 4.11 the program **Build_EuCu.exe**. This program allows the user to select the expurgated code, selection criteria, source statistics and AddSWs to use. It then generates three files. The first two files created were shown in Figure 4-19 and Figure 4-20. They dealt with the number of states in the encoder and created the θ_u, E_u and C_u matrices used to calculate the runlength probabilities of the code.

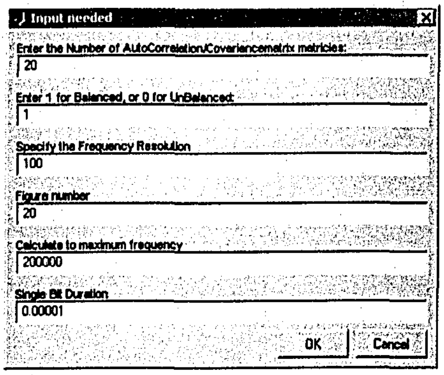
Continuing from the example in Section 4.11, the third file will contain a combination of the information found in the first two files, and it is used to evaluate the PSD. An example of this file is shown in Figure 5-5. This file begins with some comments and general information about the code used. Following this is each **Input word** (SW) along with its probability of occurrence which is read into the θ_u matrices, followed by the E_u and C_u matrices which are written in binary form.


ProbOf1_50_LCODEPWR_(7,2)_[0,4,B,F]_Lowest RDS.txt - Notepad
File Edit Format Help

```
*
* Input file (7,4) shortened to (7,2) for Line Code Power Spectral Analysis.
*
* Unbreakable Tie = 0   NumTies = 0     out of 72
*
*
Length of input words:          2
Length of encoded words:        7
Number of states in encoder:    6
Source characteristics:         0.5000 logic one probability of independent bits.
Number of plot points:          501

Input word:     00
Probability:    2.500000e-001
000100      0100111
000100      0100111
100000      1011000
000001      0100111
001000      1011000
001000      1011000

Input word:     01
Probability:    2.500000e-001
000100      1010011
000100      1010011
000001      1010011
100000      0101100
001000      0101100
001000      0101100
```

Figure 5-5: Input file for PSD calculation

The θ_u, E_u and C_u information is used by a Matlab program called **"calcCompletePSDfromLCODEPWRIN.m"**. Additional parameters are entered into the dialog box as shown in Figure 5-6. These are a) the number of autocorrelation and autocovariance matrices to compute, b) whether or not the system uses NRZ mapping (i.e. balanced mapping), c) frequency stepping resolution, d) which figure number to use, e) maximum frequency to calculate values for and f) the duration of a single bit in seconds. Note that the pulse shape is hard coded to be square and normalized to 1 volt to match with the hardware implementation. This Matlab script will also output the PSD with and without the influence of the square pulse shape. This was done in order to see the effect of the coding alone on the shape of the PSD.



Enter the Number of AutoCorrelation/Coveriencematrix matricles:
20

Enter 1 for Balanced, or 0 for UnBalanced:
1

Specify the Frequency Resolution
100

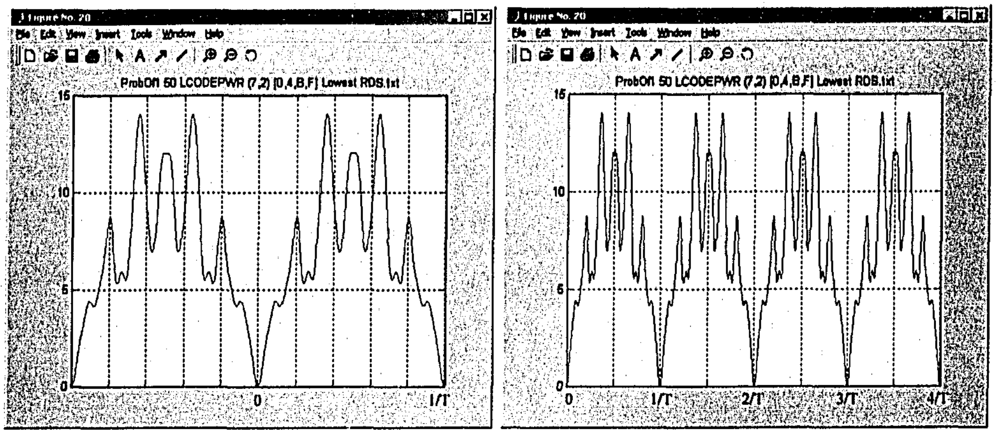
Figure number
20

Calculate to maximum frequency
200000

Single Bit Duration
0.00001

Figure 5-6: Information required for PSD calculation

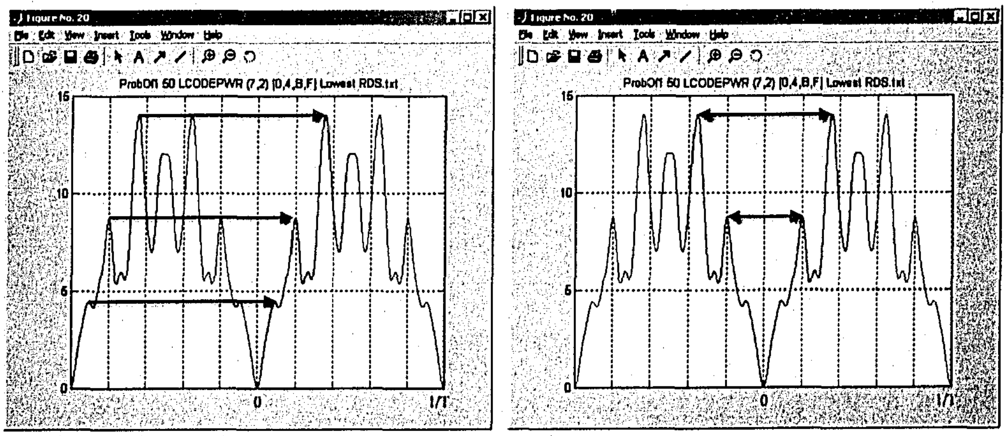
Continuing the previous example with the parameters in Figure 5-6, the continuous component $X_C(f)$ can be plotted on a linear scale without the effect of the pulse shape as shown in Figure 5-7. This plot demonstrates the influence of the combined EC and CS coding scheme alone.



(a) (b)

Figure 5-7: Linear PSD of the multimode system (no pulse shape included)

There are a few things to notice about this PSD. First of all as a result of the discrete nature of the autocorrelation **the PSD is periodic** in the frequency domain as shown in Figure 5-8a. Secondly since the symbols are all real valued the **PSD is an even function** and thus symmetrical about 0 Hz as shown in Figure 5-8b. Therefore the PSD repeats in this fashion to infinity where each lobe is a replica of the previous one as shown in Figure 5-7b.



(a) (b)

Figure 5-8: Showing that the PSD is periodic and an even function

However once the pulse shape of Figure 5-4 is taken into consideration, the PSD begins to decay as sinc squared. This effect makes the null at 0 Hz less noticeable, but since this PSD is periodic, it is clear that the null is present. This is shown in Figure 5-9 on both the linear and logarithmic scale.

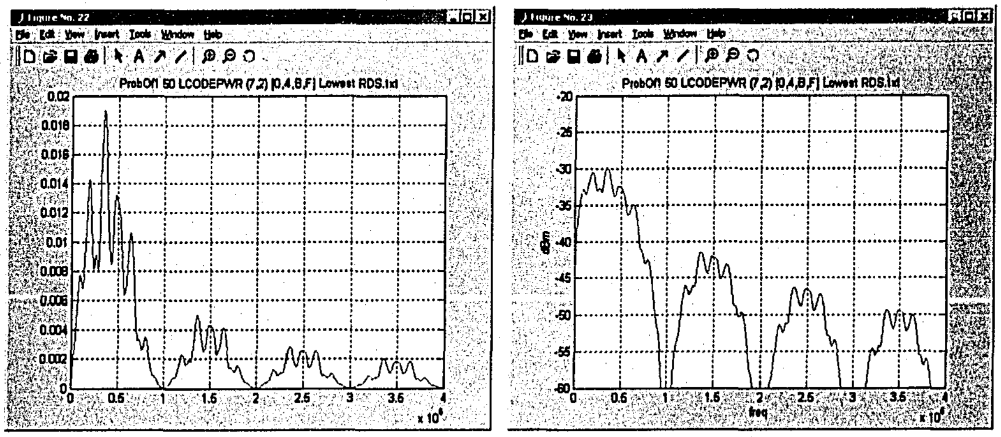


Figure 5-9: Effect of the square pulse shape plotted on a linear and logarithmic scale

Note that the logarithmic scale is used in order to compare theoretical spectra with those measured from the FPGA implementation in the lab. This is because digital oscilloscopes and spectrum analyzers often display frequency domain results in decibels. For example Figure 5-10 shows this PSD plotted for the first two lobes as generated by this Matlab script, with the measured PSD from the FPGA implementation beside it. It is clear that the Cariolaro and Tronca approach matches closely with the measured values. This aspect will be more fully discussed in Chapter 6.

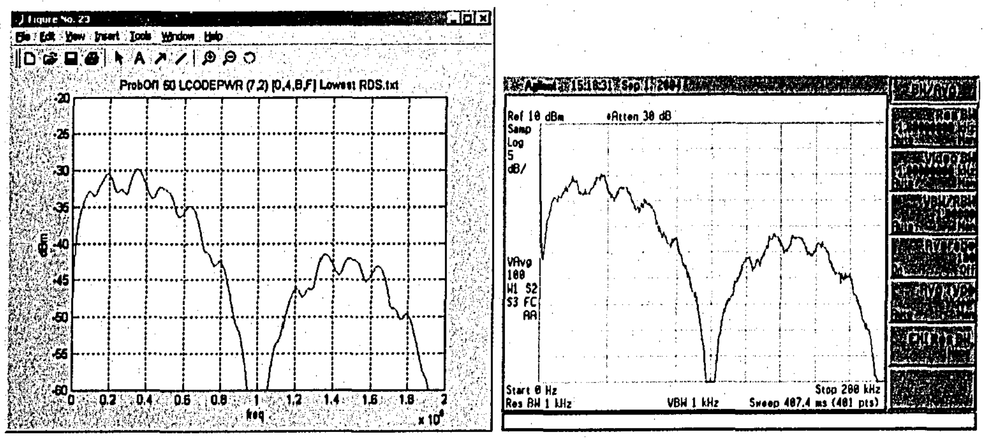


Figure 5-10: Comparing the spectrum analyzer to Matlab spectrum

5.5. Understanding the Discrete-Like Nature of the PSD

A final source of confusion is why the PSD sometimes has a "spiky" shape. Normally strong spectral spikes are indicative of periodic sequences in the time domain. These present themselves as discrete components in the PSD. However these strong spectral peaks are not caused by periodic sequences with this coding scheme. In this case the concentrated power in narrow bands in the frequency domain actually come from the sometimes limited selection of CWs. This is best seen by example.

Consider the (7,1) Hamming code from Table 3-3 using the eight AddCWs {00h,16h,27h,31h,4Eh,58h,69h,7Fh}. Figure 5-11 shows this code as a state diagram. This small code is interesting to analyze since there are only two input SWs, 0 and 1. Furthermore, even though there are eight AddCWs in this system, the simplicity of this code results in only four CWs {69h,53,2Ch,16h} ever being transmitted.

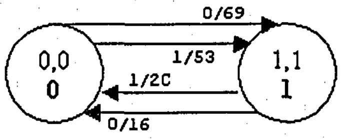


Figure 5-11: Simplest FSM model for the (7,1) code

The FSM model in Figure 5-11 indicates that the only possible CW sequences that leave state 0 and return are {69h,16h}, {69h,2Ch}, {53h,16h} and {53h,2Ch}. One may expect this system to produce a discrete PSD since the transmitted sequences appear periodic. However this is not the case since even though there are only four possible sequences, they are emitted randomly. This results in the continuous PSD shown in Figure 5-12. This PSD was calculated assuming all four sequences are equally likely.

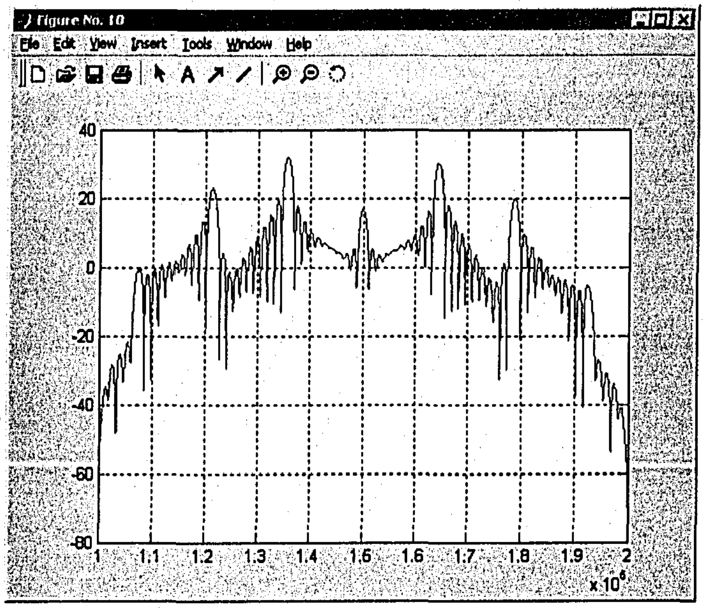


Figure 5-12: PSD of the multimode code shown in Figure 6-50

At first glance this spectrum appears to have discrete components. Understanding why this is can be found by analyzing the Fourier Series (see Appendix A) of the four sequences that can leave state 0 and return. As discussed above these sequences are {69h,16h}, {69h,2Ch}, {53h,16h} and {53h,2Ch}, and their Fourier Series representations are shown in Figure 5-13 and 5-14. All four figures have seven frequency components in common as shown in Figure 5-15.

Comparing the spectrum in Figure 5-15 with the PSD in Figure 5-12 explains where the apparent discrete components come from. These seven frequency components will be dominant in this system regardless of randomness. For example if the output of the encoder was ever forced to always be one of the four sequences, then the output would be one of the four Fourier Series spectra shown below. However, since the output at any point in time is one of the four sequences chosen in a random fashion, each one of the four spectra is prevalent, but overall yields a continuous spectrum.

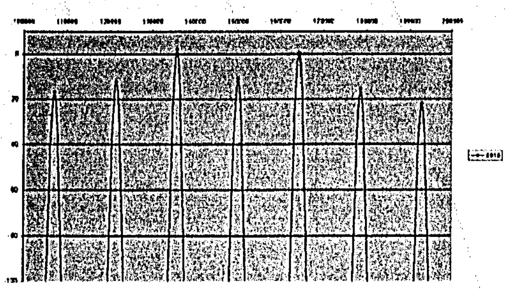
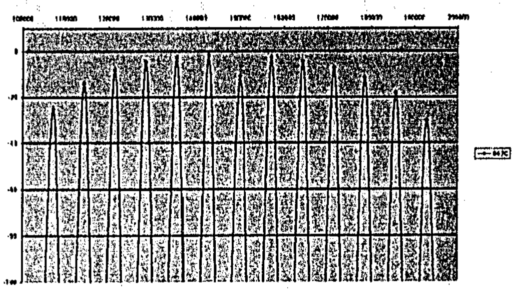



Figure 5-13: PSD of the periodic sequences {6916h} and {692Ch}

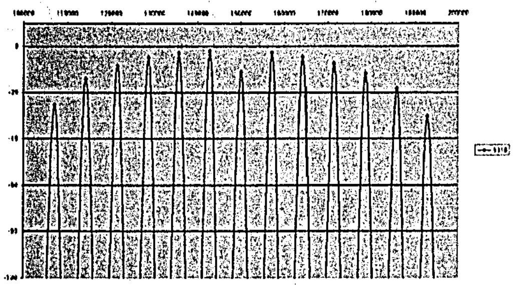
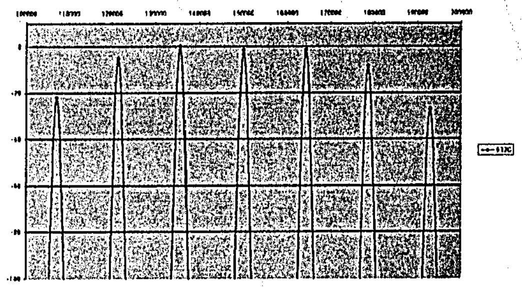



Figure 5-14: PSD of the periodic sequences {5316h} and {532Ch}

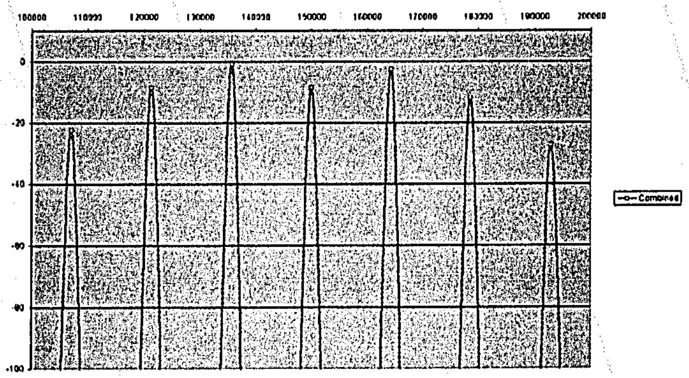


Figure 5-15: Average of the seven frequency components

5.6. Spectra of various codes

The CW search results of Section 4.14 for CWs of length 15 revealed that the coding technique on average could guarantee runlengths of 4 or less 94% of the time. Only 6% of the time would runlengths be higher than this to a maximum of 11.

Furthermore out of the 16777216 AddCW sets tested, only four sets were considered to be the worst since they could have runlengths as high as 15. However these runlengths would occur with very low probability and they could still guarantee runlengths of 4 our less 92% of the time. The four best AddCW sets on the other hand could guarantee runlengths of 4 or less 96% of the time, and no runlengths higher than 7. This section now investigates the PSD of these codes.

For comparison the PSDs are calculated using two heavily unbalanced source statistics of 10% probability of a logic 1 and 90% probability of a logic 1. Furthermore only the four best and four worst AddCW sets are calculated, as all other AddCW sets will have statistics within their range.

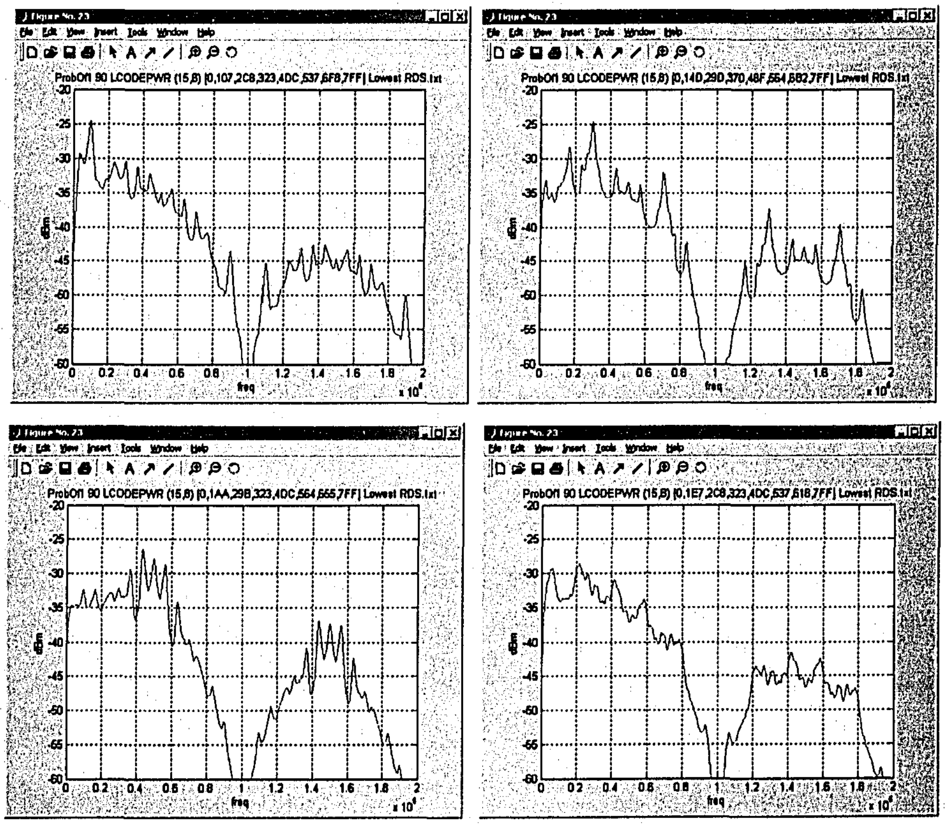


Figure 5-16: 90% ones – All four Best AddCW sets

Figure 5-16 shows the PSD of the four best AddCW sets with a source that emits logic 1s with 90% probability. It is clear that the coding scheme is working since the nulls are still present and quite wide. The "spiky" nature of the PSD however is a result of the unbalanced source. Since the likelihood a SW containing many 1s is very high, only a limited number of the available CWs are being transmitted. That is since only a subset of available SWs is being selected (i.e. SWs comprised mostly of 1s), only a subset of available CWs is being transmitted. Hence the correlation between CWs is higher and the PSD shows spiky peaks indicative of a periodic-like transmission. In other words, the same group of CWs is being repeatedly transmitted but in a random fashion.

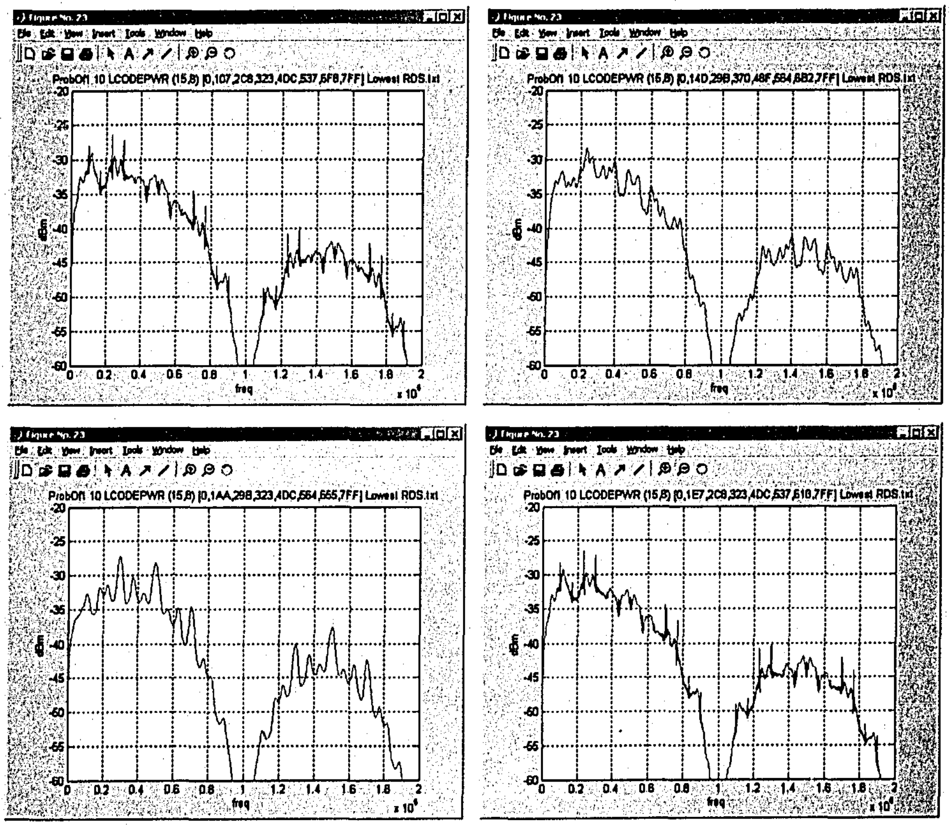


Figure 5-17: 10% ones – All four Best AddCW sets

Similarly Figure 5-17 shows the PSD of the four best AddCW sets with a source that emits logic 1s with 10% probability. This is of course the same but opposite problem as above since now the likelihood of a logic 0 is 90%. Thus the PSD still exhibits the spiky response. Nevertheless the attenuation of the PSD near dc shows that lower frequencies are less likely than higher frequencies, and hence these PSDs are indicative of a balanced sequence that has numerous transitions even with these unbalanced source statistics.

Note that when the source is balanced the PSD loses its spiky nature. Figure 5-18 shows the PSD of two of the best AddCW sets with a balanced source. Since all the SWs are equally likely the entire set of available CWs is used and hence the discrete-like components disappear from the PSD. It is now more "smooth" and similar to the PSD of a random binary waveform with the addition of the null at dc. Note that the PSD of all (15,8) codes are practically identical when the source is balanced, and as a result they all resemble those presented in Figure 5-18.

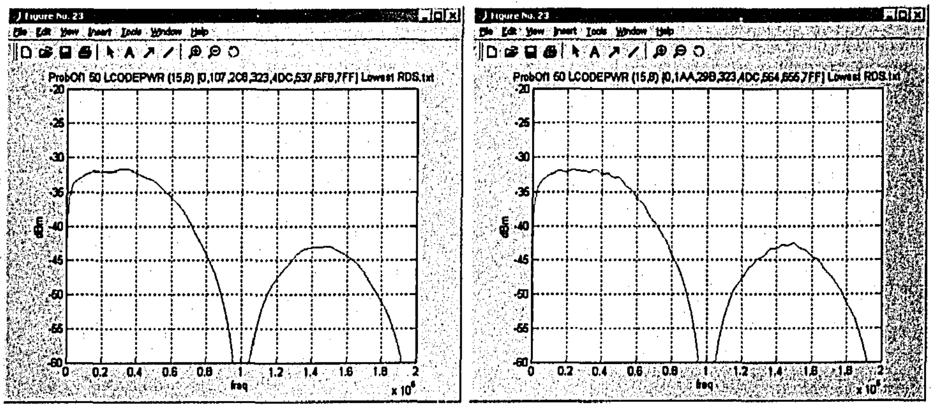


Figure 5-18: PSD of (15,8) codes with a balanced source

Continuing with this analysis Figure 5-19 shows the PSD of the four worst AddCW sets with a source that emits logic 1s with 90% probability. The null at dc indicates that the coding scheme is working, however since runlengths of up to length 15 are possible, the null is narrower. Furthermore it is clear that the worst AddCW sets have more power at low frequencies than the best AddCW sets.

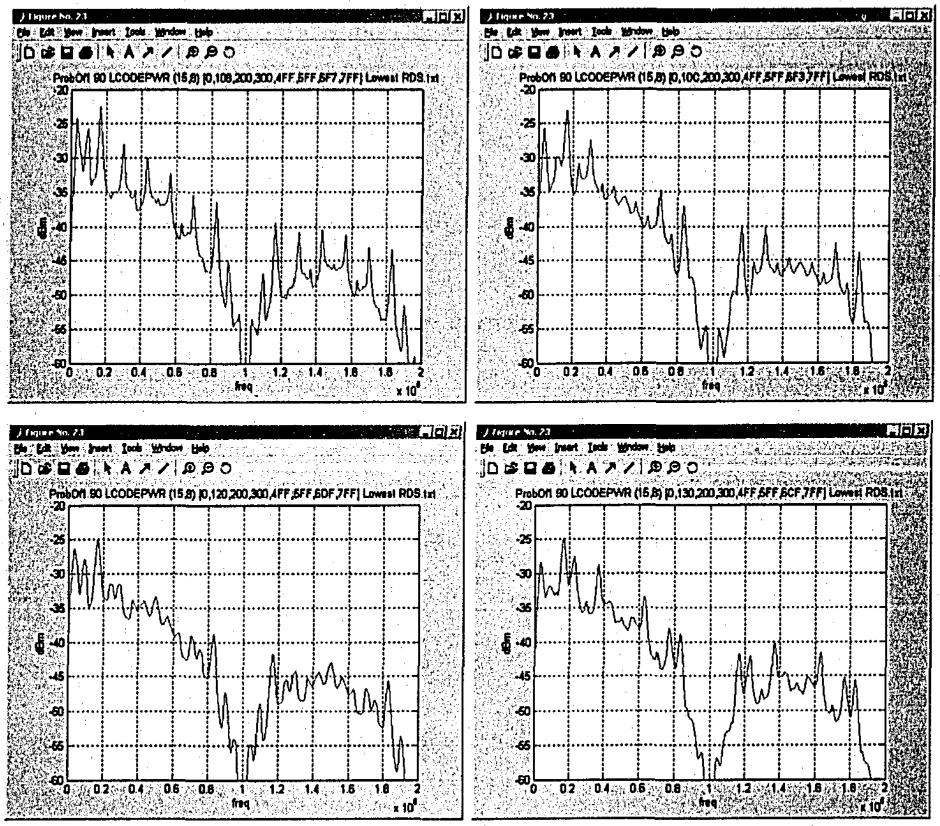


Figure 5-19: 90% ones – All four Worst AddCW sets

Similarly Figure 5-20 shows the PSD of the four worst AddCW sets with a source that emits logic 1s with 10% probability. While the best AddCW sets had PSDs with convex shapes, the worst AddCW sets have almost concave shapes. This indicates that the highest frequencies are less likely than some of the midband frequencies. On the other hand the power at low frequencies is still quite small but clearly larger than the PSD of the comparable best AddCW sets. For example Figure 5-21 shows the PSD of one of the best AddCW sets beside one of the worst AddCW sets when the probability of the source emitting logic 1s is 90%.

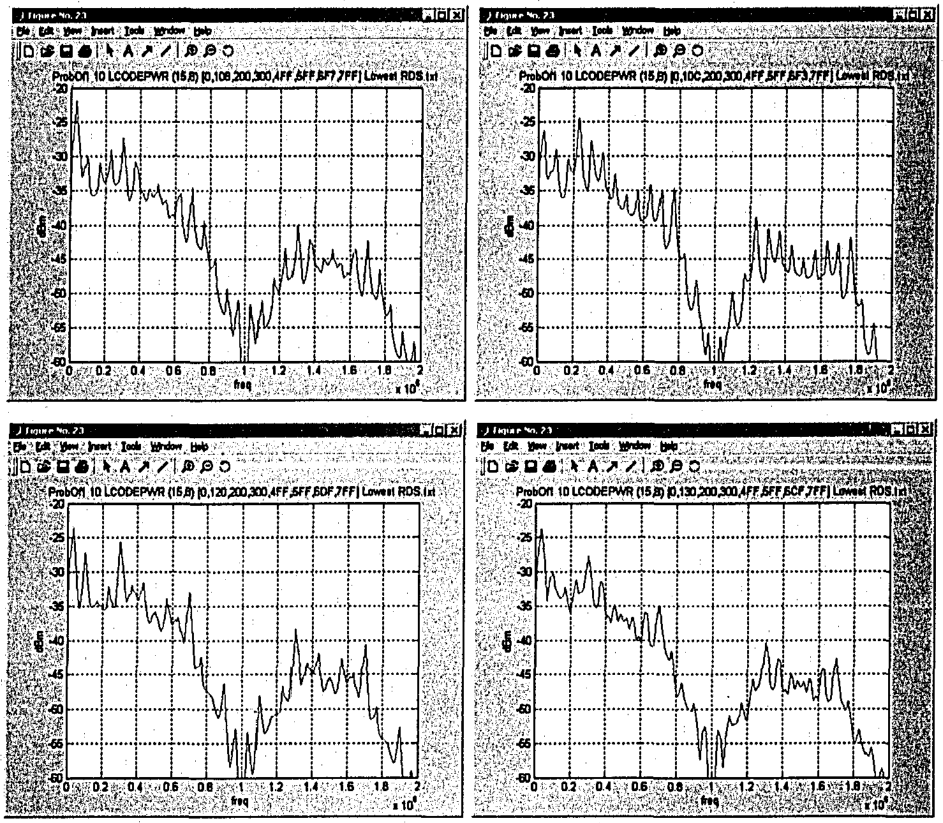


Figure 5-20: 10% ones – All four Worst AddCW sets

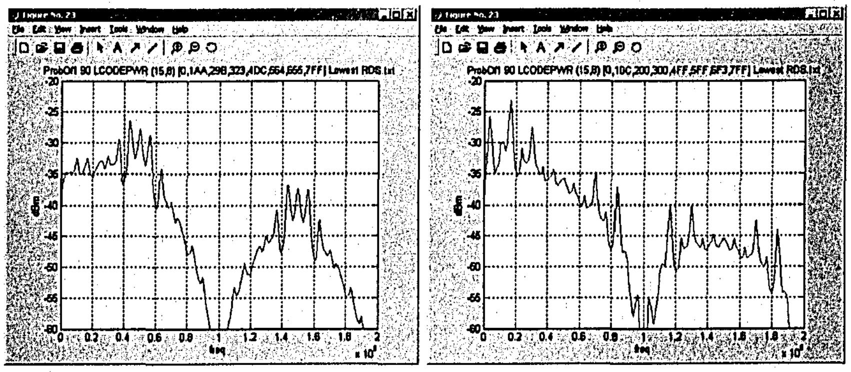


Figure 5-21: 90% ones – Best AddCW set followed by a worst AddCW set

While it is obvious that the best AddCW set has more power at high frequency it is most evident when they plotted together as shown in Figure 5-22. Here it is clearly seen that not only does the best AddCW set have most of its power at higher frequencies, it also has a wider null width indicating its ability to limit runlengths.

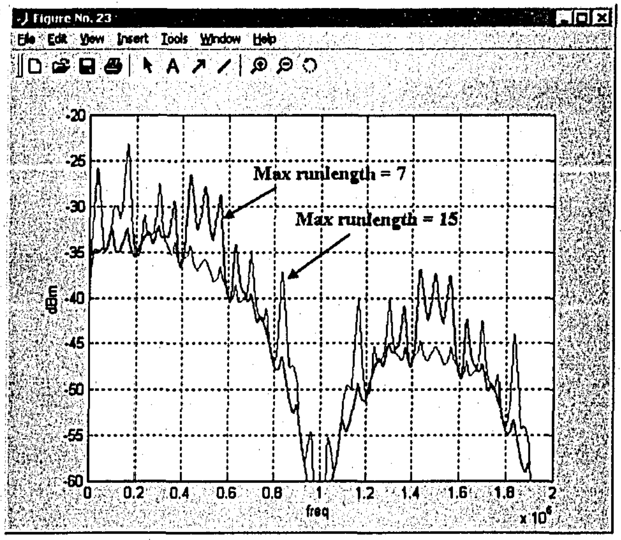
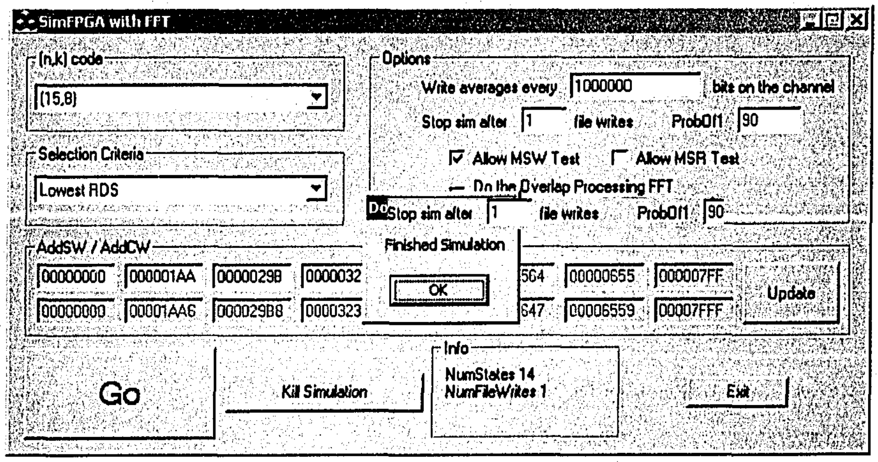


Figure 5-22: 90% ones - Best and worst AddCW set

5.7. Spectra of larger codes

Based on the AddCW recommendations of Section 4.15 the PSD of larger codes based on the (31,26) Hamming code are now investigated. The PSD of these codes however cannot easily be calculated using the Cariolaro and Tronca technique used so far. This is because their method involves modeling an encoder as a FSM and analyzing it as a Markov chain which involves numerous calculations for every possible way into and out of a state. In the case of the (15,8) codes this meant on average 2^8 ways into a state, 2^8 ways out of each state, and sometimes as many as 2^8 states. Thus the number of calculations is on the order of 256^3. Furthermore the file sizes for the intermediate θ_u, E_u and C_u matrices are on the order of 1MB . Consequently for the (31,23) codes the number of calculations is on the order of $(2^{23})^3 = 8388608^3$ with file sizes for the intermediate θ_u, E_u and C_u matrices well past the order of 1GB. Thus calculating exact

PSDs for larger codes proved to be impractical with today's computing power. Thus an alternative approach used was to simulate the encoding scheme and calculate the PSD using FFT techniques.


SimFPGA with FFT

(n,k) code: (15,8)

Selection Criteria: Lowest RDS

Options
Write averages every 1000000 bits on the channel
Stop sim after 1 file writes ProbOf1 90
[✓ Allow MSW Test] Allow MSR Test
Do the Overlap Processing FFT
Stop sim after 1 file writes ProbOf1 90

AddSW / AddCW
00000000 000001AA 00000298 0000032 564 00000655 000007FF
00000000 00001AA6 000029B8 0000323 647 00006559 00007FFF

Finished Simulation

Info
NumStates 14
NumFileWrites 1

Figure 5-23: Simulation for finding the PSD

Figure 5-23 shows the program **SimFPGAwithFFT** written to find the PSD through simulation. This program uses a technique called FFT overlap processing with a Hanning window to find an accurate spectrum representation. This technique involves windowing 1024 CWs, taking an FFT, sliding the window by 256 CWs (i.e. overlapping 768 CWs), taking an FFT, and repeating over 64 windows and averaging the results. Figure 5-24 shows two PSDs found through simulation. Comparison with Figure 5-21 shows the accuracy of this technique. The noisiness of the spectra is due to the fact that the transmission is random and an infinite number of samples would need to be averaged in order to precisely match the PSD of Figure 5-21.

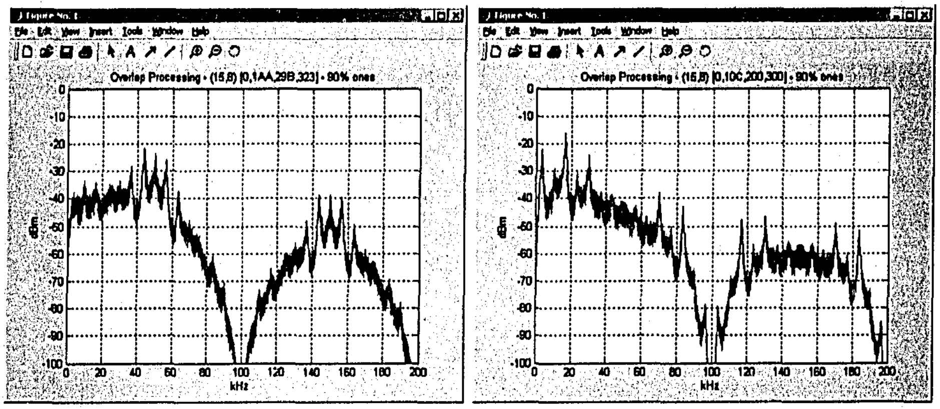


Figure 5-24: 90% ones – best AddCW set followed by a worst AddCW set

Using the AddCW recommendations of Section 4.15, Figure 5-25 shows the PSD of the (31,23) code, with a source emitting logic 1s with 10% probability, using the AddSW sets {`0000000h`, `084D4D4h`, `1323232h`, `189B9B9h`, `2764646h`, `2CDCDCDh`, `37B2B2Bh`, `3FFFFFFh`} and {`0000000h`, `08AAAAAh`, `1555555h`, `1832323h`, `27CDCDCh`, `2AAAAAAh`, `3755555h`, `3FFFFFFh`}. It is immediately obvious from Figure 5-25 that the first AddCW set appears to have many bands of high power at high frequency. On the other hand the second AddCW set has a majority of power at high frequencies. This is indicative of a transmitted sequence that has very short runlengths.

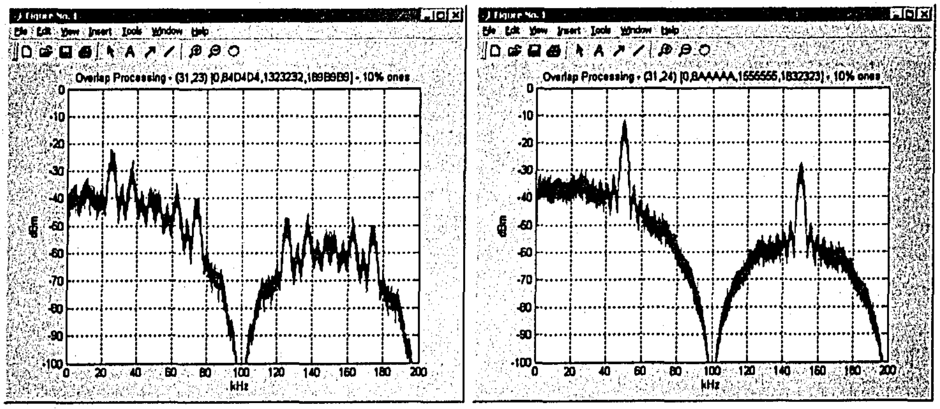
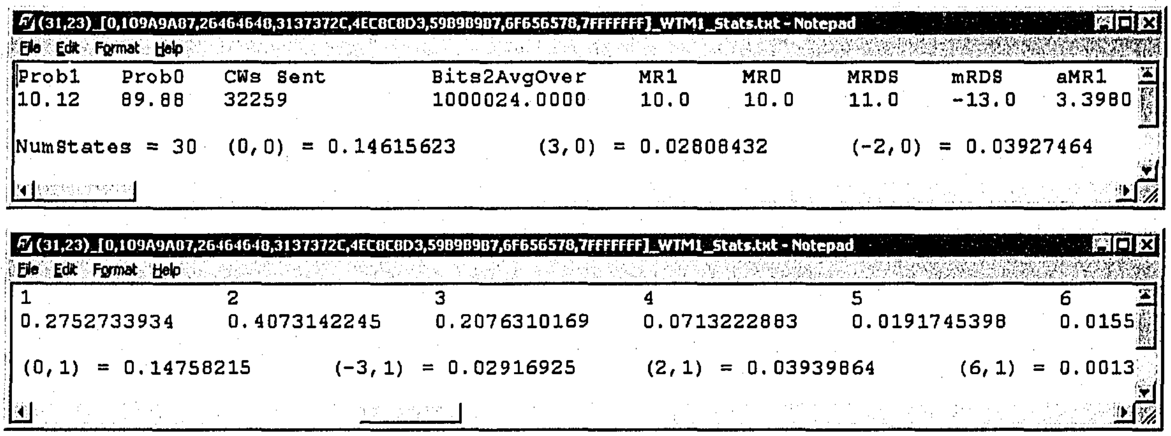


Figure 5-25: 10% ones – Two (31,23) codes

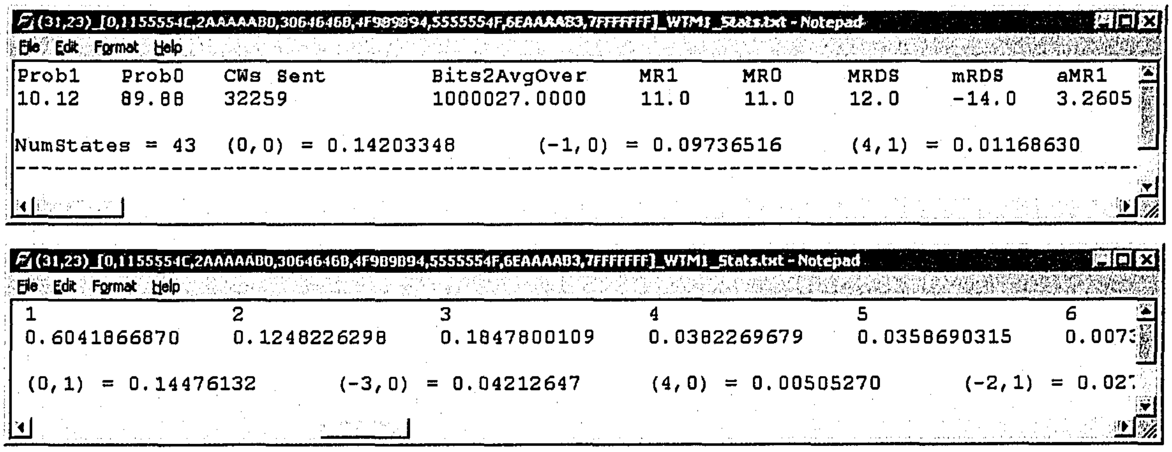
This can be further understood by inspecting the statistics of the transmitted sequence. In addition to the PSD the SimFPGAwithFFT program generates a file containing the number of states in the encoder, maximum runlengths and AddCW usage. These files are shown in Figures 5-26 and 5-27 respectively.



```
(31,23)_[0,109A9A07,26464640,3137372C,4EC8C8D3,59B9B9B7,6F656578,7FFFFFFF]_WTM1_Stats.txt - Notepad
File  Edit  Format  Help
Prob1     Prob0     CWs Sent              Bits2AvgOver     MR1       MR0       MRDS      mRDS      aMR1
10.12     89.88     32259                 1000024.0000     10.0      10.0      11.0      -13.0     3.3980

NumStates = 30  (0,0) = 0.14615623        (3,0) = 0.02808432        (-2,0) = 0.03927464
```

```
(31,23)_[0,109A9A07,26464640,3137372C,4EC8C8D3,59B9B9B7,6F656578,7FFFFFFF]_WTM1_Stats.txt - Notepad
File  Edit  Format  Help
1                2                3                4                5                6
0.2752733934     0.4073142245     0.2076310169     0.0713222883     0.0191745398     0.0155

 (0,1) = 0.14758215       (-3,1) = 0.02916925       (2,1) = 0.03939864       (6,1) = 0.0013
```

Figure 5-26: Statistics of the (31,23) code with the 1st AddCW set



```
(31,23)_[0,1155554C,2AAAAAB0,30646468,4F9B9B94,5555554F,6EAAAAB3,7FFFFFFF]_WTM1_Stats.txt - Notepad
File  Edit  Format  Help
Prob1     Prob0     CWs Sent              Bits2AvgOver     MR1       MR0       MRDS      mRDS      aMR1
10.12     89.88     32259                 1000027.0000     11.0      11.0      12.0      -14.0     3.2605

NumStates = 43  (0,0) = 0.14203348        (-1,0) = 0.09736516       (4,1) = 0.01168630
------------------------------------------------------------------------------------------------------
```

```
(31,23)_[0,1155554C,2AAAAAB0,30646468,4F9B9B94,5555554F,6EAAAAB3,7FFFFFFF]_WTM1_Stats.txt - Notepad
File  Edit  Format  Help
1                2                3                4                5                6
0.6041866870     0.1248226298     0.1847800109     0.0382269679     0.0358690315     0.007

 (0,1) = 0.14476132       (-3,0) = 0.04212647       (4,0) = 0.00505270       (-2,1) = 0.02
```

Figure 5-27: Statistics of the (31,23) code with the 2nd AddCW set

Comparison of these two files shows that the first set only has maximum runlengths of 10 $(MR1 = MR0 = 10)$, while the second set has maximum runlengths of 11 $(MR1 = MR0 = 11)$. On the other hand, the second set has a 60.4% chance of runlengths of length 1, while the first set only 27.5%. This was clearly evident from the PSD plots.

Similarly it was noticed that the first set had more power at intermediate frequencies than at its highest frequencies. This can be seen in the statistics file since the probability of a run of length 2 is 40.7% and the probability of a run of length 3 is 20.7%.

5.8. PSD Summary

The multimode coding technique introduced in Chapter 3 is designed to guarantee balanced transmission and remove long runs of like valued bits. An output sequence that has long runs of 0s and 1s will appear as a slowly varying square wave, and an output sequence with short runs of 0s and 1s will appear as a rapidly varying square wave. Hence these sequences can be analyzed in the frequency domain in terms of their PSD.

Cariolaro and Tronca [24] outline an efficient procedure for calculating the PSD of a coded system, the details of which were demonstrated in this chapter, with further examples found in Appendix A. It was shown that in systems where the number of states or size of the code is too large, it is preferred to simulate the PSD.

The following chapter looks at the FPGA hardware implementation of this multimode encoder. It also compares measured PSD results to the theoretical and simulated PSDs calculated in this chapter.

6. Implementing the Multimode Encoder and Decoder in Hardware

The previous chapters introduced the combined EC and CS coding concept, as well as outlined how to evaluate performance in both the time and frequency domain. This chapter chronicles how a working transmitter and receiver was implemented on two FPGA boards and PSD measurements were taken for comparison with results from Chapter 5. For proof of concept the goals of this communication system were simple:

1. Implement the combined CS coding and EC coding technique;
2. Transmit a file from transmitter to receiver to verify the encoding and decoding techniques;
3. Measure the PSD of the transmitted sequence to verify it agrees with calculations.

A block diagram of the communication system is shown in Figure 6-1. It consists of the two FPGA boards TX FPGA and RX FPGA, the three wires over which they communicate, and the controlling computers TX PC and RX PC. Computer control allows for a more sophisticated user interface than what is present on the FPGA boards themselves. Furthermore this setup allows a user to select any file on the local TX PC and transmit it to the RX PC using the combined EC and CS coding technique. Typically no distinction will be made between the PC and the FPGA and they will collectively be referred to as the TX and the RX.

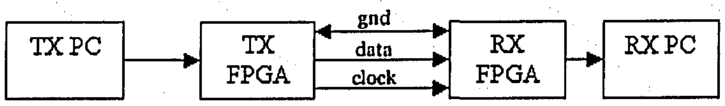


Figure 6-1: Block diagram of the FPGA communication setup

The TX and RX communicate using three wires: serial data, serial clock and ground. The serial data line transmits the actual coded sequence and the ground is simply the common line between the two FPGAs. These two wires together would form what is commonly known as a twisted pair. A third wire is also included which allows the RX clock to synchronize with the TX. This is because a typical RX must synchronize to the clock rate of the TX in order to determine the duration of a single bit. Conventional communication systems accomplish this using a phase lock loop (PLL) to derive this clock from the transitions in the incoming bit stream on the serial data line. In fact the combined EC and CS code introduced in this thesis is designed to aid PLLs to accomplish this task. However the added complexity of designing a stable PLL for use with the RX was considered unnecessary since the analysis reported in Chapters 4 and 5 has proven that the outgoing bit stream of this multimode coding technique contains numerous transitions. Furthermore this coding technique was designed to be added to an existing communication system. Therefore as long the calculated PSDs presented in Chapter 5 match with the measured PSDs of this hardware implementation, the system would be proven to be valid even without the implementation of a PLL at the receiver. As a result, to simplify the receiver circuitry a clock signal was connected directly from the transmitter.

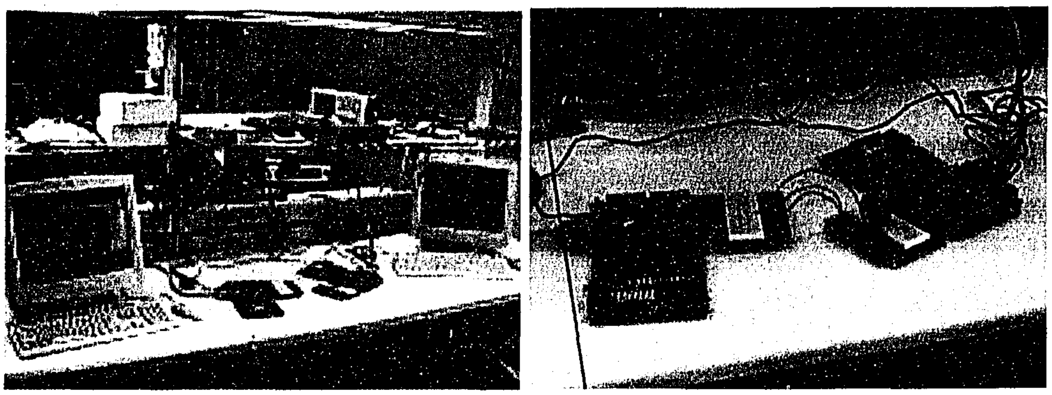


Figure 6-2: FPGA transmitter and receiver communicating

Figure 6-2 shows the laboratory setup of the block diagram of Figure 6-1. Two PCs and two FPGAs using three connecting wires are shown. The TX and RX each consisted of a Digilent Digilab 2E (D2E) development board as shown in Figure 6-3, and a Digilent Digital I/O board 2 (DIO2) expansion board shown in Figure 6-4. The D2E boards each featured a 200K-gate Xilinx Spartan 2E FPGA (XC2S200E) in a PQ208 package running at 50MHz with 143 user I/O pins. The DIO2 boards provided input and output functions with fifteen push buttons, eight toggle switches, sixteen LEDs, an LCD screen and four 7-segment displays.

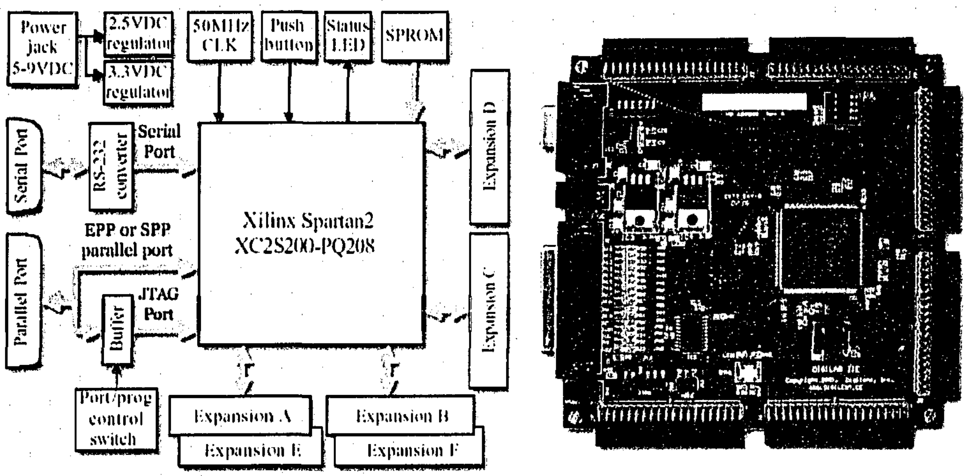


Figure 6-3: Digilent Digilab FPGA board

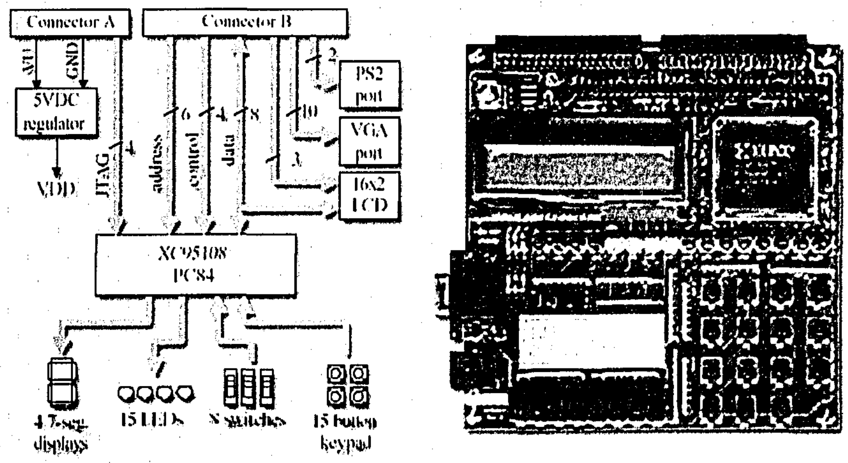
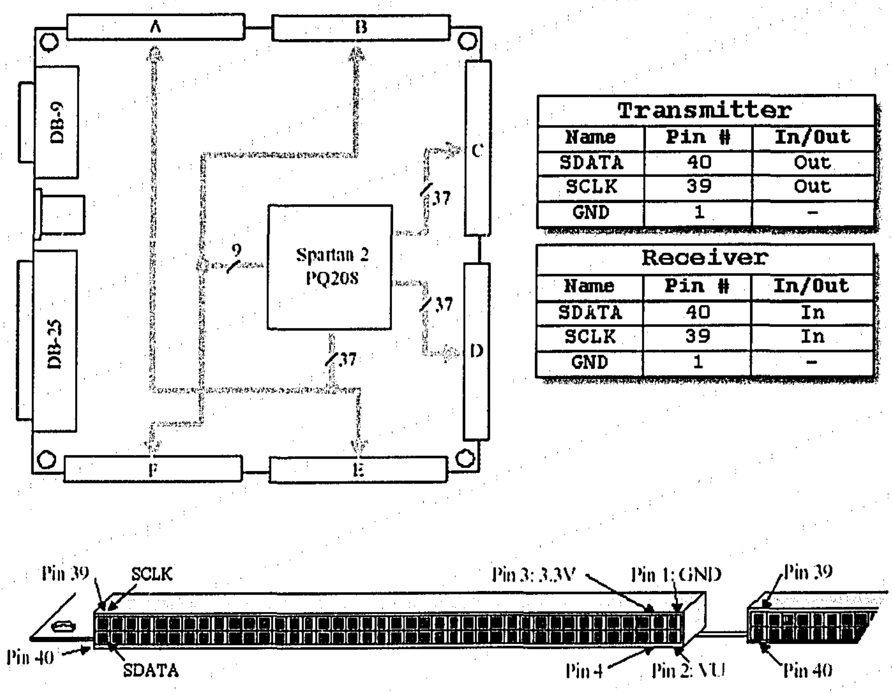


Figure 6-4: DIO2 peripheral board for input and output

6.1. Pinouts

On each FPGA board pin 1 was the common ground (**GND**), pin 73 was the serial data (**SDATA**) and pin 74 was the serial clock (**SCLK**). The data and clock pins were easily accessed from the D-connector on the development boards in locations D40 and D39 as shown in Figure 6-5. These translated into the physical pin locations on the FPGA as pin 73 and 74 respectively.



Transmitter		
Name	Pin #	In/Out
SDATA	40	Out
SCLK	39	Out
GND	1	-

Receiver		
Name	Pin #	In/Out
SDATA	40	In
SCLK	39	In
GND	1	-

Figure 6-5: The pinouts used on the FPGA boards

6.2. SDATA and SCLK signals

Figure 6-6 shows a typical data and clock sequence. It is clear from the digital oscilloscope screenshots that SCLK rises halfway through SDATA. Thus the RX samples SDATA midway through each bit on the rising edge of SCLK.

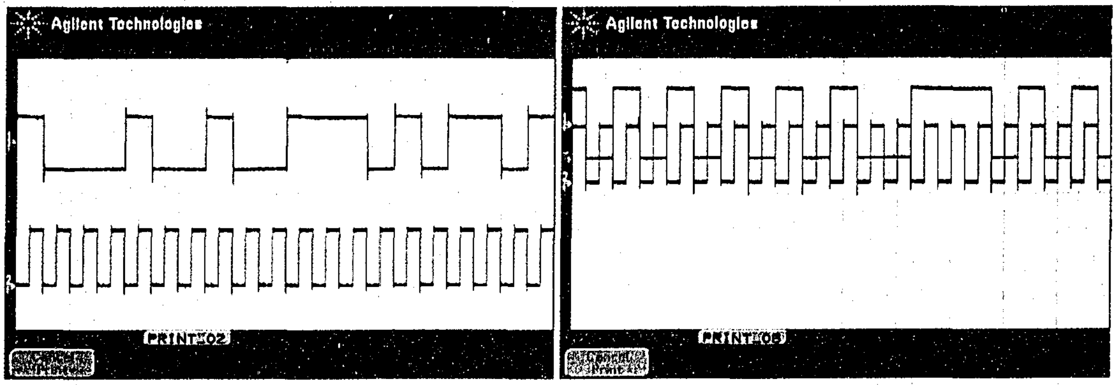


Figure 6-6: SDATA sequence (top) and SCLK sequence (bottom)

6.3. Controlling the FPGAs over the PC parallel port

The D2E development boards each had a 25-pin parallel port connector that is used during programming. This port could also be used to communicate with the FPGA after programming. The parallel port of a PC, when accessed in standard parallel port (SPP) mode, is typically located at address 378h (BIOS configurable), and is made up of three individual ports. Of the 25 pins there is an 8-bit bi-directional data port (address 378h), a 5-bit input only status port (address 379h) and a 4-bit output only control port (address 37Ah). The remaining 8 pins are all grounds as shown in Figure 6-7.

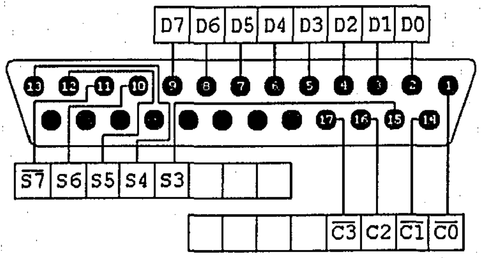


Figure 6-7: Parallel Port connector on the back of a PC

It is also important to note that some of the bits of the SPP are inverted by PC hardware (7404 inverters) for legacy printer reasons. This must be compensated in software by inverting these bits before writing to the write only control port, and by inverting these bits when reading from the read only status port.

PC parallel ports when used in SPP mode run at 125 kHz. This means that new data can be written to or read from the SPP bus every 8 us. However when writing data there is often a 2us settling time in the form of random transitions on the parallel port data bus. For example, using the Agilent Logic Wave logic analyzer it can be seen that there are transient values on the data bus when the data is changing from 65h to 00h as shown in Figure 6-8.

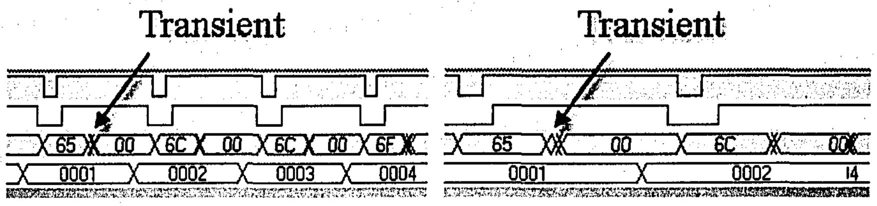


Figure 6-8: The 'transient' nature of the parallel port when switching values

This presents a small problem when dealing with an FPGA that can sample the parallel port inputs at 50 MHz. For instance, if the transient values that occur when the data is changing from 65h to 00h proceeds through the series 65h, 61h, 21h, 20h, 00h, the FPGA would see the above sequence as five separate and valid data values. This issue must be resolved by implementing a debouncing circuit in VHDL code. That is, before reading any change on the parallel port inputs, a timer should begin that will re-sample the parallel port bits after the 2us settling time has passed. Then a simple comparison with the original data determines what the new value is, and the transient values are ignored as shown in the FSM in Figure 6-9.

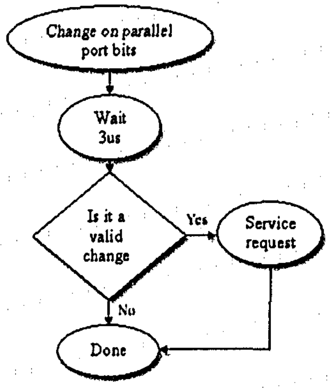


Figure 6-9: FSM model of the parallel port debounce circuit

6.4. Transmitter FPGA State Machine

The transmitter FPGA runs a main state machine that consists of four states as shown in Figure 6-10. On power up it is forced into state 1 which is the reset state that initializes the encoder. After this the FSM cycles through states 2, 3, and 4 indefinitely. Only by powering the FPGA off and on, or through user input in state 2 will the system ever enter the reset state again.

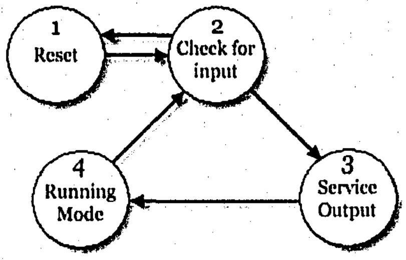


Figure 6-10: Main transmitter FPGA FSM

1. Reset State – this state is entered during power on or through specific user input (such as command Eh in Table 6-1). Once entered all variables and buffers are cleared and the system is ready to accept further commands such as transmitting a file.

2. Check for Input – in this state the transmitter checks for input commands from the parallel port as discussed in Section 6-3. User input is set up as follows. The 4-bit control port of the SPP is used to issue one of sixteen individual commands to the FPGA, with additional data for each command coming from the 8-bit data port. These are summarized in Table 6-1.

For example, to write the data value AAh to the FPGAs RAM at address 0E20h, begin by placing the FPGA into a HighZ state by issuing command Fh. This makes the FPGA back-off the bi-directional data bus so the parallel port can write data without any bus contention. The next step is to assign the RAM address to modify by writing the address to the 12-bit FilePointer. Since the data bus is only 8-bits this must be done is two steps. First the upper 4-bits of the address is written to the data bus by writing data value 0Eh followed by command 2h. Then the lower 8-bits is written to the data bus by writing data value 20h followed by command 2h. At this point the FPGA has latched the value E20h into the FilePointer register. Finally in order to write a new value AAh into this RAM location, simply write data value AAh, followed by command 1h.

Table 6-1: Commands for both the transmitter and receiver FPGAs

4 bit Control Port Command	Function	Description
0h	RAM Read mode	Latch file pointer RAM address Enable RAM read mode Increment file pointer
1h	RAM Write mode	Latch file pointer RAM address Enable RAM write mode Increment file pointer
2h	Assign File Pointer	Write to file pointer. This allows for non-sequential access to RAM
3h	Assign File Size	File size of the data written to RAM
4h,5h	Write AddSWs	Assign AddSWs
6h,7h	Read AddCWs	Read AddCWs
8h	Set number of AddCWs	Set either 0,2,4 or 8 AddCWs
9h	Initialize the Encoder	This builds the AddCWs from the AddSWs
Ah	Running Mode Continuous	Begins transmitting the file continuously. When it reaches the end of the file, it does not stop and simply wraps around.
Bh	Running Mode Once	Begins transmitting the file, and when it reaches the end of the file it stops.
Ch	Set Serial Transmission Speed	Either 1 kHz, 10 kHz, 100 kHz or 1MHz
Dh	Enable RAM	If RAM is in ReadMode (0), data is now put onto the data bus. If RAM is in WriteMode (1) it will read data from the data bus and store it.
Eh	Reset	Reset all variables
Fh	HighZ	Similar to reset, except not all variables are reset, and data bus that is connected to the parallel port is put into HighZ state.

3. Service Output – This state involves writing data to the 16 character by 2 line LCD screen (32 ASCII characters) on the DIO2 expansion board. The LCD was used for debugging and testing purposes. For instance in order to verify that the multimode coding system was choosing the correct CW at each coding interval, a debug mode was set up that allowed stepping through the CW selection process.

In this mode there needed to be a way to display up to a 32 bit number (8-hex characters). The LCD screen was the only viable choice since the DIO2 board only had four 7-segment displays and thus could only display 4-hex characters.

It is important to note that the LCD screen uses the Samsung KS0066 controller which has enough Display Data RAM (DDRAM) to handle up to a 40 character by 2 line display. Thus each of the 80 DDRAM location corresponds to a single character location on the screen even though physically the LCD screen only has 32 character locations as shown in Figure 6-11. This must be taken into account in order to see the text displayed correctly. Thus it is not immediately obvious but the simplest way of writing the 10 character string "Hello" and "World" on two separate lines is to use 45 characters, not 10. This is accomplished in the following manner.

Displaying the 5 character string "Hello" on the first line is trivial since it requires placing the LCD screen into data mode followed by sending the five ASCII characters 48h, 65h, 6Ch, 6Ch, 6Fh for 'H', 'e', 'l', 'l', 'o'. Writing the 5 character string "World" on the second line however is a bit more involved.

In order to get to the next line on the LCD screen the cursor must manually be moved 35 locations to the right since the generic KS0066 controller has 40 DDRAM locations per line. One method to do this is to switch the LCD screen into control mode followed by sending the "move cursor right" command 35 times. While this is not difficult, switching into control mode can be avoided by just writing 35 spaces instead (ASCII character 20h). It is then just a matter of writing the string "World" which is done by sending the 5 ASCII characters 57h, 6Fh, 72h, 6Ch, 64h, for 'W', 'o', 'r', 'l', 'd'. This is illustrated in Figure 6-11, where the LCD and corresponding DDRAM locations are shown.

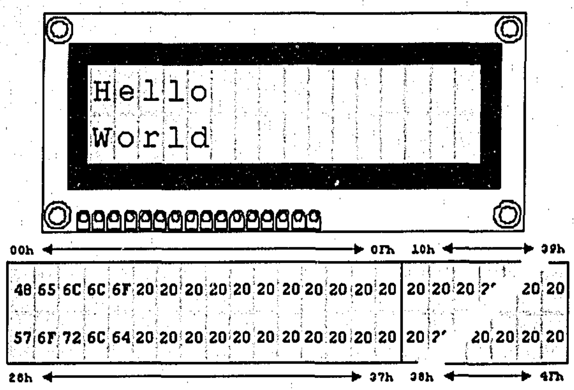

Hello
World
00h
0Fh
10h
39h
48 65 6C 6C 6F 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
57 6F 72 6C 64 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
28h
37h
38h
4Fh

Figure 6-11: LCD screen and corresponding DDRAM locations

Accordingly the simplest way to display a string on the LCD screen without any additional overhead is to pad the string out to 80 characters with spaces. In this manner each string starts at the first location of the LCD screen and overwrites every DDRAM location. This has the added benefit of erasing any past characters that were on the LCD, thus clearing the screen before displaying each new string without ever having to switch modes.

Table 6-2: String table held in the first 400h bytes of RAM

Ram Address	String
000h	Multimode Coder Online
051h	Source Word
0A2h	Code Word
0F3h	Hamming Code is
144h	Number of AddCWs
195h	RDS
1E6h	Num Tran
237h	1st Tran at loc / MSW
288h	Winning Location
2D9h	Last Bit is
32Ah	Not Updating RDS or LB for Test
37Bh	Updating RDS and LB as normal

For this reason all strings displayed on the LCD are stored in RAM as 80 bytes plus a 0 terminator for a total of 81 bytes (51h). That is, the 1st string is stored from 000h to 050h, the 2nd string is stored from 051h to 0A1h and so on. Therefore the first 1024 bytes (400h) of RAM are used to hold a string table that consists of 12 strings as shown in Table 6-2.

In accordance with the above discussion the following FSM in Figure 6-12 illustrates how to display information on the LCD screen. One of the 12 strings to display must be chosen, followed by an optional number.

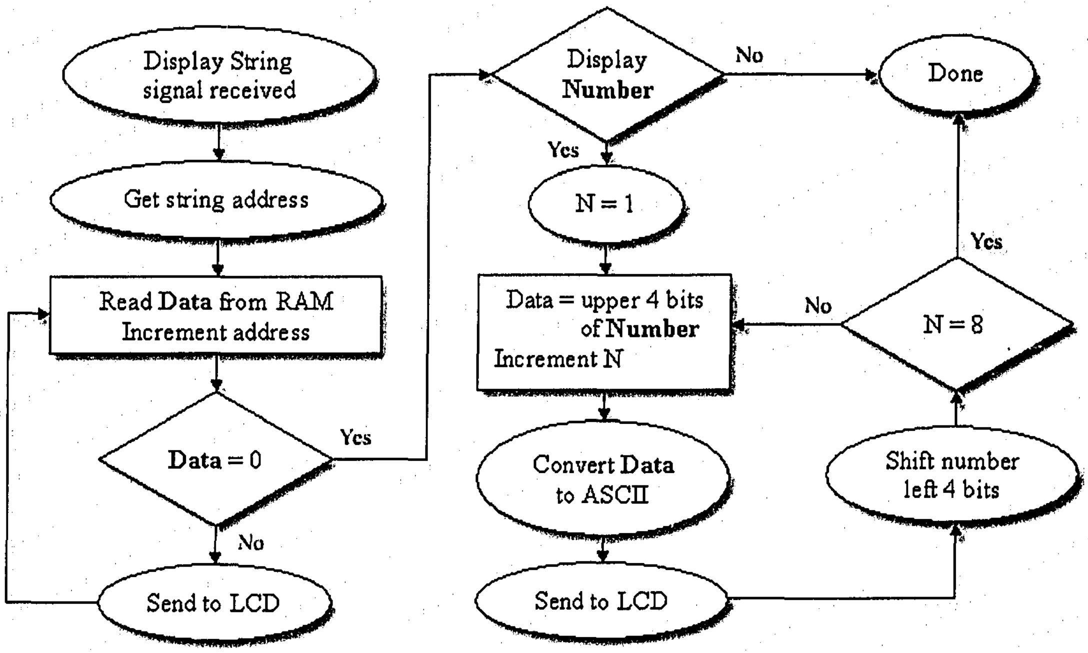


Figure 6-12: FSM for displaying strings and numbers on the LCD screen

4. Running Mode – When the TX is in this state it is transmitting data serially to the RX. This process involves reading SWs from RAM, encoding them as CWs, performing the multimode CW addition, calculating statistics for each candidate CW to see which one maximizes the CS coding requirements, and finally transmitting these CWs serially. There are two modes of operation.

I. **Run Once Mode** – This mode is used for sending a data file once from the TX to the RX. The file is stored in the FPGA's RAM, and the TX sends each CW on the channel until the end of file is reached. Once the end of file is reached the TX automatically appends a 00h data byte to indicate the end of the file. Note that depending on the code and AddCWs used, this 00h byte is transmitted fully EC and CS coded.

II. **Run Continuously Mode** – This mode is used when measuring the PSD so the spectrum analyzer has a continuous signal to measure. In this mode the file is transmitted repeatedly which means when the end of file is reached, the transmitter wraps around to the beginning of the file.

Showing a complete block diagram of the Running mode FSM is not practical due to its size and complexity. Instead Figure 6-13 shows a high-level block diagram with a text description of the functionality of each state.

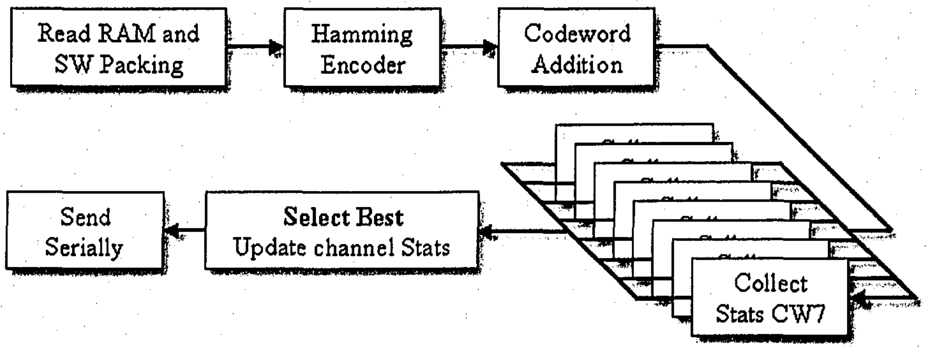


Figure 6-13: High-level block diagram of the running mode state FSM

Each block in Figure 6-13 represents a sequential FSM. As the FPGA was programmed with VHDL, each block also represents a VHDL module. The following section briefly describes the functionality of each one of these modules as well as the corresponding test benches to verify their functionality.

Read RAM and SW Packing: Since a data file is organized as a series of 8-bit bytes, the first major task is to take the file data and convert it into SWs of sizes as small as 1-bit and as large as 26-bits depending on the code being used. This is not straightforward since, for example, with the (15,11) code 11-bits are needed to form a single SW. This would require reading two bytes from RAM to get 16-bits, to then form an 11-bit SW with 5 bits left over. Building the next SW however only requires reading from RAM once (unlike twice the last time) to get an additional 8-bits for a total of 13-bits. The next 11-bit SW can now be created leaving 2-bits this time and so on. Thus the number of bits left over and the number of RAM reads varies for each coding interval depending on the code being used.

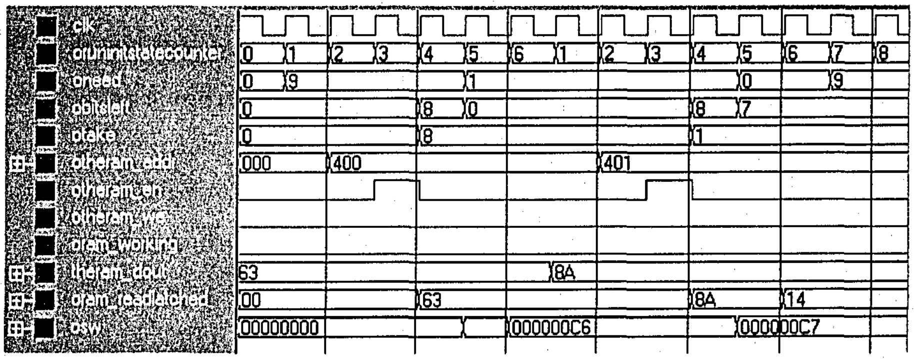


Figure 6-14: Testbench for SW packing

Figure 6-14 shows the testbench for this module called SW Packing. This particular test shows a (15,9) SW being formed. From initialization the number of bits needed to form the SW (**oneed**) is set to 9. The first byte of the file is read (**theram_dout**) from RAM address 400h (**otheram_addr**). Its value is 63h and it is latched (**oram_readlatched**). The 8-bits are then loaded into **osw** and it can be seen that one more bit is needed (**oneed**) to form the 9-bit SW. Thus another byte is read (**theram_dout**) from address 401h (**otheram_addr**) and its value 8Ah is latched (**oram_readlatched**). Here only the MSb is taken from **oram_readlatched** and it is shifted left by one bit leaving 14h. Now the complete

9-bit SW is formed (**osw**) which is 0C7h or 011000111b. Thus 7-bits remain (**oneed**) in **oram_readlatched** for building the next SW. To clarify how the two bytes 63h and 8Ah form the 9-bit SW C7h, the process is shown in Figure 6-15.

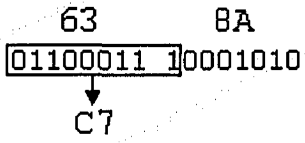


Figure 6-15: 63h and 8Ah form the 9-bit SW C7h or 011000111b

Hamming Encoder: As discussed in Section 2-14, the EC Hamming code can be implemented using very low level logic blocks. It is a configurable stand-alone module that allows selecting the generator polynomial as well as the length of the code. In addition, encoding can be done in parallel on the FPGA so a SW to CW conversion takes only a single clock cycle.

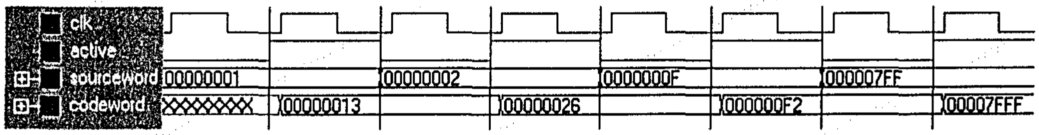


Figure 6-16: Testbench for the Hamming encoder

Figure 6-16 shows the testbench for the Hamming encoder module using a (15,11) code. Encoding of the SWs 001h, 002h, 00Fh, and 7FFh each take a single clock and are encoded to 0013h, 0026h, 00F2h and 7FFFh respectively. Note that all SWs and CWs are represented using 32 bits and are truncated before serial transmission according to the *n* and *k* values of the (*n*,*k*) code.

Codeword Addition: CW addition is the simplest block/module of Figure 6-13. As discussed in Section 2.5, modulo-2 addition can be done using XOR gates. It too can be done in parallel on the FPGA. Figure 6-17 shows the testbench for the

(15,9) encoder. The CW 0C70h (**ocw**) is added modulo-2 to the four AddCWs 0000h, 024EBh, 5B14h and 7FFFh, to form the new CWs (**onewcw0** to **onewcw3**) 0C70h, 289Bh, 5764h, and 738Fh. Note that the flexible algorithm used always adds each CW to eight AddCWs for simplicity. Thus the testbench in Figure 6-17 displays eight new CWs, but only the first four are valid. The last four (**onewcw4** to **onewcw7**) new CWs would not be considered for transmission.

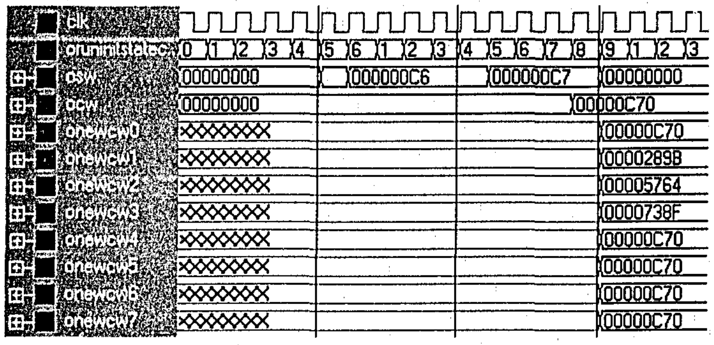


Figure 6-17: Testbench for CW addition

Collect Statistics: As discussed in Section 4-1, the multimode coder evaluates the CW statistics and selects the best CW that minimizes the RDS and/or the MSW as well as maximizes the transitions. These statistics are gathered using the module called StatsGen. This module is instantiated once for every possible AddCW and thus the statistics for all 8 AddCWs can be calculated in parallel.

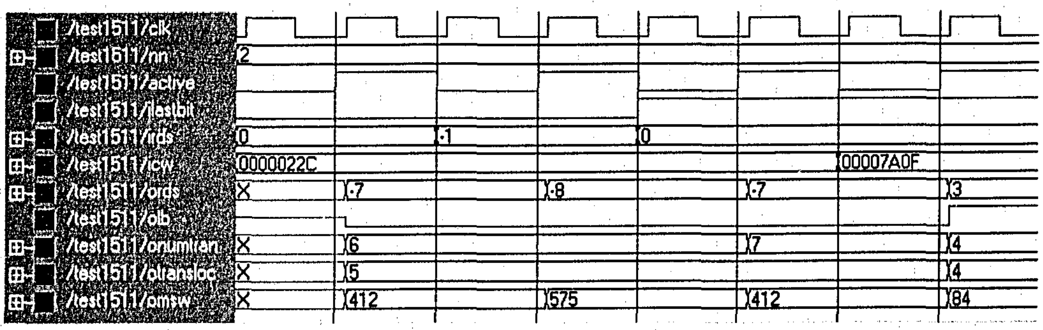


Figure 6-18: Testbench for the StatsGen module

Figure 6-18 shows the testbench for the StatsGen module. As discussed in Sections 4-2 and 4-3 the encoder keeps track of the current RDS **(irds)** and current last bit **(ilastbit)** of the previous CW transmitted on the channel (these comprise the state of the encoder). This continuous feedback is updated with every CW sent. To verify this functionality the testbench of Figure 6-18 collects the statistics on the same input CW **(icw)** 022Ch for the (15,9) code three times, but with different **irds** and **ilastbit** equaling (0,0), (-1,0) and (0,1). As a result when the first CW 022Ch is sent, the calculated RDS is -7 and the number of transitions is 6. However when the CW is sent a second time the current RDS is at –1, resulting in a calculated RDS that is one less than before at -8. This however has no effect on the number of transitions but does result in a higher MSW. The third and final time CW 022Ch is sent the RDS is back at 0 resulting in an RDS of –7 again. However since the current last bit was changed to a 1 this results in an extra transition for a total of 7. This extra transition occurs between the last bit and the start of the CW. Finally the testbench also transmits CW 7A0Fh with an **irds** and **ilastbit** of (0,1). This CW is more balanced than 022Ch resulting in an RDS of 3. However this CW has fewer transitions with only 4. The results from the testbench in Figure 6-18 are summarized in Table 6-3.

This StatsGen module is the most important block of Figure 6-13. This is because the information gathered in this module is used to decide which new CW will represent the original SW, i.e. the multimode coding. This information is passed to the next block **Select Best**, which decides which of the new CWs will be transmitted.

Table 6-3: Summary of the StatsGen module gathering statistics

CW	Current RDS	Current LB	RDS	Tran	MSW	1stTran
022C	0	0	-7	6	412	5
022C	-1	0	-8	6	575	5
022C	0	1	-7	7	412	5
7A0F	0	1	3	4	84	4

Select Best and Update Channel Statistics: Once the statistics have been collected a decision must be made to determine which CW will be sent on the channel. In the (15,x) and (7,x) encoders the decision was based on which of the new CWs had the lowest RDS, followed by the tie breaking criteria which included most transitions, lowest MSW, and the CW with the first transition. Note that the CW with the first transition test was only used to provide a final tie break in order to always be able to predict which CW the encoder would select in the unlikely event that two or more CWs had the identical RDS, MSW, and number of transitions. Simulations demonstrated that this selection metric was used less than 1% of the time with any AddCW set.

Also note that for the (31,x) codes only the lowest RDS, most transitions and first transition metrics were used. This was due to the size and resource limitations of the FPGA. Missing the MSW test has very little impact on the code performance, since the RDS and transitions test were the most important, and the MSW test was only used to break ties.

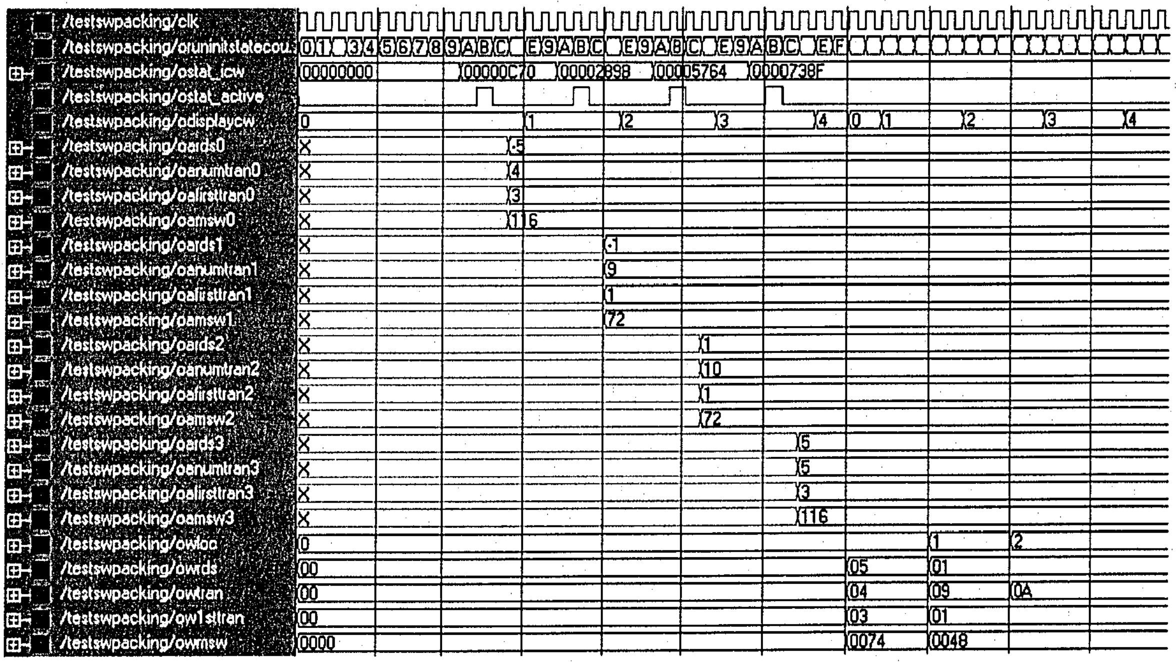


Figure 6-19: Testbench for the SelectBest module

Continuing with the example from Figure 6-17 the testbench for the SelectBest module is shown in Figure 6-19. Here the statistics are compared for the four CWs 0C70h, 289Bh, 5764h, and 738Fh. This is done sequentially. The algorithm initially chooses CW0 to be the best, and then the module compares each new CW's statistics to the current *best* statistics and only updates the *winner* if the statistics are better. For example from Figure 6-19 it can be seen that the winning location (**owloc**) is temporarily changed from 0 to 1, before location 2 is selected as the best.

This process is summarized in Table 6-4 where it can be seen that CW1 (289Bh) is better than CW0 (0C70h) since it has a lower absolute RDS of –1 as compared to –5. CW1 (289Bh) is then the current best until comparison with CW2 (5764h) indicates that they are tied in terms of lowest absolute RDS. Since both of these CWs will minimize the RDS, preference should be given to the CW that has the most transitions. Thus CW2 (5764h) with 10 transitions is chosen to be the best CW to transmit since it has one more transition than CW1 (289Bh). Comparison with CW3 (738Fh) indicates that CW2 (5764h) is still the better choice and thus it is the overall winner and is selected for transmission.

Table 6-4: Summary of the SelectBest module

CW location	CW	RDS	Tran	MSW	1stTran
0	0C70h	-5	4	116	3
1	289Bh	-1	9	72	1
2	5764h	1	10	72	1
3	738Fh	5	3	116	3

Send Serially: Once the best CW is selected it must be transmitted serially along with a serial clock to the RX FPGA. Figure 6-20 shows the testbench for the Parallel2Serial module. Here the best CW 5764h chosen from the testbench above in Figure 6-19 is being transmitted serially. Once the **go** pulse is received the **data** is sent starting with the MSb first on the **sdata** line. The **sclk** signal is also sent with a low to high transition occurring midway between each data bit. Once the entire 15-bits are transmitted the Parallel2Serial module signals that it is **ready** to transmit another CW. This hand shaking is used to prevent the FPGA that is running at 50MHz from swamping the Parallel2Serial module that transmits at a maximum rate of 1MHz.

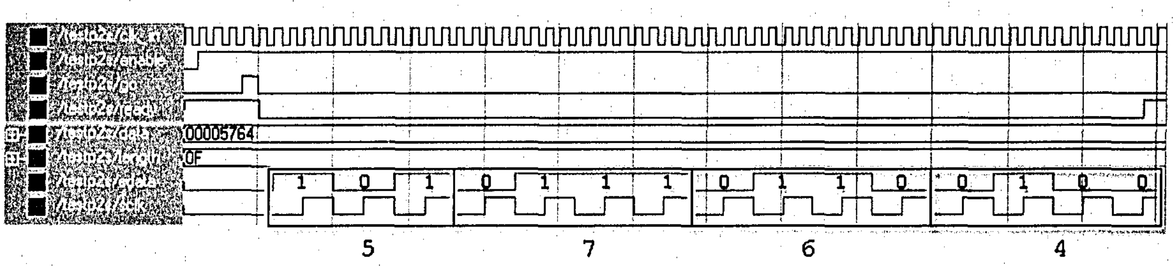
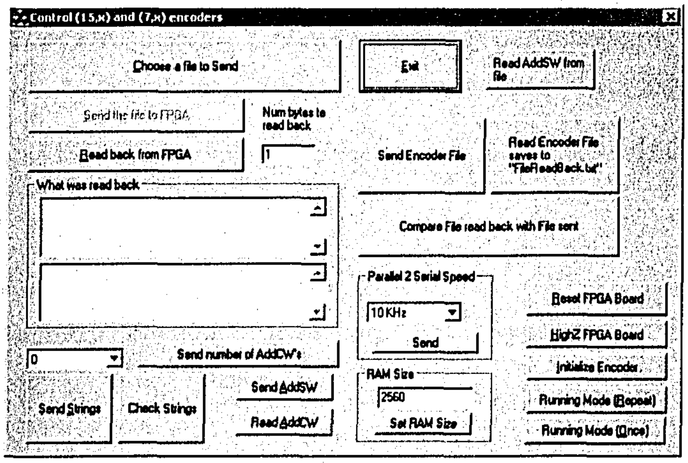


Figure 6-20: Testbench for the Parallel2Serial module

6.5. PC Software: Controlling the TX FPGA

The TX FPGA was controlled using Windows software with the interface shown in Figure 6-21. As shown in Table 6-1 all sixteen commands were issued using the control port, with additional instructions coming from the data port. In order to explain the operation and features of the software, an example is given illustrating how to transmit a file.



Num bytes to read back: 1
Parallel 2 Serial Speed: 10KHz
RAM Size: 2560
0

Figure 6-21: Software that controls the TX FPGA

6.6. Transmitting a File

This section briefly describes how to transmit a standard ASCII text file that contains the 11 byte string "Hello World". The first step after powering on the board is to ensure that the FPGA is in a known state by pressing **Reset FPGA Board**. This forces the FPGA into the reset state and initiates the main FSM of Figure 6-10. Since the FPGA has just powered up the string table from Table 6-2 must be loaded into the first 400h bytes of RAM. This is accomplished using the **Send Strings** button and pressing the **Check Strings** button reads back the string table from RAM to verify that they were stored without error. Now the number of AddCWs can be set up as 0, 2, 4 or 8 using the drop down box as shown in Figure 6-22. For example with length 31 Hamming CWs, this would set up a (31,26), (31,25), (31,24), or (31,23) code respectively.

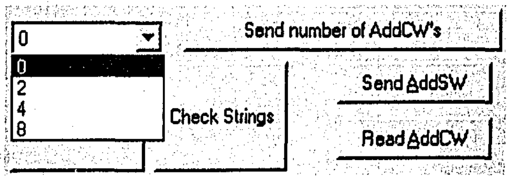


Figure 6-22: Setting up the number of AddCWs

Once the number of AddCWs has been selected, pressing the **Send number of AddCW's** button writes this value to the FPGA. The resultant code is immediately displayed on the DIO2 boards 7-segment display. For example, eight AddCWs has been selected resulting in the (31,23) code as shown in Figure 6-23.

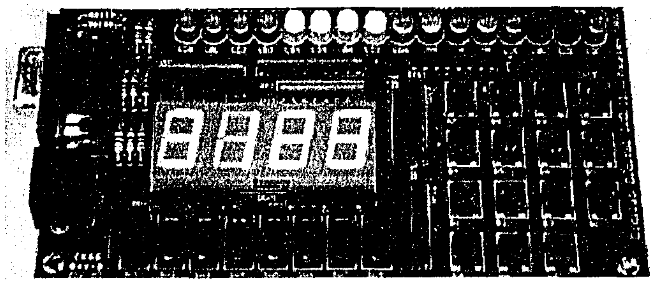
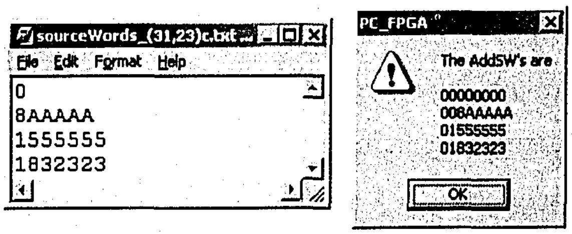


Figure 6-23: Eight AddCWs have been selected for use with the (31,x) code

Once the encoder knows the number of AddCWs to use it must be told what these AddCWs are. This is done by sending the encoder AddSWs which it then encodes into AddCWs. Pressing the **Read AddSW's from File** button allows selection of an ASCII text file that contains the desired AddSWs as shown in Figure 6-24. Note that only the first four AddSWs are stored in this file. This is because the system is set up to automatically use the CW complements. Hence these four AddSWs are encoded to four AddCWs and four AddCW complements, resulting in a total of eight AddCWs.



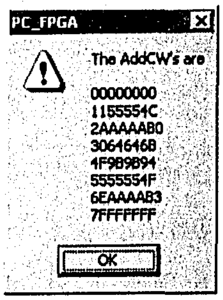
```
0
8AAAAA
1555555
1832323
```

```
The AddSW's are

00000000
008AAAAA
01555555
01832323
```

Figure 6-24: Reading the AddSWs from file to use with the (31,23) code

The **Send AddSW's** button writes these AddSWs into the FPGA memory. The **Initialize Encoder** button causes the encoder to encode the four AddSWs to eight AddCWs using the Hamming encoder module as explained above. The encoder then automatically writes these AddCWs back to the PC to allow for verification that they were encoded correctly, as shown in Figure 6-25.

AddSW	AddCW
00000000	00000000
008AAAAA	1155554C
01555555	2AAAAAB0
01832323	3064646B
027CDCDC	4F9B9B94
02AAAAAA	5555554F
03755555	6EAAAAB3
03FFFFFF	7FFFFFFF



```
The AddCW's are

00000000
1155554C
2AAAAAB0
3064646B
4F9B9B94
5555554F
6EAAAAB3
7FFFFFFF
```

Figure 6-25: Encoder writes back the 8 AddCWs it is going to use

The file to transmit is selected by pressing the **Choose a file to Send** button which brings up a file selection dialog box. Once a file is selected it is transferred to the FPGA's RAM by pressing either the **Send the file to FPGA** button or the **Send Encoder File** button as shown in Figure 6-26. The difference between these two functions is that the **Send the file to FPGA** button writes the file contents into the string tables address space. This is done to allow the loading of different strings into the string table (other than the default hard coded ones) if required. The **Send Encoder File** button writes the file contents directly above the string tables' address space.

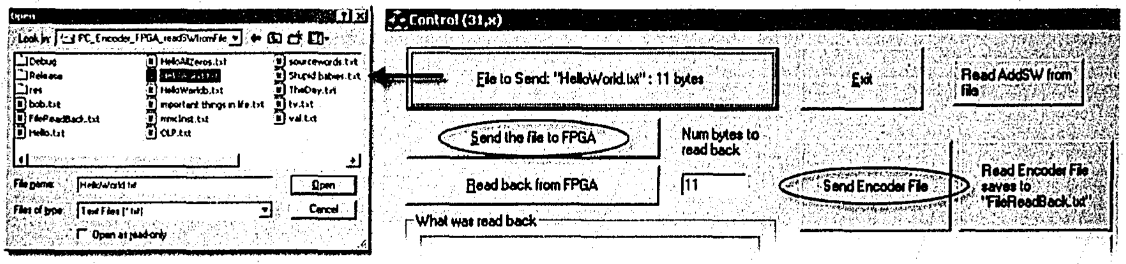
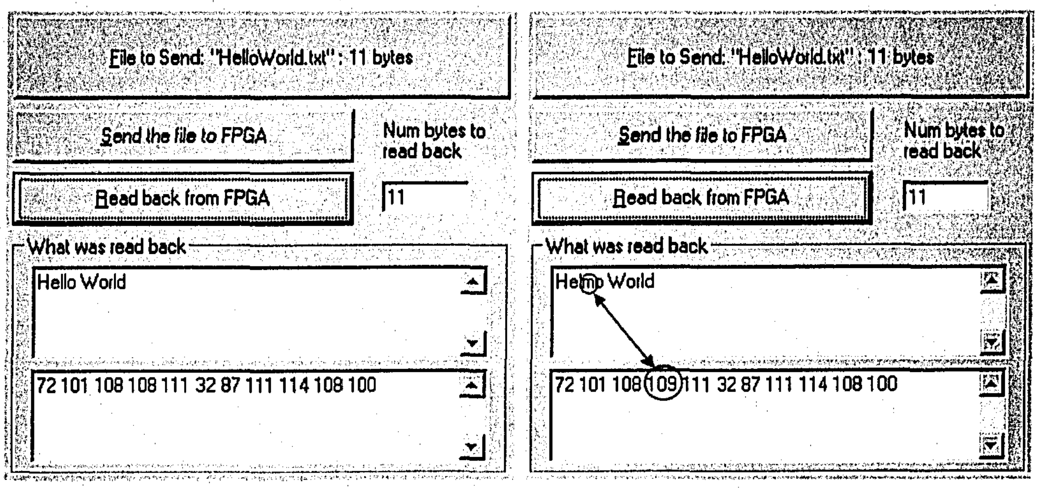


Figure 6-26: Two ways to transmit the file into FPGA RAM

There are two ways to verify that the file was loaded correctly. First by pressing the **Read Encoder File saves to "FileReadBack.txt"**, the file is read back from the FPGA and a byte by byte comparison is done with the original file that was sent. Any errors that occurred are displayed in a popup box and the file read back is saved to the file FileReadBack.txt. This allows for saving and inspection of the data at a later time.

Secondly by pressing the **Read back from FPGA** button the file is read back and displayed in the two windows as shown in Figure 6-27. The top window displays the ASCII representation of the data, while the bottom window shows the decimal representation. Figure 6-27 shows the file "Hello World" was read back and what a possible error might look like.


File to Send: "HelloWorld.txt": 11 bytes

Send the file to FPGA

Num bytes to read back: 11

Read back from FPGA

What was read back

Hello World

72 101 108 108 111 32 87 111 114 108 100

File to Send: "HelloWorld.txt": 11 bytes

Send the file to FPGA

Num bytes to read back: 11

Read back from FPGA

What was read back

Helmo World

72 101 108 109 111 32 87 111 114 108 100

Figure 6-27: Reading back the file Hello World, and what a possible error might look like

Once the file is loaded into the FPGA's RAM and the AddCWs are selected, the next step is to select the serial transmission speed. This is done by selecting the **Parallel 2 Serial Speed** as either 1kHz, 10kHz, 100kHz or 1MHz as shown in Figure 6-28. These various speeds were useful for testing and trouble shooting. For example, the logic analyzer could capture far more samples at 1kHz than it could at 1MHz. However, the spectrum analyzer could not measure frequencies below 9kHz, and thus having the ability to select different speeds proved quite useful.

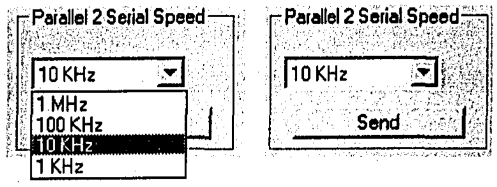


Figure 6-28: Choosing the serial transmission speed

Another debugging feature implemented was the ability to change the amount of RAM the encoder could use, specified as **RAM Size**. This is because when transmitting a file repeatedly, the end of the file is inevitable reached and the file data is deliberately wrapped around back to the start. In order to verify that the rollover occurred correctly, it was easier and faster to simply reduce the size of RAM (as far as the FPGA was concerned) to force a roll over sooner than it was to load in a smaller version of the file.

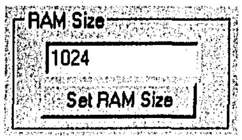


Figure 6-29: Setting the amount of RAM that the FPGA could use

The final step is to select which method to use when transmitting the file. If **Running Mode (Once)** is selected then the file is transmitted until the file in RAM is exhausted. That is, this mode is used to transmit a single file once from the TX to the RX. The other method of transmitting a file is to choose **Running Mode (Repeat).** In this mode the transmission runs continuously, sending the file over and over again until the transmitter is stopped by pressing either **Initialize Encoder** or **Reset FPGA Board.** This mode is used when measuring PSDs since the digital oscilloscope and spectrum analyzer are designed to monitor continuous signals.

6.7. Debug mode of the FPGA

Once the system was designed and implemented it had to be verified that the encoder was indeed selecting the best CW for transmission. When switch 0 (SW0) is set on the DIO2 board, the encoder is put into a debug mode where with alternate use of buttons 8 and 9, each phase of the CW selection process can be stepped through.

As discussed in Section 4-4, a simulation called **SimFPGA** was developed to emulate the way the FPGA hardware would perform the encoding. Table 6-5 shows a small sample of the output file created with this program. In this example the (31,26) code is expurgated to (31,23) using 8 AddCWs created from the set of four AddSWs which are { 0h, 0800000h, 1000000h, 1800000h}. Since this is a large code the FPGA needs to read from RAM three times in order to obtain enough data to form the 23-bit SW. In this case the three bytes read are 2Ah (00101010b), 22h (00100010b) and 56h

(01010110b). These three bytes (24-bits) form the 23-bit SW 015112Bh (00101010001000100101011b) with 1-bit left over. Table 6-5 shows the 31-bit CW added to all eight AddCWs along with the RDS, number of transitions, MSW and first transition location statistics.

Table 6-5: Output from SimFPGA

```
****** RamRead = 2A    FilePointer = 0 of 49 ******

****** RamRead = 22    FilePointer = 1 of 49 ******

****** RamRead = 56    FilePointer = 2 of 49 ******

SW = 0015112Bh  RDS    TRAN     MSW    1stTran
02A22562h      -11.0   18.0    1768.0  6.0
12A2256Ch      -7.0    20.0    880.0   3.0
22A2257Eh      -3.0    18.0    744.0   2.0
32A22570h      -7.0    18.0    448.0   2.0
4D5DDA8Fh      +7.0    19.0    448.0   2.0
5D5DDA81h      +3.0    19.0    744.0   2.0****
6D5DDA93h      +7.0    21.0    880.0   3.0
7D5DDA9Dh      +11.0   19.0    1768.0  6.0
```

Figure 6-30 shows the encoder in debug mode displaying this process on the LCD screen. Note that all values are given in hexadecimal including negative numbers. For example an RDS of -11 is really F5h (using two's complement representation) and 18 transitions would be displayed as 12h.

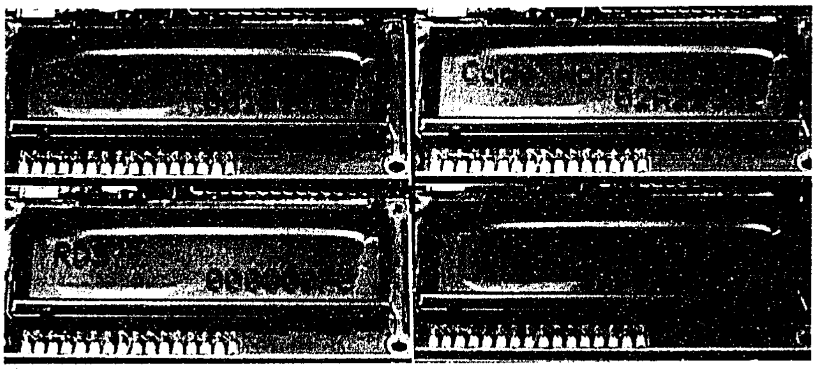


Figure 6-30: Encoder in debug mode showing the CW selection process

From Table 6-5 it can also be seen that the 6th CW or CW5 is the winner with an RDS of +3, 19 transitions and an MSW of 744. Figure 6-31 demonstrates that after stepping through all possible CWs and their statistics, the FPGA selects the Winning Location to be CW5.

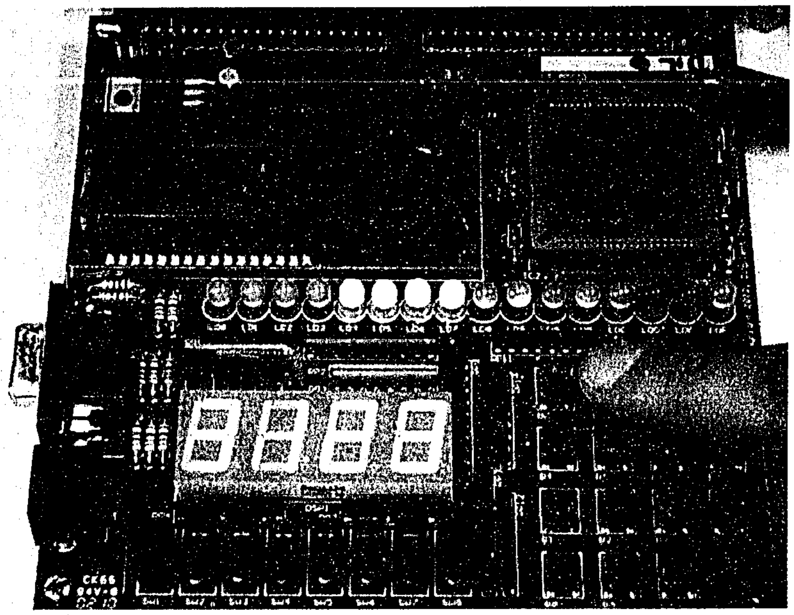


Figure 6-31: Winning location is CW5 or 5D5DDA81h

6.8. Receiver State Machine

The RX FPGA runs the same main state machine as the TX which consists of four states as shown in Figure 6-10 or Figure 6-32. On power up it is forced into state 1 which is the reset state that initializes the decoder. After this the FSM cycles through states 2, 3, and 4 indefinitely. Only by powering the FPGA off and on, or through user input in state 2 will the system enter the reset state again.

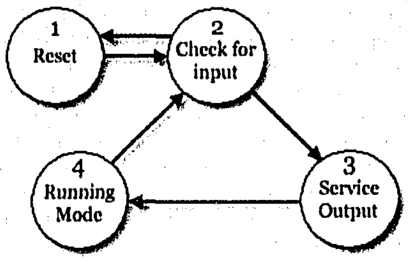


Figure 6-32: Main receiver FPGA FSM

1. Reset State – Identical to the TX reset state

2. Check for Input – Identical to the TX check for input state.

3. Service Output – Identical to the TX service output state.

4. Running Mode – This is the main mode of the communication system. It is similar to the equivalent TX state in that many of the same operations are present. The major difference however is that most functions are now in reverse. When the RX is in this state it is receiving data serially from the TX and processing it to recover the original data. This involves combining the serial data into CWs, performing error correction, extracting the MSbs to apply the CW addition again to remove the CS coding and finally storing these SWs into RAM.

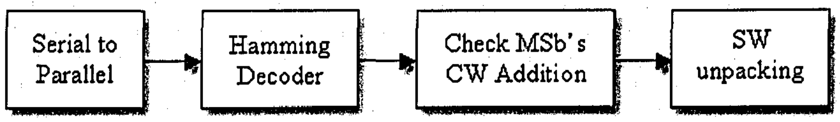


Figure 6-33: High-level block diagram of the running mode state FSM

Showing a complete block diagram of the Running mode FSM is not practical due to its size and complexity. Instead Figure 6-33 shows a high-level block diagram with a text description of the functionality of each state. Each block represents either a VHDL module and/or a sequential FSM.

Serial to Parallel Conversion: Once in running mode the RX continuously monitors for changes on the serial clock line. The serial data (**sdata**) is sampled when a rising edge (low to high transition) occurs on the serial clock (**sclk**). Whenever the RX has sampled *n* bits it packs them into an *n*-bit CW which is passed to the Hamming Decoder for error correction and conversion to a SW.

Figure 6-34 shows the testbench for the Serial2Parallel module. The CW 5D5DDA81h has been received serially and stored into **data.** At this point the Serial2Parallel module signals that a CW is **ready.**

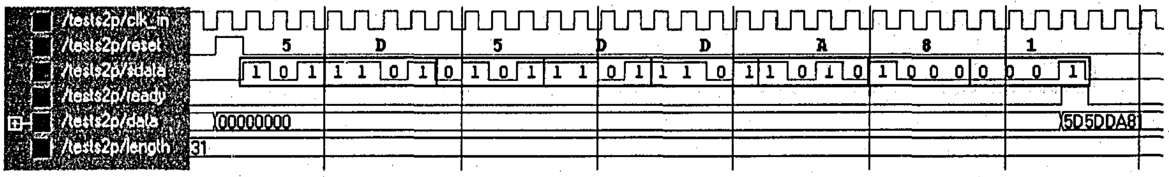


Figure 6-34: Testbench for the Serial2Parallel module

Hamming Decoder: Once a CW has been received it is sent to the Meggitt Decoder (see Figure 2-6) which attempts to correct any errors in the CW. Recall from Section 2-12 that Hamming codes can correct single bit errors per CW. Thus if the number of errors exceeds the power of the error correction code there will possibly be error extension. Therefore Figure 6-35 shows the CW 5D5DDA81h from above being decoded in three different scenarios. The first time the CW does not contain any errors and it is correctly decoded to SW 2EAEED4h. The second time however the CW is forced to contain two bit errors. This results in CW 5D5DDA87h being decoded to SW 2EAEFD4h. Note that this SW still contains one error. In the last case the CW only has a single bit error resulting in CW 5D5DDA80h being decoded to the correct SW 2EAEED4h.

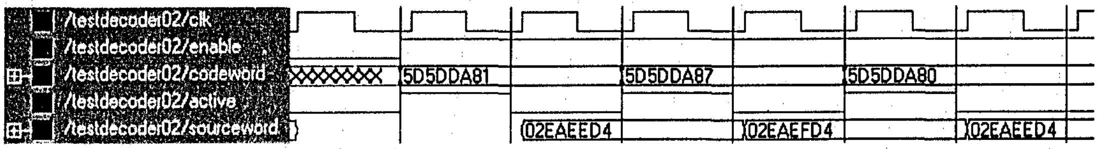


Figure 6-35: Testbench for the Meggitt Decoder module

CW Addition: As described in Section 3-2 when a CW has been received and corrected the original CW can be recovered by adding the same AddCW that was added at the TX. This is done by checking the MSbs of the CW as shown in Figure 3-20. However an alternative and equally valid approach takes advantage of the fact that CWs used in this system are systematic (see Section 2.4). Since the Meggitt Decoder module returns the error corrected SW (**ohamd_sw**), the CW addition module can recover the original SW by adding the correct AddSW. The testbench in Figure 6-36 demonstrates this.

Continuing with the example presented in Table 6-5 the original SW 015112Bh was transmitted as CW 5D5DDA81h. This was error corrected and decoded by the Meggitt Decoder module to the SW (**ohamd_sw**) 2EAEED4h. In order to recover the original SW the three MSbs of this 23-bit SW are inspected to determine that the winning location (**owloc**) must have been 5. Therefore adding AddSW5 (**oaddsw5**) which is 2FFFFFFh to this SW recovers the original SW (**ointsw**) 015112Bh.

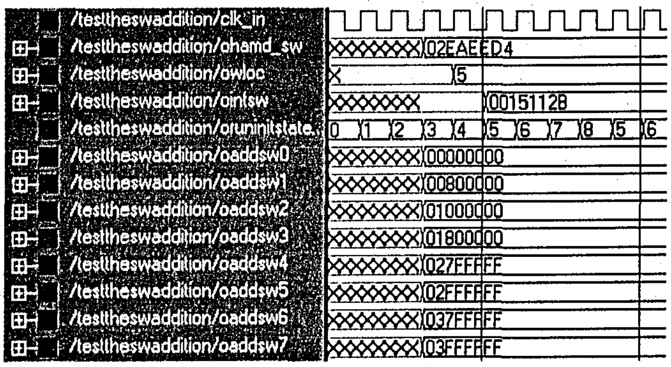

/testtheswaddition/clk_in
/testtheswaddition/ohamd_sw
/testtheswaddition/owloc
/testtheswaddition/ointsw
/testtheswaddition/orunninitstate
/testtheswaddition/oaddsw0
/testtheswaddition/oaddsw1
/testtheswaddition/oaddsw2
/testtheswaddition/oaddsw3
/testtheswaddition/oaddsw4
/testtheswaddition/oaddsw5
/testtheswaddition/oaddsw6
/testtheswaddition/oaddsw7
02EAEED4
5
0015112B
0 1 2 3 4 5 6 7 8 5 6
00000000
00800000
01000000
01800000
027FFFFF
02FFFFFF
037FFFFF
03FFFFFF

Figure 6-36: Testbench for the CW addition module

SW Unpacking: The final step in this process is to recover the original 8-bit data values (bytes) from the *k*-bit SWs and store them into RAM. However, similar to the SW Packing module, this is complicated by the variable *k*-bit SW size. For instance if *k* is a small number as in the case of the (7,1) code, it will take eight SWs to fill a single byte. However if *k* is a large number, as in the case of the (31,25) code, a single SW will fill three bytes in a row with one bit left over.

Figure 6-37 shows the testbench for the SW Unpacking module. Here SW 015112Bh from Figure 6-36 is unpacked and stored into RAM. The 23-bit SW is 8-bits + 8-bits + 7-bits. Thus 2Ah is stored at address 400h, and 22h is stored at address 401h. Referring back to Table 6-5 shows that these are the correct values. The remaining 7-bits will not be written to RAM until the next SW arrives.

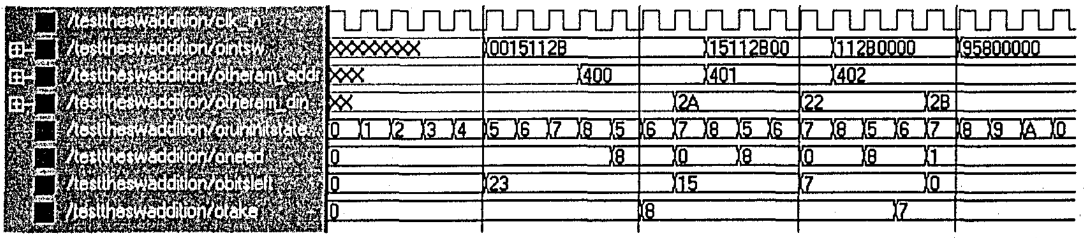
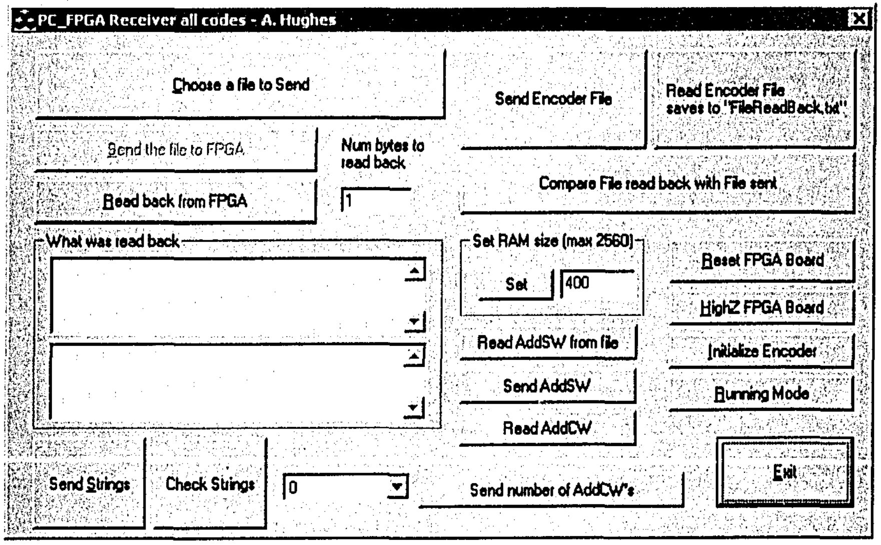

/testtheswaddition/clk_in
/testtheswaddition/ointsw
/testtheswaddition/otheram_addr
/testtheswaddition/otheram_din
/testtheswaddition/orunhitstate
/testtheswaddition/oneed
/testtheswaddition/obitsleft
/testtheswaddition/otake
XXXXXXXX
0015112B
15112B00
112B0000
95800000
XXX
400
401
402
XX
2A
22
2B
0 1 2 3 4 5 6 7 8 5 6 7 8 5 6 7 8 5 6 7 8 9 A 0
0 8 0 8 0 8 1
0 23 15 7 0
0 8 7

Figure 6-37: Testbench for the SW Unpacking module

6.9. PC Software: Controlling the RX FPGA

As shown in Figure 6-38, the RX software has many of the same functions as the TX software. This is because the RX FPGA needs to set up the same Hamming code, use the same AddCWs and so on. The difference however is that the RX can receive the bit sequence at any of the predetermined transmission rates without having to specifically select one. Furthermore the RX was constantly in a state of monitoring the clock and data lines for activity. In order to explain the operation and features of the software an example follows which outlines how to receive a file.



Num bytes to read back: 1

Set RAM size (max 2560): Set 400

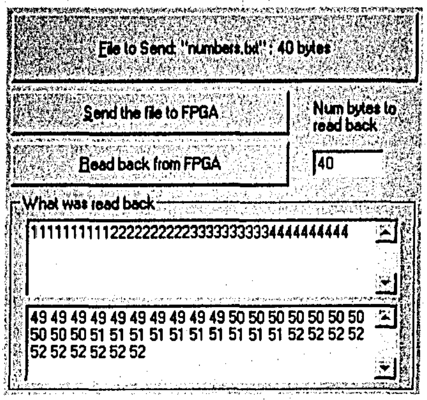
0

Figure 6-38: Software that controllers the RX FPGA

6.10. Receiving a File

This section briefly describes how to receive a standard ASCII text file such as the eleven byte "Hello World" file discussed in Section 6.6. Initializing the RX FPGA is almost identical to the TX FPGA. The first step after powering on the board is to ensure the FPGA is in a known state by pressing **Reset FPGA Board**. This forces the FPGA into the reset state shown in Figure 6-32, and therefore begins the main FSM. After this the string table from Table 6-2 is loaded, the number of AddCWs is set up and the AddSWs are loaded into the FPGA memory in the exact same manner as the TX FPGA. At this point however the setup changes.

To receive a file what is typically done is to first set up the RX RAM with a preloaded dummy file. Then any data received by from the TX will overwrite this data to verify that communication was successful. For example, Figure 6-39 shows the dummy file "**numbers.txt**" loaded into the RX FPGA's RAM. This is displayed when the **Read back from FPGA** button is pressed. The top window displays the ASCII representation and the bottom window displays the decimal representation, i.e. '1' = 49 decimal, '2' = 50 decimal and so on.



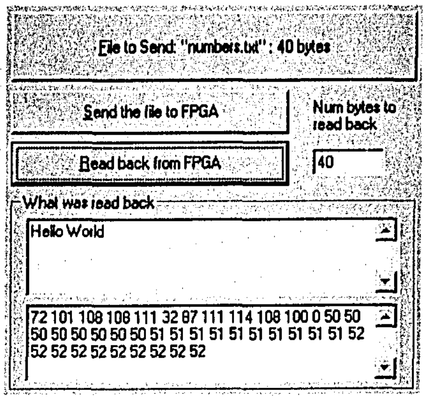
Num bytes to read back: 40

```
1111111111222222222233333333334444444444
```

```
49 49 49 49 49 49 49 49 49 49 50 50 50 50 50 50 50
50 50 50 51 51 51 51 51 51 51 51 51 51 52 52 52 52
52 52 52 52 52 52
```

Figure 6-39: Dummy file read back from the RX FPGA RAM

Pressing the **Running Mode** button places the RX into a mode where any data received on the **sdata** line is automatically stored into the RX RAM. Thus when the TX transmits the "Hello World" file in the **Run Once Mode**, the 11-byte file plus 00h byte terminator is sent to the RX. Pressing the **Read back from FPGA** button now shows the current RAM contents.



Num bytes to read back: 40

```
Hello World
```

```
72 101 108 108 111 32 87 111 114 108 100 0 50 50
50 50 50 50 50 50 51 51 51 51 51 51 51 51 51 51 52
52 52 52 52 52 52 52 52 52
```

Figure 6-40: RAM contents read back from the RX FPGA

Figure 6-40 shows how the received 11-byte file has overwritten the RAM contents. Notice that the top window only shows the string "Hello World", while the bottom window still displays the 40 byte RAM contents. This is because in the **Run Once Mode** the TX automatically appends a 00h byte to indicate the end of the transmission. In the lower window it can be seen that the 12th byte is a 0 terminator. In the top window the ASCII representation stops displaying data at this point since strings in C++ are 0 terminated. However, since it was requested that 40 bytes were to be read back from RAM, the lower window still shows all 40 bytes in decimal form.

6.11. Verify that the system could correct single bit errors

In this 3-wire system the channel length was less than 10 cm and the maximum transmission rate was 1MHz in base band. Hence the channel was practically ideal (noiseless). Therefore, in order to test the error correcting abilities of the combined code, errors were intentionally introduced into the transmitted bit stream. Using SW(6) or SW(7) a 1-bit or 2-bit error per CW could be introduced on every CW sent on the channel as shown in Figure 6-41.

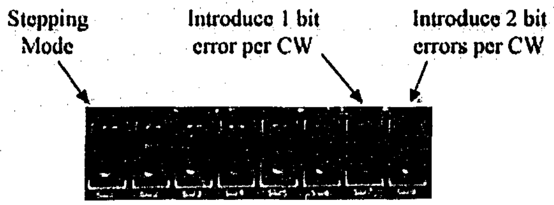


Figure 6-41: How the switches on the DIO2 board controlled the TX FPGA

To verify that bit errors were being introduced the RX was configured to store all received CWs into a file called **FileReadBack.dat**, to compare with the file transmitted. Figure 6-42 shows the same sixteen CWs transmitted using the (7,2) code with and without SW(6) on. The top line shows the CWs all have a single bit error in the LSb position while the bottom line shows the CWs without errors. Note that as a result of the error correcting ability, single bit errors have no effect on the accuracy of the SWs and they will still be decoded correctly.

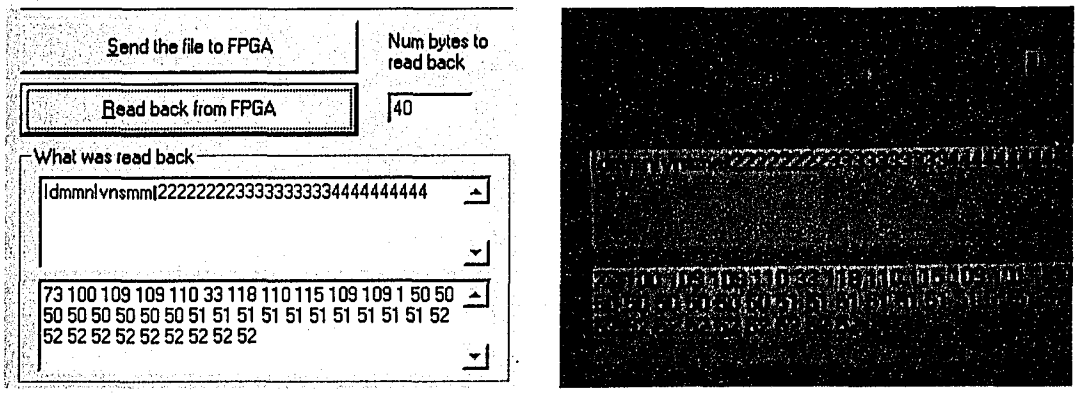
```
000000  45 3A 69 45 27 58 58 53  2C 27 3A 45 16 69 27 58  E:iE'XXS,':E.i'X
000000  44 3B 68 44 26 59 59 52  2D 26 3B 44 17 68 26 59  D;hD&YYR-&;D.h&Y
```

Figure 6-42: Top line is CWs with single bit errors, bottom line is error free CWs

6.12. The effect of more than single bit errors

When SW(7) is set the TX introduces 2-bit errors per CW into the transmitted bit stream. This number of errors exceeds the error correcting ability of the EC code, and thus the SWs will no longer be decoded correctly. This effect can be demonstrated best by observing the result of transmitting the "Hello World" file again with SW(7) on. The resulting decoded data is displayed in Figure 6-43.

It is obvious from Figure 6-43 that the text in the ASCII window is completely unrecognizable when compared to Figure 6-40. This is because the excess bit errors resulted in the wrong SWs being decoded and as a result the decoded ASCII text is garbled. Another interesting result is that the original dummy file "numbers.txt" appears in both the ASCII and decimal windows in its entirety. This is because the terminating 00h byte which followed the "Hello World" text was also corrupted (the 12th byte is decoded as a 1 as shown in the lower window in Figure 6-43). Thus without the 0 to terminate the string the entire requested 40 bytes is displayed.



40

```
Idmmnlvnsmml2222222233333333334444444444
```

```
73 100 109 109 110 33 118 110 115 109 109 1 50 50
50 50 50 50 50 50 51 51 51 51 51 51 51 51 51 51 52
52 52 52 52 52 52 52 52 52
```

Figure 6-43: "Hello World" file received with 2-bit errors per CW

6.13. Equipment used to measure the PSD

The two devices used to measure the spectral characteristics of the transmitted sequences were the **Agilent 54621A Digital Oscilloscope** (DO) and the **Agilent E4402B 9kHz – 3GHz Spectrum Analyzer** (SA) as shown in Figure 6-44.

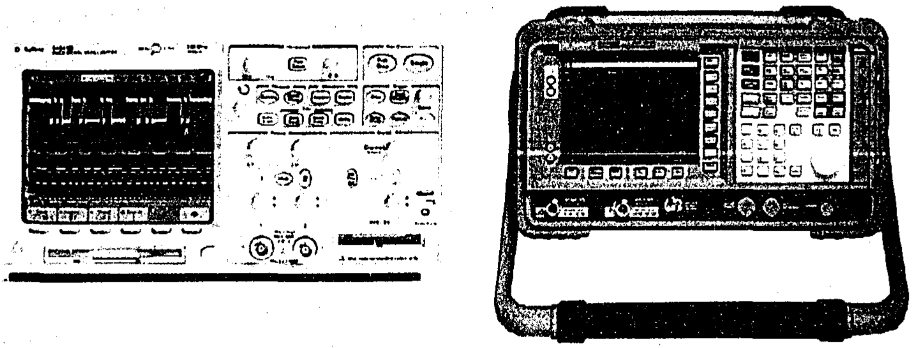
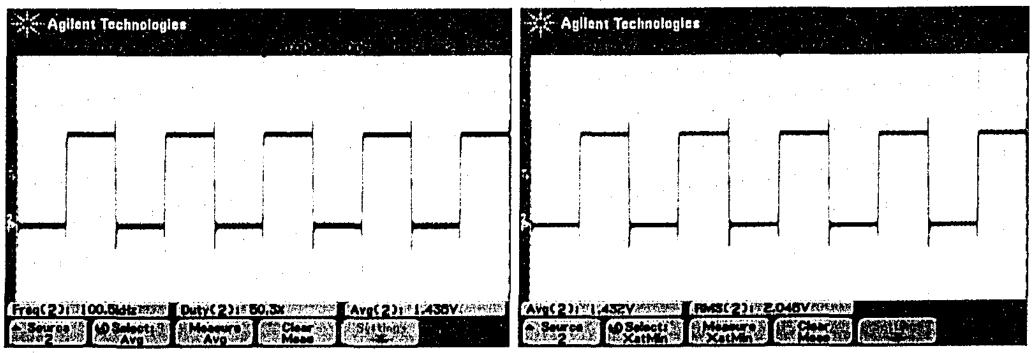


Figure 6-44: Digital oscilloscope and spectrum analyzer used

6.14. Testing discrete spectra – 50% duty cycle square wave

In order to have confidence in the measured spectra some initial tests were performed. The first step was to observe the spectrum of a unipolar 3Vpp, 100kHz, 50% duty cycle square wave on the SA and DO, then compare this spectrum to calculations. The actual lab signal measured was -31mV to 2.875V or approximately 2.9Vpp. The measured **average** voltage was 1.438V and the measured **RMS** voltage was 2.046V, as shown in the DO screenshots in Figure 6-45.



```
Freq(2): 100.5kHz   Duty(2): 50.3%   Avg(2): 1.438V
Avg(2): 1.432V   RMS(2): 2.046V
```

Figure 6-45: 50% duty cycle unipolar square wave

Calculating the ideal **average** and **RMS** value is shown in Equations 6.1 and 6.2.

$$Avg = \frac{1}{T_0}\int_{T_0} f(t)dt \qquad Avg = \frac{1}{T_0}\left[\int_0^{T_0/2} 2.875dt + \int_{T_0/2}^{T_0} -0.31dt\right] = 1.42V \tag{6.1}$$

$$RMS = \sqrt{\frac{1}{T_0}\int_{T_0}(f(t))^2\,dt} \qquad RMS = \sqrt{\frac{1}{T_0}\left[\int_0^{T_0/2}(2.875)^2\,dt + \int_{T_0/2}^{T_0}(-0.31)^2\,dt\right]} = 2.03V \tag{6.2}$$

Table 6-6: Calculated AVG and RMS versus measured

	Ideal	**Measured**
AVG	1.42V	1.43V
RMS	2.03V	2.04V

Table 6-6 shows how the actual square wave produced by the FPGA agrees with calculations in terms of average and RMS voltages.

6.15. Fourier Series Representation

Converting periodic signals from the time domain to the frequency domain can be done with a **Trigonometric Fourier Series.** The most general form is

$$f(t) = a_0 + \sum_{n=1}^{\infty}\left(a_n\cos(n\omega t) + b_n\sin(n\omega t)\right) \quad where \quad \omega = \frac{2\pi}{T_0} \tag{6.3}$$

$$a_0 = \frac{1}{T_0}\int_0^{T_0} f(t)dt$$
$$a_n = \frac{2}{T_0}\int_0^{T_0} f(t)\cos(n\omega t)dt \qquad n = 1,2,3.. \tag{6.4}$$
$$b_n = \frac{2}{T_0}\int_0^{T_0} f(t)\sin(n\omega t)dt \qquad n = 1,2,3..$$

Using the relation

$$a_n \cos(n\omega t) + b_n \sin(n\omega t) = C_n \cos(n\omega t + \theta n) \tag{6.5}$$

where

$$C_n = \sqrt{a_n^{\ 2} + b_n^{\ 2}}$$
$$\theta_n = \tan^{-1}\left(\frac{-b_n}{a_n}\right) \tag{6.6}$$

this can also be written as a **Compact Fourier Series** which has the form:

$$f(t) = C_0 + \sum_{n=1}^{\infty} C_n \cos(n\omega t + \theta n) \tag{6.7}$$

This form gives a one-sided spectrum identical to what is displayed on a DO and SA. In addition it is very easy to evaluate a magnitude and phase spectrum that together completely describe the waveform. Calculating the **Compact Fourier Series** of the 50% duty cycle square waveform yields:

$$a_0 = \int_0^{T_0/2} 2.875dt + \int_{T_0/2}^{T_0} -0.31dt = 1.42 \tag{6.8}$$

$$a_n = \frac{2}{T_0}\left(\int_0^{T_0/2} 2.875\cos(n\omega t)\,dt + \int_{T_0/2}^{T_0} -0.31\cos(n\omega t)\,dt \right) = 0 \quad \text{for } n = 1,2,\ldots \tag{6.9}$$

$$\begin{gathered} b_n = \frac{2}{T_0}\left(\int_0^{T_0/2} 2.875\sin(n\omega t)\,dt + \int_{T_0/2}^{T_0} -0.31\sin(n\omega t)\,dt \right) \\ b_n = \frac{.0098\left(\cos(2n\pi) - 93.741\left(\cos(n\pi) - 0.989\right)\right)}{n} \quad \text{for } n = 1,2,\ldots \end{gathered} \tag{6.10}$$

Using these values the C_n coefficients of the Compact Trigonometric Fourier series can be calculated. The result is shown in Table 6-7.

Table 6-7: First nine C_n terms from Compact Fourier Series (linear scale)

Harmonic	Frequency (kHz)	Calculated
1 - Fundamental	100	1.85
2	200	1.39E-15
3	300	0.616
4	400	45.26E-15
5	500	0.370
6	600	30.17E-15
7	700	0.264
8	800	46.49E-15
9	900	0.205

These C_n values represent peak values of sinusoids and can be used to recreate the waveform. Also since ideal 50% square waves contain no even harmonics, these values are extremely small (approximately zero) as shown in Table 6-7, and only exist as a result of calculator precision (i.e. machine precision).

In order to compare these values with the DO and SA they must be converted to the decibel (dB) scale. The DO displays its results in terms of dB per volt RMS (**dBV$_{RMS}$**) across a 1 Mohm load (ideally infinite), while the SA displays its results in dB per milliwatt across a 50 ohm load (**dBm**). Therefore in order to get the same units for comparison the following conversions are used.

$$dBVrms = 10\,\text{Log}\left(\left(\frac{C_n}{\sqrt{2}}\right)^2\right) \tag{6.11}$$

$$dBm = 10\,Log\left(\frac{\left(\frac{C_n}{\sqrt{2}}\right)^2}{(50/1mW)}\right) \tag{6.12}$$

Table 6-8 shows the first nine C_n values converted to dBV$_{rms}$ which are compared to measured values from the DO.

Table 6-8: First nine C_n terms from Compact Fourier Series in dBV$_{RMS}$

Harmonic	Frequency (kHz)	DBVrms (measured)	DBVrms (calculated)
Fundamental	100	2.5	2.33
2	200	<-60	<-60
3	300	-7.19	-7.21
4	400	<-60	<-60
5	500	-11.56	-11.64
6	600	<-60	<-60
7	700	-14.44	-14.57
8	800	<-60	<-60
9	900	-16.56	-16.77

Note that the Fourier Series indicates that there should only be odd harmonics. However the PSD measured on the DO clearly shows even harmonics less than -60 dBV$_{rms}$ as shown in Figure 6-46. This is due to the fact that the square waves emitted from the FPGA are not perfect, i.e. they do not have an infinitely fast rise time. Hence there will be some power in the even harmonics. For comparison the -30dBV_{RMS} value is the center line of Figure 6-46 allowing one to gauge that the first peak is near 2 dBV_{RMS}.

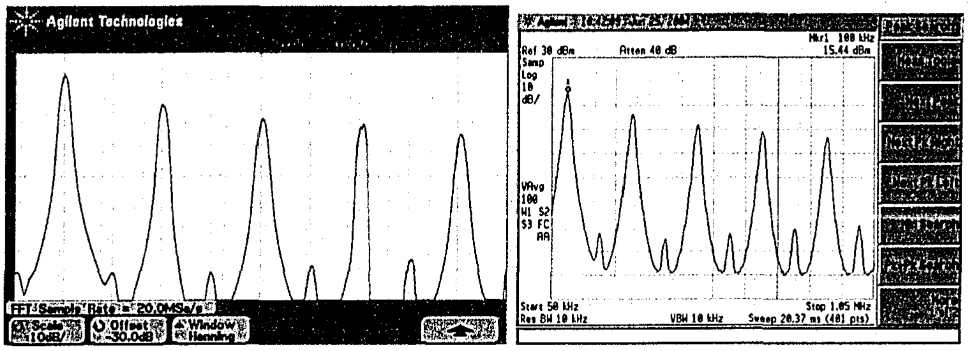


Figure 6-46: PSD measured on the DO and SA

Table 6-9: First nine C_n terms from Compact Fourier Series in dBm

Harmonic	Frequency (kHz)	DBm (measured)	DBm (calculated)
Fundamental	100	15.44	15.32
2	200	<-40	<-200
3	300	6.11	5.78
4	400	<-40	<-200
5	500	1.65	1.34
6	600	<-40	<-200
7	700	-1.37	-1.57
8	800	<-40	<-200
9	900	-3.45	-3.75

Figure 6-46 also shows the PSD as measured on the SA. In this case the first peak has a marker on it which indicates the peak is at 15.44 dBm. Table 6-9 shows how well the calculations match up with the SA measured values. The even harmonics are also noticeable on the SA which again are due to the fact that the square waves are not ideal. In addition, both PSDs should ideally be line spectra, however the resolution settings on the DO and SA and the imperfect nature of the square waves result in these broader spikes (concentrated frequency bands) in the frequency domain.

6.16. Accuracy of discrete PSD measurements

While the DO is useful for displaying time domain waveforms it is generally not considered to give the most accurate PSD. This is because the PSD is based on the FFT and using various digital signal processing (DSP) techniques such as windows. Therefore the majority of measured PSDs presented in this thesis will be from the SA since it is considered to give a more accurate representation of the PSD.

When measuring spectra with the SA one factor that has to be taken into consideration is its bandwidth (BW) of 9kHz to 3GHz. The manual stated that frequency measurements below 9kHz were not accurate. This is demonstrated in Figure 6-47 where a 6.25% duty cycle square wave is measured on the SA and compared with calculations. Clearly it can be seen that as the frequency falls below the low end of the bandwidth of the analyzer the measured values no longer agree with calculations. In addition, neither the DO nor SA is capable of giving accurate readings at DC (0 Hz). As a result the DC value shown in Figure 6-47 is the measured DC averages from the time domain.

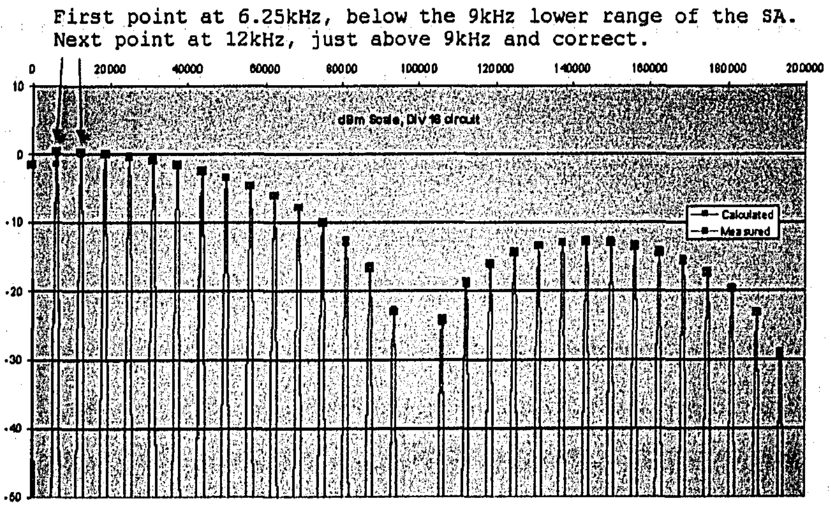


Figure 6-47: Shows how the SA is not accurate below 9kHz

6.17. Accuracy of continuous PSD measurements

The BW of the SA must also be taken into account when measuring continuous spectra. For example Figure 6-48 demonstrates the difference between the measured PSD of a random binary waveform and the calculated PSD. The 9kHz cutoff frequency of the SA has a pronounced effect on the measured PSD in that the power seems to drop dramatically, only to ramp up very high at DC. The power drop off results from the BW limitation of the SA and the ramp up at DC is due to the DC marker of the SA. This is a feature found on many SA's since PSD measurements are often double sided. That is, any spectrum appearing to left of the DC marker (negative frequency range) is essentially a mirror image of the spectrum to the right of the marker (positive frequency range). These two effects at low frequencies should not be the present as shown in the PSD plot generated by Matlab.

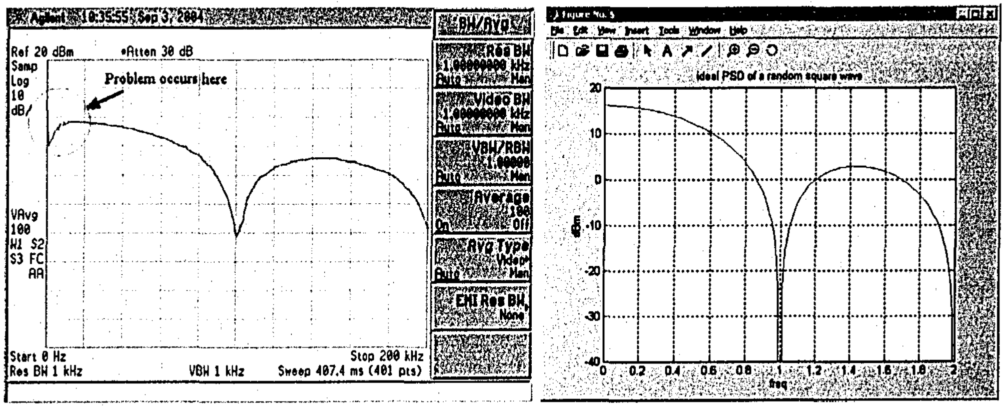
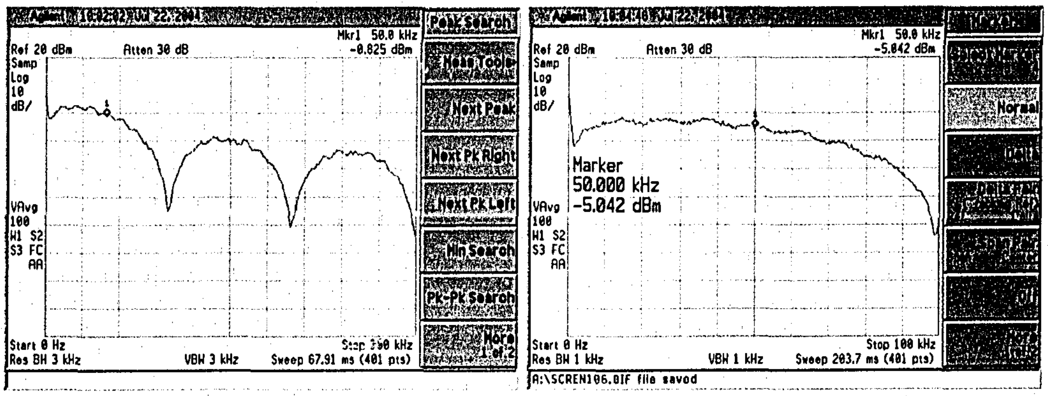

Agilent 10:35:55 Sep 3, 2004
Ref 20 dBm •Atten 30 dB
Samp Log 10 dB/
VAvg 100
W1 S2 S3 FC AA
Start 0 Hz Stop 200 kHz
Res BW 1 kHz VBW 1 kHz Sweep 407.4 ms (401 pts)
Res BW 1.00000000 kHz Auto Man
Video BW 1.00000000 kHz Auto Man
VBW/RBW 1.00000 Auto Man
Average 100 On Off
Avg Type Video Auto Man
EMI Res BW None

Figure 6-48: PSD of a random binary square wave, measured versus calculated

Another factor that needs to be taken into account is the resolution bandwidth (ResBW) of the SA. When the SA takes power measurements it does so using banks of analog filters. It dwells in certain frequency bandwidths set by ResBW, and measures the power in these bands. As a result, large ResBWs tend to give a more coarse picture of the shape of the PSD, while smaller ResBWs tends to give a more detailed picture of the shape of the PSD. Aside from the shape, the actual power measured will differ depending on the ResBW also. For example in Figure 6-49 the same PSD is measured using a

ResBW of 3kHz and 1kHz respectively over a bandwidth of 300kHz and 100KHz respectively. The power is measured at 50kHz is different in both plots at -0.825 dBm and -5.042dBm respectively.


Ref 20 dBm Atten 30 dB Mkr1 50.0 kHz -0.825 dBm
Start 0 Hz Stop 300 kHz
Res BW 3 kHz VBW 3 kHz Sweep 67.91 ms (401 pts)

Ref 20 dBm Atten 30 dB Mkr1 50.0 kHz -5.042 dBm
Marker 50.000 kHz -5.042 dBm
Start 0 Hz Stop 100 kHz
Res BW 1 kHz VBW 1 kHz Sweep 203.7 ms (401 pts)
A:\SCREN106.GIF file saved

Figure 6-49: Same PSD measured with two different ResBW's

The 3kHz ResBW plot on the left shows a coarser picture of the shape of the PSD and a high measured power of -0.825 dBm at 50kHz. However the 1kHz ResBW plot on the right shows a detailed picture of the shape of the PSD over one lobe, and a smaller measured power of -5.042dBm at 50kHz. This three times increase in ResBW directly corresponds to approximately a $10\log[3] = 4.77\text{dB}$ change in the measured power, i.e. $5.042 - .825 = 4.2\text{dB}$. As shown in the following section, this difference in measured power is to be expected with continuous spectra, and for every factor of 10 increase in ResBW, the measured power should drop by a factor of 10dB, and vice versa.

6.18. How the ResBW affects the PSD

Recall from Figure 5-12 the PSD of the (7,4) Hamming code expurgated to a (7,1) code from Section 5-5. Figure 6-50 shows the PSD measured on the SA with different ResBWs. The top line corresponds to a ResBW of 3kHz, the middle line to 1kHz and the bottom line to 300Hz. The first thing to notice is that the ResBW has a factor of 10 difference between the 3kHz plot and the 300Hz plot, and this directly relates to a 10dB difference in continuous power levels. What is happening is that when the ResBW is set to 3kHz, the SA measures power in 3kHz bands and plots the result as a single point in the spectrum. Hence it more crudely estimates where the power is located since it is unable to distinguish frequencies smaller than 3kHz. Contrast this with the 300Hz ResBW which has 10 times the frequency resolution. The SA now measures and plots power in the 300 Hz bands. While it takes longer for the SA to generate this plot, it can display more accurate information with these smaller bands. As a result the plot appears more spiky. Also the more the ResBW is lowered the more the continuous component of the PSD drops. This is because the SA is measuring and plotting the power in smaller frequency bands. However, when there are strong spectral components in narrow frequency ranges, their power will not change as the ResBW is decreased. This can be seen in Figure 6-50 where the dominant discrete like components maintain their relative power levels as the ResBW is decreased. Clearly, when comparing measured PSDs to calculated PSDs, it is only practical to compare their relative shapes and not their overall power levels.

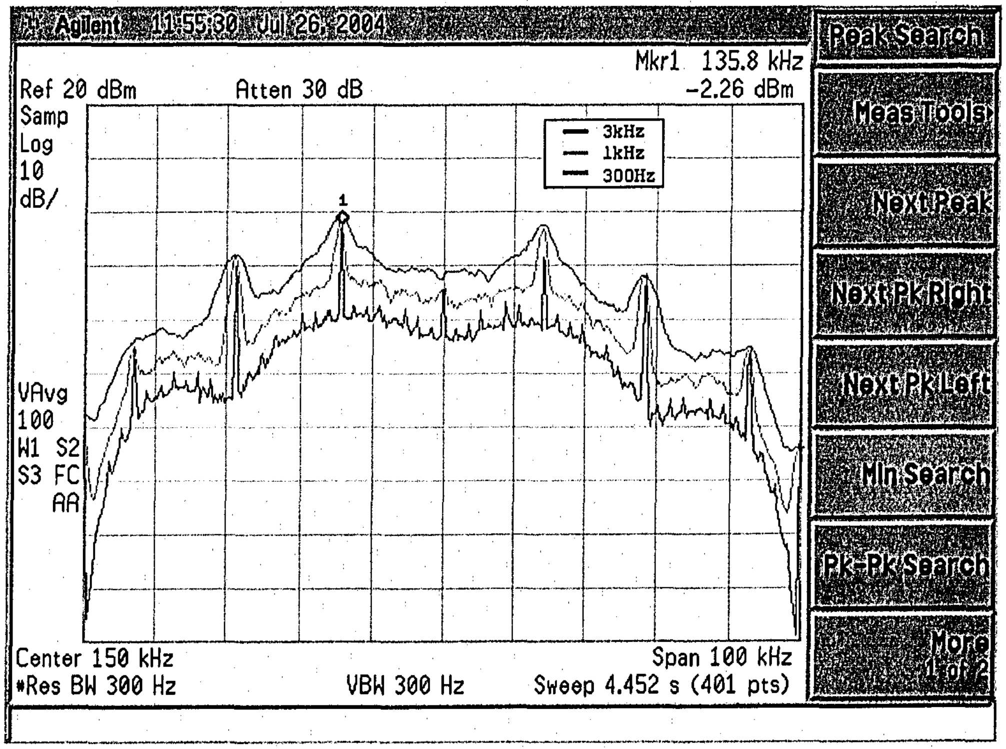
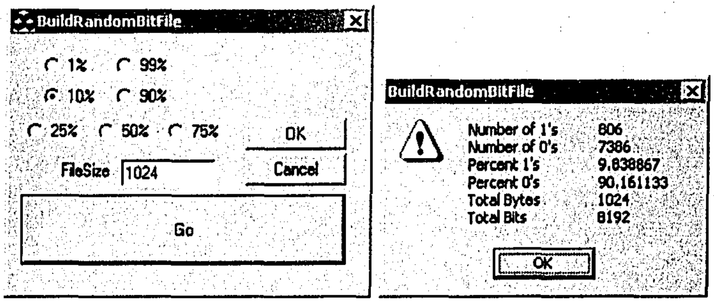


Figure 6-50: The effect of decreasing the ResBW

6.19. Comparing spectra of various codes

This section deals with measuring the PSDs of various codes with different AddCW sets and comparing the results to the theoretical PSDs presented in Section 5.6. As discussed in that section, two heavily unbalanced source statistics of 10% and 90% probability of a logic 1 are considered. The source statistics were generated using the computer program **BuildRandomBitFile** as shown in Figure 6-51.

Figure 6-51 demonstrates the use of this program to generate a file of size 1024 bytes that contains approximately 10% logic 1s. The file generated in this example actually had 9.83% probability of a logic 1. The output file consists of 1024 bytes or 8192 bits which are primarily logic 0s as shown in Figure 6-52. This file is then transmitted repeatedly on the channel using **Run Mode Continuously** by the TX.

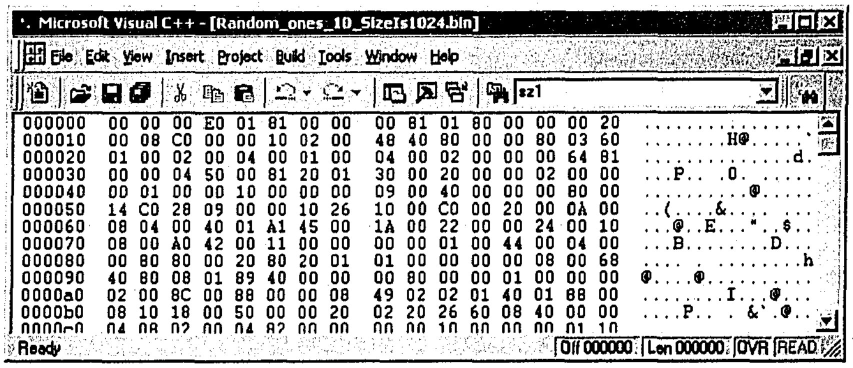


1% [✓ 10%] 25% 50% 75% 99% 90%

FileSize 1024

Number of 1's	806
Number of 0's	7386
Percent 1's	9.838867
Percent 0's	90.161133
Total Bytes	1024
Total Bits	8192

Figure 6-51: BuildRandomBitFile program for generating random data files

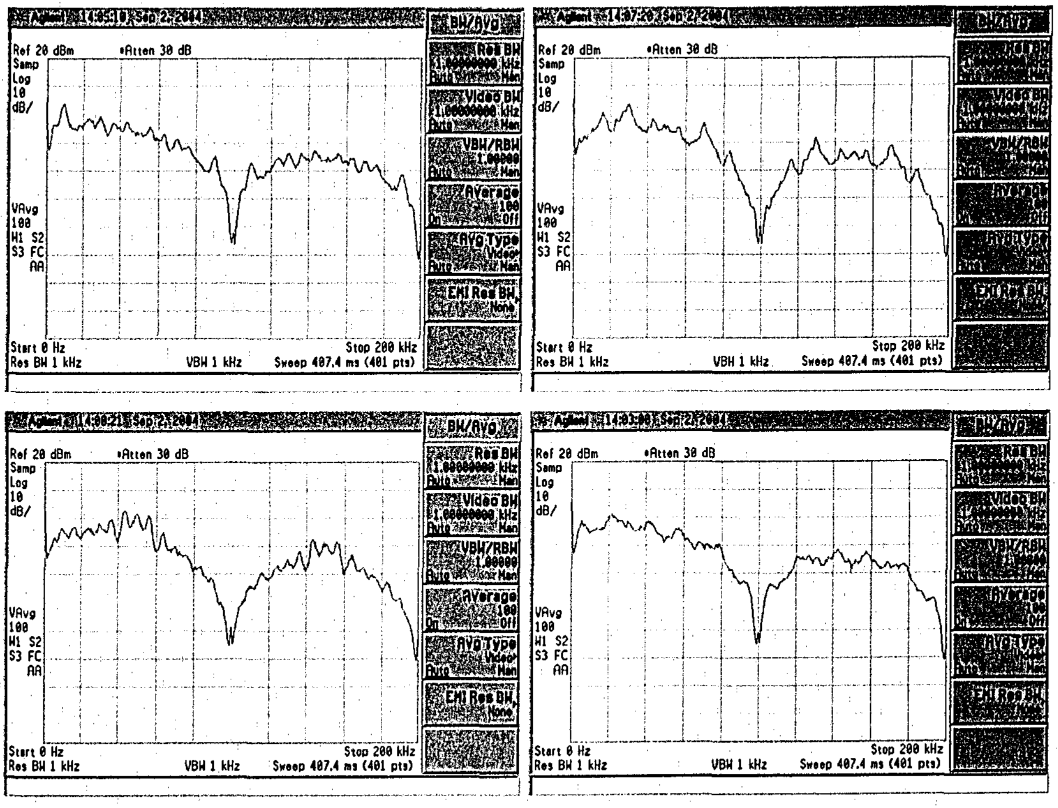

Microsoft Visual C++ - [Random_ones_10_SizeIs1024.bin]
File Edit View Insert Project Build Tools Window Help
sz1
Ready
Off 000000
Len 000000
OVR
READ

```
000000  00 00 00 E0 01 81 00 00  00 81 01 80 00 00 00 20   ................
000010  00 08 C0 00 00 10 02 00  48 40 80 00 00 80 03 60   ........H@.....`
000020  01 00 02 00 04 00 01 00  04 00 02 00 00 00 64 81   ..............d.
000030  00 00 04 50 00 81 20 01  30 00 20 00 00 02 00 00   ...P.. .0. .....
000040  00 01 00 00 10 00 00 00  09 00 40 00 00 00 80 00   ..........@.....
000050  14 C0 28 09 00 00 10 26  10 00 C0 00 20 00 0A 00   ..(....&.... ...
000060  08 04 00 40 01 A1 45 00  1A 00 22 00 00 24 00 10   ...@..E..."..$..
000070  08 00 A0 42 00 11 00 00  00 00 01 00 44 00 04 00   ...B........D...
000080  00 80 80 00 20 80 20 01  01 00 00 00 00 08 00 68   .... . ........h
000090  40 80 08 01 89 40 00 00  00 80 00 00 01 00 00 00   @....@..........
0000a0  02 00 8C 00 88 00 00 08  49 02 02 01 40 01 88 00   ........I...@...
0000b0  08 10 18 00 50 00 00 20  02 20 26 60 08 40 00 00   ....P.. . &`.@..
0000c0  04 08 02 00 04 82 00 00  00 00 10 00 00 00 01 10
```

Figure 6-52: 'Random_ones_10_SizeIs1024.bin' shows the file is mostly logic 0s

6.20. (15,8) Best AddCW sets

The following PSDs were measured with the SA using the randomly generated 8192-bit files. Analysis of these PSDs can be found in Section 5.4.



14:05:10 Sep 2, 2004
Ref 20 dBm •Atten 30 dB
Samp
Log
10
dB/
VAvg
100
W1 S2
S3 FC
AA
Start 0 Hz Stop 200 kHz
Res BW 1 kHz VBW 1 kHz Sweep 407.4 ms (401 pts)

14:07:20 Sep 2, 2004
Ref 20 dBm •Atten 30 dB
Samp
Log
10
dB/
VAvg
100
W1 S2
S3 FC
AA
Start 0 Hz Stop 200 kHz
Res BW 1 kHz VBW 1 kHz Sweep 407.4 ms (401 pts)

14:00:21 Sep 2, 2004
Ref 20 dBm •Atten 30 dB
Samp
Log
10
dB/
VAvg
100
W1 S2
S3 FC
AA
Start 0 Hz Stop 200 kHz
Res BW 1 kHz VBW 1 kHz Sweep 407.4 ms (401 pts)

14:03:00 Sep 2, 2004
Ref 20 dBm •Atten 30 dB
Samp
Log
10
dB/
VAvg
100
W1 S2
S3 FC
AA
Start 0 Hz Stop 200 kHz
Res BW 1 kHz VBW 1 kHz Sweep 407.4 ms (401 pts)

Figure 6-53: 90% ones – All four Best AddCW sets. Compare with Figure 5-16

Note that in all of the above PSD plots, the SA is set to 10dB per division, while the comparable plots of Section 5.6 are 5dB per division. The following Figure 6-54 shows how one of the spectra would compare on the same scale. The agreement of the curves is clear.

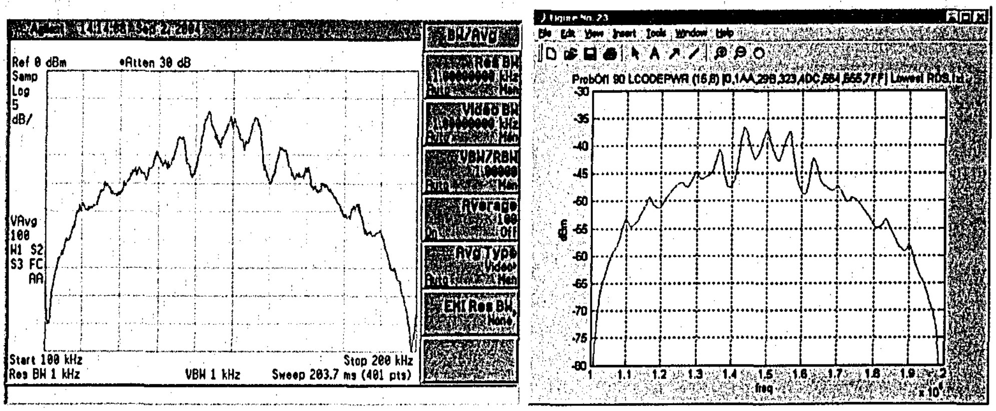


Figure 6-54: Comparing the 90% ones best AddCW set 1AA on the 5dB/div scale

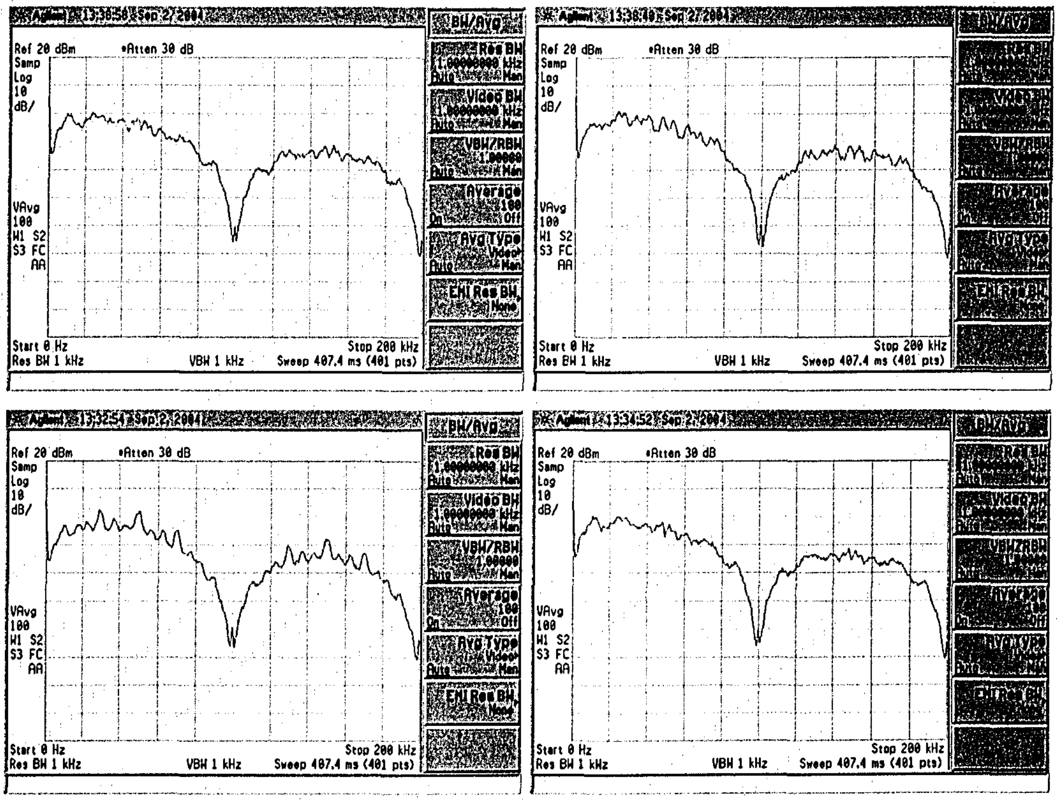


Figure 6-55: 10% ones – All four Best AddCW sets. Compare with Figure 5-17

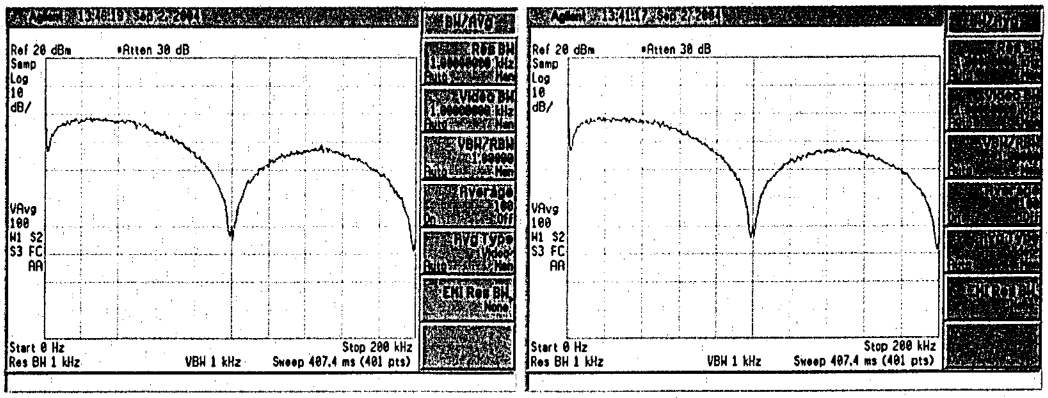


Figure 6-56: 50% ones shows little difference between codes – Compare with Figure 5-18

Figure 6-56 once again demonstrates that comparing these codes with balanced source statistics does not portray the effectiveness of this coding technique.

6.21. (15,8) Worst AddCW sets

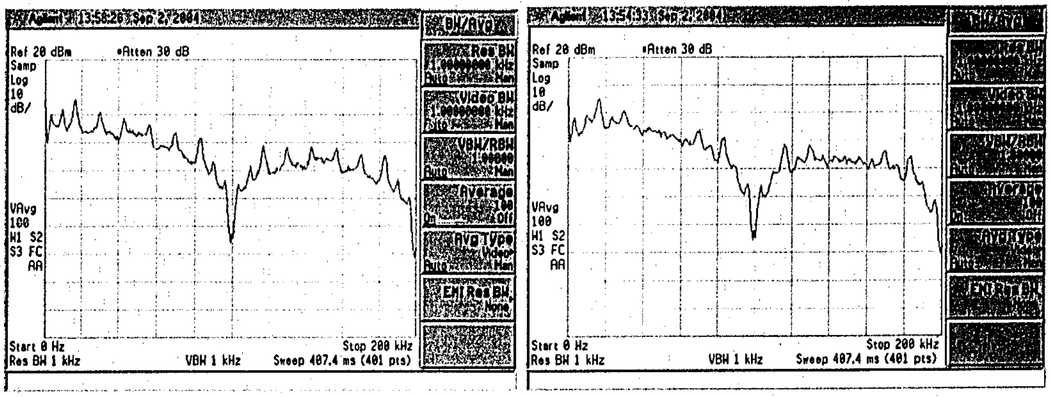


Figure 6-57: 90% ones – two of the Worst AddCW sets. Compare with Figure 5-19

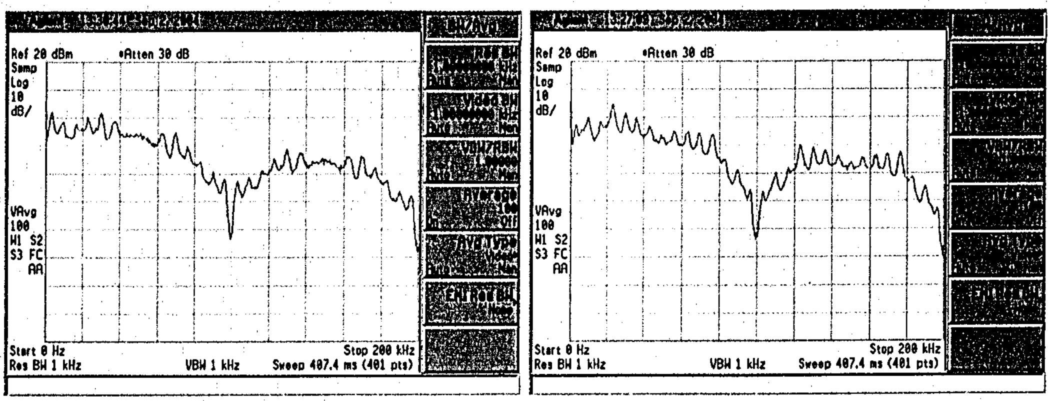


Figure 6-58: 10% ones – two of the Worst AddCW sets. Compare with Figure 5-20

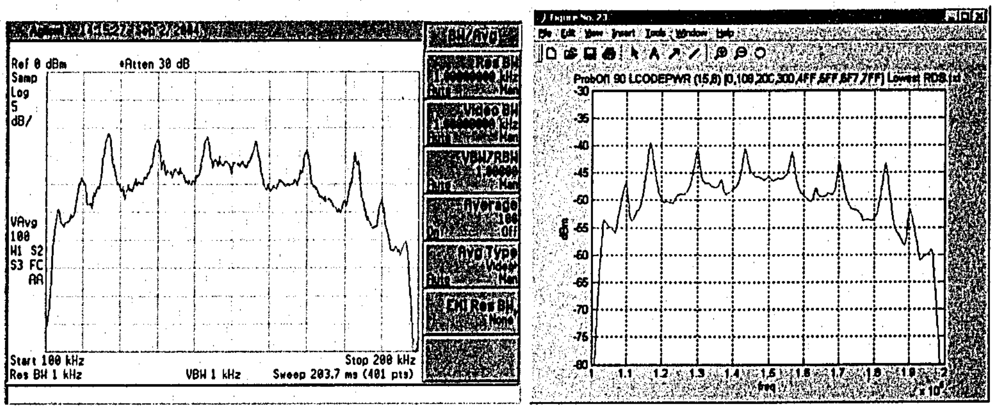


Figure 6-59: Comparing the 90% ones worst AddCW set 108 on the 5dB/div scale

Again the majority of the PSD plots were taken on the 10dB per division scale, while the comparable plots of Section 5.6 are 5dB per division. Figure 6-59 compares the PSDs on the same scale in the same manner as Figure 6-54.

As in Section 5.6 the PSDs can be contrasted side by side. For example the PSD of one of the best AddCW sets plotted beside one of the worst AddCW sets when the probability of the source emitting a logic 1 is 90% is shown in Figure 6-60. While it is obvious that the best AddCW set has more power at high frequency, it is most evident when they are both plotted on the same scale as shown in Figure 6-61.

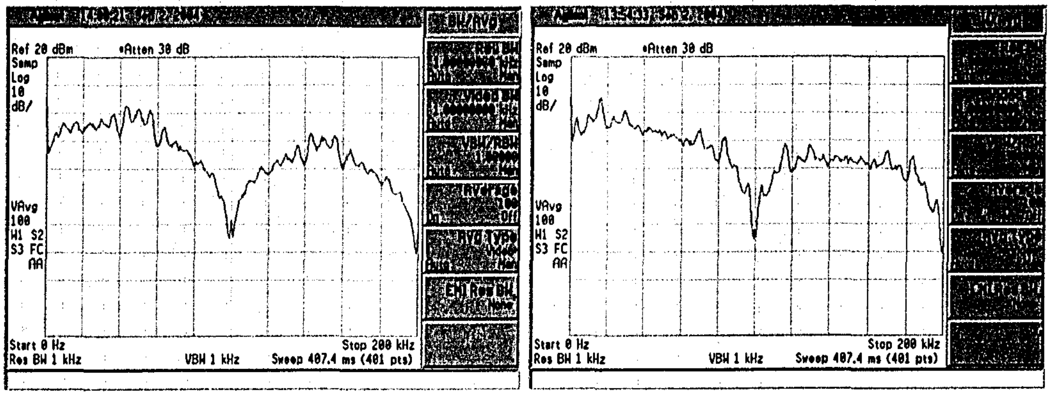


Figure 6-60: 90% ones – Best AddCW and Worst AddCW set – Compare with Figure 5-21

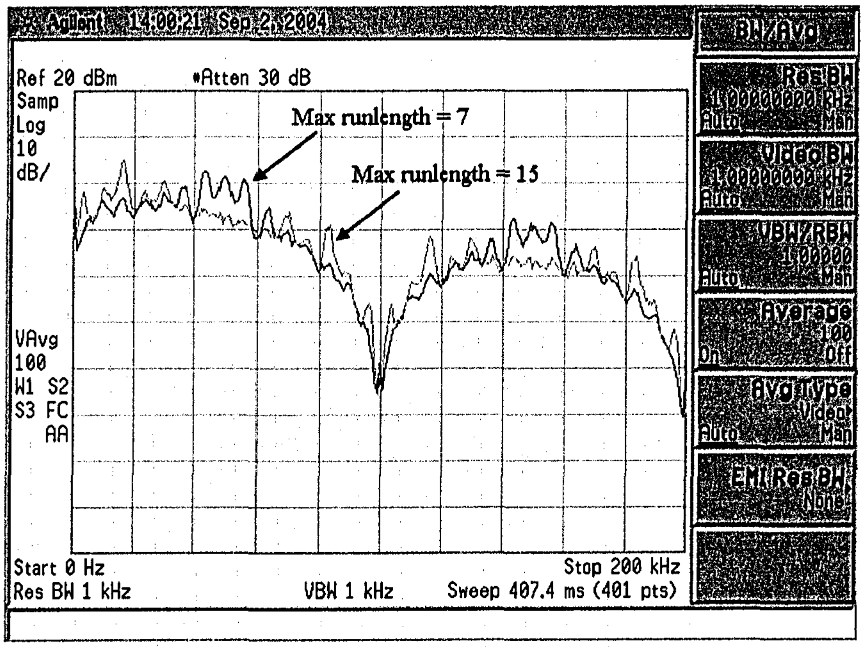


Figure 6-61: 90% ones – Best AddCW and Worst AddCW set – Compare with Figure 5-22

In Figure 6-61 it is clearly seen that not only does the best AddCW set have most of its power at higher frequencies, it also has a wider null indicating its ability to limit runlengths and generate a signal with less baseline wander.

6.22. Spectra of larger codes

Based on the AddCW recommendations in Section 4.15, the PSD of larger codes based on the (31,26) Hamming code are now compared to those of Section 5.7. Recall that these spectra could not be calculated exactly using the Cariolaro and Tronca method. Instead they were simulated using the SimFPGA program. Comparing the spectra in Figure 6-62 with Figure 5-25 show how closely they match.

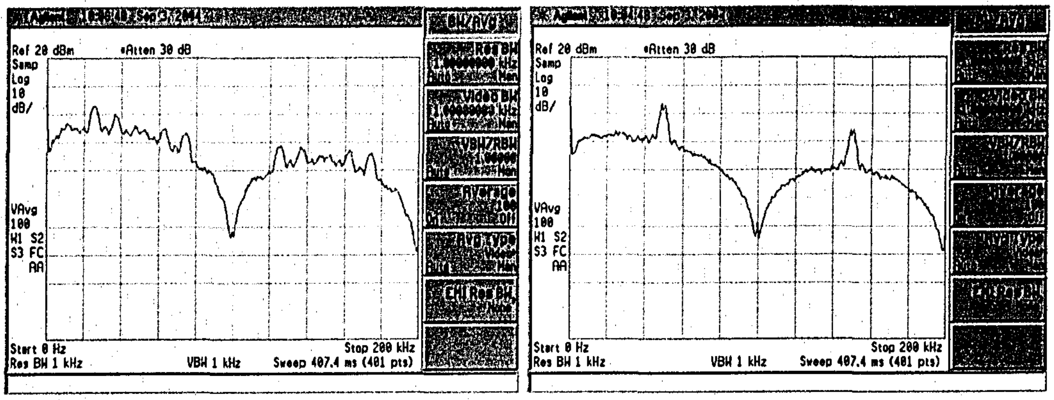


Figure 6-62: 10% ones – Two (31,23) codes. Compare with Figure 5-25

6.23. Summary

This chapter outlined and explained the methods used to create a hardware proof of concept system for the multimode coding technique introduced in Chapter 3. The communication system was implemented on two FPGA boards, one as a transmitter and the other as the receiver. There were two modes of operation. The first was used to transmit a file once on the channel to be decoded by the receiver. The second mode was used to transmit the file repeatedly on the channel in order to allow for measurement of the PSD. The PSD was measured in order to verify that the calculations presented in Chapter 5 were correct and that the coding technique operates as designed.

7. Bit Error Rate Performance

In this chapter the bit error rate (BER) performance of the multimode coding technique is investigated. This is achieved through simulations on a high-pass additive white Gaussian noise (AWGN) channel that models ac-coupled receivers.

7.1. Channel model

Recall from Section 2.19 that the serial bit stream requires periodic maintenance as it travels on the channel in order to combat the accumulation of noise and signal distortion. This requires that the signal be received and regenerated for further transmission. Receivers that derive their clock from the incoming bit stream require numerous transitions for adequate clock recovery. If there is a long series of 1s or 0s in a row the receiver's oscillator frequency may drift and lose synchronization, consequently losing track of where it is supposed to sample the transmitted data.

Recall that in the frequency domain these strings of like valued bits can be considered low frequency content as shown in Figure 5-2. Furthermore a bit stream with many transitions is considered to contain high frequency content as shown in Figure 5-3. Therefore receivers that derive their clock from the incoming bit stream can be summarized in the frequency domain as requiring a signal to have a majority of high frequency content and little low frequency content.

It is also very common for receivers to be AC coupled since they are easier to design and are capable of better performance [15,16]. However this results in the frequency response of the receiver being limited by the *RC* time constant of the dc blocking capacitor and input impedance of the receiver [15]. These types of receivers not only block the average DC value, but they also tend to integrate the detected signal giving rise to baseline wander [16].

Therefore the effects and demands of this type of receiver are simplified by considering it as a distinct type of channel referred to as a dc-constrained channel, i.e. a channel that heavily attenuates the low frequency and dc power of a signal [17]. As a result one simple model of a dc-constrained channel is that of a first-order high pass filter (HPF) concatenated with an AWGN source [17] as shown in Figure 7-1. This model will be referred to as the HPF channel model.

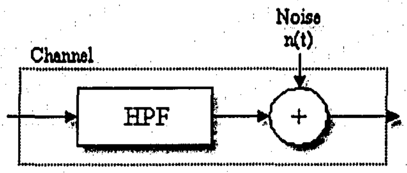


Figure 7-1: dc-constrained channel – HPF concatenated with AWGN

Using this channel model it will be illustrated that the time domain waveform of the transmitted bit stream will decay exponentially towards zero during each symbol interval, where the rate of decay will be dependent on the value of the *RC* time constant τ. For example, letting $V_{s,k}$ and $V_{e,k}$ be the voltages at the start and the end of the k^{th} bit interval $(k \geq 0)$, the voltage during the k^{th} interval is represented by Equation 7.1 [17]. The instantaneous voltage at the end of the interval is represented by Equation 7.2 [17], where *T* denotes the duration of each bit interval.

$$V(t) = V_{s,k} e^{-(t-kT)/\tau} \tag{7.1}$$

where $kT \leq t \leq (k+1)T$ and $\tau = RC/T$, and

$$V_{e,k} = V_{s,k} e^{-1/\tau} \tag{7.2}$$

At the start of the next interval the starting voltage $V_{s,k+1}$ takes on one of the values:

$$V_{s,k+1} = \begin{cases} V_{e,k} & \text{, if logic values are the same} \\ V_{e,k} + 2 & \text{, if logic values change from -1 to 1} \\ V_{e,k} - 2 & \text{, if logic values change from 1 to -1} \end{cases} \qquad (7.3)$$

Using Equations 7.1 to 7.3 the effect of this channel model can be demonstrated using Matlab. For example, consider transmitting the bit sequence 1,0,0,1,1,1,0 using bipolar coding with a square pulse shape. Figure 7-2 (a) shows the time domain waveform on a noiseless channel, while for reference Figure 7-2 (b) shows the midpoints of the symbols in order to see the individual bit locations.

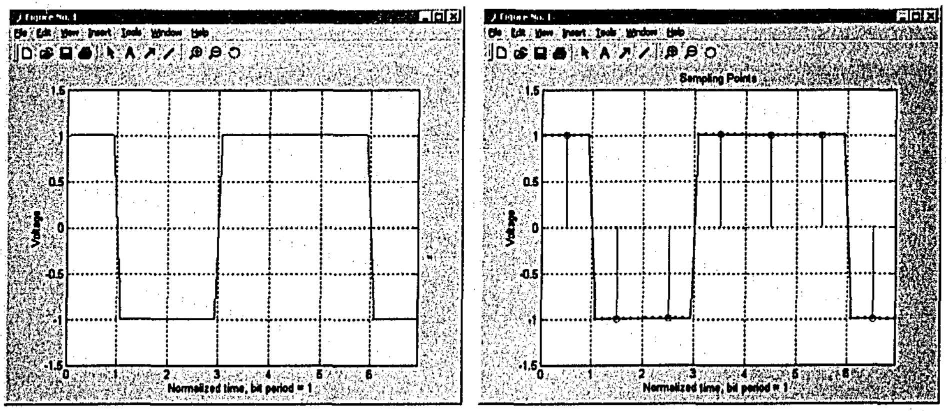


Figure 7-2: (a) Binary sequence [1, 0, 0, 1, 1, 1, 0] transmitted using bipolar coding. (b) Midpoint locations

Figure 7-3 on the other hand shows the binary sequence on the HPF channel with RC time constants of $\tau = 10$ and $\tau = 2$ respectively. In both cases it is clear that the time domain waveform decays exponentially towards zero due to the frequency response of the channel. That is, the more like valued bits in a row transmitted on this channel (i.e. the greater the low frequency content), the more severe the power loss is to the intelligible signal. Conversely the AWGN is practically unaffected on this channel since the distribution of white noise has zero mean. As a result it is clear that the output SNR of the receiver will not be constant over a symbol period. The loss of the low frequency portion of the signal power causes a fluctuating SNR that degrades the BER.

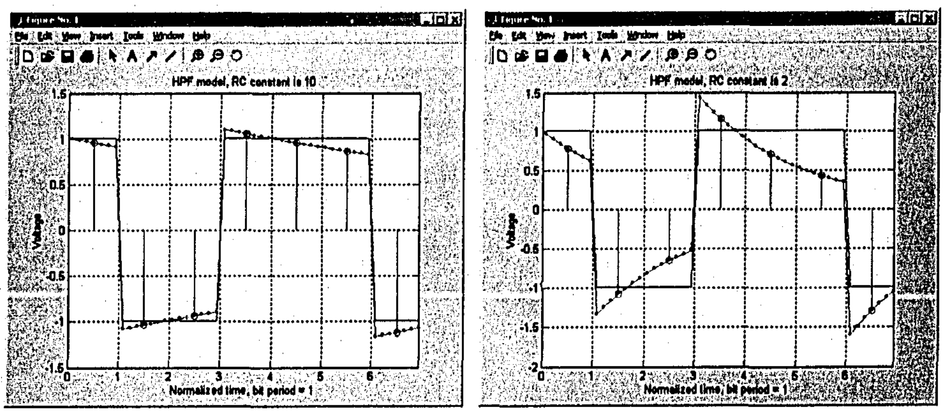


Figure 7-3: Showing the effect of the HPF channel with *RC* time constants 10 and 2, and how the SNR is not constant over a bit period

While Figure 7-3 demonstrates how the output SNR is not constant per symbol interval, a greater problem occurs if there are not adequate transitions in the output sequence. If for example there was a long run of logic 0s, the signal would decay to practically 0V as shown in Figure 7-4. In this case no amount of equalization can recover the time domain waveform since in this extreme situation the receiver is working with practically zero signal power.

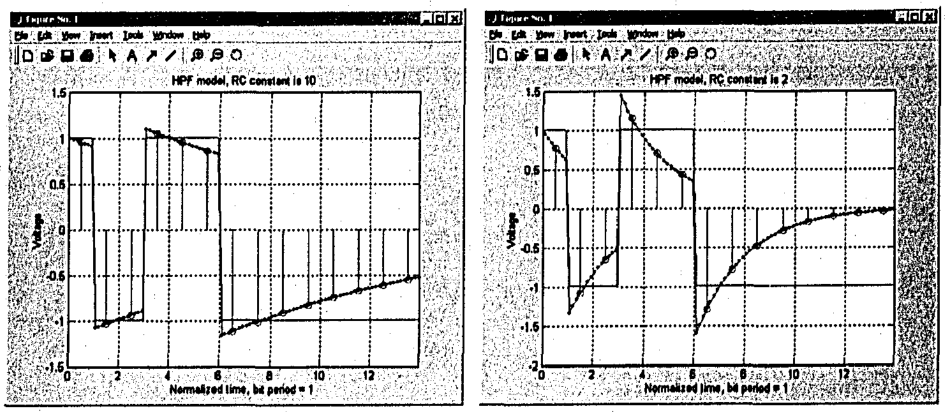
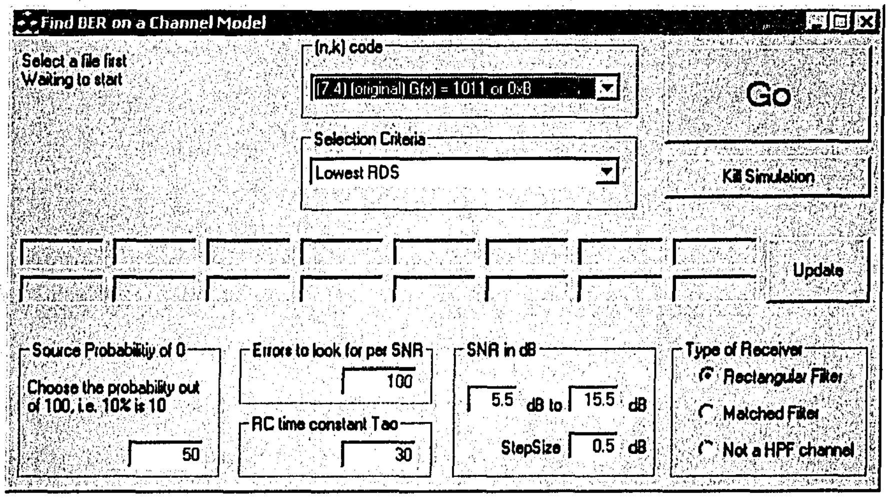


Figure 7-4: HPF channels effect of a long run of logic 0s with an *RC* constant of 10 and 2. It can be seen that the voltage is decaying to 0 Volts

Thus the motivations to create a coding technique that can combat the effects of this type of channel are clear. For improved BER performance on this channel the transmitted sequence not only needs to be balanced, but it also must contain numerous transitions and have limits on the length of consecutive like valued bits.

7.2. Simulation

In order to test the multimode coding technique on this HPF channel model a simulation called **HPF_BER_Sim.exe** was written; the user interface for this program is shown in Figure 7-5. This program allows the user to select the linear block code, selection criteria and AddCWs to use, similar to the SimFPGA program introduced in Section 4.4. The source statistics can be controlled using the **Source Probability of a 0** and the *RC* time constant of the channel can be controlled using the **RC time constant Tau.** The simulation runs at each SNR until the set number of decoding errors has been encountered. Finally, the type of filtering in the receiver can be chosen as either a **Rectangular Filter** or **Matched Filter**. Note that the channel can also be specified as **Not a HPF channel,** at which time the receiver uses a **Matched Filter.**


Find BER on a Channel Model

Select a file first
Waiting to start

(n,k) code: (7,4) (original) G(x) = 1011 or 0xB

Selection Criteria: Lowest RDS

Source Probability of 0
Choose the probability out of 100, i.e. 10% is 10
50

Errors to look for per SNR: 100

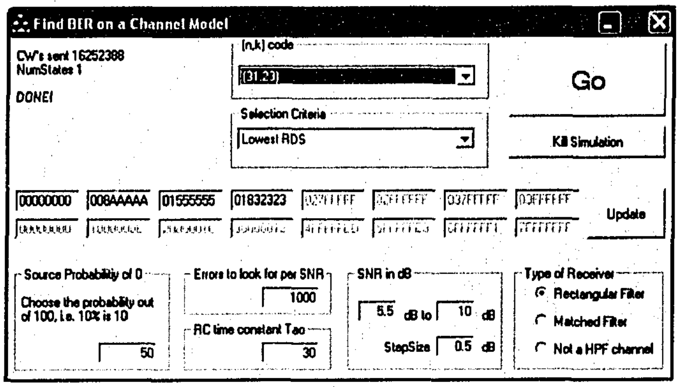
RC time constant Tao: 30

SNR in dB: 5.5 dB to 15.5 dB
StepSize 0.5 dB

Type of Receiver: [✓ Rectangular Filter] Matched Filter Not a HPF channel

Figure 7-5: HPF channel model simulation

An example is shown in Figure 7-6. Here the simulation was configured to run the (31,23) code using eight AddCWs. The source statistics were balanced and the *RC* time constant τ was set to 30.


Find BER on a Channel Model

CW's sent 16252388
NumStates 1

DONE!

(n,k) code: (31,23)
Selection Criteria: Lowest RDS

00000000	008AAAAA	01555555	01832323	027FFFFF	[illegible]	037FFFFF	03FFFFFF
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	7FFFFFFF

Source Probability of 0: Choose the probability out of 100, i.e. 10% is 10: 50
Errors to look for per SNR: 1000
RC time constant Tao: 30
SNR in dB: 5.5 dB to 10 dB, StepSize 0.5 dB
Type of Receiver: [✓ Rectangular Filter] Matched Filter Not a HPF channel

Figure 7-6: Looking for 1000 errors at each SNR from 5.5dB to 10dB with τ = 30

7.3. BPSK Simulation results

The channel was first tested without coding using bipolar signaling (BPSK) with matched filtering as a benchmark. The *RC* time constant chosen for all channel simulations was $\tau = 30$. This value was selected in order to test the performance of the multimode coding technique on a channel that was highly sensitive to baseline wander.

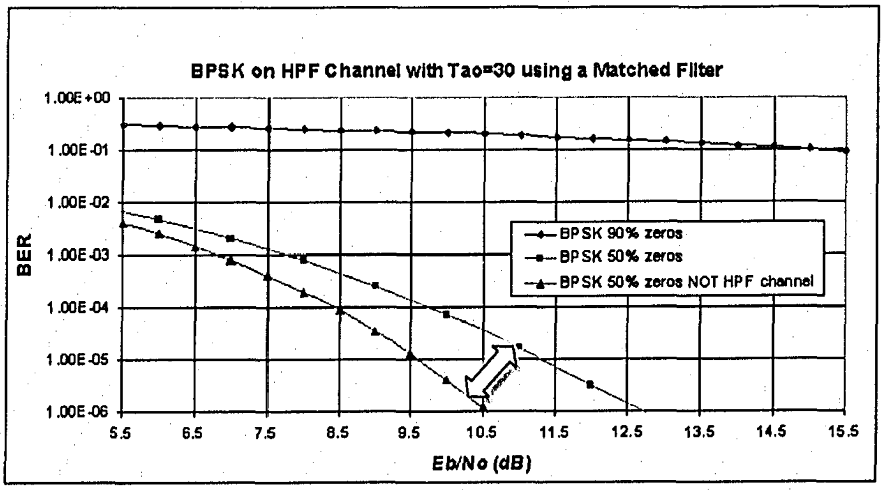


Figure 7-7: BPSK on an AWGN channel versus the HPF channel with AWGN

Figure 7-7 demonstrates the effect of the channel on a BPSK signal that does not use any form of EC or CS coding. For comparison a regular BER curve is also plotted for a BPSK signal on a typical AWGN channel (i.e. not the HPF channel).

The arrow in Figure 7-7 clearly shows how the BER of the BPSK signal on the HPF channel is worse than the BER on a standard AWGN channel. However the lower two curves only show the BER when the source emits logic 0s and 1s with equal probability. If the instantaneous source statistics become unbalanced to the point where logic 0s are transmitted 90% of the time, the BER curve almost flattens out. Thus the BER performance is highly dependent on the source statistics. Figure 7-8 shows the approximate range of BER values that a BPSK signal would experience on this channel with different statistics of the source.

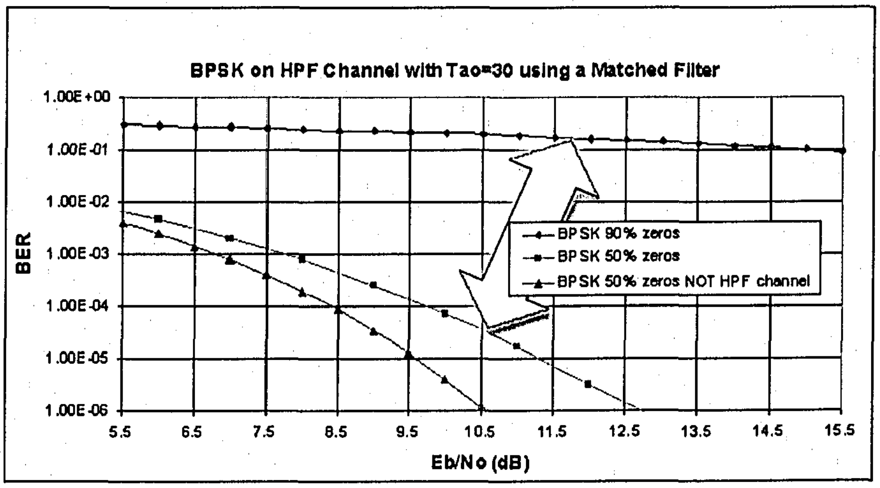


Figure 7-8: Range of BER values that occurs with changing source statistics

7.4. Effect of error control codes

It is of interest to determine if error control coding can alleviate the BER degradation on this channel. Figure 7-9 demonstrates the limited effectiveness of the simple Hamming codes used so far in this thesis. In the case of an unbalanced source only marginal improvement is observed with diminishing returns as larger error control codes are used.

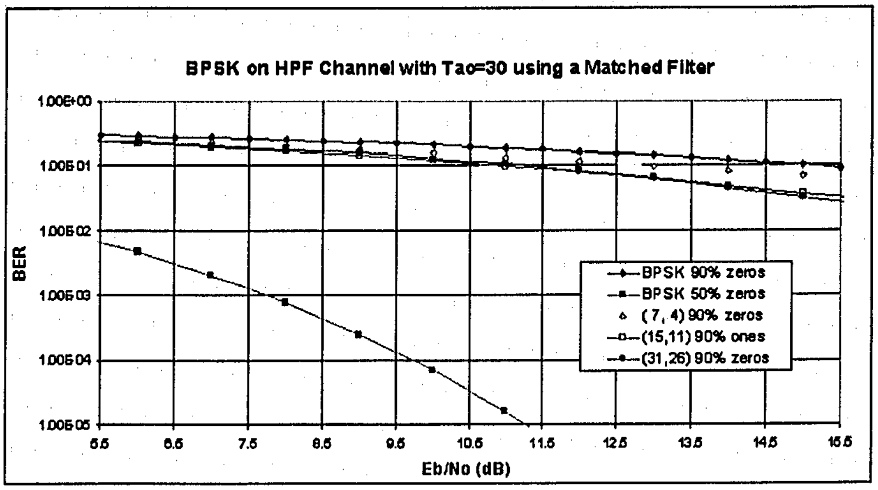


Figure 7-9: Effect of EC codes on this channel is minimal

7.5. Simple multimode code

As discussed in Section 3.1 and 3.2, a primary goal of this multimode coding technique is to guarantee balanced transmission regardless of the source statistics. It was shown that the use of a single extra bit of redundancy for CS coding is sufficient for this to be achieved. As confirmation Figure 7-10 shows the BER improvement using this technique with the (15,11) code expurgated to (15,10).

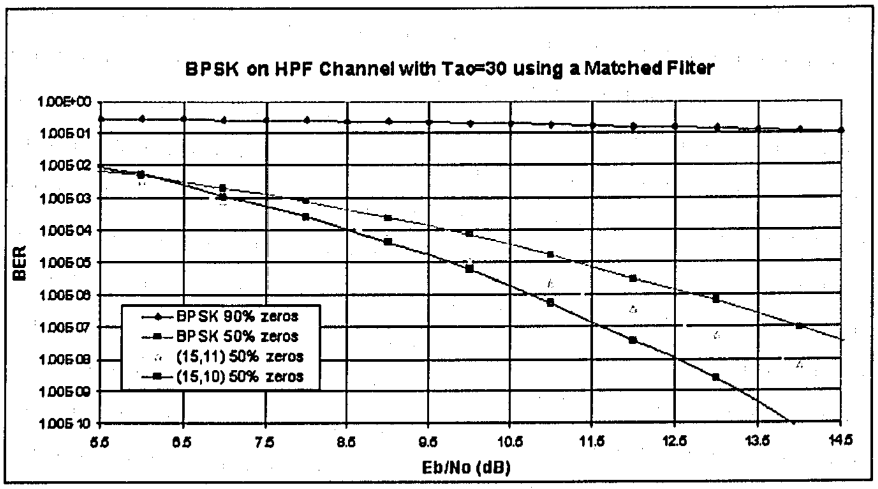
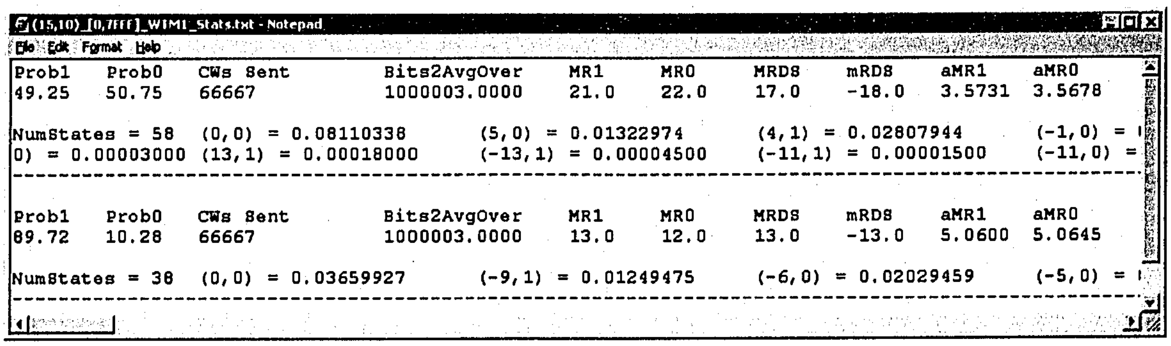


Figure 7-10: Comparison of performance of (15,11) and (15,10) codes with balanced source

This figure demonstrates that both codes have the ability to correct single bit errors per CW, and at low SNRs the (15,11) code actually performs slightly better than the (15,10) code. This is because with the extra bit of redundancy the overall per bit energy versus noise spectral density is slightly lower for the (15,10) code than the (15,11) code. At low SNRs the coding gain of the (15,10) code is not realized as the signal energy is simply too low. However around an SNR of 8.5dB a crossover occurs. It is at this point that the ability of the (15,10) code to guarantee balanced transmission results in improvement of the BER over that of the (15,11) code, and it can be seen that the BER curves are actually diverging. This is a significant improvement considering only a single extra bit of redundancy was used. It is not dramatic however since analysis in Chapter 4 showed that the (15,10) code has minimal ability to limit runlengths. As shown in Figure 7-11 maximum runlengths (MR1 and MR0) of 22 like valued bits in a row are still possible. Furthermore, the average runlength (aMR1 and aMR0) with balanced source statistics is approximately 3.6 like valued bits in a row. With unbalanced source statistics however the maximum runlengths (MR1 and MR0) are actually less at 13, but the average run (MR1 and MR0) has increased to approximately 5 like valued bits in a row. These results are still better than the (15,11) code which can have runlengths that are completely unbounded.



```
Prob1    Prob0    CWs Sent          Bits2AvgOver    MR1     MR0     MRDS    mRDS    aMR1    aMR0
49.25    50.75    66667             1000003.0000    21.0    22.0    17.0    -18.0   3.5731  3.5678

NumStates = 58   (0,0) = 0.08110338       (5,0) = 0.01322974       (4,1) = 0.02807944       (-1,0) = 
0) = 0.00003000 (13,1) = 0.00018000       (-13,1) = 0.00004500     (-11,1) = 0.00001500     (-11,0) =
-----------------------------------------------------------------------------------------------------

Prob1    Prob0    CWs Sent          Bits2AvgOver    MR1     MR0     MRDS    mRDS    aMR1    aMR0
89.72    10.28    66667             1000003.0000    13.0    12.0    13.0    -13.0   5.0600  5.0645

NumStates = 38   (0,0) = 0.03659927       (-9,1) = 0.01249475      (-6,0) = 0.02029459      (-5,0) = 
-----------------------------------------------------------------------------------------------------
```

Figure 7-11: SimFPGA results for the (15,10) code showing runlength probabilities

Figure 7-12 compares the range of BER values the (15,10) code can assume compared with BPSK. Clearly the range of results for the (15,10) code is less than the corresponding range of the BPSK signal. As well, when the SNR is greater than 9dB, the performance of the (15,10) code with unbalanced source statistics (90% chance of a logic 0) is actually better than the performance of the BPSK signal with balanced source statistics (50% chance of a 0). It can be concluded that the multimode coding technique is removing the BER dependency on the source statistics.

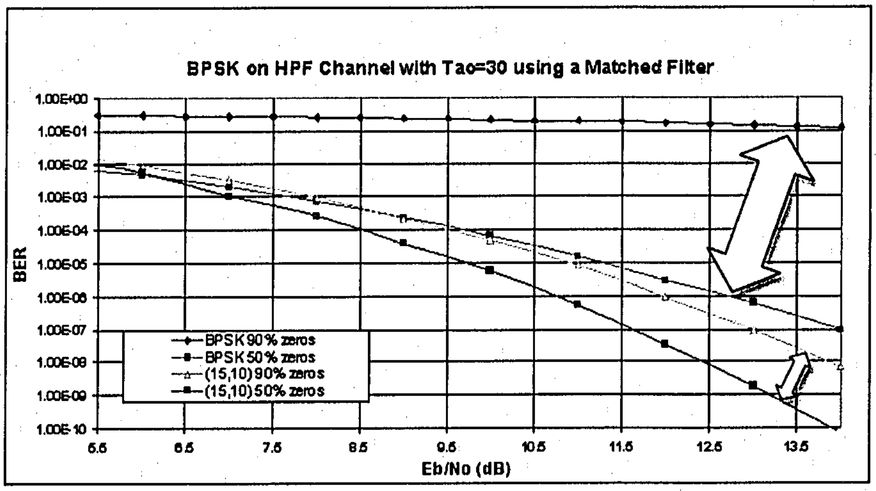


Figure 7-12: Range of BER results for the (15,10) code

7.6. Effect of 8 AddCWs with the (15,8) code

In Chapter 4 and Chapter 5 the (15,11) code expurgated to a (15,8) code with eight AddCWs was analyzed in great detail. Using the (15,8) code with any one of the four best AddCW sets results in the BER curve shown in Figure 7-13. With balanced source statistics the first test demonstrates the ability of this code to further improve the BER curve on the HPF channel beyond the performance of the (15,10) code at high values of SNR. This ability is a result of the fact that the (15,8) code is not only balanced, but that the worst case maximum runlength is only 7, versus 22 with the (15,10) code.

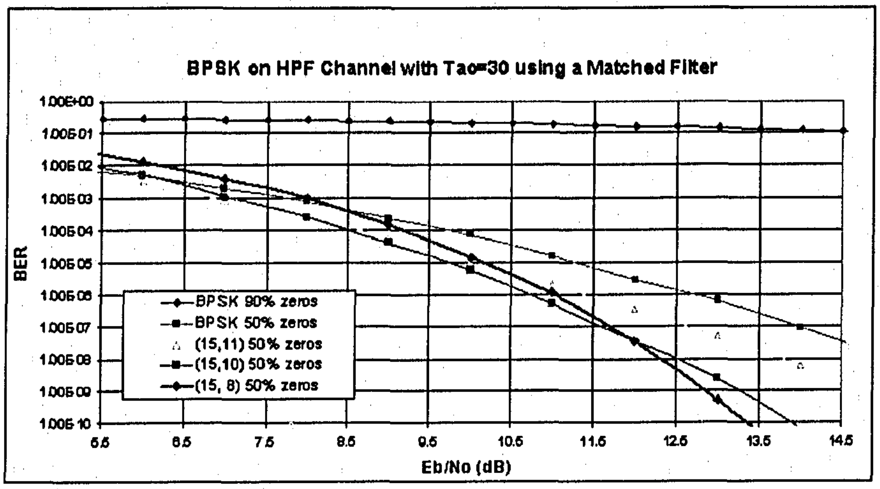


Figure 7-13: BER performance of the (15,8) code

The benefits of this coding technique are appreciated more when compared to BPSK signaling on a regular AWGN channel. In Figure 7-14 it can be seen that the BER curve of the (15,8) code is approaching the BER curve of the original BPSK signal on a regular AWGN channel.

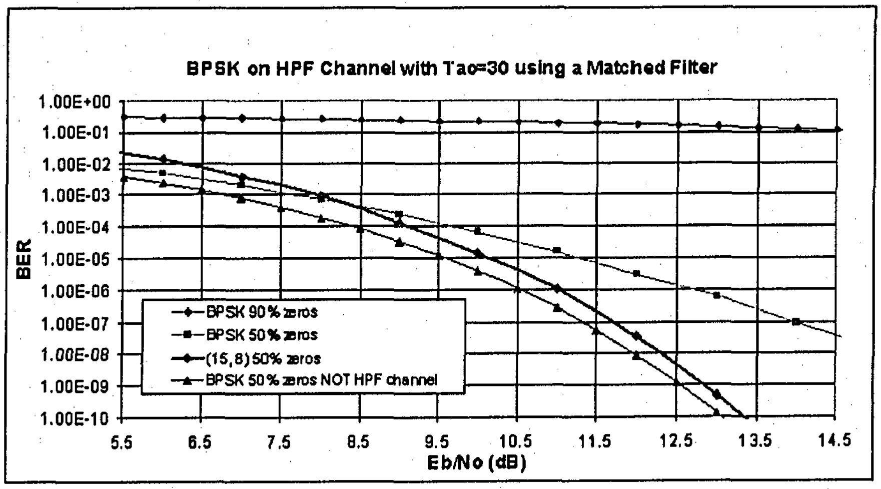


Figure 7-14: The (15,8) code on the HPF channel compared to BPSK on an AWGN channel

Finally, when using the (15,8) code with an unbalanced source, simulation results show that the BER curve remains fixed. That is, the BER curve for the 50% zeros and

90% zeros are identical. This is shown in Figure 7-15 which demonstrates that the source statistics no longer have any effect of the BER curve performance as a result of the CS coding features of this multimode coding technique.

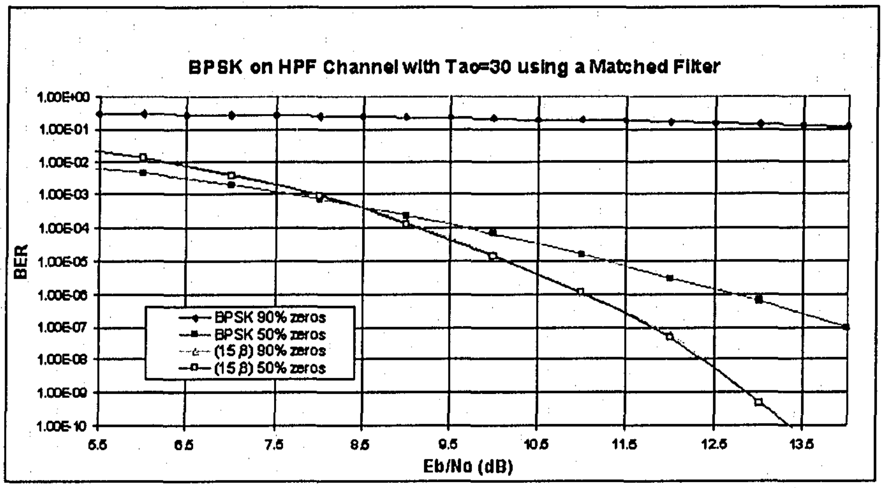


Figure 7-15: Performance of the (15,8) code does not change with varying source statistics

7.7. Performance of larger codes

It was suggested in Chapter 3 that the additional redundancy of the multimode coding technique can be minimized by using larger codes. Based on Section 7.5 and the results of the (15,10) code, the performance of the (31,26) code expurgated by 1-bit to a (31,25) code with two AddCWs is now investigated.

Figure 7-16 shows the range of values the BER curve can assume for the (31,25) code. At first glance it may be surprising to see that the range of BER values is far greater than the corresponding (15,10) code considered in Section 7.5. This can be understood by looking at the SimFPGA results. As shown in Figure 7-17 when the source statistics are balanced the maximum runlength (MR1 and MR0) of the (31,25) code is approximately 20, which is 2 less than the (15,10) code. As well, while the average runlength (aMR1 and aMR0) of approximately 4.5 is slightly higher than that of the (15,10) code, the redundancy in the (31,25) code is considerably less. Thus with a balanced source the (31,25) code performs slightly better than the (15,10) code.

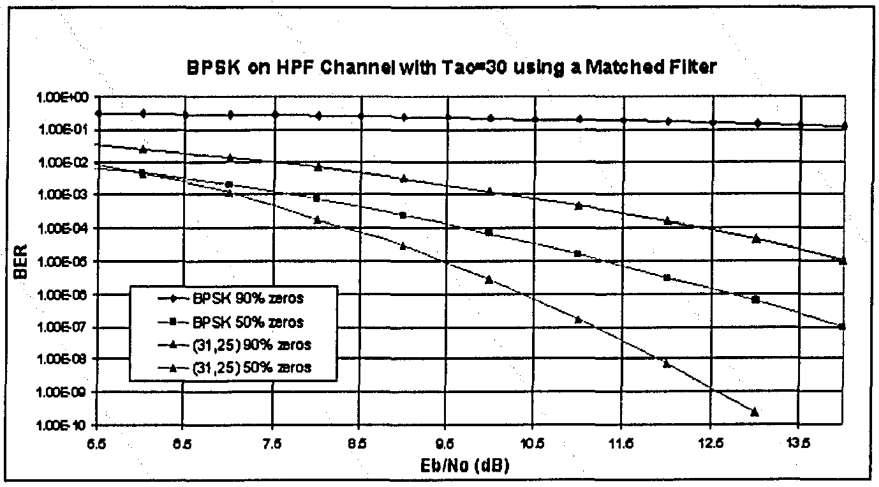
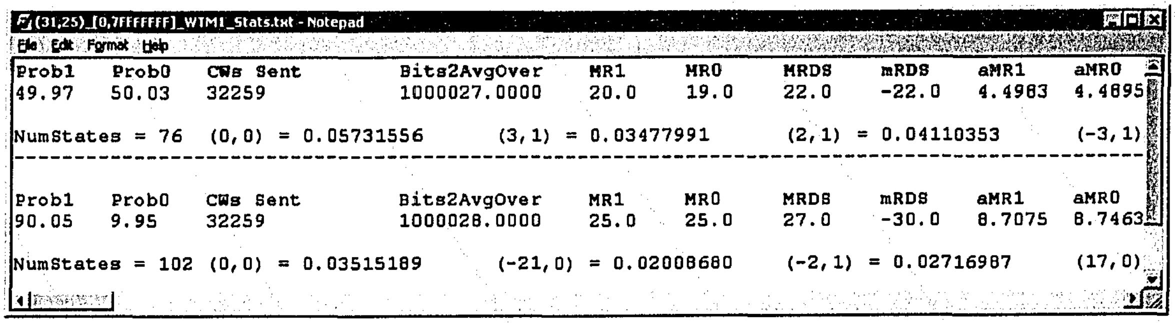


Figure 7-16: BER performance of the (31,25) code

Conversely when the source statistics become unbalanced, the (31,25) code generated maximum runlengths (MR1 and MR0) of approximately 25 like valued bits in a row, versus only 13 for the (15,10) code. In addition the average runlength (aMR1 and a MR0) increases to approximately 8.7, versus only 5 for the (15,10) code. Hence the BER curve for the (31,25) code with unbalanced source statistics is significantly worse than the curve for the corresponding (15,10) code, and hence the range of BER values for the (31,25) code is much greater as shown in Figure 7-18.



```
Prob1    Prob0    CWs Sent         Bits2AvgOver     MR1      MR0      MRDS     mRDS     aMR1     aMR0
49.97    50.03    32259            1000027.0000     20.0     19.0     22.0     -22.0    4.4983   4.4895

NumStates = 76   (0,0) = 0.05731556      (3,1) = 0.03477991       (2,1) = 0.04110353        (-3,1)
--------------------------------------------------------------------------------------------------

Prob1    Prob0    CWs Sent         Bits2AvgOver     MR1      MR0      MRDS     mRDS     aMR1     aMR0
90.05    9.95     32259            1000028.0000     25.0     25.0     27.0     -30.0    8.7075   8.7463

NumStates = 102 (0,0) = 0.03515189      (-21,0) = 0.02008680     (-2,1) = 0.02716987       (17,0)
```

Figure 7-17: SimFPGA results for the (31,25) code showing runlength probabilities

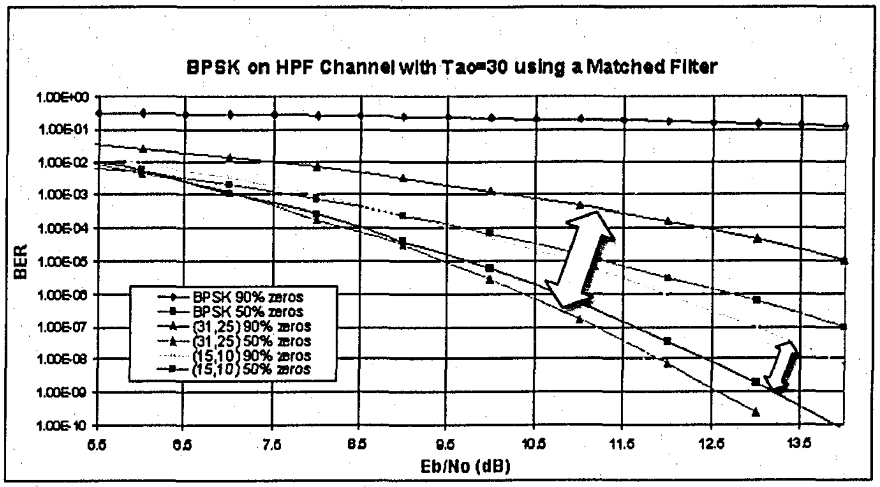


Figure 7-18: Range of BER values for the (31,25) code versus the (15,10) code

7.8. Effect of 8 AddCWs with the (31,23) code

Using the AddCW recommendations of Section 4.15, the (31,26) code expurgated to (31,23) with eight AddCWs is now investigated. In general it can be seen from Figure 7-19 that this code performs better than the (15,8) code. This is due to the fact that the (31,23) code has less redundancy than the (15,8) code, while still providing EC and CS abilities. Figure 7-19 also demonstrates that like the (15,8) code, the performance of the (31,23) code is not affected by the source statistics.

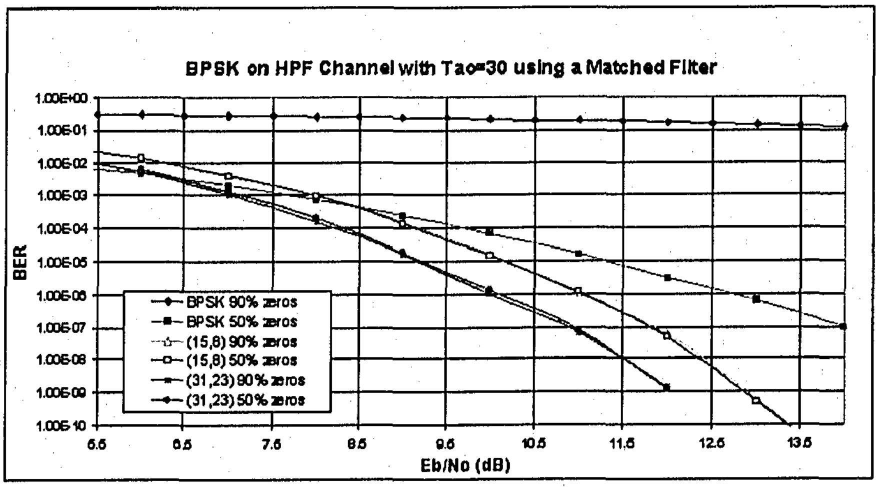


Figure 7-19: BER performance of the (31,23) code is better than that of the (15,8) code

Finally it can be seen in Figure 7-20 that the BER performance of the (31,23) code is actually better than BPSK signaling on an AWGN channel only. This clearly demonstrates how with larger codes the drawbacks associated with redundancy in this multimode coding technique can be minimized. With larger codes the efficiency increases and consequently the BER penalty incurred decreases.

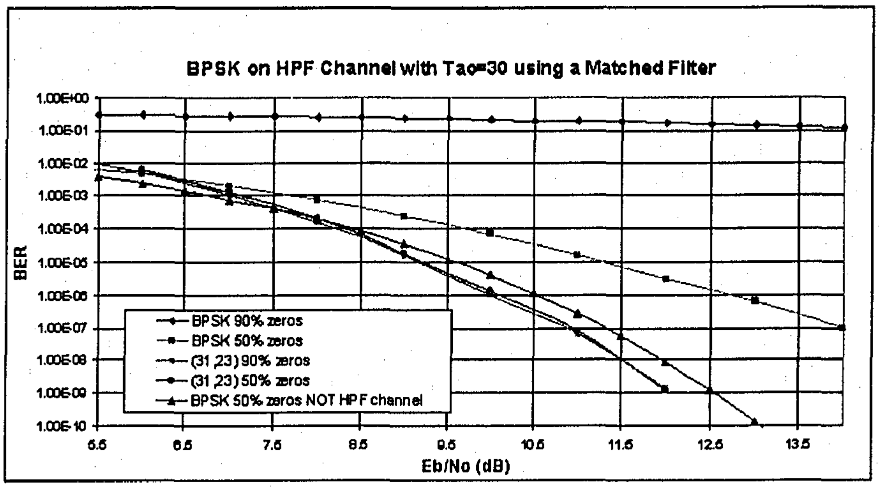


Figure 7-20: BER performance of the (31,23) code on the dc-constrained channel is better than the BER performance of BPSK on an AWGN channel

7.9. Conclusions

This chapter has investigated the BER performance of the multimode coding technique on a dc-constrained AWGN channel. On this type of channel the source statistics play a significant role in BER performance. As the source statistics change, so too does the BER experienced in the system.

It was shown that on this type of channel EC coding techniques alone do little to overcome this dependency on source statistics. However by using the multimode coding technique introduced in Chapter 3, the BER performance of the system can be improved. With a single extra bit of redundancy it was shown that the range of BER values experienced can be restricted and improved versus a system that does not use this coding.

In addition, it was shown that once the system incorporates three additional bits of redundancy, the BER performance can be improved and completely lose its dependency on the source statistics. Furthermore, the redundancy required to obtain this CS coding effect can be minimized by using larger block codes.

8. Conclusion

This thesis has introduced a new combined EC and CS code based on linear block codes. This chapter summarizes the development of this coding technique and offers suggestions for future work.

8.1. Thesis summary

Following the introduction and discussion of the concepts of EC and CS coding presented in Chapter 1 and 2, Chapter 3 introduced a novel combined EC and CS code which is based on multimode coding. It was apparent that with a single extra bit of redundancy any linear block code that includes the all-zero and all-one codeword can be used to construct a balanced dc-free code since the CWs are effectively partitioned into two complementary sets, giving the encoder two CW choices per SW. It was also shown that this partitioning can be accomplished through linear code word addition. Furthermore this technique was extended by giving the transmitter four and eight CW choices per SW through the use of four and eight AddCWs respectively, thus it is called a multimode coding technique.

Chapter 4 reviewed the mathematics required for analyzing the encoder as a Markov chain and evaluating its performance in the time domain. This involved a computer search to find the best AddCW sets, and results were presented contrasting the performance between the best and worst AddCW sets. The chapter concluded with recommendations and applications of this technique to other systems.

Chapter 5 investigated the frequency domain characteristics of the multimode coding scheme, and the power spectral densities of various code configurations were evaluated and analyzed. Chapter 6 presented the hardware implementation that used two FPGAs and compared their measured time domain and frequency domain results to the results in Chapter 4 and 5, demonstrating how they agreed with calculations.

Finally Chapter 7 looked at the bit error rate performance of this combined coding technique on a dc-constrained channel and clearly showed how its performance is superior to uncoded systems.

8.2. Future work

In this thesis thorough analysis was completed on the (15,11) code expurgated to a (15,8) code. It was shown that four sets of AddCWs had the best performance in terms of limiting runlengths. However no specific pattern emerged from these results. While most of the AddCWs were balanced and had numerous transitions, there was the odd exception that did not adhere to this rule. Therefore only general guidelines could be recommended for selecting AddCWs for use with larger codes where a computer search was impractical. While it was shown in Chapter 7 that these recommendations with the (31,23) code produced excellent results, the optimum AddCW sets for this code and larger codes remain unknown. Further work could quantify the results from the analysis of the (15,8) code and extend them to larger codes.

Finally, as proof of concept, the only linear block code considered in this thesis was the simple Hamming code. Future work could apply this multimode coding technique to other more powerful error control block codes.

References

[1] R. Togneri, C. Desilva, *Fundamentals of Information Theory and Coding Design*, Chapman & Hall/CRC, London, 2003.

[2] P. Sweeney, *Error Control Coding: From Theory to Practice*, John Wiley & Sons Ltd, Chichester, 2002.

[3] T. Cover, J. Thomas, *Elements of Information Theory*, Wiley-Interscience, New York, 1991.

[4] B. Lathi, *Modern Digital and Analog Communications Systems*, Third Edition, Oxford University Press, New York, 1998.

[5] B. Sklar, *Digital Communications: Fundamentals and Applications*, Second Edition, Prentice Hall PTR, New York, 2001.

[6] I. J. Fair, Y. Xin, "A method of integrating error control and constrained sequence coding," *Elec Letters*, vol. 36, pp. 210-215, February 2003.

[7] F. Zhai, Y. Xin, I. J. Fair, "DC-Free Multimode EC Block Codes," *Proceedings of the 2003 IEEE International Symposium on Information Theory*, p. 76, Yokohama, Japan, June 29 – July 4, 2003.

[8] I. J. Fair, D. R. Bull, "DC-Free Error Control Coding through Guided Convolutional Coding," *Proceedings of ISIT 2002*, p. 297, Lausanne, Switzerland, June 30 – July 5, 2002.

[9] I. J. Fair, Y. Xin, "Constrained Sequences with Embedded Redundancy for Error Control," *Elec Letters*, vol. 36, pp 215-217, February 2000.

[10] T. Wadayama, A. J. Han Vinck, "DC-Free Binary Convolutional Coding," *IEEE Trans. Infor. Theory*, vol 48, no. 1, pp. 162-173, January 2002.

[11] R. H. Deng, M. A. Herro, "DC-Free Coset Codes," *IEEE Trans. Infor. Theory*, vol. 34, pp. 786-792, July 1988.

[12] CE Shannon. "A mathematical theory of communication," *Bell System Technical Journal*, Journal 27 pp. 379-423, July and October 1948.

[13] G. Clark Jr., J. Cain, *Error-Correction Coding for Digital Communications*, Springer Plenum US, 1981.

[14] I. J. Fair, W. D. Grover, W. A. Krzymien, R. I. MacDonald, "Guided Scrambling: a new line coding Technique for High Bit Rate Fiber Optic Transmission Systems," *IEEE Trans. Commun.*, vol. 39, no. 2, pp. 289-297, February 1991.

[15] S. D. Personick, "Receiver design for optical fiber systems,'' *Proc. IEEE*, vol. 65, pp. 1670-1678, December 1977.

[16] T. Van Muoi, "Receiver design for High-Speed Optical-Fiber Systems," *Journal Lightwave Tech*, vol. LT-2, no. 3, pp 243-267, June 1984.

[17] F. Zhai, Y. Xin, and I. J. Fair, "Performance Evaluation of DC-Free EC Block Codes," *Proceedings of the 2004 Ninth International Conference on Communication Systems*, pp. 451-455, Singapore, September 6 – 8, 2004.

[18] G. Byeong, C. Seok, *Scrambling Techniques for Digital Transmission*, Springer-Verlag Ltd, Great Britain, 1994.

[19] K. A. S. Immink, *Codes for Mass Data Storage System*, Shannon Foundation Publishers, The Netherlands, 1999.

[20] I. J. Fair, V. K. Bhargava, Q. Wang, "Evaluation of the power spectral density of Guided Scrambling Coded Sequences," *IEE Proc.-Commun.*, vol. 144, no. 2, April 1997.

[21] S. Al-Bassam, B. Bose, "Design of Efficient Balanced Codes," *IEEE Trans. Comput*, vol. 43, no. 3, March 1994.

[22] T. Floyd, *Digital Fundamentals*, Eighth Edition, Prentice Hall, New York, 2002.

[23] A. Papoulis, S. Pillai, *Probability, Random Variables and Stochastic Processes*, Fourth Edition, McGraw-Hill Science, New York, 2001.

[24] G. L. Cariolaro, G. P. Tronca, "Spectra of Block Coded Digital Signals," *IEEE Trans Commun*, vol. Com-22, no. 10, pp 1555-1564, October 1974.

Appendix A - Calculating Power Spectral Density

The **energy spectral density** (ESD) is a measure of the **energy** contributed by all spectral components of an energy signal. If a signal has infinite energy it is considered to be a power signal and its **power spectral density** (PSD) is a measure of the **power** contributed by all spectral components. Both the ESD and PSD are important quantities in evaluating the performance of a communication system. For example, in Figure A-1, modulation shifts the spectrum of a baseband signal to a higher frequency band. If the allocated BW for this system was 100kHz, measuring the ESD or PSD on a spectrum analyzer would indicate whether or not the system is operating within this limit.

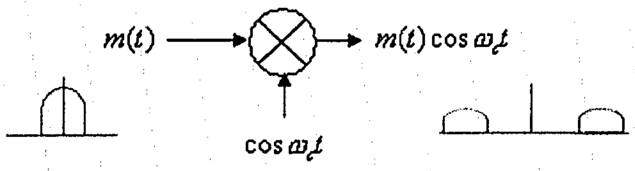


Figure A-1: PSD displays the contribution of power at each frequency

A.1. Determining the Power Spectral Density

There are three ways to find spectral densities

1. Measure them with a **Digital Oscilloscope** or **Spectrum Analyzer**
2. Calculate them mathematically
3. Simulation

The first approach can be difficult since it involves actually building a working system. This may not be practical if you are only interested in the PSD of some new communication approach. The second approach can also be difficult simply due to the mathematics involved, and it becomes more challenging once coding is introduced. For example, what is the PSD of a Hamming code? The third approach is therefore often used. In this Appendix, the second approach is considered in detail.

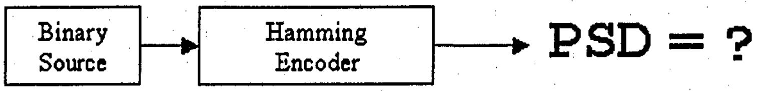


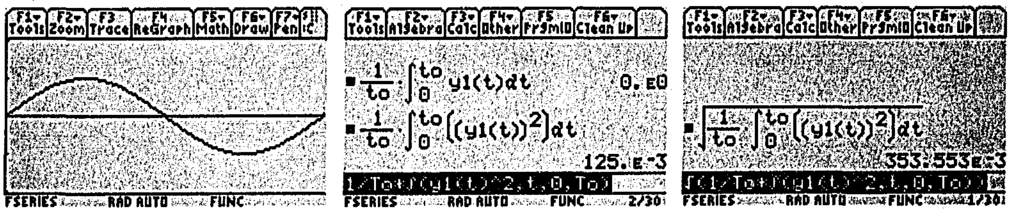
Figure A-2: Once coding is introduced, calculating the PSD can be challenging

A.2. How to calculate RMS for non-sinusoidal signals

Quite often measuring equipment such as oscilloscopes work in RMS when dealing with time varying voltages. Therefore to compare calculations with measured values it is convenient to also work in RMS. Thus it is useful to review this concept.

The **RMS (root mean square)** value of a signal is defined in general as the square root of the power of a time varying signal. It was originally used to determine the equivalent DC voltage required to deliver the same power as a sinusoidal signal.

Example 1 - RMS: Consider a 1Vpp sine wave at 1MHz. This is a signal that varies from $-0.5V$ to $+0.5V$ and would be written as $y_1(t) = 0.5\sin(2\pi ft)$. Find the power and RMS voltage with $t_0 = 1/f$.



```
■ 1/to · ∫[0,to] y1(t)dt                    0.E0
■ 1/to · ∫[0,to] ((y1(t))^2)dt
                                          125.E-3
1/To*∫(y1(t)^2,t,0,To)
FSERIES      RAD AUTO      FUNC        2/30
```

```
■ √(1/to · ∫[0,to] ((y1(t))^2)dt)
                                        353.553E-3
√(1/To*∫(y1(t)^2,t,0,To))
FSERIES      RAD AUTO      FUNC        1/30
```

Figure A-3: Calculating the power and RMS of a sine wave

As shown in Figure A-3 the power is found to be 125mW assuming the voltage is dissipated over 1 ohm. The square root of the power gives 353.55mV RMS. Thus 353.55mV is the equivalent DC voltage that would give the same power over 1 ohm, that is $(353.55\text{mV DC})^2 = 125\text{mW}$.

A simplified way to find the RMS voltage for sinusoidal waveforms is:

$$RMS = \frac{Vpp}{2\sqrt{2}} \qquad RMS = \frac{Vpeak}{\sqrt{2}} \tag{A.1}$$

Either way the result is $\boxed{\frac{1Vpp}{2\sqrt{2}} = 353.55mV}$ $\boxed{\frac{0.5V_{peak}}{\sqrt{2}} = 353.55mV}$

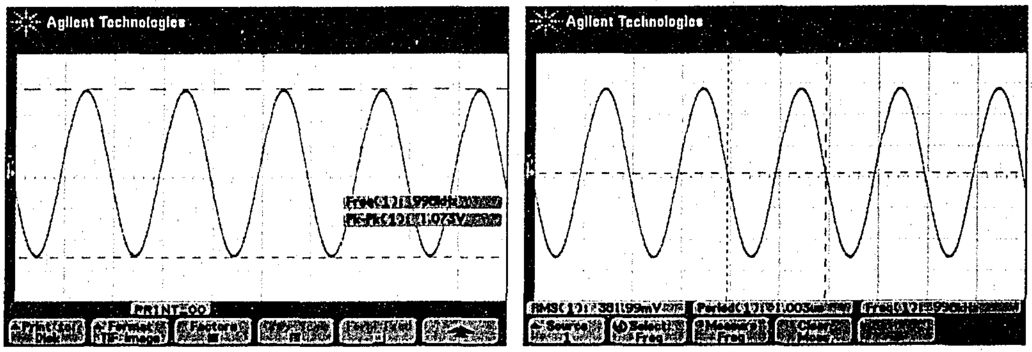


Figure A-4: Digital oscilloscope measurements

Oscilloscope screen shots shown above in Figure A-4 have a peak to peak value of 1.073V and as a result the measured the RMS value is 381mV. This matches very closely to 379.36mV and $1.073/\left(2\sqrt{2}\right) = 379.36\text{mV}$.

A.3. Comparing Fourier calculations to spectrum analyzer output

Converting periodic time domain signals to the frequency domain is done with Fourier Series representations. These are classified as the Trigonometric Fourier Series, the Compact Fourier Series and the Exponential Fourier Series.

A.4. Trigonometric Fourier Series

$$f(t) = a_0 + \sum_{n=1}^{\infty}\left(a_n \cos(n\omega_o t) + b_n \sin(n\omega_o t)\right) \tag{A.2}$$

where $\omega_o = \dfrac{2\pi}{T_o}$, T_o is the period

$$\begin{aligned} a_0 &= \frac{1}{T_o}\int_0^{T_o} f(t)dt \\ a_n &= \frac{2}{T_o}\int_0^{T_o} f(t)\cos(n\omega_o t)dt \qquad n = 1,2,3.. \\ b_n &= \frac{2}{T_o}\int_0^{T_o} f(t)\sin(n\omega_0 t)dt \qquad n = 1,2,3.. \end{aligned} \tag{A.3}$$

A.5. Compact Fourier Series

An equivalent form of the trigonometric Fourier series is to combine the sine and cosine terms into a single sinusoid using the identity

$$a_n \cos(n\omega_o t) + b_n \sin(n\omega_o t) = C_n \cos(n\omega_o t + \theta_n) \tag{A.4}$$

$$\begin{aligned} C_n &= \sqrt{a_n^{\,2} + b_n^{\,2}} \\ \theta_n &= \tan^{-1}\left(\frac{-b_n}{a_n}\right) \end{aligned} \qquad f(t) = C_0 + \sum_{n=1}^{\infty} C_n \cos(n\omega_o t + \theta_n) \tag{A.5}$$

A.6. Exponential Fourier Series

This is the most compact form of the Fourier series and it is based on the Euler identities since each sinusoid is expressed as the sum of two exponentials.

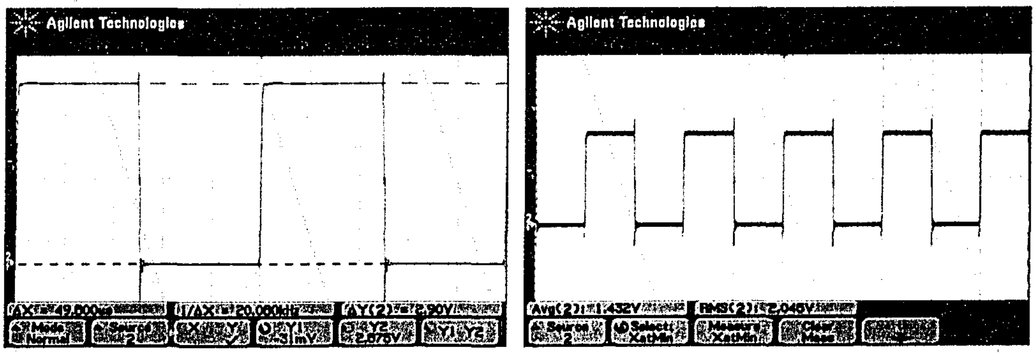
$$\sin(\theta)=\frac{e^{j\theta}-e^{-j\theta}}{2j} \quad \text{and} \quad \cos(\theta)=\frac{e^{j\theta}+e^{-j\theta}}{2} \tag{A.6}$$

$$f(t)=\sum_{-\infty}^{\infty} D_n e^{jn\omega_o t} \quad \text{where} \quad D_n=\frac{1}{T_o}\int_{T_o} f(t)e^{-jn\omega_o t}dt \tag{A.7}$$

The important thing to note with an Exponential Fourier Series is that it is a two-sided series. Thus there are negative and positive frequency components that range from –∞ to +∞. This is in contrast to the Compact and Trigonometric Fourier Series that are one-sided series that only have positive frequency components in the range 0 to +∞. Note that these two forms are related by Equation A.8 for real valued signals.

$$C_n=2D_n \qquad n=1,2,3... \tag{A.8}$$

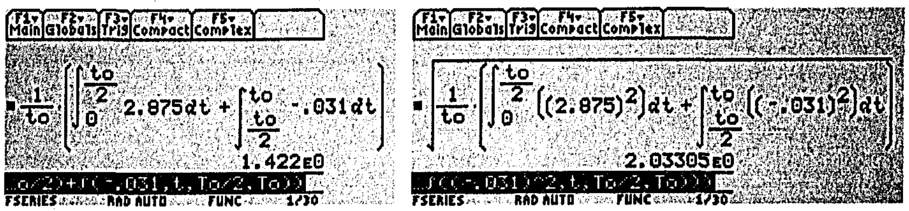
Example 2 – Compact Fourier Series: Calculate the Fourier series of a 100kHz, 2.9V, 50% duty cycle unipolar square wave.



```
ΔX = 49.000us    1/ΔX = 20.000kHz    ΔY(2) = 2.90V
```

```
Avg(2): 1.432V    RMS(2): 2.046V
```

Figure A-5: 50% duty cycle square wave on scope



$$\frac{1}{to}\left(\int_0^{\frac{to}{2}} 2.875\,dt+\int_{\frac{to}{2}}^{to} -.031\,dt\right) \qquad 1.422E0$$

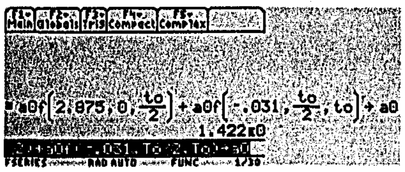
```
o/2)+∫(-.031,t,To/2,To))
```

$$\sqrt{\frac{1}{to}\left(\int_0^{\frac{to}{2}} \left((2.875)^2\right)dt+\int_{\frac{to}{2}}^{to} \left((-.031)^2\right)dt\right)} \qquad 2.03305E0$$

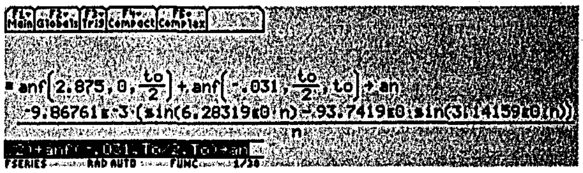
```
∫((-.031)^2,t,To/2,To)))
```

Figure A-6: Calculating the average and RMS

Figures A-5 and A-6 show that the calculated average and RMS values are practically identical to the measured values. Calculating the Trigonometric Fourier Series by Equation A.3 with $f_0 = 100\text{kHz}$ and $t_0 = 1/f_0 = 10\mu\text{s}$ is shown below in Figures A-7 and A-8.

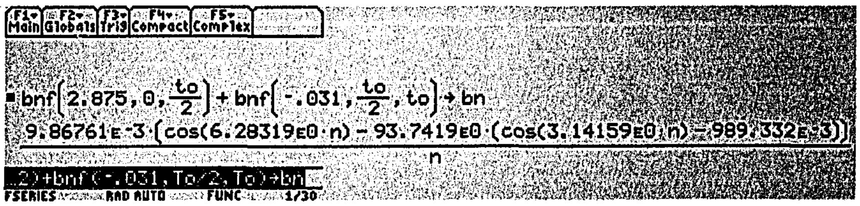


```
■a0f(2.875,0,to/2)+a0f(-.031,to/2,to)→a0
                    1.422E0
2)+a0f(-.031,To/2,To)→a0
FSERIES   RAD AUTO   FUNC   1/30
```



```
■anf(2.875,0,to/2)+anf(-.031,to/2,to)→an
 -9.86761E-3·(sin(6.28319E0·n)-93.7419E0·sin(3.14159E0·n))
 ─────────────────────────────────────────────────────────
                            n
2)+anf(-.031,To/2,To)→an
FSERIES   RAD AUTO   FUNC   1/30
```

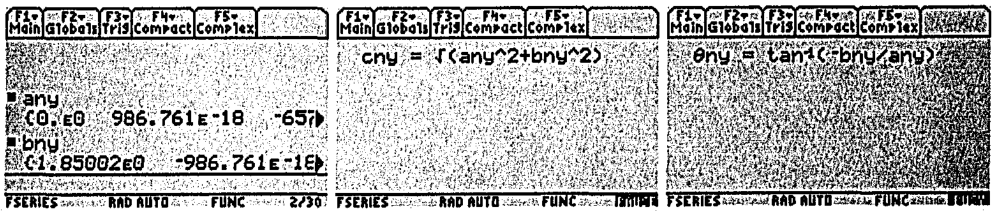
Figure A-7: Calculating a_0 and a_n



```
■bnf(2.875,0,to/2)+bnf(-.031,to/2,to)→bn
 9.86761E-3·(cos(6.28319E0·n)-93.7419E0·(cos(3.14159E0·n)-989.332E-3))
 ─────────────────────────────────────────────────────────────────────
                                n
2)+bnf(-.031,To/2,To)→bn
FSERIES   RAD AUTO   FUNC   1/30
```

Figure A-8: Calculating b_n

Figure A-7 shows that $a_0 = 1.422V$ which is identical to the DC average shown in Figure A-6. As well, the a_n and b_n values are calculated for $n = 1,2,3,...$ which are used to calculate the Compact Fourier Series using Equation A.5.

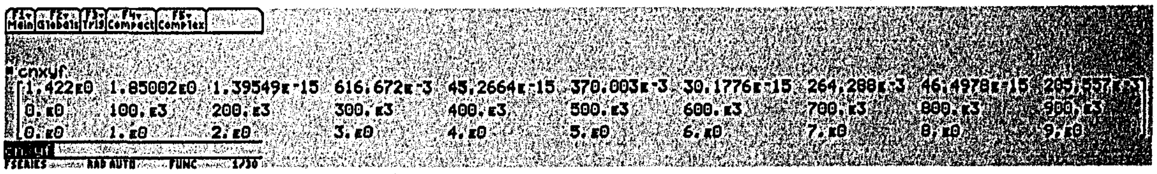


```
■any
 {0.E0   986.761E-18   -657▸
■bny
 {1.85002E0   -986.761E-1▸
FSERIES   RAD AUTO   FUNC   2/30
```

```
cny = √(any^2+bny^2)
```

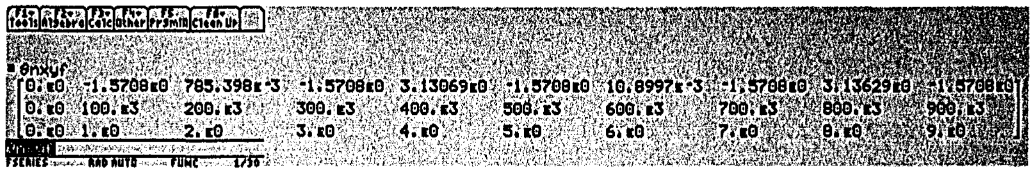
```
θny = tan⁻¹(-bny/any)
```

Figure A-9: Shows the a_n and b_n values in a list, and the conversion to find the C_n values



```
■cnxyf
[1.422E0  1.85002E0  1.39549E-15  616.672E-3  45.2664E-15  370.003E-3  30.1776E-15  264.288E-3  46.4978E-15  205.557E-3]
[0.E0     100.E3     200.E3       300.E3      400.E3       500.E3      600.E3       700.E3      800.E3       900.E3    ]
[0.E0     1.E0       2.E0         3.E0        4.E0         5.E0        6.E0         7.E0        8.E0         9.E0      ]
FSERIES   RAD AUTO   FUNC   1/30
```

Figure A-10: C_n values in a list with frequency values and *n* index below



```
■ θnxyf
[0.E0  -1.5708E0  785.398E-3  -1.5708E0  3.13069E0  -1.5708E0  10.8997E-3  -1.5708E0  3.13629E0  -1.5708E0]
[0.E0  100.E3     200.E3      300.E3     400.E3     500.E3     600.E3      700.E3     800.E3     900.E3   ]
[0.E0  1.E0       2.E0        3.E0       4.E0       5.E0       6.E0        7.E0       8.E0       9.E0     ]
FSERIES    RAD AUTO    FUNC    17/30
```

Figure A-11: θ_n values shown in list with frequency values and *n* index below

Figures A-10 and A-11 above show the Compact Fourier Series representation of the signal. These magnitude and phase values indicate exactly how to recreate $f(t)$. That is you would need a DC level of 1.422V, plus a 1.85V peak cosine wave of frequency 100kHz with initial phase –1.57 rad, plus a 0.616V peak cosine wave of frequency 300kHz with initial phase –1.57 plus and so on. Clearly these are peak values which is important when converting to RMS.

A.7. Comparing to a Spectrum Analyzer and Digital Oscilloscope

In order to compare measured results to calculations, two things must be taken into account. First, most spectrum analyzers (SA) and digital oscilloscopes (DO) only show one-sided spectra, and second, SAs and DOs often work in RMS, and display their results in **decibels** (dB). Specifically the DOs considered here work in **dBV$_{\text{RMS}}$**, which is dB per 1V$_{\text{RMS}}$ normalized to 1 ohm, and SAs work in **dBm**, which is dB per 1mW normalized to 50 ohms.

As a result the Compact Fourier Series is a natural choice since the C_n values are already one sided. All that is required for comparison is to convert them to dB. This can easily be done by dividing them by $\sqrt{2}$ first (Equation A-1) since they are Vpeak, and then converting them to the appropriate dB scale as shown in Equations A-9 and A-10.

$$\text{dBV}_{RMS} = 10\,\text{Log}\left(\frac{\left(\frac{C_n}{\sqrt{2}}\right)^2}{1}\right) \tag{A.9}$$

$$\text{dBm} = 10\,\text{Log}\left(\frac{\left(\frac{C_n}{\sqrt{2}}\right)^2}{50*.001}\right) \tag{A.10}$$

For example Figure A-12 shows the square wave from Example 2 in the frequency domain. Figure A-13 shows the dBV_{RMS} values calculated from the C_n values, and Table A-1 shows how these calculated values match up with the measured values from the DO.

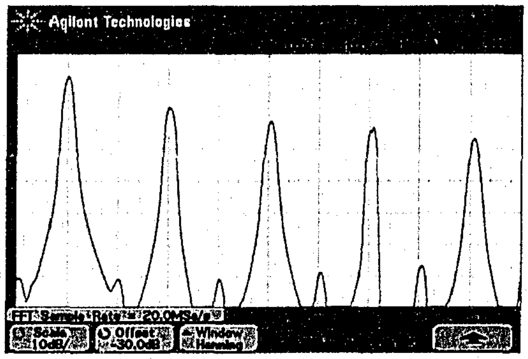
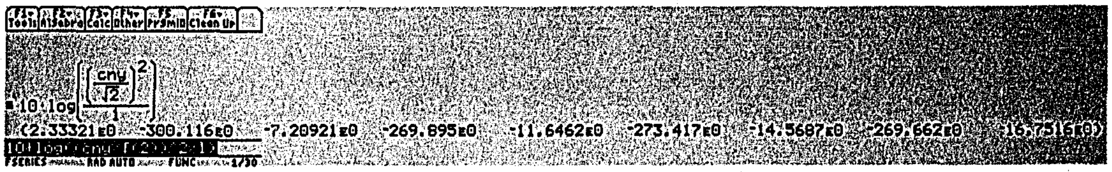


Figure A-12: DO showing the frequency domain PSD of this square wave


Tools Algebra Calc Other PrgmIO Clean Up
FSERIES RAD AUTO FUNC 1/30

```
■10·log(((cny/√2)^2)/1)
{2.33321E0  -300.116E0  -7.20921E0  -269.895E0  -11.6462E0  -273.417E0  -14.5687E0  -269.662E0  -16.7516E0}
10*log((cny/√(2))^2/1)
```

Figure A-13: Calculated dBV_{RMS} values of the square wave

Table A-1: Showing how the measured and calculated values match up

Harmonic	Frequency (kHz)	DBV_{RMS} measured	Calculated
Fundamental	100	2.5	2.31
2	200	< 60	< 60
3	300	-7.19	-7.22
4	400	< 60	< 60
5	500	-11.56	-11.66
6	600	< 60	< 60
7	700	-14.44	-14.58
8	800	< 60	< 60
9	900	-16.56	-16.76

Clearly this approach is very accurate.

Similarly Figure A-14 shows the same output on the SA. Figure A-15 shows the dBm values being calculated from the C_n values, and Table A-2 shows how these calculated values match up with the measured values.

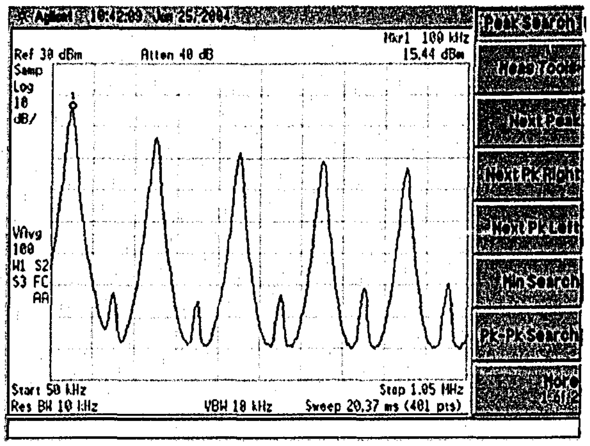
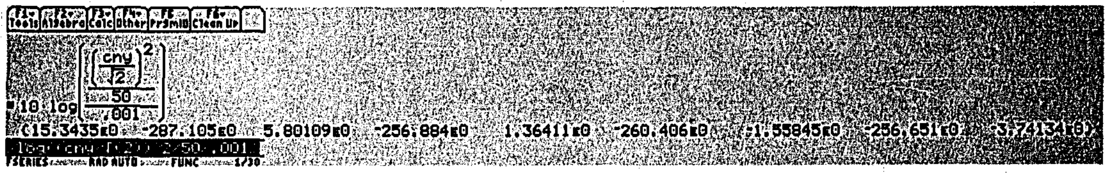


Figure A-14: SA showing the frequency domain PSD of this square wave



```
        ( ( cny )2 )
        ( ( ─── )  )
        ( ( √2  )  )
        ( ───────  )
        (   50     )
■10·log ( ───────  )
        (  .001    )
{15.3435E0  -287.105E0  5.80109E0  -256.884E0  1.36411E0  -260.406E0  -1.55845E0  -256.651E0  -3.74134E0}
10log((cny/√(2))^2/50/.001)
FSERIES      RAD AUTO      FUNC      1/30
```

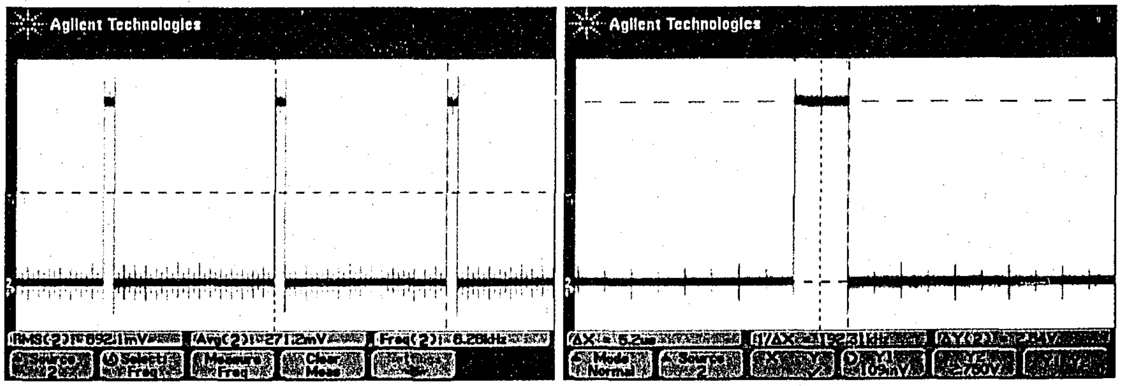
Figure A-15: Calculated dBm values of the square wave

Table A-2: Showing how the measured and calculated values match up

Harmonic	Frequency (kHz)	dBm measured	Calculated
Fundamental	100	15.42	15.32
2	200	< 40	< 200
3	300	6.11	5.78
4	400	< 40	< 200
5	500	1.65	1.34
6	600	< 40	< 200
7	700	-1.37	-1.57
8	800	< 40	< 200
9	900	-3.45	-3.75

Clearly this approach is very accurate.

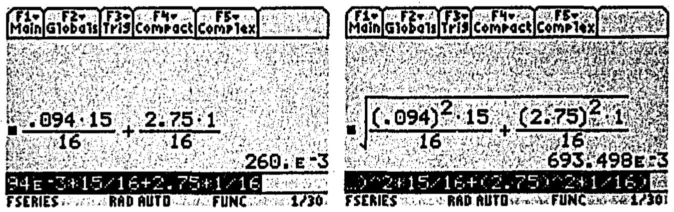
Example 3 – Repeat Example 2 with a square wave that does not have a 50% duty cycle:



```
RMS(2): 692.1mV    Avg(2): 271.2mV    Freq(2): 6.28kHz
ΔX: 5.2us    1/ΔX: 192.31kHz    ΔY(2): 2.64V
```

Figure A-16: Square wave with a 6.25% duty cycle

Figure A-16 shows a square wave with one 10us pulse every 160us. This can also be thought of as a 100kHz square wave with a 6.25% duty cycle. Looking at Figure A-16 shows that there is a slight DC offset of approximately 94mV, so the 2.65V peak to peak signal swings from 94mV to 2.75V.



$$\frac{.094\cdot 15}{16}+\frac{2.75\cdot 1}{16} \qquad 260.\text{E}{-3}$$

```
94E-3*15/16+2.75*1/16
```

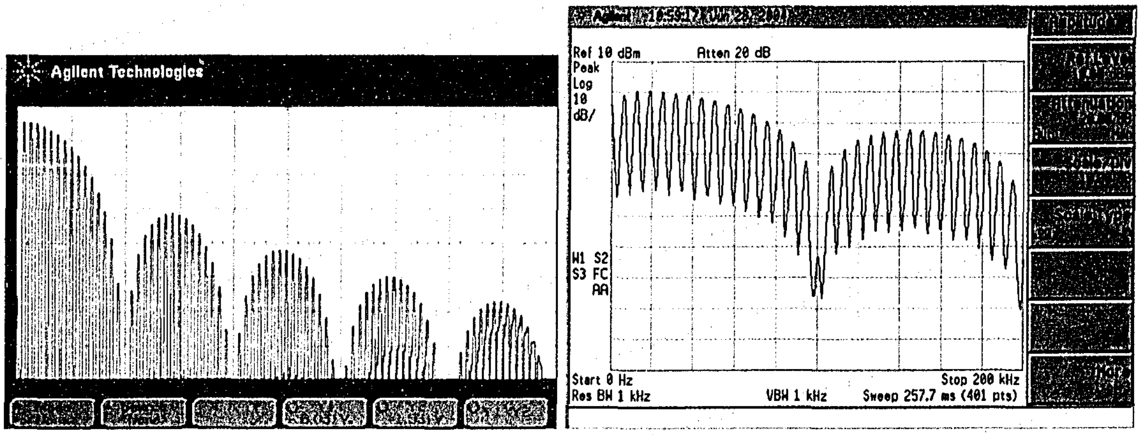
$$\sqrt{\frac{(.094)^2\cdot 15}{16}+\frac{(2.75)^2\cdot 1}{16}} \qquad 693.498\text{E}{-3}$$

```
…)^2*15/16+(2.75)^2*1/16)
```

Figure A-17: Calculating Average and RMS values for the 6.25% duty cycle square wave

Figure A-17 shows how to calculate the average. The signal has 94mV for 15/16th of the period, and 2.75V for 1/16th of the period for an average of 260mV, or 271.2mV measured. As well, the same approach can be used for the RMS voltage where 693.5mV is calculated and 692.1mV is measured. Note that 15/16th of the period can also be considered as 15 out of 16 bits and 1/16th of the period can be considered 1 out of 16 bits.

As shown in Figure A-18 the measured spectra appear more dense and more like the spectrum that corresponds to a single square wave. This result is pleasing since as the duty cycle decreases the pulses become farther apart and the time domain signal itself begins to appear more and more like a single square wave. Therefore the **Fourier Transform** could almost be used to calculated the spectrum (or "frequency content of the signal") instead of the Fourier Series.



Ref 10 dBm Atten 20 dB
Peak
Log
10
dB/
W1 S2
S3 FC
AA
Start 0 Hz Stop 200 kHz
Res BW 1 kHz VBW 1 kHz Sweep 257.7 ms (401 pts)

Figure A-18: 6.25% duty cycle square wave in the frequency domain on the DO and SA

Table A-3: Measured vs calculated for dBm and dBV$_{RMS}$ for 6.25% duty cycle square wave

Harmonic	Frequency (kHz)	dBm measured	dBm calculated	DBV$_{RMS}$ measured	DBV$_{RMS}$ calculated
1	6.25	-1.5	0.347	-12.81	-12.66
2	12.50	-0.1	0.178	-12.81	-12.83
3	18.75	-0.11	-0.105	-13.13	-13.11
4	25.00	-0.373	-0.508	-13.44	-13.51
5	31.25	-0.857	-1.039	-13.75	-14.05
6	37.50	-1.52	-1.708	-14.38	-14.71
7	43.75	-2.38	-2.527	-15.31	-15.53
8	50.00	-3.44	-3.519	-16.25	-16.52
9	56.25	-4.67	-4.710	-17.5	-17.72
10	62.50	-6.16	-6.14	-19.06	-19.15
11	68.75	-7.82	-7.88	-20.94	-20.89
12	75.00	-9.91	-10.05	-22.81	-23.06
13	81.25	-12.64	-12.84	-25.62	-25.85
14	87.50	-16.47	-16.72	-29.69	-29.73
15	93.75	-22.76	-23.17	-35.94	-36.18

A.8. Random Binary Signals

Examples so far have considered only the cases of repeating periodic signals. A different approach is needed when these signals are random. Consider a random binary waveform that takes on two different values $+A$ and $-A$, with transitions that occur only at integer multiples of the symbol period T_b with symbol values that are equally likely. This can be considered a **discrete time random process** with the probability density function (PDF) shown in Figure A-19.

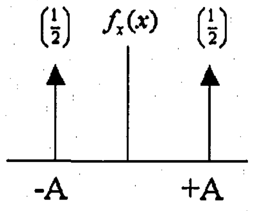


Figure A-19: PDF for the discrete time random process

Since this is now a random process (random signal) there is no easy way to do Fourier analysis. Therefore how does one find the PSD of a signal that is not deterministic? Clearly this signal can take on an infinite number of sequences. The PSD would be known if the duty cycle was always 50% or 6.25% like in Example 1 and 2, however now the signal can have any duty cycle at any given time. Since each sequence is different and not periodic the best we can realistically do is to evaluate this signal in terms of some average statistics of bit sequences. In order to do this we need the concept of a **random process.**

A random process (also known as a **stochastic process**) is a mapping of an experimental outcome to a function of time. Each waveform associated with an experiment is called a **sample function**. The collection of all sample functions is called the **ensemble** of the random process. In general there are two approaches to evaluating average statistics of random processes:

1. **Time average:** Examine one sample function from the ensemble (collection of sequences) over an extended period of time, i.e. examine a typical sequence
2. **Ensemble average:** Take averages over all possible different sample functions. This is usually the more comprehensive approach since we don't know if any one sample function is representative of the entire ensemble.

A process whose ensemble statistics don't change with time is a **stationary process**. They are usually called **strict sense stationary** (SSS) processes if all statistics do not change with time. i.e. PDF's and CDF's do not change with time. They are **wide sense stationary** (or weak sense stationary) (WSS) if at least the first and second order statistics do not change with time, i.e. the mean and variance. Processes in which all time averages of all sample functions are equal to the ensemble averages are called **strict sense ergodic** processes. If at least the first and second order time averages are equal to the first and second order ensemble averages then the process is called **wide sense ergodic** (or weak sense ergodic).

Note that in reality there is no such thing as a stationary process. However many processes can be considered stationary for the time interval of interest and this stationary assumption allows for a manageable mathematical model.

Some common statistics about random processes that are often useful

- $E[x]$ – Expected value or average. Indicates the DC level of the signal
- $E[x^2]$ – Expected square value – Total power (AC+DC) of the signal
- $\sigma^2 = E[x^2] - (E[x])^2$ – Variance – Total AC power (i.e. subtracting DC power)
- σ = Standard Deviation – RMS value of the AC components

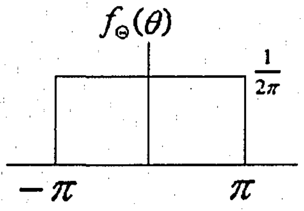
NOTE: When processes are zero mean (such as noise) the variance is the total power and the standard deviation is the total RMS value.

In general only two statistics are usually considered which are the **mean** and the **autocorrelation** of the random process. The autocorrelation of a random process is a measure of how similar the process is to itself as time increases. That is, correlating the function with itself as a function of time separation gives an indication of how quickly one can expect the random process to change. This will be demonstrated by example.

Example 4 – A random process: Consider the transmission of a signal from a TX to an RX where the frequency is given as f Hz and the attenuation is minimal. The received phase can be any value from 0 to 2π. This is shown in Equation A.11:

$$X(t) = A\cos(wt + \Theta) \qquad \text{(A.11)}$$

where A and ω are constants and the phase Θ is a uniform random variable over $[-\pi,\pi]$. Therefore the PDF is shown in Figure A-20.



$$f_\Theta(\theta) = \begin{cases} \frac{1}{2\pi} & -\pi \le \theta \le \pi \\ 0 & \textit{otherwise} \end{cases}$$

Figure A-20: PDF of the random process for Example 4

As shown in Equations A.12 and A.13 respectively, the first step is to find the time averaged mean and ensemble mean.

Time averaged mean:

$$\begin{aligned} \bar{x} &= \langle x(t) \rangle = \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{T/2} A\cos(\omega t + \theta)dt \\ \bar{x} &= \frac{A}{T_0} \int_{-T_0/2}^{T_0/2} \cos(\omega t + \theta)dt = 0 \end{aligned} \qquad \text{(A.12)}$$

Ensemble mean:

$$\mu_x(t) = E[X(t)] = \int_{-\infty}^{\infty} A\cos(\omega t + \theta) f_\Theta(\theta) d\theta$$

$$\mu_x(t) = \frac{A}{2\pi}\int_{-\infty}^{\infty} \cos(\omega t + \theta) d\theta = 0 \tag{A.13}$$

By inspection the average value of any cosine wave regardless of phase will be 0, i.e. no DC value. Thus even when considering the ensemble mean (all sample functions) there will be an equal number of sample functions above and below zero and thus the mean is still 0. Therefore this process is **Ergodic in the mean** since $\langle x(t) \rangle = E[X(t)]$

The next step is to find the time averaged autocorrelation and ensemble autocorrelation as shown in Equations A.14 and A.15 respectively.

Time averaged autocorrelation:

$$\bar{R}_{xx}(\tau) = \langle x(t)x(t+\tau) \rangle$$

$$\bar{R}_{xx}(\tau) = \lim_{T\to\infty}\frac{1}{T}\int_{-T/2} A^2\cos(\omega t + \theta)\cos(\omega(t+\tau)+\theta) dt$$

$$\bar{R}_{xx}(\tau) = \frac{A^2}{T_0}\int_{-T_0/2}^{T_0/2} \frac{1}{2}\left[\cos\omega t + \cos(2\omega t + 2\theta + \omega\tau)\right] dt \tag{A.14}$$

$$\bar{R}_{xx}(\tau) = \frac{A^2}{T_0}\int_{-T_0/2}^{T_0/2} \frac{1}{2}\left[\cos\omega t + \cos(2\omega t + 2\theta + \omega\tau)\right] dt$$

$$\bar{R}_{xx}(\tau) = \frac{A^2}{2}\cos\omega\tau$$

Ensemble autocorrelation:

$$R_{xx}(t,t+\tau)=E[X(t)X(t+\tau)]$$
$$R_{xx}(t,t+\tau)=\frac{A^2}{2\pi}\int_{-\pi}^{\pi}\cos(\omega t+\theta)\cos(\omega(t+\tau)+\theta)\,d\theta$$
$$R_{xx}(t,t+\tau)=\frac{A^2}{2\pi}\int_{-\pi}^{\pi}\frac{1}{2}\cos\omega\tau+\cos(2\omega t+2\theta+\omega\tau)\,d\theta \qquad \text{(A.15)}$$
$$R_{xx}(t,t+\tau)=\frac{A^2}{2}\cos\omega\tau$$

This process is **Ergodic in the autocorrelation** since $\langle x(t)x(t+\tau)\rangle = E[X(t)X(t+\tau)]$.

Furthermore since the mean is constant (does not depend on time) and the autocorrelation is a function of time separation τ only (does not depend on time) then we can conclude that this process is **wide sense stationary** (WSS).

Example 5 – Random Binary Signal. Recall from Section 2.19 the random binary waveform where 0 is mapped to –A and 1 is mapped to +A. Transitions occur only at integer multiples of the symbol period T_b and the symbol values are equally likely. If we let $A=1$ then by inspection we can see that if 50% of the time there are -1s, and 50% of the time there are +1s, then the mean value (DC value, expected value) of this random process should be 0. This is found to be true as shown in Equation A.16.

$$E[x]=\sum_{i=0}^{1}xf_x(x)=-1\left(\frac{1}{2}\right)+1\left(\frac{1}{2}\right)=0 \qquad \text{(A.16)}$$

Note that Equation A.16 finds the ensemble average and not the time average. Time averages are not as useful as ensemble averages in this case since a single sample function will not be indicative of all sample functions. Thus only ensemble averages will be considered from this point on.

The autocorrelation of a random binary waveform is not as straightforward as the previous example since it is known as a **cyclostationary random process.** This means that by the above definitions this random binary process is not strictly stationary, or even wide sense stationary. However it turns out that the statistics (mean, autocorrelation etc) are periodic. Therefore an average over mean and an average autocorrelation can be evaluated to obtain a stationary result. Hence the term cyclostationary (cyclically stationary).

To analyze random binary processes it is useful to interpret discrete time signals as trains of delta functions as shown in Figure A-21 and Equation A.17. This maps a **discrete time random process** to a **continuous time random process.**

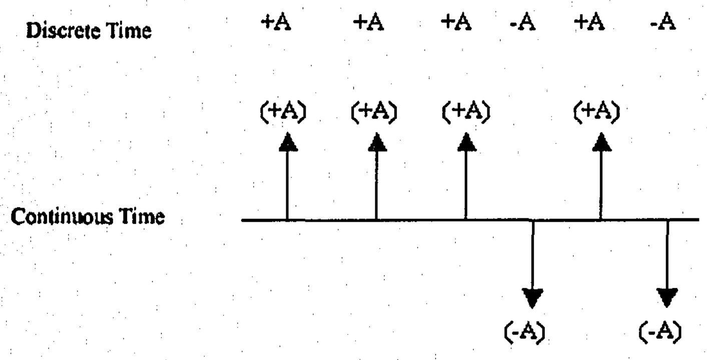


Figure A-21: Discrete time random process mapped to continuous time with delta functions

$$x(t) = \sum_{k=-\infty}^{\infty} a_k \delta(t - kT_b) \tag{A.17}$$

This works well since we can consider the final waveform to be the convolution of the train of delta functions with a pulse shape $p(t)$ as shown in Figure A-22 and Equation A-18.

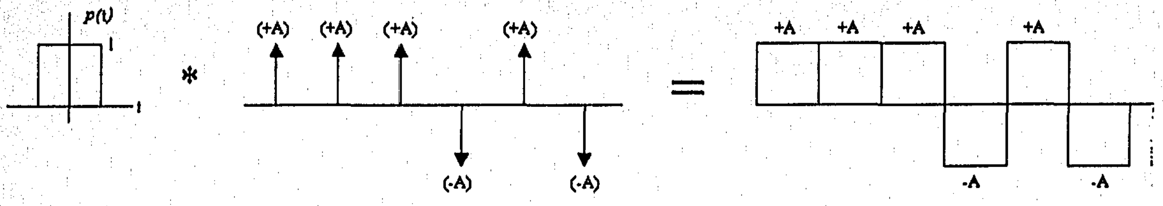


Figure A-22: Waveform is a convolution of the pulse shape and train of delta functions

$$x(t) = p(t) * \sum_{k=-\infty}^{\infty} a_k \delta(t - kT_b) \tag{A.18}$$

Therefore the average autocorrelation function for cyclostationary random processes as a function of time separation τ is defined as

$$R_{xx,avg} = p(\tau) * p(-\tau) * \left(\frac{1}{T} \sum_{k=-\infty}^{\infty} E\left[x_n x_{n+k}\right] \delta(\tau - kT_b) \right) \tag{A.19}$$

Equation A.19 is indicating that to evaluate the autocorrelation of this cyclostationary process, first convolve the pulse shape with itself, then convolve the result with the autocorrelation of the source symbols averaged over all time. An example of a square pulse shape being convolved with itself is shown in Figure A-23.

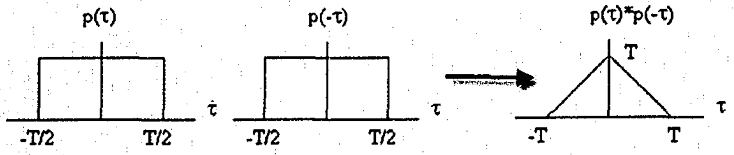


Figure A-23: Square pulse shape being convolved with itself

Equation A.20 shows the autocorrelation of the source symbols.

$$E[x_n x_{n+k}] = \begin{cases} k = 0 & E[x_n^2] = A^2 \\ k \neq 0 & E[x_n x_{n+k}] = E[x_n]E[x_{n+k}] = 0 \end{cases} \qquad \text{(A.20)}$$

Finally the autocorrelation by Equation A.19 is shown graphically in Figure A-24.

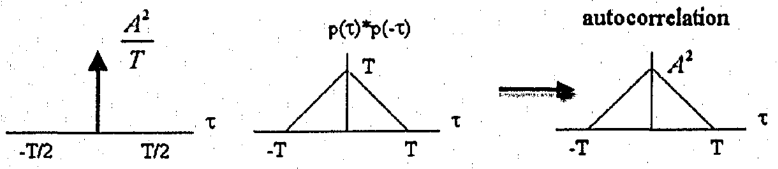


Figure A-24: Graphical depiction of Equation A.19

The Fourier Transform of the autocorrelation function of a random process is the power spectral density.

$$S_X(\omega) = \mathcal{F}\,[R_X(t)]$$

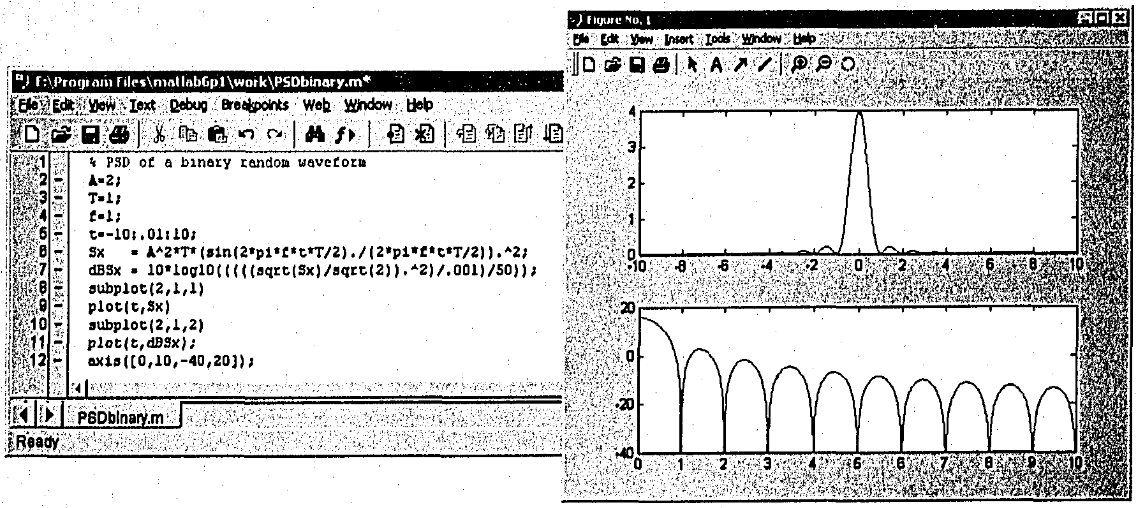
only place where delta function exists

$$S_X(\omega) = \frac{|P(\omega)|^2}{T}\sum_{n=-\infty}^{\infty} E[a_k a_{k+n}]\cdot e^{-j\omega nt}$$

effect of pulse shape

correlation of symbols

Figure A-25: Fourier transform of the autocorrelation function is the PSD



F:\Program Files\matlab6p1\work\PSDbinary.m*
File Edit View Text Debug Breakpoints Web Window Help
PSDbinary.m
Ready
Figure No. 1
File Edit View Insert Tools Window Help
4
3
2
1
0
-10
-8
-6
-4
-2
0
2
4
6
8
10
20
0
-20
-40
0
1
2
3
4
5
6
7
8
9
10

```
% PSD of a binary random waveform
A=2;
T=1;
f=1;
t=-10:.01:10;
Sx   = A^2*T*(sin(2*pi*f*t*T/2)./(2*pi*f*t*T/2)).^2;
dBSx = 10*log10(((((sqrt(Sx)/sqrt(2)).^2)/.001)/50));
subplot(2,1,1)
plot(t,Sx)
subplot(2,1,2)
plot(t,dBSx);
axis([0,10,-40,20]);
```

Figure A-26: Matlab script showing PSD that matches with Example 3

Figure A-25 shows the process to find the PSD from the autocorrelation function. Figure A-26 shows the Matlab script that performs this calculation and plots the PSD. Comparing the shape of this PSD to that found in Example 3 shows their similarity. The difference is that this PSD is continuous since this random process contains every possible duty cycle square wave. Thus Example 5 demonstrated how to find the spectrum of a random binary waveform.

A.9. Finding the PSD of a Coded Binary Signal

This challenging topic was first covered by Cariolaro and Tronca. Their approach is to first model the communication system as a finite state machine, then analyze it using the theory of Markov Chains.

A.10. Finite State Machines

Any system which operates at discrete instants of time and takes on a finite number of configurations can be represented as a Finite State Machine (FSM). A general description of a FSM is a system that operates on a finite number of inputs and assumes a specific internal state based on those inputs. The output of the FSM can be determined in two ways.

1. **Moore machine**: The output is a function of the internal state only.
2. **Mealy machine**: The output is a function of the internal state and the present input.

A general Mealy FSM is shown in Figure A-27. In this model let S_n, C_n and L_n represent the input, output and internal state values of the system respectively. The system then operates in time intervals T, with a new state being generated every $t = nT$.

Therefore using this FSM to model a coded system, S_n would be the input SWs, C_n would be the output CWs, and L_n would represent the internal state of the encoder.

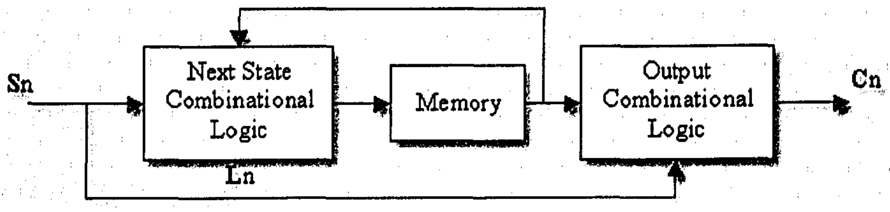


Figure A-27: Mealy Finite State Machine (FSM)

$$\begin{aligned} C_n &= f[L_n, S_n] \\ L_{n+1} &= g[L_n, S_n] \end{aligned} \tag{A.21}$$

A FSM can also be represented graphically by a **state diagram** which shows the progression of states through which the system operates and the resulting outputs based on specific inputs.

Example 6 – FSM encoder: Below is a typical ½ rate convolutional encoder with constraint length 2. The ½ comes from the fact that 1 source bit (SW) is encoded to 2 coded bits (CW).

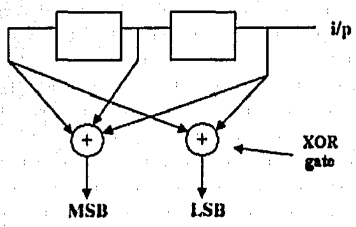


Figure A-28: Rate ½ convolutional encoder

For example, consider the encoded output corresponding to the following input sequence

10100

Details of the operation of the circuit are:

Clk	Regs	i/p	o/p
	00		
1	01	1	11
2	10	0	10
3	01	1	00
4	10	0	10
5	00	0	11

and the output sequence would be

1110001011

Modeling this convolutional encoder as a FSM allows creation of the state diagram as shown in Figure A-29. This is done by considering all possible states with the corresponding inputs and outputs.

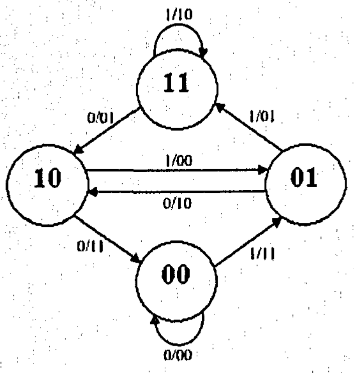


Figure A-29: FSM of the convolutional encoder of Example 6

Clearly the FSM model of a communications system is useful since once the FSM model is complete, the behavior of the system can be determined very easily. As well, Cariolaro and Tronca require modeling the coding system as a FSM and analyzing it as a Markov Chain in order to find the PSD.

A.11. Markov Chains

A random process $X(t)$ is a **Markov Process** if the future value (or state) of the process is dependent only on its immediate present value (or state). In other words, the process is independent of the past, and its future value is dependent only on its present value. Equation A.22 indicates that the probability of $X(t)$ assuming a new value given all the past values for all time is equal to the probability of $X(t)$ assuming a new value if only the immediate value x_k is considered.

$$P[X(t_{k+1})=x_{k+1} \mid X(t_k)=x_k,\ldots,X(t_1)=x_1]=P[X(t_{k+1})=x_{k+1} \mid X(t_k)=x_k] \quad \text{(A.22)}$$

An integer valued Markov Process is a discrete time random process that is called a **Markov Chain**. Here the random variable X_n takes on a countable number of values at discrete moments in time, where T is the interval between discrete time events. The value of X_n at the discrete time n is referred to as the **state of the process** at time n.

$$X_n = X(nT) \quad \text{(A.23)}$$

In Markov chains the **probability distribution functions** (PDFs) that are conditioned on several time instants always reduce to one PDF that is conditioned only on the most recent time instant. This is known as the **Markov property**. For this reason the value of X_n at the discrete time n is referred to as the **state of the process** at time n.

A.12. Markov Chain Properties

In general a system can be modeled by a Markov chain if the sequence of trials satisfy the following properties:

1. Each outcome belongs to a finite set of outcomes $\{a_1, a_2, \ldots, a_m\}$ called the **state space** of the system. If the outcome on the n^{th} trial is a_i, then the system is defined to be in state a_i at time n, or the system is in state a_i at the n^{th} step.
2. The outcome of any trial depends at most upon the outcome of the immediately preceding trial and not upon any other previous outcome.
3. There is a given probability p_{ij} that state a_j occurs immediately after the occurrence of a_i. Therefore if the system is currently in state a_i, then p_{ij} is the probability of moving to state a_j.

The numbers p_{ij} from requirement 3 are called the transition probabilities and they can be arranged in a matrix called the **transition matrix** as shown in Equation A.24.

$$P = \Pi = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix} \tag{A.24}$$

The transition matrix Π is a **stochastic matrix** since each row is a **probability vector** that sums to 1. In addition to the transition matrix a Markov chain has an **initial probability vector** p that indicates the starting state. This is best shown in an example.

Example 7 – Simple Markov Chain: After work a woman enjoys either working out or playing tennis. Since the workout can be quite tiring she never works out 2 days in a row. However if she plays tennis one afternoon, she is just as likely to play tennis the next afternoon as she is to work out. This can be modeled as a Markov chain since the outcome on any day depends only on what happened the preceding day. The transition matrix is shown in Figure A-30.

$$P = \begin{array}{c c} & \begin{array}{cc} W & T \end{array} \\ \begin{array}{c} W \\ T \end{array} & \begin{pmatrix} 0 & 1 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \end{array}$$

Figure A-30: Transition matrix for Example 7

In this figure, W denotes working out and T denotes playing tennis. The rows denote the activity on the current day, and the columns denote the activity on the next day. This matrix is then read in the following manner. If the woman worked out one afternoon she definitely plays tennis the next afternoon (i.e. she never works out 2 days in a row). If however she played tennis one afternoon, 50% of the time she will play tennis the next afternoon and 50% of the time she will work out the next afternoon. This can also be shown as a state diagram as shown in Figure A-31.

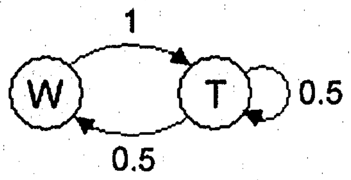


Figure A-31: State diagram for Example 7

Here it is very easy to see that if she works out one day she does not work out the next. However if she plays tennis one day she is just a likely to play tennis the next day as she is to work out.

Now consider this. What is the likelihood that she will play tennis two days from the current day? This is easy to solve. If she works out today she definitely plays tennis the next day (100%), and then two days later there is a 50% chance of playing tennis. If she plays tennis today she works out the next day 50% of the time and plays tennis 50% of the time. Two days from now if she had worked out she definitely plays tennis (100%), otherwise there is a 50% chance that she will play tennis. In terms of a probability tree there are two ways she can play tennis again and one way she will work out again in 2 days. So to summarize

If the woman is currently working out, two afternoons from now:

- the probability that she will work out again is 1/2.
- the probability that she will play tennis is 1/2.

If the woman is currently playing tennis, two afternoons from now:

- the probability that she will work out is 1/4.
- the probability that she will play tennis again is 3/4.

Notice that this can be done elegantly using matrix powers as shown in Figure A-32.

$$P = \begin{array}{c} \\ W \\ T \end{array} \begin{array}{c} \begin{array}{cc} W & T \end{array} \\ \begin{pmatrix} 0 & 1 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \end{array} \qquad P^2 = \begin{pmatrix} 0 & 1 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 0 & 1 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} = \begin{array}{c} \\ W \\ T \end{array} \begin{array}{c} \begin{array}{cc} W & T \end{array} \\ \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{4} & \frac{3}{4} \end{pmatrix} \end{array}$$

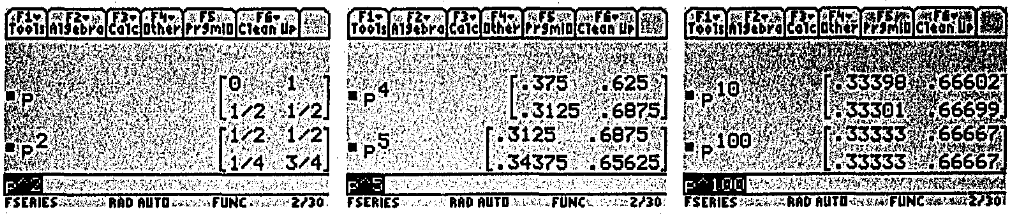
Figure A-32: Matrix powers can find the two step probability

The component p_{ij} in the transition matrix P of a Markov chain is the probability that the system changes from the state a_i to the state a_j in one step (i.e. one time instant). From the above example in one time instant we can move from either state 1 to 2, or from state 2 to 1, or stay in state 2. The probabilities in the transition matrix then show the **one-step probabilities.**

However we may want to know the probability of moving from state a_i to state a_j in two or more steps. This can be accomplished simply by matrix multiplication as shown in Equation A.25

$$P^2 = PP \qquad \text{or} \qquad \Pi^2 = \Pi\Pi \tag{A.25}$$

Based on this result, it is easy to evaluate the probability of this woman working out or playing tennis in four days, five days or even one hundred days using P^4, P^5, and P^{100} as shown in Figure A-33.



F1▾ Tools | F2▾ Algebra | F3▾ Calc | F4▾ Other | F5 PrgmIO | F6▾ Clean Up

```
■ P                [0     1  ]
                   [1/2   1/2]
■ P^2              [1/2   1/2]
                   [1/4   3/4]
P^2
FSERIES    RAD AUTO    FUNC    2/30
```

```
■ P^4              [.375    .625  ]
                   [.3125   .6875 ]
■ P^5              [.3125   .6875 ]
                   [.34375  .65625]
P^5
FSERIES    RAD AUTO    FUNC    2/30
```

```
■ P^10             [.33398  .66602]
                   [.33301  .66699]
■ P^100            [.33333  .66667]
                   [.33333  .66667]
P^100
FSERIES    RAD AUTO    FUNC    2/30
```

Figure A-33: 4-step, 5-step and 100-step probabilities for Example 7

The interesting thing to note in Figure A-33 is that these matrix powers are converging to a special matrix that has identical rows. After a certain number of steps (matrix powers), the probability of being in state W or T approaches a constant regardless of the starting state. This special matrix is called the **transition matrix** Π (sometimes called the T matrix) and the rows of this matrix are known as the **invariant probability vector** π (also sometime called λ_∞).

A.13. Partial Summary

It would be useful to summarize the most important concepts so far.

- Oscilloscopes and spectrum analyzers generally work in RMS and dB
- Communication systems must be analyzed as random processes
- Coding in systems adds an extra dimension of difficulty

- Cariolaro and Tronca developed a way to find the PSD of these systems, they require modeling the system as a FSM, and then analyzing the system using Markov Chains.

With these definitions the PSD of systems that use coding can be performed using the approach developed by Cariolaro and Tronca. Their approach is quite involved and two more examples will be explored to explain how their approach works.

A.14. PSD of Coded Systems – Simplest Example

Example 8 – 50% duty cycle square wave revisited: Evaluate the PSD of the same 50% duty cycle square wave from Example 2, with 2.9V peak and fundamental frequency of 100kHz.

While this is not really a coding system it can be considered a coding operation with the mapping $1 \rightarrow 10$ and $0 \rightarrow 10$. Hence this is a 1:2 mapping and each CW is 2-bits long. Secondly we can assume that the source outputs a 1 or a 0 bit (SW) with equal probability. The first step is to represent the system as a finite state machine and draw a state diagram. However in this example system there would only be one state no matter what the input symbols are and the output would always be 10 as shown in Figure A-34.

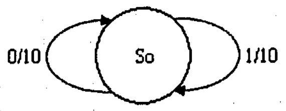


Figure A-34: Original state diagram of the example system

This would lead to a very trivial example and therefore the model used in this example will be modified to have two states as shown in Figure A-35. This does not change the functionality since the system is still always outputting a 50% duty cycle square wave.

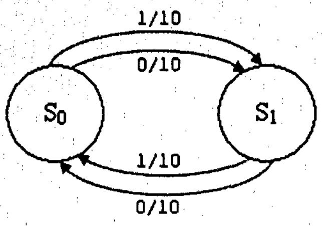


Figure A-35: Expanded state diagram for the example system

This model is adequate since we can consider:

- 1s and 0s are equally likely
- In state S_0 a 1 is mapped to a 10 and a 0 is mapped to a 10
- In state S_1 a 1 is mapped to a 10 and a 0 is mapped to a 10

A couple observations:

- Due to this coding procedure (mapping), the source statistics are irrelevant (on purpose). That is, even if the source outputs 1s constantly, the output sequence will always be 10101010 etc.
- Since both a 1 or a 0 forces a move to the other state, we can intuitively determine that 50% of the time is spent in each state.
- Since the output is always 10 repeatedly, if we had to determine the average output it would have to be 10.

The next step is to define the **input probability matrix** θ_u for $u = 1,2,...,S$ as shown in Equation A.26, where S is the number of SWs. In this example, $S = 2$ since the 1-bit SWs are either 0 or 1. Equation A.26 is showing the probability of a particular SW being chosen given the current state. It is written in diagonal form for mathematical convenience.

$$P(SW \mid l_i)_{diag} = \theta_u(i,j) = \begin{cases} P(SW \mid l_i), & i = j \\ 0 & otherwise \end{cases} \tag{A.26}$$

Also define E_u for $u = 1,2,...,S$ as the **next-state matrices**, where $E_u(i,j) = 1$ if and only if state l_j is entered from state l_i given input S_u. Thus for each possible SW S_u, the system will move to only one state so the E_u matrices will have a single 1 on each row. Finally define C_u for $u = 1,2,...,S$ as the **CW matrix**, where $C_u(i,j)$ is the CW generated when state l_j is entered due to a given input S_u. These are shown in Equations A.27 and A.28.

$$\theta_0 = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \quad E_0 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad C_0 = \begin{bmatrix} 10 \\ 10 \end{bmatrix} \tag{A.27}$$

$$\theta_1 = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \quad E_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad C_1 = \begin{bmatrix} 10 \\ 10 \end{bmatrix} \tag{A.28}$$

Therefore by considering all the ways in which these state transitions can occur and the probability of their occurrence, it is clear that a **transition probability matrix** can be constructed by Equation A.29.

$$P = \Pi = \sum_{u=1}^{S} \theta_u E_u \tag{A.29}$$

$$P = \Pi = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{A.30}$$

The transition probability matrix P in Equation A.30 indicates that if the system is currently in state 0, the probability of moving to state 1 is 100%. Likewise if the system is currently in state 1 the probability of moving to state 0 is 100%, as expected.

Once the transition probability matrix is found the invariant probability distribution can be found by taking P to high powers. However from **Markov Chain Theory** it is found that P is a non-regular, irreducible, periodic Markov Chain with d=2.

Thus the subsequent powers will oscillate between d different matrices. However like most things in probability, we can once again average over these matrices to find a stationary distribution. Therefore finding the invariant stationary distribution is

$$\frac{1}{n}\sum_{k=1}^{n}\Pi^k \rightarrow M \tag{A.31}$$

$$M=\frac{1}{2}\left(\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}+\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\right)=\begin{bmatrix}.5 & .5\\ .5 & .5\end{bmatrix} \quad \text{therefore} \quad \pi=[.5 \quad .5] \tag{A.32}$$

Thus vector π indicates, as expected, that this system spends 50% of the time in each state. As well the CW $c_u(i)$ will occur with probability $\pi(i)\theta_u(i,i)$ and thus the **average CW** or **mean symbol vector** can be found by summing over all CWs multiplied by their likelihood as shown in Equation A.33.

$$\eta_c=\pi\sum_{u=1}^{S}\theta_u c_u=[.5 \quad .5]\begin{bmatrix}.5 & 0\\ 0 & .5\end{bmatrix}\begin{bmatrix}20\\ 20\end{bmatrix}=[.5 \quad .5]\begin{bmatrix}10\\ 10\end{bmatrix}=[10] \tag{A.33}$$

As expected the **average CW** is 10. As already discussed this is the only CW so it must obviously be the average CW.

Now consider evaluation of the symbol sequence autocorrelation $R_{c,k}=E[c_n c_{n+k}]$. Consider first $R_{C,0}$ or evaluation of $E[c_n c_n]$. Since symbol $c_u(i)$ occurs with probability $\pi(i)\theta_u(i,i)$, the contribution of this symbol to $R_{C,0}$ is $c_u^2(i)\pi(i)\theta_u(i,i)$. Therefore to find the contribution from all CWs to $R_{C,0}$, Cariolaro and Tronca first define Λ as the L-square diagonal matrix such that

$$\Lambda(i,j) = \begin{cases} \pi(i) & i = j \\ 0 & \text{otherwise} \end{cases} \tag{A.34}$$

Then summing over all CWs gives

$$R_{C,0} = \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u c_u \tag{A.35}$$

Therefore when the input symbols are independent of state the autocorrelation is be shown to be

$$R_{C,k} = \begin{cases} \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u c_u & k = 0 \\ \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u E_u \Pi^{k-1} \sum_{v=1}^{S} \theta_v c_v & k \geq 1 \\ R_{C,-k}^{\mathrm{T}} & k \leq -1 \end{cases} \tag{A.36}$$

Continuing with the example above, there are two SWs so the autocorrelation at zero time separation $R_{C,0}$ is found by summing up $c_0^{\mathrm{T}} \Lambda \theta_0 c_0$ + $c_1^{\mathrm{T}} \Lambda \theta_1 c_1$.

$$\begin{gathered} c_0^{\mathrm{T}} \Lambda \theta_0 c_0 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix} \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} .5 & 0 \\ 0 & 0 \end{bmatrix} \\ c_1^{\mathrm{T}} \Lambda \theta_1 c_1 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix} \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} .5 & 0 \\ 0 & 0 \end{bmatrix} \\ R_{C,0} = c_0^{\mathrm{T}} \Lambda \theta_0 c_0 + c_1^{\mathrm{T}} \Lambda \theta_1 c_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \end{gathered} \tag{A.37}$$

Equation A.37 depicts the correlation of all the CWs. In order to continue this evaluation for the first few time separations it is important to notice that in Equation A.36 the second summation term does not change with k, and thus it can be computed separately as shown in Equation A.38.

$$\sum_{v=1}^{S}\theta_v c_v = \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = 2\begin{bmatrix} .5 & 0 \\ .5 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \tag{A.38}$$

Thus $R_{C,1}$ is found to be the same as $R_{C,0}$ as shown in Equation A.39.

$$\begin{aligned}
R_{C,1} &= \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u E_u \Pi^{k-1} \sum_{v=1}^{S} \theta_v c_v \\
R_{C,1} &= \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u E_u \Pi^{1-1} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \\
R_{C,1} &= \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u E_u \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \\
R_{C,1} &= \sum_{u=1}^{S} c_u^{\mathrm{T}} \Lambda \theta_u E_u \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = 2\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \\
R_{C,1} &= 2\begin{bmatrix} .5 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}
\end{aligned} \tag{A.39}$$

Continuing with this approach it is found that

$$R_{C,0} = R_{C,1} = R_{C,2} = R_{C,3} = \ldots = R_{C,\infty} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{A.40}$$

where

$$R_{C,\infty} = \eta_c^{\mathrm{T}} \eta_c \tag{A.41}$$

which is the transpose of the mean symbol vector multiplied by itself. Finally the autocovariance is found as

$$\begin{aligned} K_{c,k} &= R_{c,k} - \eta_c^2 \\ K_{c,k} &= R_{c,k} - R_{c,\infty} \end{aligned} \tag{A.42}$$

This means that if the mean symbol vector is zero, the autocovariance matrices are equal to the autocorrelation matrices. As discussed previously, the mean symbol vector will be zero if the output waveforms are balanced, i.e. if 0 is mapped to –1 and 1 is mapped to +1.

Thus $K_{C,0}$ is found by $R_{C,0} - R_{C,\infty}$ as $K_{C,1} = R_{C,1} - R_{C,\infty}$ and $K_{C,1} = R_{C,1} - R_{C,\infty}$ and thus as k tends to infinity the all zero matrix results as shown below.

$$K_{c,k} = R_{c,k} - R_{c,\infty} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad as \quad k \to \infty \tag{A.43}$$

In the example above $K_{C,k}$ is all zero for all k. This makes sense since the output never varies. In other words the output is periodic and in the frequency domain there is no continuous component. Finally the PSD is found by

$$W(f) = \frac{|P(f)|^2}{NT}\left(X_C(f) + \frac{X_D(f)}{NT} \sum_{m=-\infty}^{\infty} \delta\left(f - \frac{mf_0}{N} \right)\right) \tag{A.44}$$

where

$$X_C(f) = v\left[K_{C,0} + \sum_{k=-\infty}^{\infty} K_{C,k} e^{-j2\pi fkNT} \right] v^* \qquad X_D(f) = vR_{C,\infty}v^* \tag{A.45}$$

$$v = \begin{bmatrix} 1 & e^{j2\pi fT} & e^{j4\pi fT} & e^{j6\pi fT} & \dots & e^{j2\pi f(N-1)T} \end{bmatrix} \qquad \text{(A.46)}$$

In this example since the covariance matrices are all zero there will be no continuous component and $X_C(f) = 0$ and therefore

$$W(f) = \frac{|P(f)|^2}{NT}\left(0 + \frac{X_D(f)}{NT}\sum_{m=-\infty}^{\infty}\delta\left(f - \frac{mf_0}{N}\right)\right) \qquad \text{(A.47)}$$

and the discrete PSD component is found to be

$$X_D(f) = \begin{bmatrix} 1 & e^{j2\pi fT} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ e^{-j2\pi fT} \end{bmatrix} = [1] \quad \text{where} \quad v = \begin{bmatrix} 1 & e^{j2\pi fT} \end{bmatrix} \qquad \text{(A.48)}$$

Therefore the overall PSD reduces to

$$W(f) = \frac{|P(f)|^2}{NT}\left(\frac{1}{NT}\sum_{m=-\infty}^{\infty}\delta\left(f - \frac{mf_0}{N}\right)\right) \qquad \text{(A.49)}$$

Equation A.49 indicates that the PSD of this signal is a train of delta functions weighted by the Fourier Transform of the pulse shape squared. Since the pulse shape is a square wave we can find the Fourier transform from tables or by Equation A.50.

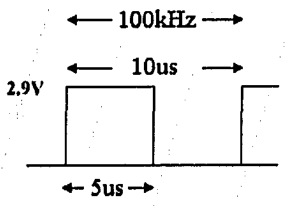


Figure A-36: Square pulse shape used in Example 8

$$\begin{aligned}
&Arect\left(\frac{t}{\tau}\right) \leftrightarrow A\tau \mathrm{sinc}\left(\frac{\omega\tau}{2}\right)\\
&2.9rect\left(\frac{t}{5\mu s}\right) \leftrightarrow 2.9(5\mu s)\mathrm{sinc}\left(\frac{2\pi f 5\mu s}{2}\right)\\
&P(f) = (14.5\mu s)\mathrm{sinc}(\pi f 5\mu s)
\end{aligned} \tag{A.50}$$

With this pulse shape define the *weight* value as

$$weight = \frac{|P(f)|^2}{NT} \quad \text{from} \quad W(f) = \frac{|P(f)|^2}{NT}\left(\frac{1}{NT}\sum_{m=-\infty}^{\infty}\delta\left(f - \frac{mf_0}{N}\right)\right) \tag{A.51}$$

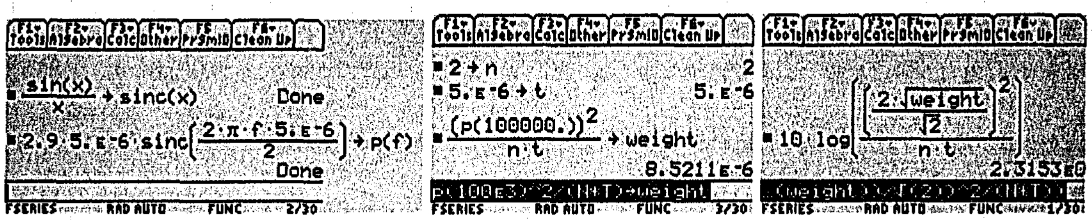
Since it is a magnitude function squared it is now in the form of power dissipated over 1 ohm. However to match the oscilloscope we want it to be in dB and RMS. Therefore we need to take the square root $\sqrt{weight}$.

Furthermore Cariolaro and Tronca define this PSD as a two sided spectrum, i.e. the Fourier Transform of the pulse shape is related to the complex exponential Fourier Series. Therefore to match a one sided spectrum the $\sqrt{weight}$ needs to be multiplied by two as indicated by Equation A.8 i.e. $2\sqrt{weight}$. Therefore the final form to match the digital oscilloscope and spectrum analyzer is

$$W_{V_{RMS}}(f) = 10\log\left(\left(\frac{2\sqrt{weight}}{\sqrt{2}}\right)^2\left(\frac{1}{NT}\sum_{m=-\infty}^{\infty}\delta\left(f - \frac{mf_0}{N}\right)\right)\right) \tag{A.52}$$

$$W_{dBm}(f) = 10\log\left(\frac{\left(\frac{2\sqrt{weight}}{\sqrt{2}}\right)^2\left(\frac{1}{NT}\sum_{m=-\infty}^{\infty}\delta\left(f - \frac{mf_0}{N}\right)\right)}{50(.001)}\right) \tag{A.53}$$

Note: Equations A.52 and A.53 are general forms for this example only. Some values for dBV_{RMS} are given in the following figures.



$\frac{\sin(x)}{x} \rightarrow \mathrm{sinc}(x)$ Done

$2.9 \cdot 5.E{-6} \cdot \mathrm{sinc}\left(\frac{2 \cdot \pi \cdot f \cdot 5.E{-6}}{2}\right) \rightarrow p(f)$ Done

$2 \rightarrow n$ 2

$5.E{-6} \rightarrow t$ 5.E-6

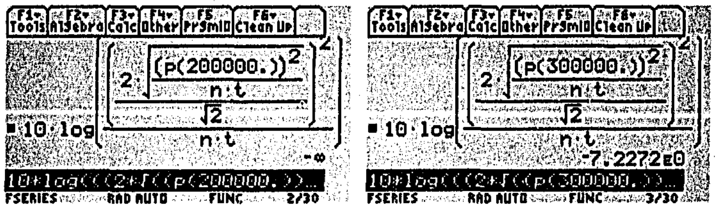
$\frac{(p(100000.))^2}{n \cdot t} \rightarrow weight$ 8.5211E-6

```
p(100E3)^2/(N*T)→weight
```

$10 \cdot \log\left(\frac{\left(\frac{2 \cdot \sqrt{weight}}{\sqrt{2}}\right)^2}{n \cdot t}\right)$ 2.3153E0

```
(weight))/√(2))^2/(N*T))
```

Figure A-37: Stores the pulse shape *P(f)* and weight, finds the dBV$_{RMS}$ value at 100kHz



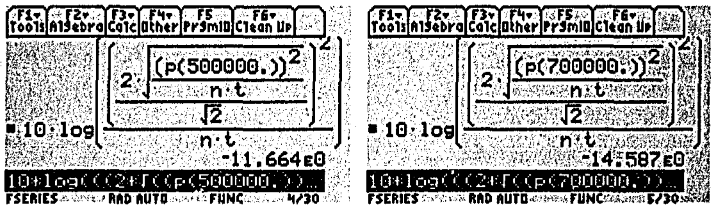
$10 \cdot \log\left(\frac{\left(\frac{2 \cdot \sqrt{\frac{(p(200000.))^2}{n \cdot t}}}{\sqrt{2}}\right)^2}{n \cdot t}\right)$ -∞

```
10*log(((2*√((p(200000.))...
```

$10 \cdot \log\left(\frac{\left(\frac{2 \cdot \sqrt{\frac{(p(300000.))^2}{n \cdot t}}}{\sqrt{2}}\right)^2}{n \cdot t}\right)$ -7.2272E0

```
10*log(((2*√((p(300000.))...
```

Figure A-38: Find dBV$_{RMS}$ values at 200kHz and 300kHz



$10 \cdot \log\left(\frac{\left(\frac{2 \cdot \sqrt{\frac{(p(500000.))^2}{n \cdot t}}}{\sqrt{2}}\right)^2}{n \cdot t}\right)$ -11.664E0

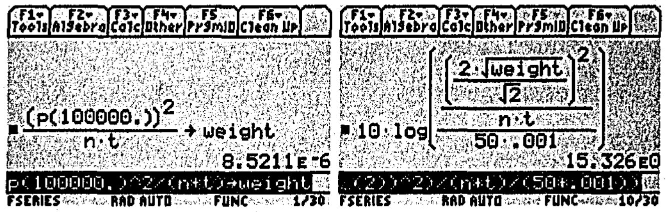
```
10*log(((2*√((p(500000.))...
```

$10 \cdot \log\left(\frac{\left(\frac{2 \cdot \sqrt{\frac{(p(700000.))^2}{n \cdot t}}}{\sqrt{2}}\right)^2}{n \cdot t}\right)$ -14.587E0

```
10*log(((2*√((p(700000.))...
```

Figure A-39: Find dBV$_{RMS}$ values at 500kHz and 700kHz

Continuing in this fashion for dBm:



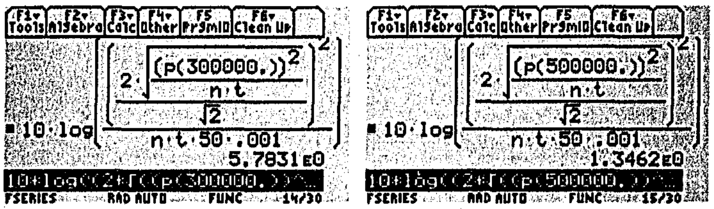
$\frac{(p(100000.))^2}{n \cdot t} \rightarrow weight$ 8.5211E-6

```
p(100000.)^2/(n*t)→weight
```

$10 \cdot \log\left(\frac{\frac{\left(\frac{2 \cdot \sqrt{weight}}{\sqrt{2}}\right)^2}{n \cdot t}}{50 \cdot .001}\right)$ 15.326E0

```
(2))^2)/(n*t)/(50*.001))
```

Figure A-40: Finding the dBm value at 100kHz



$10 \cdot \log\left(\frac{\left(\frac{2 \cdot \sqrt{\frac{(p(300000.))^2}{n \cdot t}}}{\sqrt{2}}\right)^2}{n \cdot t \cdot 50 \cdot .001}\right)$ 5.7831E0

```
10*log((2*√((p(300000.))^...
```

$10 \cdot \log\left(\frac{\left(\frac{2 \cdot \sqrt{\frac{(p(500000.))^2}{n \cdot t}}}{\sqrt{2}}\right)^2}{n \cdot t \cdot 50 \cdot .001}\right)$ 1.3462E0

```
10*log((2*√((p(500000.))^...
```

Figure A-41: Finding the dBm values at 300kHz and 500kHz

Comparing the dBV_{RMS} results in Figures A-37, A-38 and A-39 to Table A-1, and the dBm results in Figures A-40 and A-41 to Table A-2 demonstrates the accuracy of this method.

While this example is much more complicated than Example 2, it does show the generality of the Cariolaro and Tronca method of finding PSDs. Their approach however is intended for more complicated systems as the next example will show.

A.15. PSD of Coded Systems – Real World Example

Example 9 – Random Binary Signal: From Example 5 consider a $\pm A$ random square wave with pulse width 5us, modeled as a FSM as shown in Figure A-42.

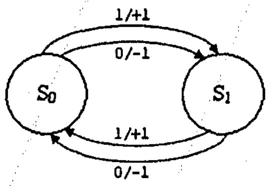


Figure A-42: State diagram of the random binary signal

The first thing to notice here is that this output will be truly random. Therefore there will definitely be a continuous component in the PSD. As well since the output is balanced, i.e. $\pm A$, there will be no discrete component in the PSD. Like Example 8, the first things to define are the θ_u, E_u and C_u matrices as shown in Equations A.54 and A.55.

$$\theta_0 = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \qquad E_0 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad C_0 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \tag{A.54}$$

$$\theta_1 = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \qquad E_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad C_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{A.55}$$

By considering all the ways these state transitions can occur and the probability of their occurrence it is clear that a transition probability matrix will be the same as in Example 8.

$$\Pi = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{A.56}$$

$$\frac{1}{n}\sum_{k=1}^{n}\Pi^k \to M$$

$$M = \frac{1}{2}\left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \begin{bmatrix} .5 & .5 \\ .5 & .5 \end{bmatrix} \quad \text{therefore} \quad \pi = [.5 \quad .5] \tag{A.57}$$

As expected this system spends 50% of the time in each state. As well the CW $c_u(i)$ will occur with probability $\pi(i)\theta_u(i,i)$ and thus the average CW or mean symbol vector can be found by summing over all the 1-bit CWs.

$$\eta_c = \pi\sum_{u=1}^{S}\theta_u c_u = [.5 \quad .5]\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \end{bmatrix} + [.5 \quad .5]\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = [0] \tag{A.58}$$

As expected the average CW is 0 since 50% of the time the system outputs a -1 and the other 50% of the time the system outputs a +1.

Furthermore, since there are two SWs the autocorrelation at zero time separation $R_{c,0}$ is found by summing up $c_0^T\Lambda\theta_0 c_0$ + $c_1^T\Lambda\theta_1 c_1$ as shown in Equation A.59.

$$c_0^{\mathrm{T}}\Lambda\theta_0 c_0 = [-1 \quad -1]\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \end{bmatrix} = [.5]$$

$$c_1^{\mathrm{T}}\Lambda\theta_1 c_1 = [1 \quad 1]\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = [.5] \tag{A.59}$$

$$R_{C,0} = c_0^{\mathrm{T}}\Lambda\theta_0 c_0 + c_1^{\mathrm{T}}\Lambda\theta_1 c_1 = [1]$$

Equation A.59 depicts the correlation of all the CWs at zero time separation. Like Example 8 the second summation term does not change with k so it can be computed separately.

$$\sum_{v=1}^{S}\theta_v c_v = \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \end{bmatrix} + \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{A.60}$$

Thus $R_{c,1}$ is found to be

$$R_{C,1} = \sum_{u=1}^{S} c_u^{\mathrm{T}}\Lambda\theta_u E_u \Pi^{k-1}\sum_{v=1}^{S}\theta_v c_v$$

$$R_{C,1} = \sum_{u=1}^{S} c_u^{\mathrm{T}}\Lambda\theta_u E_u \Pi^{1-1}\begin{bmatrix} 0 \\ 0 \end{bmatrix} = [0] \tag{A.61}$$

Continuing with this approach shows that

$$R_{C,1} = R_{C,2} = R_{C,3} = R_{C,\infty} = [0] \tag{A.62}$$

Finally the autocovariance is found as

$$K_{C,0} = R_{C,0} - R_{C,\infty} = [1] - [0] = [1]$$

$$K_{C,1} = K_{C,2} = K_{C,\infty} = R_{C,1} - R_{C,\infty} = [0] - [0] = [0] \tag{A.63}$$

This shows that there will be a continuous component. Noting that the only covariance matrix that is non-zero is $K_{C,0}$, the result is

$$\begin{aligned} X_C(f) &= v\left[K_{C,0} + \sum_{k=-\infty}^{\infty} K_{C,k} e^{-j2\pi fkNT}\right] v^* \\ X_C(f) &= vK_{C,0}v^* \\ X_C(f) &= \begin{bmatrix} 1 & e^{j2\pi fT} \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 1 \\ e^{-j2\pi fT} \end{bmatrix} = 2 \end{aligned} \tag{A.64}$$

Therefore the PSD in this case reduces to

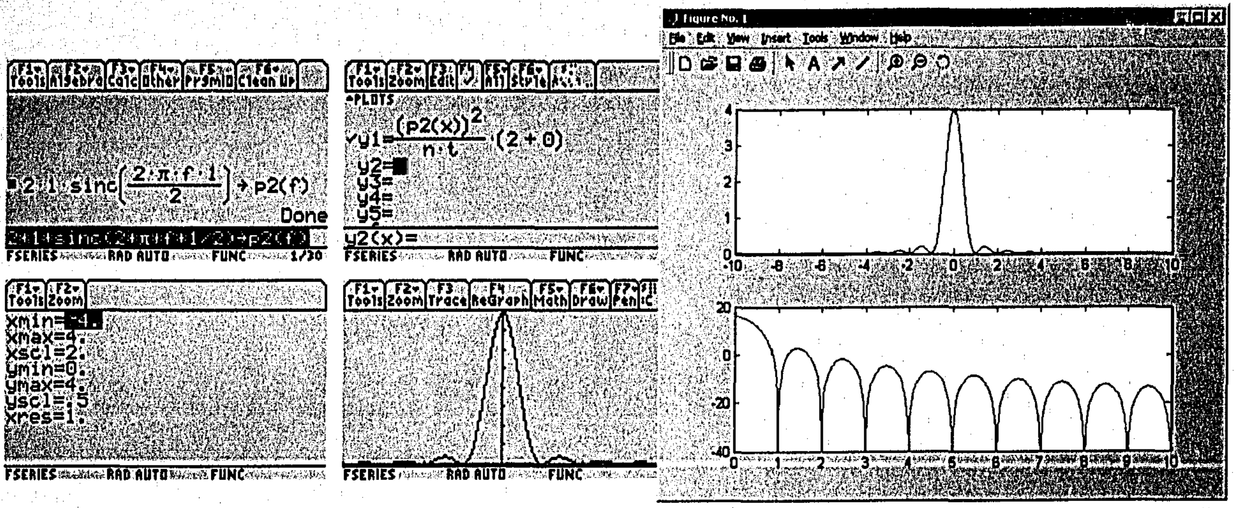
$$W(f) = \frac{|P(f)|^2}{NT}(2+0) \tag{A.65}$$

Like Example 8, since the pulse shape is a square wave, from Fourier Tables we find that

$$\begin{aligned} &A\mathit{rect}\left(\frac{t}{\tau}\right) \leftrightarrow A\tau \mathrm{sinc}\left(\frac{\omega\tau}{2}\right) \\ &2\mathit{rect}\left(\frac{t}{1}\right) \leftrightarrow 2\mathrm{sinc}\left(\frac{2\pi f}{2}\right) \\ &P(f) = 2\mathrm{sinc}(\pi f) \end{aligned} \tag{A.66}$$

The final expression for the PSD in this example is then

$$W(f) = \frac{8\mathrm{sinc}^2(\pi f)}{2T} \tag{A.67}$$



```
■2·1·sinc(2·π·f·1/2)→p2(f)
                          Done
2*1*sinc(2*π*f*1/2)→p2(f)
```

```
y1=(p2(x))^2/(n·t)·(2+0)
y2=
y3=
y4=
y5=
y2(x)=
```

```
xmin=-4.
xmax=4.
xscl=2.
ymin=0.
ymax=4.
yscl=.5
xres=1.
```

Tools Algebra Calc Other PrgmIO Clean Up
Tools Zoom Edit Style
PLOTS
FSERIES RAD AUTO FUNC 1/30
FSERIES RAD AUTO FUNC
Tools Zoom
Tools Zoom Trace ReGraph Math Draw Pen
Figure No. 1
File Edit View Insert Tools Window Help
4
3
2
1
0
-10 -8 -6 -4 -2 0 2 4 6 8 10
20
0
-20
-40
0 1 2 3 4 5 6 7 8 9 10

Figure A-43: Ti-89 and Matlab plotting the spectrum

Inspection of the PSD from Example 5 shows the identical result. Therefore this approach by Cariolaro and Tronca can be applied to very complex coding systems to obtain the PSD.

Appendix B – AddSWs, AddCWs and Codes

This Appendix lists all possible AddSWs that can be used with each Hamming code considered in this thesis. Encoding to AddCWs is accomplished as outlined in Section 2.14 using the generator polynomial $g(x)$ = 1011 = Bh for (7,x) codes, the generator polynomial $g(x)$ = 10011 = 13h for (15,x) codes, and the generator polynomial $g(x)$ = 101111 = 2Fh for (31,x) codes. They are presented in the format introduced by Figures 3-18 and 3-22. The number of AddSW/AddCW sets is defined in Equation 3.1.

B.1. (7,x) codes

Table B-1: (7,3) AddSWs – $(2^3)^0$ = 1 set

00...00	11...11
0	F

Table B-2: (7,2) AddSWs – $(2^2)^1$ = 4 sets

00...00	a	$\overline{a}$	11...11
0	4	B	F
0	5	A	F
0	6	9	F
0	7	8	F

Table B-3: (7,1) AddSWs – $(2^1)^3$ = 8 sets

00...00	a	b	c	$\overline{c}$	$\overline{b}$	$\overline{a}$	11...11
0	2	4	6	9	B	D	F
0	2	4	7	8	B	D	F
0	2	5	6	9	A	D	F
0	2	5	7	8	A	D	F
0	3	4	6	9	B	C	F
0	3	4	7	8	B	C	F
0	3	5	6	9	A	C	F

B.2. (15,x) codes

Table B-4: (15,10) AddSWs – $(2^{10})^0$ = 1 set

00...00	11...11
0	7FF

Table B-5: (15,9) AddSWs – $(2^9)^1$ = 512 sets

00...00	a	$\overline{a}$	11...11
0	200	5FF	7FF
0	201	5FE	7FF
0	202	5FD	7FF
0	...	...	7FF
0	...	...	7FF
0	3FE	401	7FF
0	3FF	400	7FF

Table B-6: (15,8) AddSWs – $(2^8)^3$ = 16777216 sets

00...00	a	b	c	$\overline{c}$	$\overline{b}$	$\overline{a}$	11...11
0	100	200	300	4FF	5FF	6FF	7FF
0	100	200	301	4FE	5FF	6FF	7FF
0	100	200	302	4FD	5FF	6FF	7FF
0	...	...	...	...	...	...	7FF
0	100	201	300	4FF	5FE	6FF	7FF
0	100	201	301	4FE	5FE	6FF	7FF
0	100	201	302	4FD	5FE	6FF	7FF
0	...	...	...	...	...	...	7FF
0	101	200	300	4FF	5FF	6FE	7FF
0	101	200	301	4FE	5FF	6FE	7FF
0	101	200	302	4FD	5FF	6FE	7FF
0	...	...	...	...	...	...	7FF
0	1FF	2FF	3FF	400	500	600	7FF

B.3. (31,x) codes

Table B-7: (31,25) AddSWs – $(2^{25})^0$ = 1 set

00...00	11...11
0	3FFFFFF

Table B-8: (31,24) AddSWs – $(2^{24})^1$ = 16777216 sets

00...00	a	$\overline{a}$	11...11
0	1000000	2FFFFFF	3FFFFFF
0	1000001	2FFFFFE	3FFFFFF
0	...	...	3FFFFFF
0	...	...	3FFFFFF
0	1FFFFFE	2000001	3FFFFFF
0	1FFFFFF	2000000	3FFFFFF

Table B-9: (31,23) AddSWs – $(2^{23})^3$ = 590295810358705651712 sets

00...00	a	b	c	$\overline{c}$	$\overline{b}$	$\overline{a}$	11...11
0	800000	1000000	1800000	27FFFFF	2FFFFFF	37FFFFF	3FFFFFF
0	800000	1000000	1800001	27FFFFE	2FFFFFF	37FFFFF	3FFFFFF
0	800000	1000000	1800002	27FFFFD	2FFFFFF	37FFFFF	3FFFFFF
0	...	...	...	...	...	...	3FFFFFF
0	...	...	...	...	...	...	3FFFFFF
0	FFFFFF	17FFFFF	1FFFFFF	2000000	2800000	3000000	3FFFFFF