TIME DELAYED ERROR DETECTION CODES

by

Charles L. Rahm, III

Thesis submitted to the Graduate Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in
Electrical Engineering

APPROVED:

R. A. Thompson, Chairman

F. G. Gray

E. A. Manus

November, 1976
Blacksburg, Virginia

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to Dr. Richard A. Thompson for his time and contributions (especially spelling corrections) he made to this thesis, and to and for helping to type this work.

TABLE OF CONTENTS

Title Page i
Acknowledgements ii
Table of Contents iii

1. Introduction

1.1 Opening 1
1.2 Previous Work 1
1.3 Definitions 3
1.4 General Problem 7

2. Nonexhaustive Codes

2.1 Determining the Decoder Properties . 15
2.2 Average Code Length Versus Error Detection Probability 20
2.3 Alphabet and Code Characteristics . . 21

3. Nonsynchronizing Codes

3.1 General 30
3.2 Even Codes 31
3.3 Higher Error Detecting Codes 35

4. Implementation And Hardware

4.1 Encoder-Decoder 47
4.2 Phase Check 49

5. Conclusion

5.1 The Use of Time Delayed Error Detection 52
5.2 Open Areas and Needed Future Work . . 52

Quotations 54
References 55
Vita 56

1. INTRODUCTION

1.1 Opening

A great deal has been done in the area of error detection of binary codes. A simple example is the even parity code. All common error detection schemes detect the code word (or block) in which the error has occurred. The error is detected where it occurs and when it occurs. With Time Delayed Error Detection, TDED, the error is detected at some time (position) after the error occurs. The time delay provides disadvantages, but the binary code using TDED can be made with greater code efficiency (lower average code lenth, acl) than a more standard error detection code for the same amount of error detection capability. The disadvantages of TDED include (but are not limited too) decoder complexity and the difficulty of creating a code which detects errors efficiently. The small amount of previous effort in this area as seen in the next section can probably be blamed on these two, above disadvantages.

1.2 Previous Work

Time Delayed Error Detection was never mentioned in previous work. However, non-exhaustive codes were discussed

in two previous papers [3 and 4]. Nonexhaustive codes (to be defined later) are a very important part of TDED. In both papers very little was said but they do show the early problems with error detection using nonexhaustive codes.

> "The above remarks apply strictly to exhaustive encoding, but may not apply where there are certain sequences of digits which can never occur. For if such a sequence of digits does occur, this may be used by the circuit as a special indication that it is out of phase, and hence it may be possible to build auxiliary circuits which can cause resynchronization, even when a fixed length encoding is used. So a more complete treatment of synchronization would allow such auxiliary circuits, but here we will consider only self-synchronization, which is carried out inherently by the same means as is used for deciphering." [1]

> "It is alternatively possible to add redundancy by restricting the sequences of inputs to the sequential machine generating the code. Although this approach requires no framing if the resulting code is error limiting, it appears in general to be less fruitful than the above. It is difficult, for example, to obtain error-limiting codes in which all single errors are detected without greatly increasing the average code-word length." [2]

The first reference [3] describes resynchronization by detecting whether the circuit is out of phase and points out the problem of auxiliary circuits. This could just as easily be error detection because if the decoder is out of phase it means an error has occurred. The second reference [4] deals

with self synchronizing codes which go out-of-phase but points out that it is difficult to formulate the codes. This problem is dealt with later.

1.3 Definitions

Before going any futher some terms need to be defined or explained. Figure 1.31 shows a typical encoder-decoder configuration. The encoder and decoder are finite state machines described by the following.

Encoder, M = <A,C,Q,F,d,w,go>

A = the set of input symbols or input alphabet

C = the set of binary code words (which are output to the decoder) corresponding to the input alphabet characters, $C \subseteq \{0+1\}^*$

Q = the set of all states in the encoder

F = the set of all final (completion of character) states in the encoder, $F \subseteq Q$

d, w are functions such that:

$d: Q \times A \rightarrow Q$
$d: F \times A \rightarrow F$
$w: F \times A \rightarrow C$
$w: \overline{F} \times A \rightarrow \overline{C}$
$w: Q \times A \rightarrow B^* = \{0,1\}^*$

qo = the starting state of the encoder, $qo \in Q$

Decoder = M = <A,C,Q,F,d,w,go>

A = the set of output symbols of the decoder which is the input alphabet of the encoder

C = the set of valid input binary code words representing characters of A from the encoder

Q = the set of all states in the decoder

F = the set of all final (that is completion of a binary encoded symbol and output of an alphabet character) states in the decoder, $F \subseteq Q$

d, w are functions such that:

$$d: Q \times \{0,1\} \dashrightarrow Q$$
$$d: F \times C^{*} \dashrightarrow F$$
$$d: F \times \overline{C^{*}} \dashrightarrow \overline{F}$$
$$w: F \times C \dashrightarrow A$$
$$w: F \times \overline{C} \dashrightarrow \emptyset$$

qo = decoder starting state, $qo \in Q$

The encoder, simply stated, takes the input alphabet, A, and outputs binary code in the form of a series of binary sequences one code word for each input character) found in set C. the set C, called a code, becomes very important in coding theory. The properties of C determine how well the decoder can detect or correct errors as well as the decoder complexity. In this paper the set C is assumed to be a nonsingular (one and only one code word for each character), uniquely decodable, instantaneous code (w:FxC-->A) as described in reference [1]. The elements of C are called code words and the distance (hamming distance) between two words of the same length is the number of places in which the two code words differ. The distance between code words becomes important when the code is subjected to errors while traveling through a channel (as shown in figure 1.31). A channel is defined here as any place between the encoder and

the decoder. Errors will be assumed to occur only in the channel. Errors will also be limited to amplitude errors in which a 1 is changed into a 0 or a 0 into a 1. A classical example is the Binary Symmetric Channel. Phase errors (the loss or addition of a 1 or 0) are not considered, nor are transpositions (a common typing error).

The alphabet, A, is considered to have a set of probabilities corresponding to the probability of receiving each character. These probabilities are assumed to be of zero order (not depending on previous inputs) for most of the examples. However nth order statistics can be handled by repeating the procedure for each markov state. From the character probabilities an optimal huffman code can be found as shown in most coding theory texts [1]. An optimal code is a code which has a minimum value of average code length, acl. The acl is determined by taking the length of a code word times the probability of the character which it represents and then summing for all the code words of a particular code. Figure 1.32 shows an optimal huffman code on the alphabet, $A = \{a,b,c,d\}$ where the probability of every character is equal. From the probabilities of the alphabet characters and their code words, a state transition graph can be drawn. The probability of being in any particular state of the graph is called the state distribution probability. The probability, given that you are in a particular state, of going to a particular state is called

the state transition probability.

The codes which are used for TDED are nonexhaustive codes. An exhaustive code is defined as a code which has any binary sequence as a valid prefix to a valid chain of code words. The TDED codes may have the effect of synchronizing, going in-phase, after an error has occurred. A decoder is synchronized (in-phase) if it is in a final state, (q) F, when it recieves the output from the encoder which put the encoder in a final state. Lemma 1.3 further clarifies the definition of synchronized.

LEMMA 1.3:

Let i be a complete sequence, that is, it starts and finishes when the encoder has finished a character code word output. Let a decoder receive such an input sequence i and, further, let the decoder be synchronized (in-phase) upon the completion of i. Then i must have been a valid code word ($i \in C$).

Proof:

By the property $d{:}F\text{x}\overline{C^{*}}\text{-->}\overline{F}$ and $d{:}F\text{x}C^{*}\text{-->}F$ (which means the code is uniquely decipherable) if $i \notin C$, then $d((q),i)=(q1)$, $(q1) \notin F$. So, the decoder is not in phase because it should be in a state (q), where $(q) \in F$.

1.4 Error Detection Using Nonexhaustive Codes.

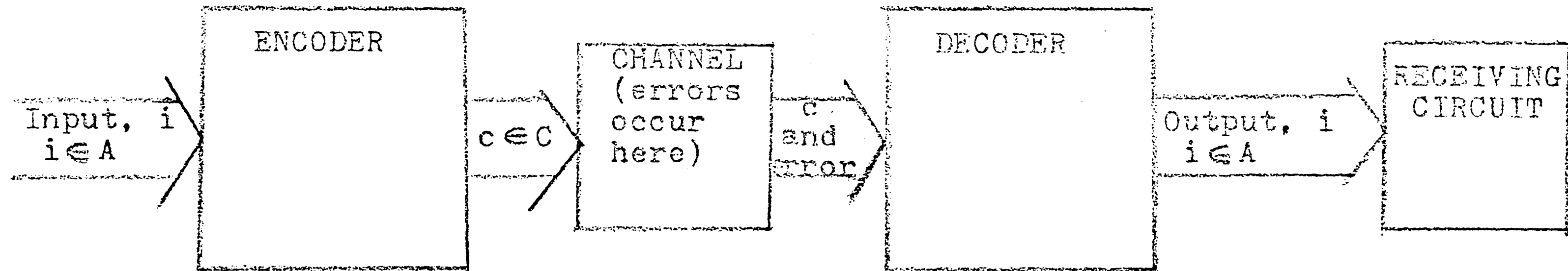
All error detection codes are nonexhaustive codes. An exhaustive code, by definition, cannot detect errors because every possible sequence of binary digits is the beginning of some valid string of characters of the alphabet, A. A simple even parity bit code is a common example of a standard nonexhaustive code shown in figure 1.41 which detects single errors on the alphabet, $A=\{a,b,c,d\}$. This code detects single errors because the code distance between any two code words is two. Thus one error in any code word produces a binary sequence which is not a valid code word. This invalid code word is then detected in the decoder whenever its state is (E) and so the even parity bit code detects all single errors. If we now change the above code to the code found in figure 1.42, called code Y, we find that we cannot immediately tell the distance between code words or the probability of detecting an error.

Looking at the Y code, we can see that only if you receive a 0 input while in state (01) do you enter (E), the error detected state. A 010 input is thus required to enter state (E) starting from state (S), the starting and finishing state ((S)=qo, (S)$\in$F). If we assume that an error has occurred in (S) with an input of 11 (which corresponds to the code word for d), we find that instead of going (S) to (1) to (S) the decoder goes (S) to (0) to (01). At this

point the decoder is said to be out-of-phase or nonsynchronized. The decoder should be in state (S) but because of the error the decoder is in state (01). If the next code word sequence is either a 00 (the code for a), or 011 (the code for b), or if another error occurs and a 0 is received, the decoder will go to (E) and thus the error has been detected (error-detected-phase). If a 11 (the code for d) is received, the decoder goes to (1) and if a 10 (the code for c), the decoder goes to state (0). in both cases the decoder is still nonsynchronized because it should be in (S). Now if a 011 is received while in (1) Or a 11 while in (0) the decoder will now go to (S) and thus the decoder has synchronized. When the decoder synchronizes there is no way to detect the original error.

The three conditions or phases of a decoder after an error has occurred were demonstrated above. The decoder is either in-phase (synchronized), out-of-phase (not synchronized), or error-detected-phase (in the error state). It is desirable for the decoder not to become in phase after an error has occurred because once in-phase the decoder will never enter the error detected state and will never detect the error. The code of figure 1.43, called an even code, has the property that for single errors the decoder will never become in-phase. This property also gives the code the ability to detect any single errors with time delay. Just how much time delay is discussed later along with proofs of

the properties of even codes.



Encoder and Decoder configuration
Figure 1.31

A= a,b,c,d

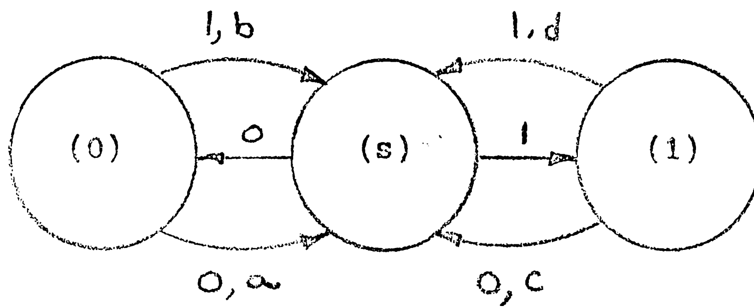
C= 00,01,10,11

Q= (s),(0),(1)

F= (s)

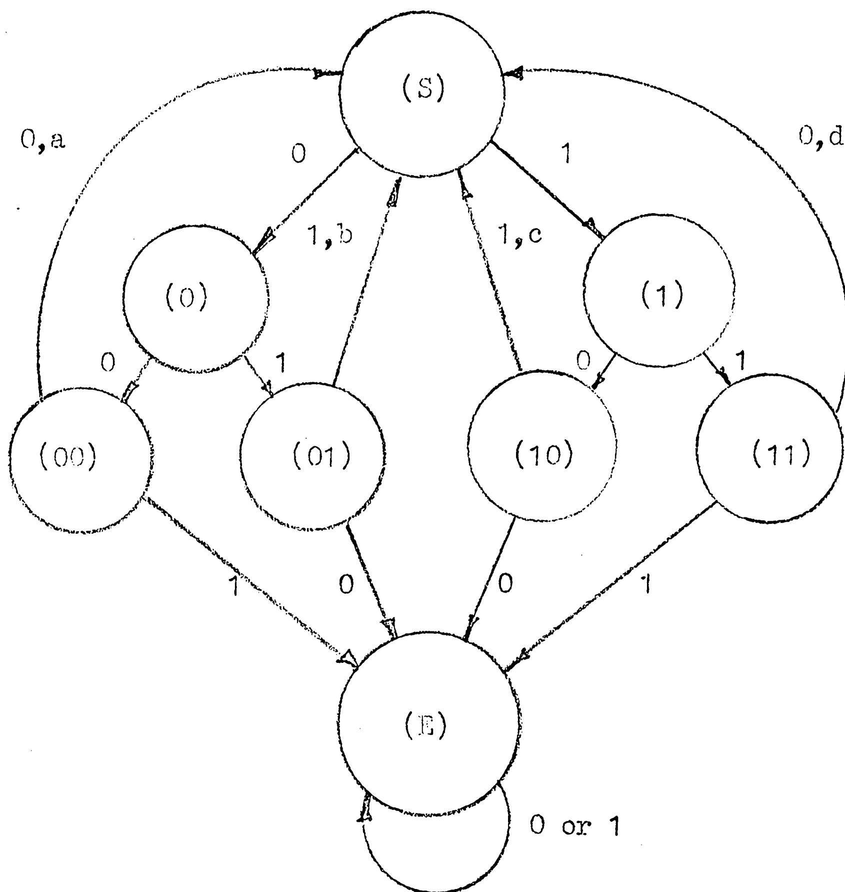
qo= (s)

A	C
a	00
b	01
c	10
d	11



Huffman Code

Figure 1.32



$A = \{a, b, c, d\}$

$C = \{000, 011, 101, 110\}$

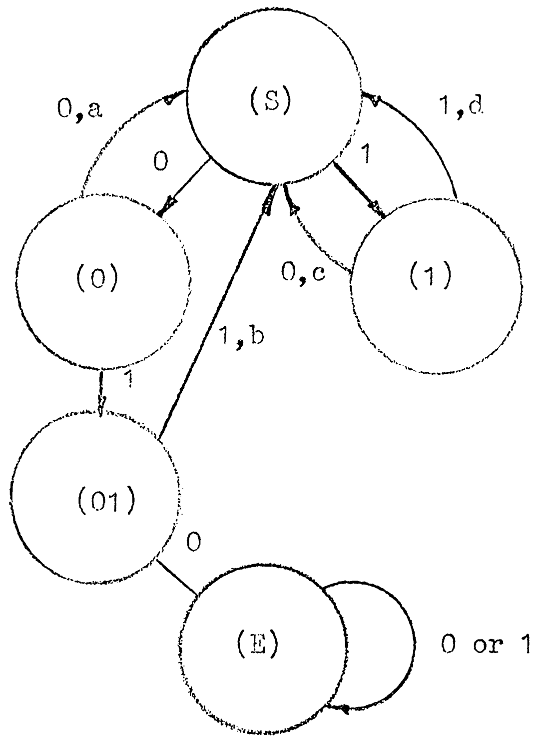
$F = \{(S)\}$

$q_o = (S)$

A	C
a	000
b	011
c	101
d	110

EVEN PARITY CODE

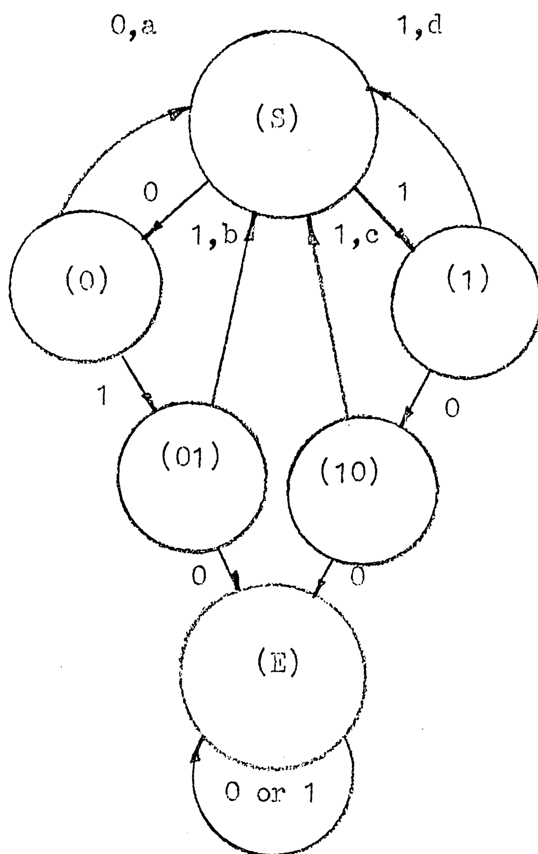
Figure 1.41



A	C
a	00
b	011
c	10
d	11

Y Code

Figure 1.42



A	C
a	00
b	011
c	101
d	11

Even Code

Figure 1.43

2. NONEXAUSTIVE CODES

2.1 Determining the Decoder Properties

The determination of decoder properties is one of the difficulties of using nonexhaustive codes in TDED. More specificly, determining the probability of detecting an error as a percentage of the errors that can be detected poses computational problems because of the out-of-phase condition. Most present error detection schemes do not have an out-of-phase condition past the code word where the error or errors occurred as shown by the simple Huffman code and even parity code discussed earlier. This is because they are either of fixed length [2] or automatically synchronized [4 and 5]. The out-of-phase condition is however very important because it allows greater efficiency in a code as compared to more common codes while retaining the same amount of error detection capability. Therefore the computational difficulties are a necessary evil.

One way of computing the properties of the code would be to simulate its operation and, after many millions of inputs and outputs, the probability of error detection could be determined by comparing the input with the output. This has its drawbacks in the form of time and energy. However, if we have an accurate probability distribution of the input alphabet, some shortcuts can be made. First, the state

transition probabilities, $P(q,i)$, (that is, the probability of receiving a certain input while in a certain state) and the state distribution probabilities, $S(q)$, (that is the probability of being in a certain state at any time) are determined. A detailed analysis of how to go about this using just the input alphabet probability distribution can be found in [1] for zero order codes and [6] for higher orders. We can then determine the probability, $R(q,i,t)$, of being in the out-of-phase, in-phase or error-detected-phase condition for any given state, (q) (where $(q) \in Q$), error ($e=i$ and $i=$input) and time (t, after the error) combination. These probabilities can be weighted by the probability of being in that state, $S(q)$, and the proability of receiving that paticular input while in that state, $P(q,i)$. once the weighting has been done we can sum all of the weighted probabilities, $R(q,i,t) \times S(q) \times P(q,i)$, and come up with the probabilities of being in the in, out or error-detected phase for some value of time. Figure 2.11 shows a flow chart of this procedure which is easily adapted into a computer program.

$R(q,i,t)$ can be determined in two ways. The first way would involve generating every binary sequence of length $t-1$. Each sequence would be run along paths in the decoder graph to determine the probability of the sequence ocurring, Pr, and determine an error, no-error pair of final states. Pr is equal to the product of all the individual path

probabilities taken by the sequence starting in the no-error starting state, (qi). (qi) is the state in which the decoder would be in after the error occured if no error had occured. (qj) is the state which the decoder entered after the error ocurred. The error and no-error final states, (q1) and (q2), are determined by starting in (qj) and (qi) respectively and following the input sequence through the decoder. If (q1) = (q2) than the decoder has synchronized so Pr will be added to the in-phase probability, Pi. If (q1) = (E), the error detected state, then Pr is added to the error detected probability, Pd. If none of the above occurs then the decoder is nonsyncronized and thus is in the out-of-phase condition, so Pr is added to the out-of-phase probability, Po. After the above procedure has been done for all sequenced of length t-1 then R(q,i,t) is obtained as <Pi,Po,Pd>.

The sequence generator approach requires a great deal more computation than required. There are three reasons for the wasted computation. First, once an error input sequence enters state (E) it will always stay their no matter what the time. Second, if at any time the decoder becomes synchronized it will remain synchronized (unless another error occurs). Third, some of the decoder state input transitions have zero probability and thus some sequences of binary numbers will have zero probability, Pr. These three reasons are used in the graphical method of determining

R(q,i,t).

The graphical method sets up a graph where nodes are (q1,q2) pairs ((q1)= error state, (q2)= no-error state) and arcs (one directional connections between nodes) are nonzero probability inputs to the decoder. The graph is started at node (qj,qi) where (qj) and (qi) are defined above (in the sequence generator method) as the error, no error starting states. From this node an arc is drawn to node (q3,q4) if and only if d((q1),i)=(q3), d((q2),i)=(q4), and P((q2),i)>0 (where i is the input associated with the arc). The node (q3,q4) is terminated (no leaving arcs allowed) if (q3)=(E), error detected, or if (q3)=(q4), in-phase condition. Every arc has associated with it an input and a probability. The arc probability is equal to P((qa),ia) where (qa) is the (q2) of the node in which the arc starts and ia is the input associated with the arc. The value of R(q,i,t)=<Pi,Po,Pd> can now be determined by taking every path of length t-1 starting from node (qj,qi) and determining Pr. Pr, the path or route probability is equal to the product of each of the arcs probabilities taken in the path. If the path ends or goes into a terminal node, it is assumed to stay there with the product probability it had upon entering. If the path goes into an error detected node, (q1)=(e), then Pr is added to Pd. If the path goes into a in-phase node, (q1)=(q2) then Pr is added to Pi. If niether of the above occurs Pr is added to Po. When all of the paths of length t-1 have

had their outcomes computed, we have R(q,i,t). The graph method is better because the number of paths in the graph is usually much smaller than the number of binary sequences and can never be larger.

The above procedure deals mainly with single errors. The problem of multiple or burst errors is one which requires a great deal of knowledge about the types of burst errors. If such knowledge is available in the form of a set of probabilities, BP(i,t) (where i is a paticular input at time, t, after the error has started), the procedure given above can be modified to obtain the overall probabilities. The modification involves the use of BP(i,t) to determine Pr in place of P(q,i) when determining R(q,i,t). The modification would be made in determining R(q,i,t) the rest of the procedure remains the same.

At this point, an example is called for to demonstrate the above procedure. Because of the large amounts of computations required for a large code with many code words, this example will be very simple as the code in figure 2.12 shows. Figure 2.12 shows the code, the P(q,i) values and the S(q) values. The R(q,i,t) values for single errors are shown in figure 2.13. Figure 2.12 also shows the total error detection probability calculation. The results achieved by this code at time one, (t=1) are very inconclusive with the total probability, TP(1)=<.0,.77,.23>. At time two or greater (t>1) the total probability, TP(2)

=<.6083,0,.3917> which is not very much considering the cost in average code lenth, acl.

2:2 Average Code Length Versus Error Detection Probability.

The example in the last section had an average code length, acl, of (.7x1)+(.3x2)=1.30. An optimal Huffman code (0,1) would have acl=1. Therefore, the example code traded .3 bits per code word for an error detection capability (for single errors) of 0.3917. If we now choose the b encoding to be 11 instead of 10 as shown in figure 2.21 we find that at time two we have <0.,.355,.645> and as time approaches infinity the probability vector rapidly approaches <0.,0.,1.>. The new code detects all single errors while keeping the acl=1.3. Obviously our new code is more advantageous. One of the major differences between the new code and the one of figure 2.12 is the probability of going into the in-phase condition. The new decoder never can go in-phase after an error has occurred unless there is a second error. Codes having this property of never going in phase will be called nonsynchronizing codes (nonsynch codes for short). It appears, but has not yet been proven, that nonsynch codes represent a minimum acl cost for a given error detection capability. This will be very apparent later for single error detecting even codes but for higher

detecting codes the problem of creating a code is compounded and not readily apparent. The next section will provide some guidance and theorems to help to better understand which codes provide a good acl, error detection capability trade off.

2.3 Alphabet and Code Characteristics

There are many desirable and undesirable alphabet-code characteristics but they all seem to center around one characteristic, the probability of synchronizing. We have already seen that the nonsynch property is desirable and theorem one is a very important extension of this property.

Definitions:

1) A phase check determines if the decoder is in-phase or out-of-phase.

2) A final phase check is a phase check made at the end of an input sequence.

Theorem 1:

In any time delayed error detection scheme requiring a final phase check, an error goes undetected if and only if the decoder synchronizes after the error and before the decoder has entered the error detection state, (E).

proof: (Necessary)

Assume the error is not detected and the decoder is not synchronized. It is either out-of-phase or has entered the error detected state, (E). It, however, could not have entered the error detected state because the error is not detected. So it is out-of-phase but the final phase check will see that it is out of phase and conclude an error has occurred. Thus proof by contradiction.

(Sufficient)

If the decoder resynchronizes before the error is detected the system will not discover this error by the phase check because the code will be in-phase and once in-phase, the decoder cannot enter the error detection state, once in phase the decoder will never go out of phase unless another error occurs because $d:F\mathrm{x}C^{*} \dashrightarrow F$.

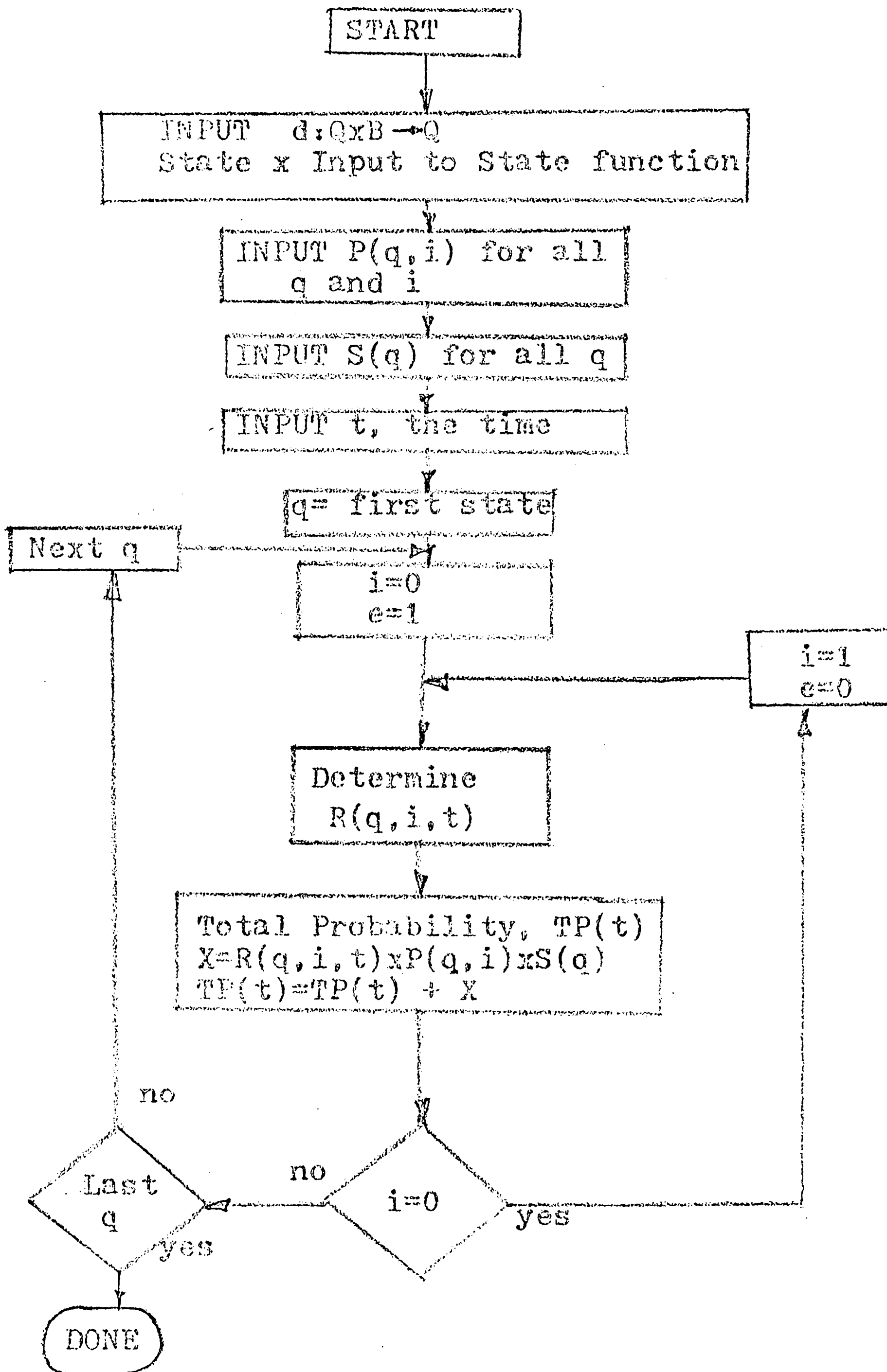
Theorem 1 may seem like a restatement of what was said earlier but there is one important difference. We can now prove that a code will detect e errors if after e or less errors occur the decoder never becomes in-phase (where e is any positive integer).

The phase check used in Theorem one may seem unnecessary because it only takes care of out-of-phase conditions. All of the examples, and in fact all of the codes that use TDED thus far observed, have the out-of-phase probability approach or equal zero for increasing values of

time. However this approach to zero may not always exist as shown by Neuman [5]. Secondly, the out-of-phase probability only equals zero at time equal to infinity as shown in the last example, figure 2.21. So for a finite sequence of code words the out-of-phase probability is still not zero. The out-of-phase check is therefore necessary and will be discussed in greater detail later.

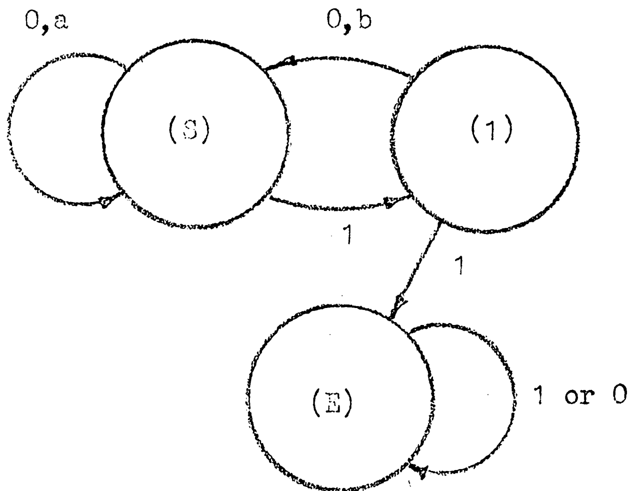
Providing a final phase check leaves only the in-phase probability as the major consideration. For a given code it is desirable to reduce this probability to maximize the error detection capability of the code. This can be done by making the code as close to a nonsynch code as possible. However, in most problems it is not the acl which is given, known, or wanted but instead the alphabet, A, and a level of desired error correcting capability, ecc, is given. The problem now becomes how to encode the alphabet, A, so as to meet ecc but minimize acl. The general approach would be to construct an optimal Huffman code [1] and then add redundancy to the lower probability (least probable to occur) code words until you had met your goal of error detection capability. The low probability encodings are used because they contribute smaller additions to the acl. How good the redundancies are at detecting errors seems to be independent of the code word probability or at least not strongly correlated. This comes about as a result of the interchanging of the state transitional probabilities after

the decoder goes out-of-phase. This interchanging differs with every input and continues until the decoder goes into the in-phase or error-detected condition. The above process is not meant as an algorithm for obtaining a code, but instead the above should be thought of as a few hints to keep in mind while constructing a TDED code. The next section, however, does give an optimizing algorithm using nonsynchronizing codes which detect any single error.



General Algorithm

Figure 2.11



A	C	Probability
a	0	.7
b	10	.3

$P((S),0) = .7$

$P((S),1) = .3$

$P((1),0) = 1.$

$P((1),1) = 0.$

$$S((S)) = \frac{.7+.5(.3)(2)}{1.3} = .77$$

$$S((1)) = 1. - S((S)) = .23$$

$$S((E)) = 0.$$

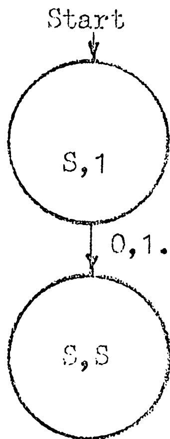
Time	Total Probability Vector, TP(t)
1	$\langle 0.,.77,.23 \rangle$
≥ 2	$\langle .608,0.,.392 \rangle$

R(q,i,t) Values are shown in figure 2.13

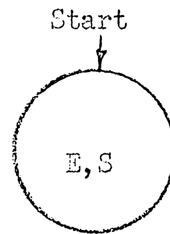
Example Decoder

Figure 2.12

R((S),1,t)

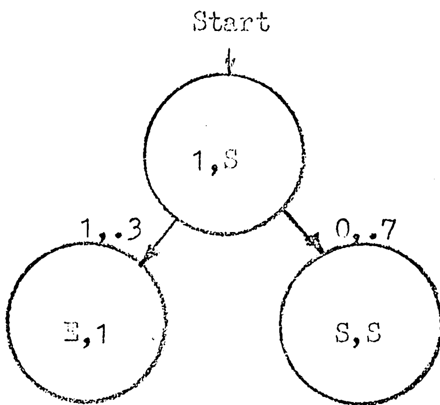


R((1),0,t)



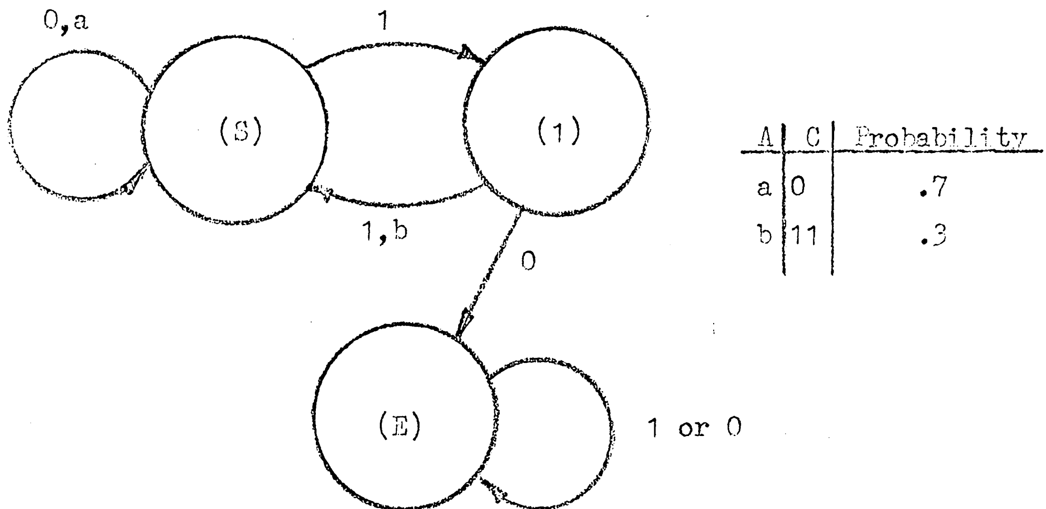
Time	R(q,i,t)
1	R((1),0,1)= <0.,0.,1.>
	R((S),0,1)= <0.,1.,0.>
	R((S),1,1)= <0.,1.,0.>
≥2	R((1),0,2)= <0.,0.,1.>
	R((S),0,2)= <.7,0.,.3>
	R((S),1,2)= <1.,0.,0.>

R((S),0,t)



R(q,i,t) Determination

Figure 2.13



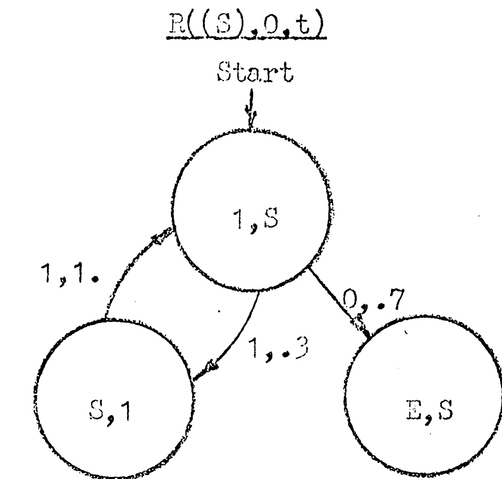
A	C	Probability
a	0	.7
b	11	.3

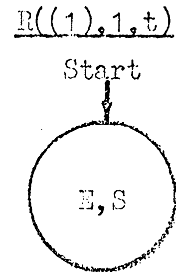
Time	Total Probability Vector, TP(t)
1	<0.,.77,.23>
2	<0.,.355,.645>
3	<0.,.231,.769>
1+2n	$\langle 0.,.77(.3)^n,1-.77(.3)^n \rangle$
$t\to\infty$	<0.,0.,1.>

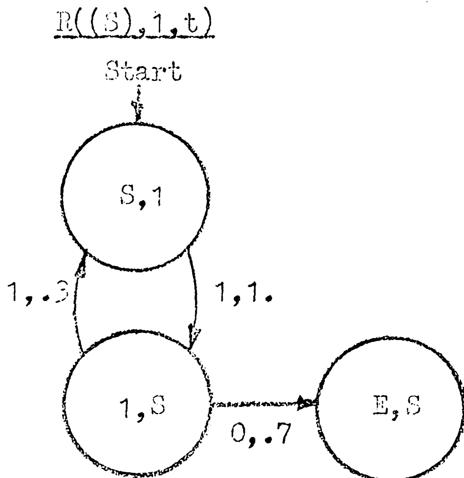
R(q,i,t) Values are shown in figure 2.22

New (Even) Code Example

Figure 2.21







Time	R(q,i,t)
1	$R((S),0,1) = \langle 0.,1.,0.\rangle$
	$R((S),1,1) = \langle 0.,1.,0.\rangle$
	$R((1),1,1) = \langle 0.,0.,1.\rangle$
2	$R((S),0,2) = \langle 0.,.3,.7\rangle$
	$R((S),1,2) = \langle 0.,1.,0.\rangle$
	$R((1),1,2) = \langle 0.,0.,1.\rangle$
3	$R((S),0,3) = \langle 0.,.3,.7\rangle$
	$R((S),1,3) = \langle 0.,.3,.7\rangle$
	$R((1),1,3) = \langle 0.,0.,1.\rangle$
1+2n	$R((S),0,1+2n) = \langle 0.,(.3)^n,1-(.3)^n\rangle$
	$R((S),1,1+2n) = \langle 0.,(.3)^n,1-(.3)^n\rangle$
	$R((1),1,1+2n) = \langle 0.,0.,1.\rangle$

Even Code R(q,i,t) Determination
Figure 2.22

3. NONSYNCHRONIZING CODES

3.1 General

Nonsynchronizing (nonsynch) codes provide many improvements over other codes. Nonsynch codes are really just a subset of nonexhaustive codes. The simplified analysis of the properties of nonsynch codes leads to a set of clear properties that can be proven. The simplified analysis is brought about because nonsynch codes have, for any time, a zero probability of going in-phase provided there are certain restrictions on the errors they will receive. The time between when the error occurs and when the error is detected (the time to error detection), appears to be inversely proportional to the amount of acl difference between the nonsynch code and an optimum Huffman code. The probability of being out of phase reduces or decays almost exponentially. The time constant of the decay is again inversely proportional to the acl difference. However, no proof or even strong evidence is provided in the paper beyond the experience of the few examples which have been done to date. The nonsynch codes have been divided into two areas, Even Codes and codes which detect more than one error.

3.2 Even Codes

Even codes are very important in Time Delayed Error Detection. Even codes represent to single error detection what Huffman codes represent to no or zero error detection. Huffman codes are optimal (minimum acl) for no error detection and even codes can be found which are optimal for single error detection. Two examples of even codes have already been shown but now the definition of what constitutes an even code will be presented.

Definitions:

The set of all even code words, E, is represented by the regular expression $E = 0^* + (0^*10^*10^*)^*$. A code, C, is called even if every code word, c, in C is an element of E (for all $c \in C \rightarrow c \in E$). An even identity function, PHI(i), equals 1 for any code word, i, $i \notin E$ and equals 0 for $i \in E$. (PHI(i) adds all the ones of a code word modulo 2.)

Lemma 2.1:

For any two code words, $i, j \in E$ and $i \neq j$, the distance between i and j will be greater than or equal to 2.

Proof:

If i and j differ in no (zero) bit positions, they are the same but $i \neq j$. If they differ in 1 bit position, then one of them has one more 1 than the other, but they both

have an even number of ones, so proof by contradiction.

Lemma 2.2:

For any $i \in E$ if i is changed in one place (either a 1 to a 0 or a 0 to a 1, one error in i) to become j, then $j \notin E$.

This follows directly from the definition of coding distance and Lemma 2.1. It may also be shown by PHI(j)=1 thus $j \notin E$.

Theorem 2:

Even codes detect all single errors. This can be restated as shown in the next sentence. Even codes never synchronize (go in-phase) after a single error has occurred provided that no additional errors occur.

Proof:

Assume that we have a valid string, of codewords, r. We also have a string, s, which is r but with one error. Both the string r and the string s start in the same state, (s). For the error in s to go undetected the s and r strings must finish in the same state. String r finishes in a valid final state besause r is a string of valid codewords. Now if all the valid code word sequences, C, accepted by the decoder are even (as is the case with an even code), then $C^* \subseteq E$. Because $s \notin E$ by Lemma 2.2 then $s \in C^*$ so $s \in \overline{C^*}$. By definition $d: F \times \overline{C^*} \dashrightarrow \overline{F}$ so $d((s), s) = (qs)$, $(qs) \notin F$

but $d((s),r) = (qr)$, where $(qr) \in F$ so $(qs) = (qr)$ thus the decoder cannot be in-phase with s as an input string. Then by Theorem 1, the single error in s is detected and therefore even codes detect single errors.

Now that the properties of even codes are known, the next step is to obtain an optimal (minimum acl) even code from a given input alphabet, A, and its input probabilities. An algorithm is available to obtain an optimal even code for A. First, a Huffman code must be constructed for A. This procedure is outlined by Abramson [1] for zero order systems. The algorithm for obtaining a minimum even code applies for the zero order system. The higher order even code is obtained by repeating the zero order procedure for each markov state code [6]. The Huffman code is now divided into classes by the length of its code words. All code words with the same length are in the same class. Starting with the smallest length class and working to larger classes an algorithm is used as found in figures 3.21, 3.22 and 3.23. The algorithm may seem complicated at first but once the first nonempty length class has been encoded the rest of the code words follow a very simple form. The simple form is arrived at by taking code words from the last nonempty class (of length p), complementing their last bit and then adding (concatenating) an odd-number-of-ones binary sequence at the end of the code word. The length of the odd length sequence

is equal to the length of the new class (of length l) minus the length of the last nonempty length class (of length p). If at any time the number (n) of characters in a length class (of length l) excede the number, $(m+r)(2^{l-p-1})$, then the characters of least probability are removed from the class l and put into the next larger class (l+1) until only $(m+r)(2^{l-p-1})$ characters remain in l (m=the number of real and dummy code words and r=the number of dummy code words in the last nonempty length class). An example of the use of the algorithm is found in figure 3.24 and 3.25. In this example, all of the length classes after one are non empty. In a second example of figure 3.26 and 3.27, this is not the case. The second example also demonstrates the use of dummy code words. Dummy code words are formed if a length class is not empty and has less than the maximum number of code words allowed for the class $(0 \leq n \leq (m+r)(2^{l-p-1}))$. The extra or dummy code words are created and saved for later use in the algorithm. They never have a character assigned to them but instead are used later to form code words but also the dummy code words can be made into code words by simply adding a binary string of length, l-p, which is made up of an even number of ones. The length is equal to the odd string length added to the code word as discussed before. Therefore a dummy code word can be used to make twice as many code words as a normal code word.

A comparison is made in figures 3.25 and 3.27 between

the acl of the Huffman code and the even code. Figure 3,24 also shows the even parity code. The even code can be seen as very close to the Huffman code in acl and much smaller than the even parity code. The difference in acl is very apparent and TDED using even codes is shown to be a far more efficient code for the capability to detect a single error. It should be noted that even codes as defined are not the only nonsynch code which detects any single error. If all the code words have every bit complemented (an even number of zeroes code) of any even code, the new code has all the properties of an even code. However, because the above code displays the same characteristic and useful properties as even codes (which are already presented) there is no need to be redundant. Therefore their properties will go unproven. The next section however deals with codes which have the property of detecting more than one error.

3.3 Higher Error Detection Codes

Higher error detection code can detect e-1 errors where e is some number larger than two. For fixed length codes e must be less than or equal to the distance between any two code words of the code. The restrictions on the fixed length code is given by the Hamming Bound which is described in many coding theory texts including reference [1]. However,

these restrictions do not apply entirely to TDED codes.

Given a TDED code, it can be divided up into length classes as were Huffman and even codes. Each length class can be thought of as a separate code which has the same restrictions as a fixed length code having the same number of code words as the length class. Every code word in the length class must have a distance of e or more from any other code word in its class if the code is to detect e-1 errors. However, the relationship between two code words in different classes is not so restrictive. If the code is a nonsynch code then the only way for an error to be detected is if enough errors occur to cause the input sequence to be a different but valid sequence. Theorem three gives a sufficient but not necessary restriction on code words of different length classes. Even codes are a good example of theorem three not being necessary as shown in almost every example of even codes.

Theorem 3

A code will detect e-1 errors if the code words can be broken into length classes such that:

1) $A = C1 \cup C2 \cup C3 \cup \ldots \cup Cn$, n=length of the largest code word

2) if $c \in Ci$, then $c \notin Cj$ where $i \neq j$

3) $c \in C$, if and only if c is i bits long

4) $\Delta(c,d) \geq e$ for $c,d \in Ci$ for any i

5) $\Delta(c,d) \geq e/2$ for $c \in Ci$, $d \in Cj$ and $i \neq j$

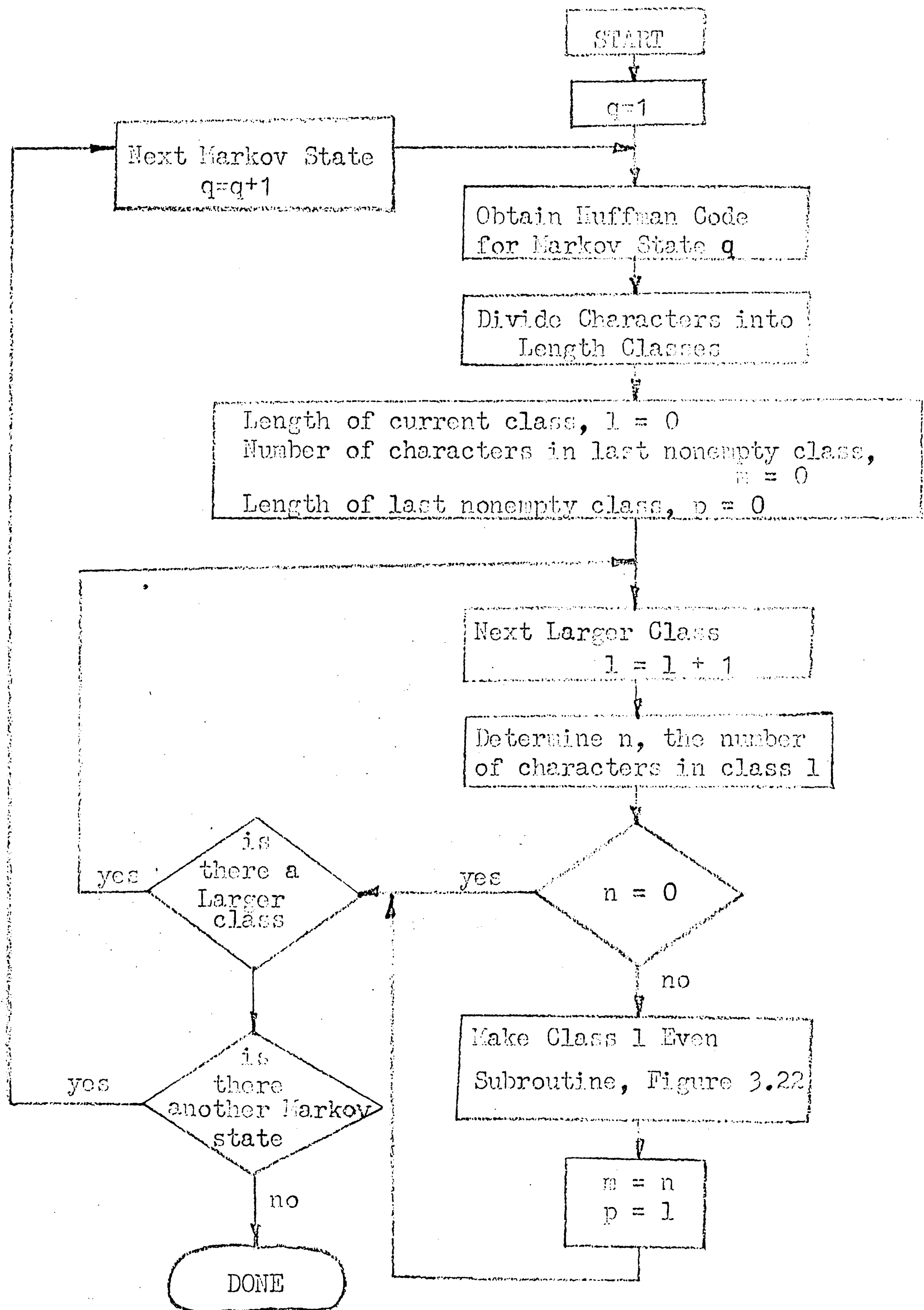
where $\Delta(c,d)$ is the Hamming distance between c and d, as compared from the front and the back.

Proof:

Assume the code will not detect e-1 errors. Then there must be a sequence, s, which can be corrupted to a valid sequence, r, with e-1 or fewer errors. Sequence s must be a valid sequence of code words in-phase with r (if the decoder is out-of-phase or in the error detected state, the errors are discovered by theorm 1). Compare s and r, starting at their beginings and continuing until we find the first and the last encoded characters in r which differs from s. If the first such character of r has the same lenth as the first one of s, then they differ by $\geq$e (by 4), but they cannot by our assumption. So, they must be different lengths and thus in different classes of A. They therefore differ by e/2 (by 5). However, because they are of different lengths, somewhere after this point there must be code words of s and r which end in the same place or position and which are not of the same length. To become in-phase both the r and s strings must cause the decoder to end in the same state. Sequence r is a valid string so it ends in a valid final state as must s. To end in a valid final state after starting out-of-phase at some point (the first different characters). The lengths must add up at

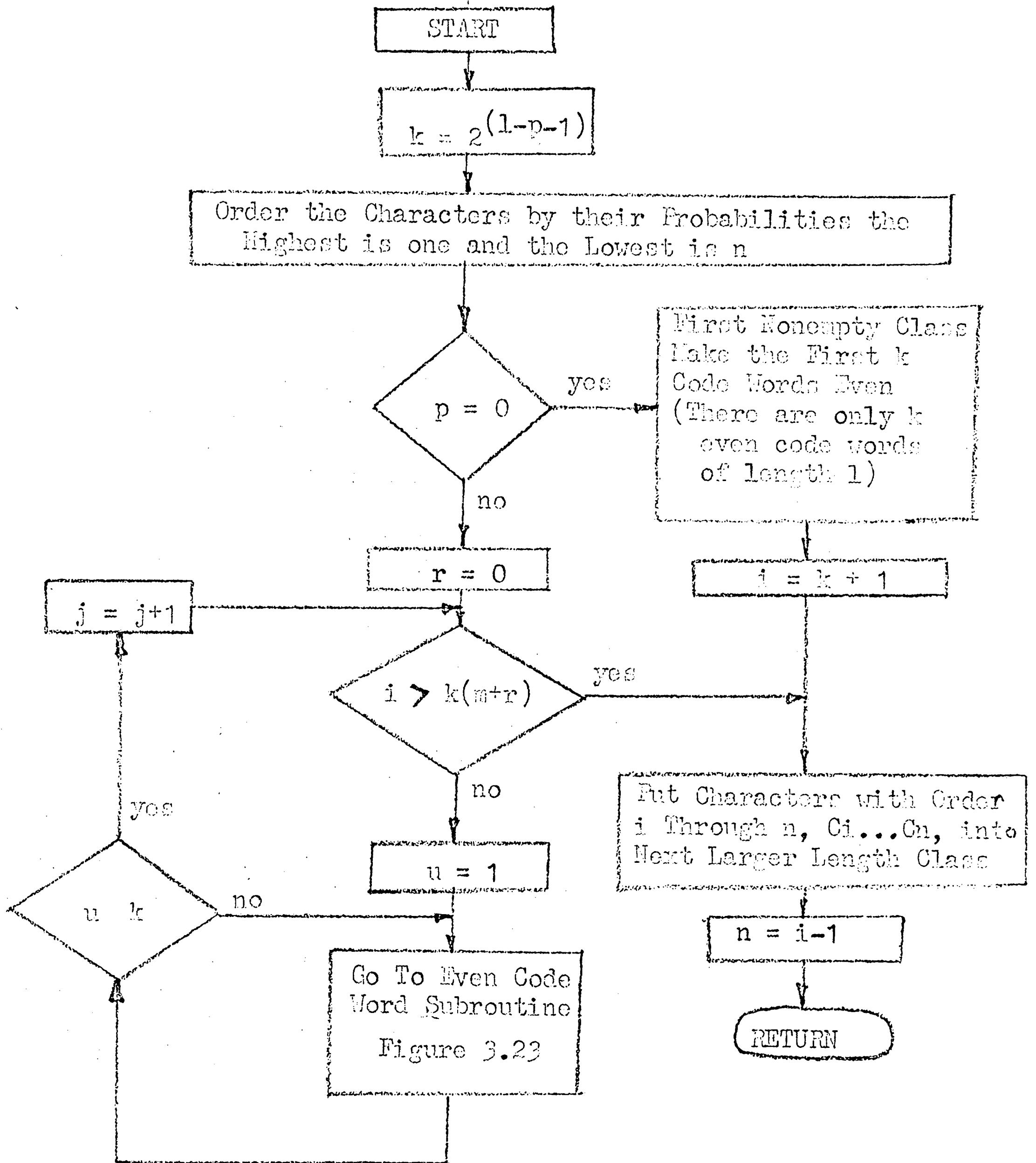
some later point to the same value. This means the r string character must end with a character of different length than the s string character ending at the same point and again (by 5) these two characters vary by at least e/2. This makes the minimum distance between r and s e/2+e/2 = e, thus proof by contradiction.

All code words for which theorem three applies are obviously not optimal. Which codes are optimal and an algorithm for obtaining a TDED code which detects e-1 errors (where e is larger than 2) was not found. However, using theorem three and approaching the problem by dividing the characters into length classes did produce the codes of figure 3.31. The Y code classes are arrived at by taking the even code length classes of figure 3.23 and adding two to their lengths. The code detects 4-1=3 errors or less with time delay. The code words themselves are the same forwards and backwards which gives them their e/2 differences between classes (regardless of being compared from the front or back). A second code which detects three errors is also shown, code Z, but because the a character code word was picked as 00, the other code words were overly restricted. This is apparent from the acl of the two codes.

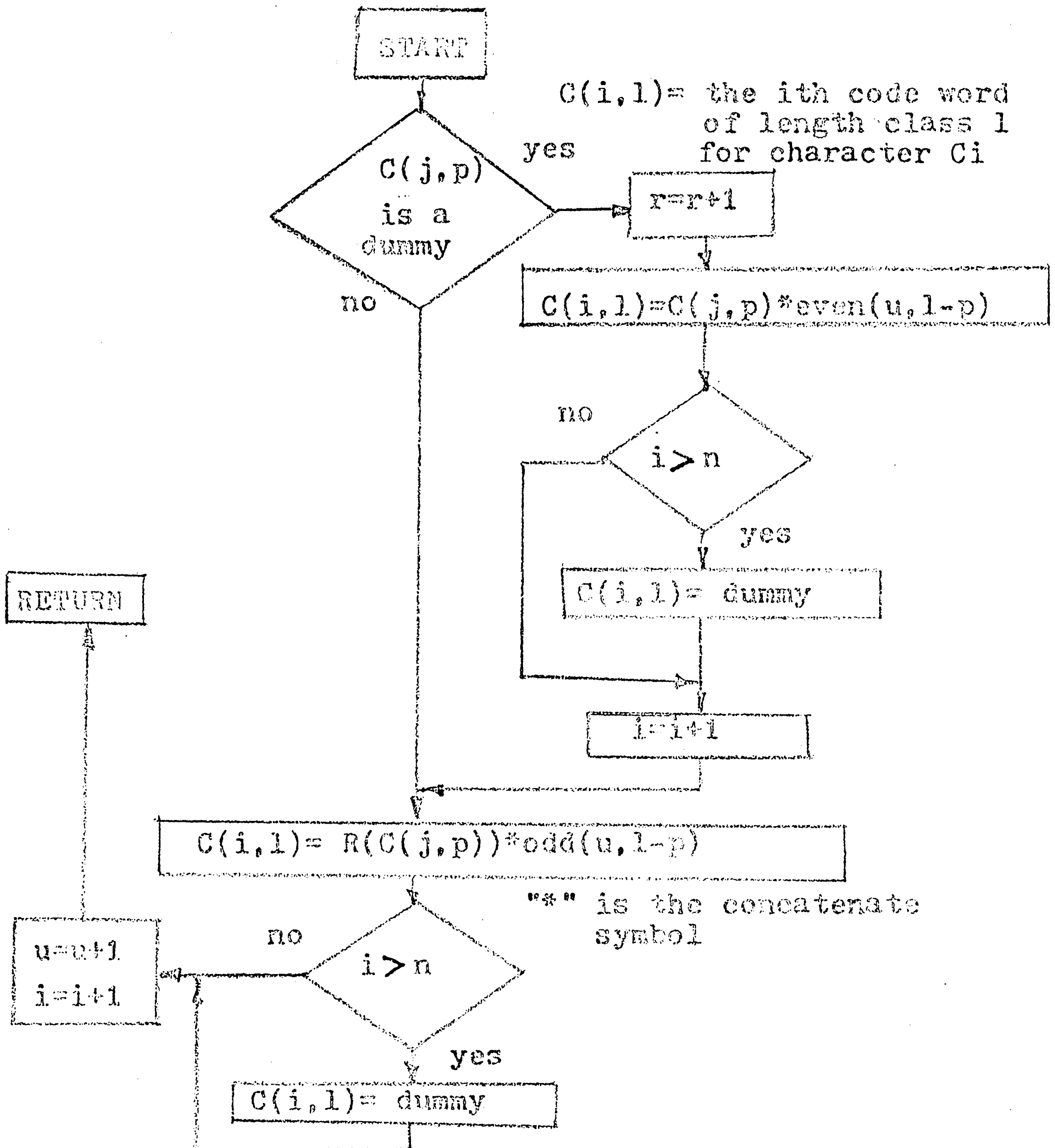


Even Code Algorithm

Figure 3.21



Even Class Subroutine
Figure 3.22



R(i)= i but with the last bit complemented
even(u,l-p)= the uth occuring "even number of ones" binary number of lenth l-p
odd(u,l-p)= the uth occuring "odd number of ones" binary number of lenth l-p

Even Code Word Subroutine

Figure 3.23

$A = \{a,b,c,d,e,f,g,h\}$

A	Probability	Huffman Code	Length Classes
a	8/32	01	
b	8/32	11	class 1 = ∅
c	4/32	001	class 2 = {a,b}
d	4/32	101	class 3 = {c,d,e}
e	3/32	100	class 4 = {f}
f	3/32	0001	class 5 = {g,h}
g	1/32	00001	
h	1/32	00000	

Start

p=0; m=0; l=1; n of 1 = 0, so go to next length class

p=0; m=0; l=2; n=2

C(1,2) = 00 These are the only two even code words of length

C(2,2) = 11 two

p=2; m=2; l=3; n=3

$C(1,3) = 0\bar{0}*1$; $R(C(1,3)) = 0\bar{0} = 01$ and odd(u,l-p)=odd(1,1)=1

$C(2,3) = 1\bar{1}*1 = 101$

$3 = i > (m+r)k = (2+0)2^{(3-2-1)} = 2x1 = 2$, so character e (e=C3=Ci...Cn)

goes into the next higher length class, 4

p=3; m=2; l=4; n=2

$C(1,4) = 01\bar{1}*1 = 0101$

$C(2,4) = 10\bar{1}*1 = 1001$

no characters left in length class 4 and i > (m+r)k, so next class

p=4; m=2; l=5; n=2

$C(1,5) = 010\bar{1}*1 = 01001$

$C(2,5) = 100\bar{1}*1 = 10001$

five is the last length class, so done

Example Using the Algorithm

Figure 3.24

A	Even Code	Even Parity Code
a	00	0000
b	11	0011
c	011	0101
d	101	0110
e	0101	1001
f	1001	1010
g	01001	1100
h	10001	1111

acl of the Huffman code = 87/32 = 2.71875

acl of the Even code = 87/32 + 3/32 = 2.8125

acl of the Even Parity code = 4

Even and Even Parity Codes for the Example

Figure 3.25

A	Probability	Huffman Code	Length Classes
a	40/64	1	
b	8/64	000	class 1 = {a}
c	6/64	001	class 2 = ∅
d	4/64	010	class 3 = {b,c,d}
e	2/64	01100	class 4 = ∅
f	2/64	01101	class 5 = {e,f,g,h}
g	1/64	01110	
h	1/64	01111	

Start

p=0; m=0; l=1; n=1

C(1,1) = 0 this is the only even code word of length one

p=1; m=1; l=2; n=0, so go to the next length class, 3

p=1; m=1; l=3; n=3

$C(1,3) = \overline{0}*01 = 101$

$C(2,3) = \overline{0}*10 = 110$

$3=i > (m+r)k=(1+0)2^{(3-1-1)}=2$, so character d goes into class 4

p=3; m=2; l=4; n=1

$C(1,4) = 10\overline{1}*1 = 1001$

$C(2,4) = 11\overline{0}*1 = 1111$; but i n, so C(2,4) is a dummy code word

p=4; m=2; l=5; n=4

$C(1,5) = 100\overline{1}*1 = 10001$

C(2,5) = 1111*0 =11110; even(u,l-p)=even(1,1)=0; an even ending is added because C(2,4)=C(i,p) is a dummy; also r=1

$C(3,5) = 111\overline{1}*1 = 11101$

$4=i > (m+r)k=(2+1)2^{(5-4-1)}=3$, so character h goes to class 6

r=0; p=5; m=3; l=6; n=1

$C(1,6) = 1000\overline{1}*1 = 100001$

six is the last class, so done

Second Example Using the Algorithm

Figure 3.26

A	Even Code
a	0
b	101
c	110
d	1001
e	10001
f	11101
g	11110
h	100001

acl of the Huffman code = 126/64 = 1.96875

acl of the Even code = 126/64 + 5/64 = 2.046875

Even Code for the Second Example

Figure 3.27

A	Probability	Even Code e=2	Code Y e=4	Code Z e=4
a	8/32	00	0000	00
b	8/32	11	1111	1111
c	4/32	011	01010	110011
d	4/32	101	10101	11000011
e	3/32	1001	110011	1100000011
f	3/32	0101	001100	110000000011
g	1/32	01001	0110110	11000000000011
h	1/32	10001	1001001	1100000000000011

acl of the Y code = 154/32 = 4.8125

acl of the Z code = 200/32 = 6.25

Higher Order Codes Examples

Figure 3.31

4. IMPLEMENTATION AND HARDWARE

4.1 Encoder and Decoder

TDED, because it is interactive in nature, would probably require a large amount of time and hardware if it were to be made up of flip-flops and gates as are many encoder-decoder combinations. A better system would involve the use of microprocessors or minicomputers for the encoder and the decoder. The state transition graphs used to describe the encoder or decoder could easily be modeled by a microprocessor. Figure 4.11 shows a simple encoder-decoder configuration or system which uses a microprocessor for the encoder and the decoder. The input and output of the system are strings of alphabet characters which may themselves be coded (for instance EBDIC or ASCII symbol codes). The memory of the processors could be either internal or external registers and/or addressable external memory. In both the encoder and decoder, the memory would hold the d(q,i), state cross input to new state, function and the w(q,i), state cross input to output function. These functions could be in the form of lookup tables or as described by [6] in the form of instructions to implement the building of the tables depending on the overall state of the machine. The memory would also hold the state and program instructions. In addition the decoder memory could hold the input until it is

ready to be operated on and/or the output until the receiving end (where the output goes) is ready to take the output. The encoder memory would also be required to hold the input to the channel (the encoder output) until the message is completed. This restriction of holding the whole input guarantees that if an error is detected then the encoder can retransmit the whole message. In practice a buffer space of some large size would be sufficient. For a long buffer, there is a small chance that an error could remain undetected for a time greater than the time covered by the buffer. For this to be a valid procedure the average time between error and detection would have to be much smaller than the time covered by the buffer. The buffers would also be required to store the state of the first bit of the buffer.

The micro processors would be required to perform the instructions to update the state of the machine and output the necessary code. The encoder would have to in addition monitor the error detect line and once an error detect sign is received the system must go back and output from the beginning of the input memory buffer. The decoder will have to output an error detected signal to the encoder once the error detected state is obtained. It must also give some indication to the receiving circuit that the last n output characters, where n is not known to the decoder, are wrong. This prospect can be avoided by having the decoder memory

hold the last n characters where the n characters have a sequence code length which is the size of the encoder buffer.

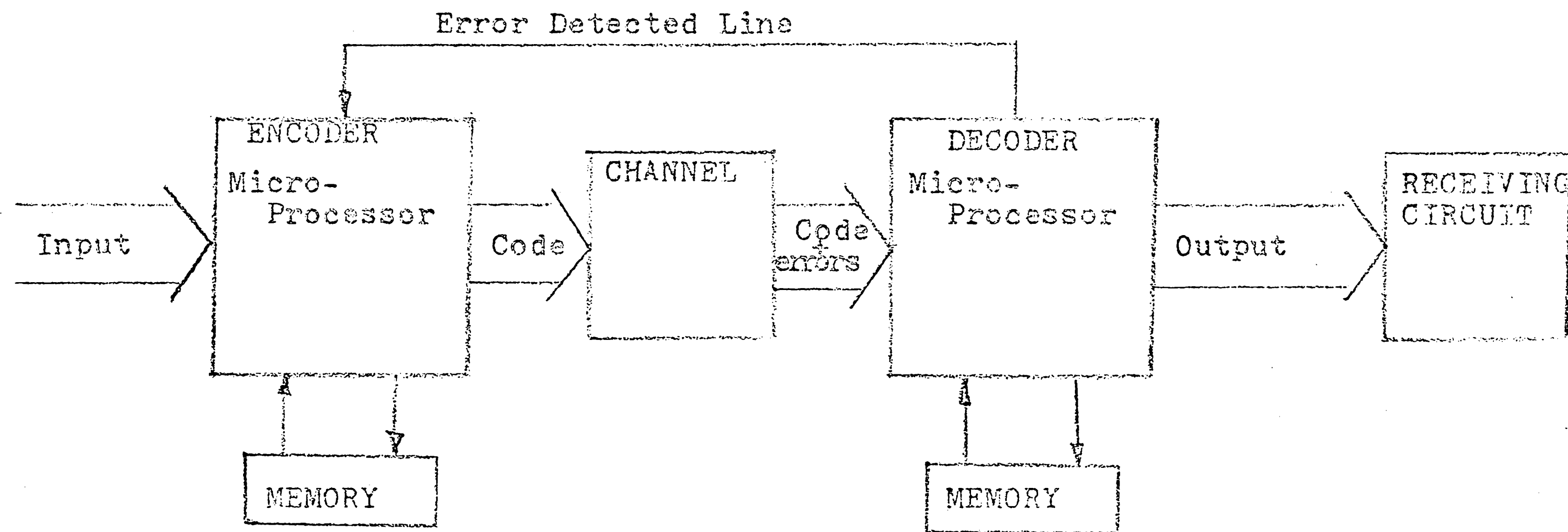
The problem of having an error go undetected for a long period of time can be very troublesome. The buffer spaces required to neglect the possibility of an error going undetected will probably be large for a very efficient code. If the error should go undetected beyond the buffer space then the encoder and decoder will remain out of phase, that is the encoder's output will not correct the decoder's problem because the error will still have occurred because it occurred before the beginning of the present buffer. The decoder will then continue to go into the error detected state after the buffer has been sent to the decoder. The problem above can be solved in many ways but one simple way would be to require a periodic phase check to see if the state of the decoder is correct.

4.22 Phase Check

There are many ways by which the decoder condition or phase may be checked. One way would be for the encoder to output to the decoder upon request its state as suggested by Ott [6]. The encoder could check this but the procedure is very complicated and awkward. A better way would be for the

encoder to output a special code which would take the decoder from a final state to a phase-check-state. The phase-check-state would act like the final state from which the special code occurred but no output would be generated. The special code would be given after every n code words from the encoder (where n is some predetermined integer). A register in the decoder would count the number of code words received. Once n code words were received the decoder would expect the special code. If it did not occur the decoder would go into the error detected state. The phase-check code would be added to the input alphabet of the encoder as a character and would have a probability of $1/n$. The other alphabet characters would have their probabilities multiplied by $(1-1/n)=(n-1)/n$. The new alphabet resulting from the addition of the phase-check character could now be dealt with like any other alphabet.

The choice of n would depend on the channel characteristics and the desired error detection capability. A small value of n would result in a significant increase in acl and a much lower probability of an error going undetected (assuming rate smaller than channel capacity). Also when an error occurs and is detected the amount of code which must be retransmitted is proportional to n and thus as n becomes smaller the length of the memory buffers and time lost retransmitting becomes smaller.



TDED, Encoder and Decoder Configuration

Figure 4.11

5. CONCLUSION

5.1 The Use of Time Delayed Error Detection

Time delayed error detection codes can be used in almost every application that any present day error detection code is used. TDED codes, because they are of variable length and can be optimized, are better adapted to encode real life alphabets which do not have equal character probabilities. TDED codes cannot be used in FEC, Forward error control, systems but could be used in ARQ, automatic repeat request, systems. The inability to be used in FEC systems, however, does not seem to be that important. For most applications, as discussed by Burton and Sulivan !2!, ARQ is or will be the way to do it.

5.2 Open Areas and Needed Work

The material covered so far has left quite a few loose ends untied. The greatest area of needed work envolves the time to error detection and the trade off of acl versus time to error detection. The brute force method of chapter two must be reduced to analyse these time problems. The capability of TDED codes to handle multiple burst errors needs to be analyzed further and some code characteristics

which help or hinder multiple error detection need to be formulated. A general algorithm needs to be thought out for higher error detection codes similar to the algorithm for even codes and A proof is needed for the even code algorithm of 3.2. A procedure to find the optimum value for n, the number of characters between phase checks, is the next step towards implementing TDED and so it should be explored. Phase errors (errors where a binary bit is lost and thus the encoder and decoder are out of phase) and transposition errors are other areas of concern. There are many areas which could use additional work and refinement which were covered but not completely. In general the problems are still wide open to further work beyond this paper.

QUOTATIONS

1. E. Gilbert and E. F. Moore, "Variable Length Binary Encodings", The Bell System Technical Journal, Vol. 37, July, 1959, page 958.

2. Peter G. Neumann, "Efficient Error-Limiting Variable-Length Codes", IRE Transactions on Information Theory, July, 1962, page 300.

REFERENCES

1. Norman Abranson, Information Theory and Coding, McGraw Hill Book Company, 1963.

2. H. O. Burton and D. D. Sullivan, "Errors and Error Control", Procedings of the IEEE, Nov. 1967.

3. E. Gilbert and E. F. Moore, "Varible Lenth Binary Encodings", The Bell System Technical Journal, Vol 37, July 1962.

4. Peter G. Neumann, "Efficient Error-Limiting Variable Length Codes", IRE Transactions on Information Theory, July 1962.

5. Peter G. Neumann, "Self-Synchronizing Sequencial Coding with Low Redundancy", The Bell System Technical Journal, Vol. 50, No. 3, March, 1971

6. Gene Ott, "Compact Encoding of Stationary Markov Sources", IEEE Transactions on Information Theory, Vol. IT-13, No. 1, Jan. 1967.

The vita has been removed from the scanned document

TIME DELAYED ERROR DETECTION CODES

by

Charles L. Rahm, III

(ABSTRACT)

The use of nonexhaustive codes which have special types of redundancies are formulated. The properties of these codes include the ability to detect amplitude errors in binary sequences by driving the decoder into an error detected state. However, the errors are detected at some time after the the error has reached the decoder and thus the name time delayed error detection, TDED. When an error is received, the decoder is found to be in one of three phases or conditions, in-phase, out-of-phase, or error-detected-phase. A type of nonexhaustive code, called an even code, is capable of detecting any single error. An algorithm is presented which can construct an optimum single error detecting even code for any alphabet. This optimum even code is found to be very close in average code lenth to the Huffman code for the same alphabet. TDED codes which detect more than one error are included and a general theorem is proved about higher error detecting TDED codes. The hardware needed to implement a TDED system is dealt with and a phase check which checks to see if the decoder is in the in-phase condition is discussed. TDED codes are shown to be usable in most applications which require error detection.