A STUDY OF THE USE OF ERROR-CORRECTING CODES IN BROADCAST TELIDON: FINAL REPORT

DR. BRIAN MORTIMER
NSERC RESEARCH FELLOW

Final Report

February, 1982

DSS Contract No. OSU81-00095

Scientific Authority:

Dr. Mike Sablatash
Communications Research Centre,
Department of Communications,
Ottawa.

Principal Investigator:

Dr. Brian Mortimer
NSERC Research Fellow,
Department of Mathematics and Statistics,
Carleton University, Ottawa

Research Associates:

Dr. Mike Moore
Brian Leroux
Lee Oattes
Tom Ritchford
Department of Mathematics and Statistics,
Carleton University, Ottawa

TABLE OF CONTENTS

information gives a rather detailed picture of how the codes would perform in a variety of error environments.

The problem of decoding was also examined very carefully. Software decoders were implemented in MC6809 machine code for the three codes. Moreover hardware decoders for PRODUCT and CARLETON code were sketched in some detail. Decoding of CARLETON code was shown to be possible well within the engineering constraints.

Further studies were made of the possibility of using long block codes to correct a half-page at a time. The performance of such a code was analyzed both by itself and in conjunction with the prefix Hamming code and the data block code. A preliminary study of decoding was made. It was found that the code choosen (a Reed-Solomon code) gave extremely good performance to the extent of overpowering the rest of the error control scheme.

Future work is needed on the 2 and 3 byte suffices for the data blocks and on a weaker half-page or page code with a quick decoder all of which harmonizes with the rest of the error-correction scheme.

Statement of Work from the Proposal for the Contract

DESCRIPTION

The research to be carried out under this contract will consist of a thorough examination of the use of Error-correcting codes in broadcast Telidon. To-date several codes have been identified as being especially suitable. Theoretical calculations have shown their ability to improve performance. The additional constraints to be considered are the amount of time necessary for decoding, the increase cost of a Telidon terminal which uses the code, and the amount of redundancy introduced into the data by the code. All must be minimized. This research will use both theoretical calculations and simulations to measure these parameters. These simulations will use a microprocessor of the same type as is used by Telidon. Field data of channel error statistics will be used if and when it is available. In their absence a variety of theoretical channel models will be used to exercise the codes.

PURPOSE OR OBJECT

The purpose of this project is to select and analyze a range of ECC for use in improving the performance and extending the range of broadcast Telidon. A careful choice of error-correcting codes and a detailed analysis of their impact on the Telidon system will allow the simultaneous satisfaction of the many constraints; economic, engineering and acceptability for international standards. At the same time, thorough and thoughtful exploration of coding options at this time will allow for a quick response as the Telidon requirements change in the future.

OUTLINE OF MAIN CONCLUSIONS

1) One-byte suffix codes:

a) The CARLETON code gives performance superior to the PRODUCT code specified in the provisional BS-14 on all but one of the model channels considered. When there is a background of white gaussian noise the only situation in which the PRODUCT code makes fewer decoding errors is when there are numerous bursts of length 5-10.

b) It has been shown that CARLETON code can be decoded in 0.97 msec with a 6809 microprocessor at a clock speed of 1.29 MHz using 512 bytes of look-up table. Longer decoding times with less look-up table are also possible. Efficient hardware decoding is possible.

c) The CARLETON code has been shown to be essentially optimal on the white gaussian channel as a one byte data block code which has overall known parity (for better error detection) and few codewords which are short bursts. (for better performance on bursty channels).

d) The SAB code recommended by Seguin, Allard and Bhargava was originally defined for 25 byte data blocks and it is not clear whether a 28 byte version is possible. In any case a code of the SAB type gives near optimal performance on the white gaussian channel but has a serious probability of decoding error when there are short bursts. An effective software decoder for SAB code was developed.

2) Codes for a Half-Page:

a) A 14 byte-error correcting Reed-Solomon code defined on a set of 9 data blocks would virtually eliminate decoding errors at bit error rates $\leq 10^{-3}$.

b) Decoding of the Reed-Solomon code would require more time than the 4 millisecond interval between packets from the same packet so a whole page would have to be captured first then be decoded.

3) The System as a Whole:

a) When using the Prefix code and one-byte data block code together the latter is limiting in terms of delays caused by detected uncorrectable errors.

b) When the Prefix code and Reed-Solomon code are used together (with or without the data-block code) it is the Prefix code which is limiting. To this must be added the decoding delay for the Reed-Solomon code. The conclusion is that the Reed-Solomon code is inappropriate and should be replaced by a less powerful code with a quicker decoder.

ACKNOWLEDGEMENTS

I am happy to take this opportunity to acknowledge the contributions to this report made by my four co-workers on the project: Brian Leroux, Mike Moore, Lee Oattes and Tom Ritchford. They all worked hard on the research presented in the Preliminary Report [2] and their influence on this Final Report can easily be seen. In particular Brian Leroux gave invaluable assistance in the calculation of many of the numbers presented here. It is also with pleasure that I express my appreciation to André Vincent, John Storrey and especially to Mikle Sablatash of the Communications Research Centre for their interest, enthusiasm and assistance throughout this research project. Finally the skill and efficiency of Susan Jameson and Jo-Ann Haynes has greatly eased the production and improved the quality of these reports and is gratefully acknowledged.

Chapter 1. Summary of Results.

1.1 Introduction

The object of the research project reported here (and covered by DSS Contract No. 0SU81-00095) was to select and analyze as thoroughly as possible error-correcting codes which are appropriate for the Canadian Broadcast Telidon system. The lion's share of the work done was reported in the preliminary report "A Study of the Use of Error-correcting Codes in the Canadian Broadcast Telidon System",August,1981 [2].This final report should be viewed as being supplementary and complimentary to that earlier report.

In analyzing the performance of the codes we identify certain possible outcomes of the decoding process. We then calculate the probabilities of these various events occuring under appropriate assumptions. The possible outcomes of the decoding process are the following.

Correct Decoding: the error pattern (if any) that occured was one which the code deals with correctly passing on the actual codeword that was sent.

Decoding Fault: the other case, i.e. an uncorrectable error is corrupting the codeword. The faults are then sub-divided as follows.

Decoding Failure: the error pattern was recognized as an error but the code cannot correct it. The decoder warns the system that the data is falaceous.

Decoding Error: the error pattern occured was mistaken by the decoder for a correctable pattern when it was not. The decoder introduces at least one more bit error and passes the codeword on as if it was correct.

The probability of a correct decoding depends only on the coding strategy (e.g. single error correction, double error correction etc.) used and not on the details of the code. The same applies to decoding faults so these parameters are easiest to calculate. At a fixed bit error rate if the channel is bursty then the errors will corrupt fewer codewords and hence, generally speaking, show an increase in the frequency of correct decoding. Thus the assumption that errors are independent (i.e. white gaussian channel) is reasonable for assessing performance as regards correct decoding and decoding faults.

When we move to the more refined (and interesting) level of analysis which consider three possible outcomes (correct decoding, decoding failure and decoding error) several complications arise. At this level it is the way in

which the code is defined which determines how it will perform. Thus detailed (and hard to get) information about the code is required. Moreover the performance of a given code is much more difficult to predict when the channel is bursty if we are interested in the frequency of decoding errors. In general though, decoding errors are a small fraction of the decoding faults. This must be traded-off against the fact that in the context of a videotex system decoding failures result in delays in delivery of a page to a user while a decoding error results in rubbish on the screen which may be either gross or quite subtle and undetectable by the user as a false page. More will be said below (Sectin 1.3) on the relative frequency of decoding errors and failures.

As a reference point when comparing the performance of a code on a number of channels we assume that the overall bit error rate is constant for all channels. This corresponds to the fact that overall bit error rate tends to be the first (and often only) parameter relevant to error patterns, which is measured on a communications channel. An alternate assumption would be that a burst noise phenomenon is added onto a background of random errors. The background bit error rate would then be held constant. We tried this second approach with the new channel model of Chapter 2,

Section 3. The results obtained didn't show a significant change over the previous assumption.

The unit of the broadcast videotex system (as specified in the provisional version of BS-14 [1]) which is of interest for coding purposes is the data packet. Each packet consists of 33 bytes. There are 5 prefix bytes then a series of (28-S) data bytes and finally S suffix bytes. Here S may be 0,1,2, or 3 and is specified by two of the bits in the prefix. We deal with the various parts of this packet one at a time.

1.2 The Prefix.

The five prefix bytes are each encoded with an odd-parity variant of the (8,4) Hamming code as specified in Appendix B of BS-14. Thus if one of these bytes is (b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1) then the bits b_8, b_6, b_4, b_2 carry information while the other four bits are check bits. The code is defined so that (arithmetic mod 2):

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_8 \\ b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Here $\overline{b_2}$ is the complement of b_2; that is $\overline{b_2} = 1-b_2$. Complementing this bit results in an odd parity byte. (For purposes of error-correction performance calculations we leave the bytes with even parity since this gives us a linear code).

Assuming that errors arise independently with probability p, the probability of a correction reception for one such byte is q^8 where $q = 1-p$. The probability of a single error in a byte is $8pq^7$. Thus the overall probability of a prefix being correctly decoded is $(q^8 + 8pq^7)^5$. The probability of decoding fault is then $1-(q^8 + 8pq^7)^5$. Again assuming independent errors, the probability of decoding error can be estimated. Strictly in terms of the Hamming code this is,

$$P_{DE} = 4A_4p^3q^5+A_4p^4q^4+4A_4p^5q^3+ 8A_8\ p^7q + A_8p^8$$
$$= 56p^3q^5 + 14p^4q^4 + 56p^5q^3 + 8p^7q + p^8$$

for a single byte and $1-(1-P_{DE})^5$ for 5 bytes. On the other hand the videotex decoder will not accept all decoding errors in the prefix as valid messages. The five bytes are interpreted as indicated in Figure 1.1.

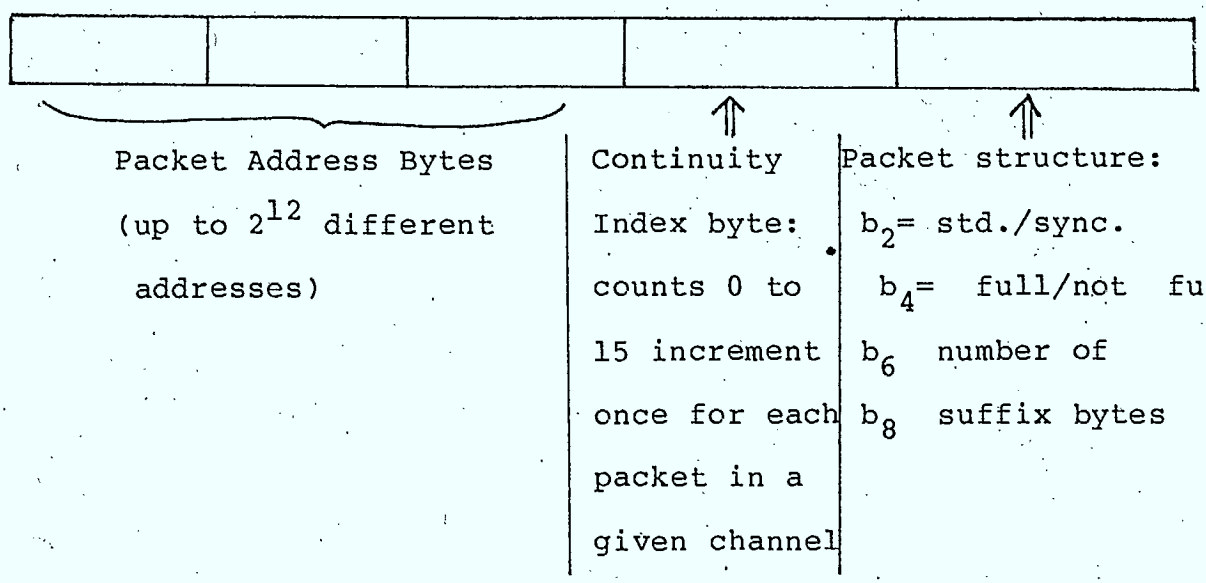


Figure 1.1 The Packet Prefix

The first three bytes specify a channel number then the fourth byte is a continuity index count on the data packets of that channel as they are broadcast. A decoding error in the first four bytes must change the continuity index by the exact quantity required to make it the next packet from the new channel (i.e. the channel specified after the errors are introduced). Moreover, either the next packet actually transmitted on the new channel must be lost or all subsequent packets must have their continuity index incremented by one by a further decoding error. The probability of such an event is negligible (say, $< 10^{-17}$ at BER = 10^{-3} assuming 1000 different channels in use).

Thus the only decoding errors that might occur are those affecting only the packet structure byte. Again many

of these decoding errors will be caught by the decoder because of other ambiguities in the subsequent processing of the packet. Thus we can take as a very rough upper bound on the probability of a decoding error coming from the prefix the estimate $56p^3q^7$. Careful implementation of the videotex decoder would reduce this much further.

1.3 Coding the Data Block

The provisional version of BS-14 [1] specifies that 0,1,2 or 3 suffix bytes may be appended to the data block of a data packet to be used for error correction. We have not yet looked seriously at the cases of 2 or 3 suffix bytes and in fact BS-14 leaves specification of these bytes to the future.

If no suffix bytes are used then error correction within the data block is impossible. If an odd number of errors corrupt one of the bytes then these errors will be detected. In an environment of random errors with bit error rate p (and q =1-p), the probability of correct decoding is just q^{224} while the probability of a decoding error is approximately

$$28\binom{8}{2}p^2q^{222} + \left[28\binom{8}{4} + \binom{28}{2}\binom{8}{2}^2\right] p^4q^{220}.$$

Explicitly this is $784p^2q^{222} + 892584p^4q^{220}$.

The case of a one-byte suffix has been intensively studied by a numer of research groups [2],[3],[4],[5]. The provisional version of BS-14 specifies a simple but effective PRODUCT code for this purpose [1], [4]. Allard Bhargava and Seguin [3] suggested another code which they called SAB. This code was defined for data packets of 30 bytes and it is not clear that it can be extended to 33 bytes in a reasonable way. The SAB code is very good for a channel with random errors but has a high probability of making a decoding error on a short burst. A third code was suggested in [2], [5] by our group. Giving it the (temporary) name CARLETON code the preliminary report [2] presented an exhaustive comparison of this code with SAB code and PRODUCT code. The assumed number of bytes in data block plus suffix in [2] was 25. Since BS-14 specifies 28 the results of that preliminary report have had to be updated here. They are presented in Chapter 2. Since the SAB code cannot be immediately extended to these longer data blocks it was dropped from the discussion. PRODUCT code and CARLETON code have essentially the same probability of

decoding failure on any channel (and hence would produce delays with the same frequency). When we look at decoding errors we find that the expected number of pages (defined as units of 18 packets) before a decoding error occurs is five times larger for CARLETON code than for PRODUCT code on the white gaussian channel. Looking at bursty channels with a background of white gaussian noise we found that this superiority persists. The best tool to use for seeing this is the new channel model of Chapter 2, Section 3. There we observe that, unless the channel is systematically introducing pathologies, the white gaussian performance characteristics of th codes persist as the channel becomes more bursty until the peformance quite abruptly degenerates to a common (poor) value for both codes (Figure 2.4.2).

We have also shown that Carleton code be decoded by a Motorolla 6809 microprocessor using 1254 machine cycles which represents 0.97 milliseconds at a 1.29 MHz clock speed using a look-up table of 512 bytes. In fact the PRODUCT code can be decoded in a similar way using approximately the same amount of time and look-up storage.

Chapter 2 is devoted entirely to the case of a one byte suffix. We conclude the discussion here with a few remarks on "optimality". By optimality we mean minimal

probability of decoding error. It can be asked: what is the optimal choice for a code using a one byte suffix? The answer is that "optimal" can only be applied to a code working on a particular channel. The historical approach seems to have been that a code is choosen to work well on a white gaussian channel and other channels are either ignored or treated incidentally. Optimizing a code on the white gaussian channel translates simply to the minimization of the number of low weight codewords. For any channel, in general, to obtain an optimal code with respect to decoding errors select a code with a minimal number of codewords which occur among the most common class of error patterns on the channel. So really you have to have a specific channel before you can talk about optimization.

Consider now the special case of a white gaussian channel. Thus the errors in the channel arise as independent events. Let A_i stand for the number of codewords of weight i in some one-byte data block code. Certainly we would take a single error correcting code so $A_1=A_2=0$. In order to detect all double errors we take $A_3=0$. Can we have $A_4=0$? No. This means that double error correction is not possible with only one byte in the suffix. How small can we make A_4? Chapter 4 of the preliminary report [2] was devoted to this question. We will up-date those results here to the

case of data packets of 33 bytes (data blocks of 27 bytes plus 1 suffix byte) as follows:

(i) for an arbitrary one byte suffix code: $A_4 \geq 285$

(ii) if the code has only codewords of known parity: $A_4 \geq 676$

(iii) if the code has no weight 4 codewords with all ones confined to a single byte: $A_4 \geq 853$

(iv) if the code satisfies (ii) and (iii): $A_4 \geq 2028$

The first case, (i), gives a bound on how good a code we can expect t. find for the white gaussian channel. The other cases deal with added conditions which would give th code vauable characteristics on other channels. CARLETON code has $A_4 = 2{,}154$ and satisfies (iv). PRODUCT code also satisfies (iv) and has $A_4 = 10{,}584$. So sticking to codes with the properties (iv) we can't improve much (for the white gaussian channel) on CARLETON code. In order to get a better code for the white gaussian channel we must weaken its ability to detect bursts by either allowing arbitrary overall parity (not (ii)) or allowing burst-like weight 4 codewords (not (iii)).

It is also essential to realize that there is a trade-off between decoding failures and errors. The probabilities of correct decoding P_{CD}, decoding failure P_{DF} and of decoding errors P_{DE} satisfy the simple

relation

$$P_{CD} + P_{DF} + P_{DE} = 1$$

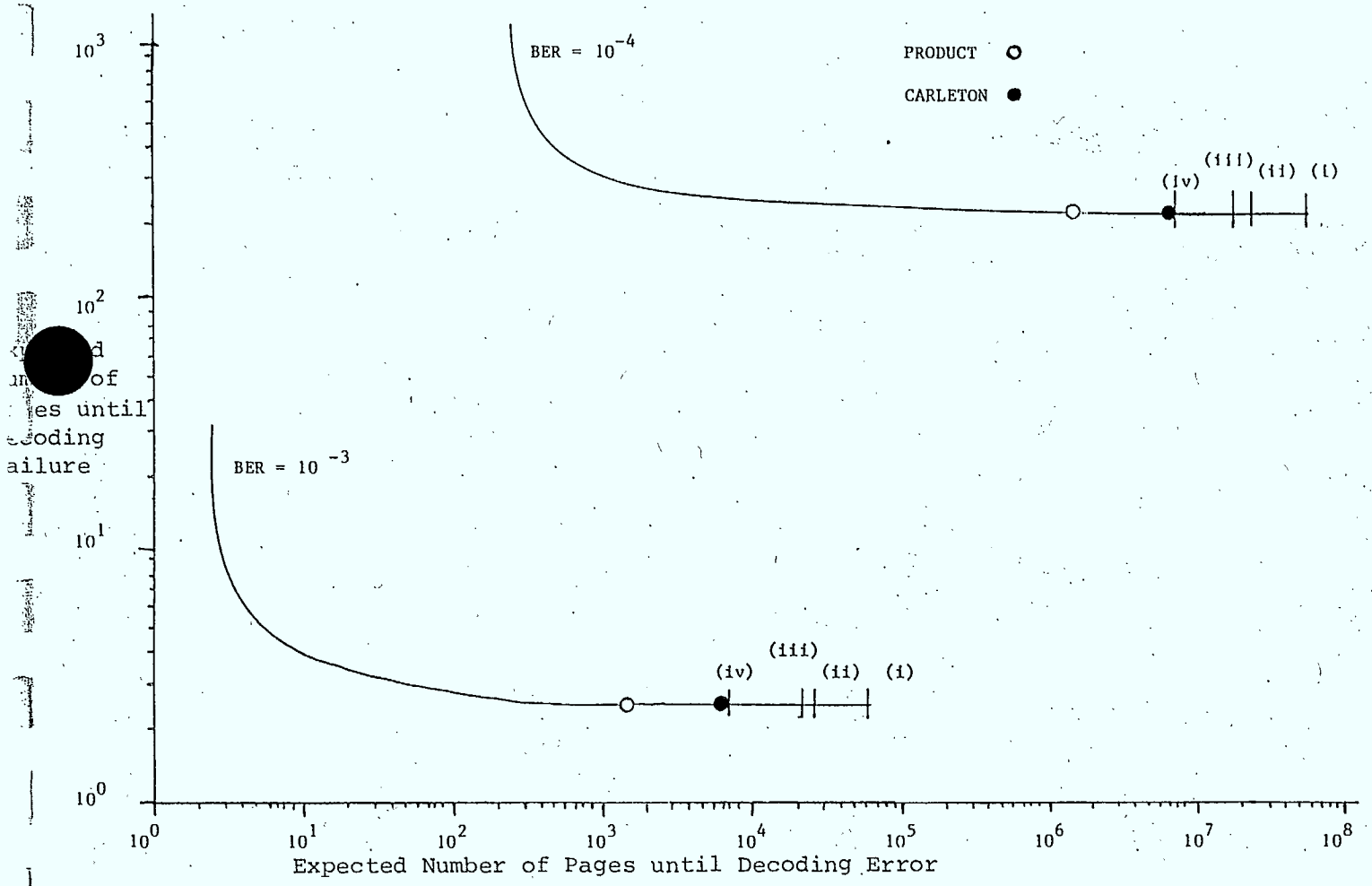


FIGURE 1.2 Expected Number of Pages until Decoding Failure/Error for Single Error Correcting Codes on the White Gaussian Channel at Bit Error Rate 10^{-3} and 10^{-4}.

since they represent all the possible outcomes of the decoding process. Now P_{CD} depends only on the frequency with which error free blocks and correctable error patterns arise in the channel. So P_{CD} is the same on any given channel for, say, all single eror correcting codes. Consider the case of the white gaussian channel and single error correcting codes of length n. Then $P_{CD} = q^n + npq^{n-1}$ where p is the bit error rate and $q = 1-p$. Therefore $P_{DF} + P_{DE}$ is a constant for all such codes. Now we take $B_F = 1/(18\ P_{DF})$ and $B_E = 1/(18\ P_{DE})$ which are respectivey the expected number of pages (= 18 data packets) until decoding failure and until a decoding error. We see that $(1/B_F + (1/B_E)$ is a constant so the graph of the points (B_E, B_F) is a hyperbola. Thus we cannot increase B_E without decreasing B_F. Moreover no code has simultaneously good performance as regards both decoding failures and errors. Figure 1.2 is a plot of B_F against B_E for bit error rates BER = 10^{-3} and 10^{-4} for the white gaussian channel. The points representing several codes are marked. Also included are the points for imaginary codes which satisy the bounds (i), (ii), (iii), and (iv) above. While keeping B_E acceptably large we may be forced to make B_F unacceptably small. The only remedy is to do more

correction. This has the effect of reducing the size of the constant C in the formula

$$\frac{1}{B_F} + \frac{1}{B_E} = C \ .$$

which leads us back to the old problem. Which error patterns are the most common? These are the ones which should be corrected first. But these patterns will depend on the channel. Unless we have explicit information about the actual broadcast videotex channels we are more likely to be led astray by further correction than directly to better performance. These remarks apply of course to the choice of 2 and 3 suffix byte codes as well.

1.4 Codes on a Page and Putting the System Together.

We have discussed the prefix and data block codes and now turn to larger units containing a number of data packets. The strategy is to use a powerful code of greater length to cover a set of say 8 data blocks by adding a cmplete data packet of redundancy. Decoding would be too complex to be done in software in the statutory 4 milliseconds allowed for "on-line" processing of incoming data packets. The whole videotex page would be extracted from the channel first and it could then be decoded "off line". The idea is to use a code which corrects so many

errors that only extremely rarely will it be necessary to re-extract a data block from the channel. Thus an extra, say, 2 seconds would be required to deliver the picture to the screen but that picture could be virtually guaranteed to be error free. The fact that such a code was in use could be specified by a Record Type specification in the Record Header.

We have examined one such code, a Reed-Solomon code over the field GF(256). This code can correct up to 14 byte errors in a set of 9 data blocks . These could be 14 random bit errors or a continuous run of 98 bit errors. Decoding was not exhaustively examined but can be accomplished in a matter of a second or two by a software decoder.

Putting all this together we will show in Chapter 4 that this Reed-Solomon code overpowers the prefix code in the sense that it provides very much better performance than the prefix can deliver. We conclude that a different code would be more effective and such a code should be sought out and analyzed.

Chapter 2. One-byte Data Block Codes

2.1 Introduction

The provisional issue of BS-14 "Television Broadcast Videotex" [1] stipulates that a suffix of 0, 1, 2 or 3 bytes is added to each data block to be used for error correction and detection, (the number of such bytes being specified by the bits B6 and B8 of the Packet Structure Byte). Together they form what we will call a data block code. This chapter is devoted to a discussion of codes appropriate to the case of a one-byte suffix. Since the Preliminary Report [2] submitted in August 1981, consists of a thorough analysis of this problem we will present here information which is supplementary to that Report. In particular we move the discussion to the case of longer data blocks of 27 bytes plus a suffix byte as specified by BS-14. Also we present hardware versions of our decoders, a software encoder and a new burst channel model.

2.2 The Codes Defined.

Two particular codes are the main object of discussion in this Section, namely PRODUCT and CARLETON codes. The first is a simple combinatorially defined code which is specified in BS-14 (paragraph 3.3) as the code used to define a single suffix byte. CARLETON code is

an algebraically defined code whose definition (first given in [2] and [5]) contains the pseudo-random element inherent in using a finite field to give the code better behavior. Both these codes are single error correcting and use the fact that the data bytes have odd parity. The so-called SAB code which was defined in [3] as a possible code for Telidon and was compared with PRODUCT and CARLETON codes in [2] is not discussed in this report. This is principally because it is far from transparent whether the SAB code can be extended to 28 bytes in a sensible way.

2.2.1 PRODUCT Code

The PRODUCT code is the simplest species of the genus of product codes. It uses an (odd) parity check on bytes and a longditudinal (odd) parity check across the bytes to correct single errors. Write the data blocks as $B_i = (b_{i8}, b_{i7}, \ldots, b_{i1})$ for $i = 1,\ldots,27$ and let B_{28} represent the suffix byte. Then with additions taken mod 2 (exclusive -or) we define the eighth bit of each block by

$$b_{i8} = 1 + b_{i7} + b_{i6} + \ldots + b_{i1}$$

for $i = 1,\ldots,27$ and define the suffix byte by

$$b_{28j} = 1 + b_{1j} + \ldots + b_{27j}$$

for $j = 1,\ldots,8$. Since there are an odd number of data bytes of odd parity the check byte also has odd parity automatically.

The use of odd parity is a nuisance to the coding theorist since it is a non-linear condition and removes the equivalence of minimum weight and minimum distance. Thus we will always consider the even parity version of the PRODUCT code (so the ones are removed from the equations above). This even-parity code has the same distance structure as the odd parity version (and hence performs in the same way) but is a linear code and hence its analysis is easier to describe.

For additional information and discussion of this code consult [4] and the literature cited there.

2.2.2 CARLETON Code

The CARLETON code is an algebraically defined code which is allied to a shortened Hamming code but makes use of the odd parity of the data bytes B_i. (It was first defined in March, 1981 during an attempt to show that the PRODUCT code is optimal as a data block code on the white Gaussian channe; which it isn't in fact.)

We first give a terse if somewhat mysterious and unmotivated definition then present an algebraic definition.

The suffix byte B_{28} of CARLETON code is obtained by successively adding each of the 27 data bytes to an accumulator $(a_8, a_7, \ldots, a_1)$, which starts off set to zeroes. After each addition the bits of the accumulator are transformed according to the following rules (sums are mod 2):

$$a_1' = a_1 + a_2 + a_6 + a_8$$

$$a_2' = a_1 + a_3 + a_6$$

$$a_3' = a_2 + a_3 + a_4 + a_7$$

$$a_4' = a_1 + a_4 + a_5 + a_8$$

$$a_5' = a_5 + a_7 + a_8$$

$$a_6' = a_2 + a_7$$

$$a_7' = a_1 + a_3 + a_4 + a_8$$

$$a_8' = a_1 + a_5 + a_6 + a_7 + a_8 .$$

After the 27th byte has been processed the accumulator contains the suffix byte B_{28}.

The transformation of the accumulator involves each bit an odd number of times. Therefore it preserves the parity of the accumulator.

The accumulator starts with even parity and the effect of the additions and transformations is to change this parity 27 times. Thus B_{28} has odd parity automatically.

Where did the transformation rules come from? Let GF(128) denote the field of $2^7 = 128$ elements. The non-zero elements of this field form a cyclic group which can be generated by α where α is a root of $X^7 + X^3 + 1$ over the field GF(2) of two elements. Thus $\alpha^7 = \alpha^3 + 1$ in GF(128) and the powers α^i run through all the non-zero field elements with period 127. If a codeword of CARLETON code is written as a bit string as $(B_1, \ldots, B_{28}) = (C_{223}, C_{222}, \ldots, C_1, C_0)$ then the code is so defined that

$$\sum_{i=0}^{223} C_i \alpha^i = 0 .$$

If the ith byte is written $B_i = (b_{i7}, b_{i6}, \ldots, b_{i0})$ this is

$$\sum_{i=1}^{28} \sum_{j=0}^{7} b_{ij} \alpha^{224-8i+j} = 0$$

More explicitly this is

$$(b_{17}\alpha^{223} + \ldots + b_{10}\alpha^{216}) + \ldots + (b_{28,7}\alpha^7 + \ldots + b_{28,0}\alpha^0) = 0 .$$

We write this equation in the form

$$\begin{aligned} & (b_{1,7}\alpha^7 + b_{1,6}\alpha^6 + \ldots + b_{1,0}\alpha^0)\,\alpha^{8\cdot 27} \\ + \; & (b_{2,7}\alpha^7 + \ldots\ldots\ldots\ldots + b_{2,0}\alpha^0)\,\alpha^{8\cdot 26} \\ + \; & \ldots\ldots \\ + \; & (b_{28,1}\alpha^7 + \ldots\ldots\ldots\ldots + b_{28,0}\alpha^0)\,\alpha^{8\cdot 0} = 0 . \end{aligned}$$

Given the first 27 bytes we wish to select the bits of the 28th byte so that this equation is valid with the parity of B_{28} odd.

Each byte B_i represents an element of the field GF(128), namely $b_{i7}\alpha^7 + \ldots + b_{i0}\alpha^0$, which we again denote by B_i. We then see that the check (suffix) byte is determined by the above equation,

$$B_{28} = B_1(\alpha^8)^{27} + B_2(\alpha^8)^{26} + \ldots + B_{27}(\alpha^8)^1$$

$$= (\ldots((B_1\alpha^8) + B_2)\alpha^8 + \ldots + B_{27})\alpha^8 \quad \ldots (*)$$

In fact each field element x has two representations as $x = u_7\alpha^7 + u_6\alpha^6 + \ldots + u_1\alpha^1 + u_0$ since $0 = \alpha^7 + \alpha^3 + \alpha^0$. One representation has an odd number of ones (non-zero coefficients) and the other has an even number of ones. Therefore B_{28} can be represented by an odd parity byte and we have found the suffix byte.

The "Horner's method" definition of B_{28} ((*) above) is the expression used in the first definition of Carleton code. The transformation specified in that definition is just "multiplication by α^8". As an 8×8 binary matrix the "multiply by α^8" transformation has rows equal to $\alpha^{15}, \alpha^{14}, \ldots, \alpha^8$. Using odd parity representations for these field elements the matrix is

$$(\text{mult. by } \alpha^8) = \begin{bmatrix} \alpha^{15} \\ \alpha^{14} \\ \vdots \\ \alpha^9 \\ \alpha^8 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} .$$

This matrix then determines the linear relations in our first definition.

As with the PRODUCT code we actually work with even parity bytes in CARLETON code when carrying out performance calculations.

2.3 PERFORMANCE

The main thrust of our progress report [2] was the analysis of the relative performance of PRODUCT, CARLETON and SAB codes. We have repeated the calculations for the first two codes extended to a length of 28 bytes and will report our results here. For details on methods of calculations etc. the reader is referred to the earlier report [2].

One measure used to assess the codes is the probability that a burst of length b results in decoding error. Here a burst is a continuous string of b bits each of which is in error with probability $\frac{1}{2}$. For PRODUCT and CARLETON codes these probabilities

are independent of the length of a code so no new calculations are necessary. Thus Figure 2.3.1 is repeated from the Progress Report unchanged. The actual numbers are given in Table 2.3.1 and are used in the channel model of Section 2.4 below.

The probability of a decoding error (or decoding failure) on a variety of model channels was the other performance measure used. This requires (in the absence of field data) that quite arbitrary assumptions be made about the nature of the error patterns in the channel so that the calculations can actually be carried out. Hence a range of channels was used. The channels are described briefly below (and in more detail in Appendix 2D). The results appear in Table 2.3.2. Note that the Burst Channel #2 of the Progress Report has been removed from those considered. Each channel depends on a choice of parameters and this choice was always made so that the overall bit error rate would be one of 10^{-3}, 10^{-4}, 10^{-5} or 10^{-6}.

White Gaussian Channel.

The errors occur as independent events with a fixed probability of 10^{-3}, 10^{-4}, 10^{-5} or 10^{-6}.

Burst Channel 1.

This channel is of the type introduced by Gilbert in Reference [6]. The channel has two states; "good" and "bad". In the "good"

FIGURE 2.3.1 The Probability that a Burst Causes a Decoding Error

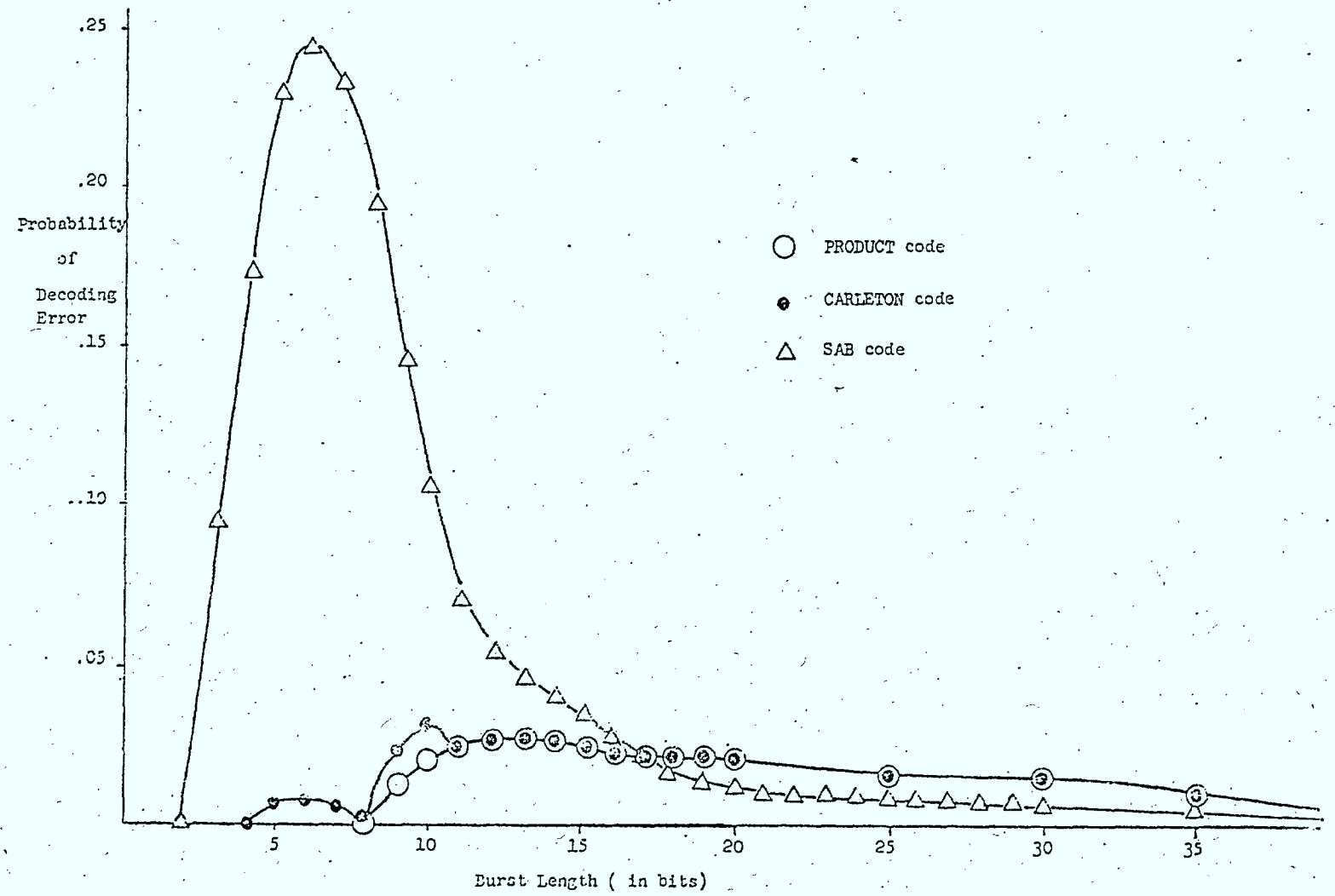


TABLE 2.3.1 The Probability that a Burst of Length b Causes Decoding Error

b	PRODUCT code	CARLETON code
3	0	0
4	0	0
5	0	.00781
6	0	.00781
7	0	.00586
8	0	.00122
9	.0137	.0237
10	.0213	.0309
11	.0251	
12	.0257	
13	.0271	
14	.0268	
15	.0261	
16	.0257	
17	.0243	
18	.0233	
19	.0223	
20	.0213	
25	.0161	
30	.0123	
35	.0090	
40	.0066	
45	.0047	
50	.0033	

	Bit Error Rate	Code	Channels: White Gaussian	Burst Channel #1	Burst Channel #2	Burst Channel #3	Burst Channel #4	Burst Channel #5
Expected No. of Pages until Decoding Error	10^{-3}	PRODUCT	1.61×10^{3}	154	90	337	43	1.74×10^{3}
		CARLETON	8.04×10^{3}	755	451	1660	254	8.57×10^{3}
	10^{-4}	PRODUCT	1.4×10^{6}	1.28×10^{4}	7.81×10^{3}	1.12×10^{5}	3.81×10^{3}	1.46×10^{6}
		CARLETON	7.0×10^{6}	6.30×10^{4}	3.94×10^{4}	5.49×10^{5}	2.26×10^{4}	7.18×10^{6}
	10^{-5}	PRODUCT	1.3×10^{9}	1.27×10^{6}	7.70×10^{5}	4.32×10^{7}	3.77×10^{5}	1.15×10^{10}
		CARLETON	6.4×10^{9}	6.22×10^{6}	3.89×10^{6}	2.12×10^{8}	2.23×10^{6}	5.64×10^{10}
	10^{-6}	PRODUCT	1.3×10^{12}	1.27×10^{8}	7.69×10^{7}	1.68×10^{10}	3.76×10^{7}	1.19×10^{13}
		CARLETON	6.4×10^{12}	6.21×10^{8}	3.88×10^{8}	8.22×10^{10}	2.23×10^{8}	6.25×10^{13}
Expected No. of Pages until Decoding Failure	10^{-3}	The same for both codes	2.6	2	2.4	3.4	1.4	3.7
	10^{-4}		226	16	30	197	9	284
	10^{-5}		2.2×10^{4}	165	308	1.28×10^{4}	84	1.12×10^{5}
	10^{-6}		2.2×10^{6}	1653	3100	6.83×10^{5}	838	4.03×10^{6}

TABLE 2.3.2 Performance Parameters on a Variety of Model Channels

state errors occur as in White Gaussian channel. In the "bad" state every bit is in error. The result is a channel with numerous short bursts (lengths 2 and 3 being the most popular).

Burst Channel 2.

This channel is a Markov chain with a one bit memory. Again it is characterized by short bursts (lengths 3 and 4 dominating.)

Burst Channel 3.

This channel is based on a Pareto distributions (see Reference 7). Its most frequent burst lengths are 5,6 and 7.

Burst Channel 4.

This channel is a White Gaussian channel in which some phenomenon converts half the isolated single errors into consecutive double errors. It has a curious distribution of burst lengths with even length bursts favoured over odd.

Burst Channel 5.

This channel has occasional long bursts (of about 10 bits) and is approximately White Gaussian the rest of the time. The collection of the errors into these longer bursts at a fixed bit error rate effectively cleans up large stretches between the bursts. The result is a channel on which codes behave largely as they would on a White Gaussian channel with a bit error rate much less than the observed bit error rate.

The low-weight distributions of the codes is given in Table 2.3.3. There A_i is the number of codewords of weight i.

	A_4	A_5	A_6	A_7
PRODUCT	10,584	0	1,100,736	0
CARLETON	2,154	0	622,733	0

Table 2.3.3 Low Weight Distribution of the Codes .

2.4 A New Burst Channel Model

One of the problems with our earlier burst-error model channels is that the bit error rate within a burst is too high (generally it is 1 or close to 1). This drawback is overcome by the channel model we now describe. We assume that each data block (i.e. codeword) is received in either good or bad condition. In the good state the errors are independent events while in the bad state a burst error of some length b occurs (and all other bits are correct). The necessary parameters of the channel model and code are the following:

P_G = probability that a good block is received,

$P_B = 1 - P_G$ = probability that a bad block is received,

σ = bit error rate in a good block,

$p(b)$ = probability that a bad block contains a burst of length b,

$E_B(b)$ = probability that the code being used turns a burst of length b into a decoding error,

$E_G(\sigma)$ = probability of decoding error for the code being used on a white gaussian channel with bit error rate σ.

We assume that there is no correlation between the states of successive blocks. We also assume that the bits of a good block outside the burst are correct and that inside the burst the bit error rate is $\frac{1}{2}$ with all 2^b burst error patterns equally likely to arise. Writing n for the length of the code the overall bit error rate is

$$BER = P_G\sigma + P_B \sum_b \frac{b}{2n} p(b) .$$

The probabilities of decoding error and of correct decoding are.

$$P_{DE} = P_G \cdot E_G(\sigma) + P_B \cdot \sum_b p(b) E_B(b)$$

$$P_{CD} = P_G((1-\sigma)^n + n\sigma(1-\sigma)^{n-1}) + P_B \sum_b \frac{b+1}{2^b} .$$

The parameters $E_B(b)$ are in fact numbers calculated earlier. They appear in Table 2.3.1 and Figure 2.3.1. The parameters $E_G(\sigma)$ is easily calculated from the weight distribution of the code. Yet to be determined are σ, P_G and the values $p(b)$ for $b = 1,2,\ldots$. The value of σ is determined from the others by fixing the over-

all bit error rate (at say 10^{-3}, 10^{-4}, 10^{-5} or 10^{-6}) and solving the bit error rate equation for σ. Since $0 \leq \sigma \leq 1$ not all choices for P_G are feasible.

Knowing the distribution $E_B(b)$ in advance means that we can produce almost any result by choosing the distribution $p(b)$ appropriately. In comparing CARLETON and PRODUCT Codes the question is: does the better performance of CARLETON code on the good blocks compensate for its relative weakness on the bad blocks with bursts of lengths 4 to 10? Or more to the point, what happens to the performance as the channel becomes more bursty i.e. as P_B increases?

Three distributions $p(b)$ have been used to demonstrate this model. We call them Channels A, B and C, and they are plotted in Figure 2.4.1. This channels have B.E.R. = 10^{-3}. As we move from Channel A to Channel C there is an increasing trend to longer bursts. Note that Channel A has bursts only in the range ($b = 5,6,7,8$) in which CARLETON code produces decoding errors and PRODUCT code does not. Hence, this channel produces a rather remarkable curve in Figure 2.4.2 where the mean number of (18 data block) pages until decoding error is plotted against the probability P_B that a block contains a burst error. Once we move the preponderance of bursts to longer lengths as in Channels B and C we see a consistent shape

Figure 2.4.1 New Channel Model :

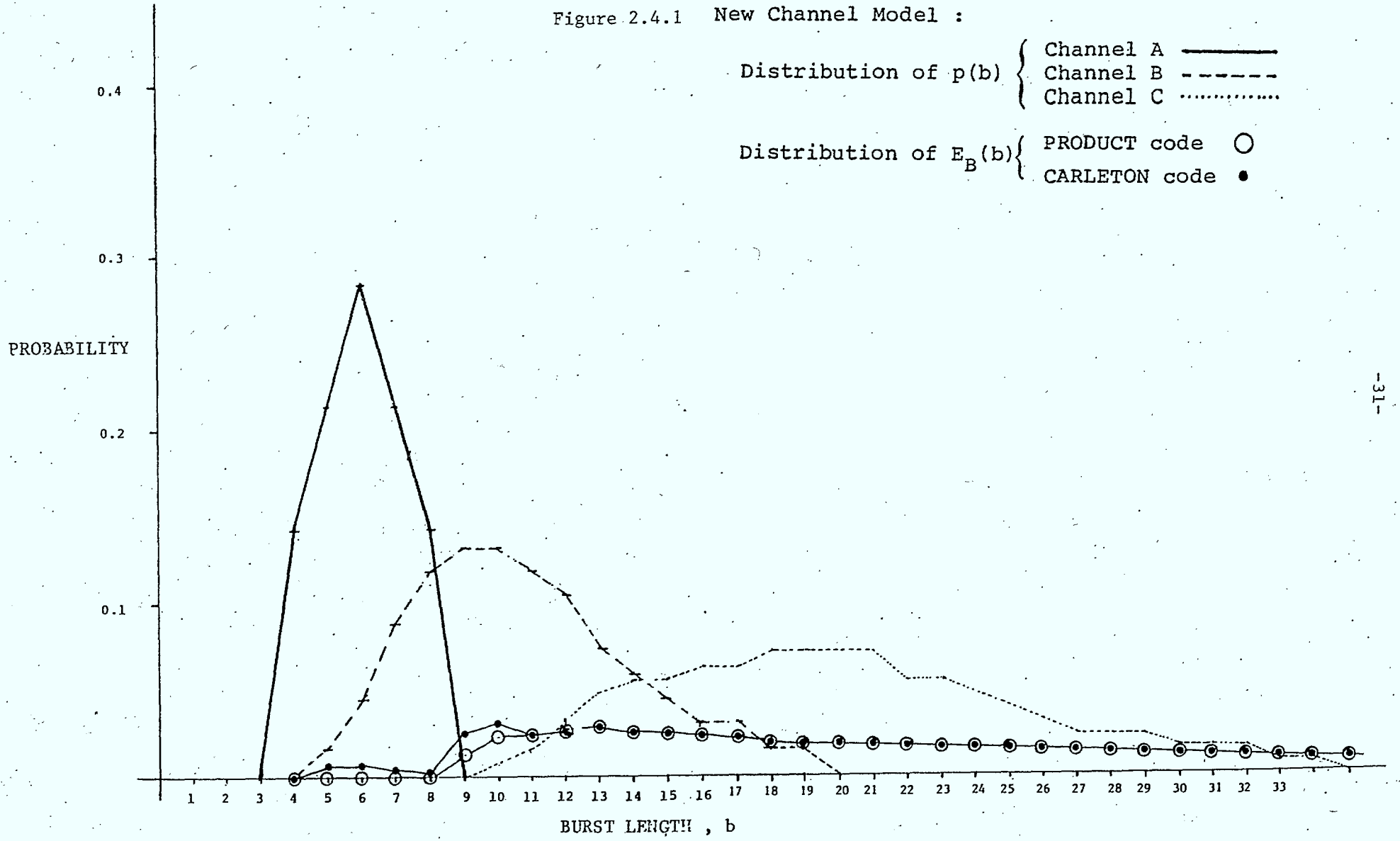
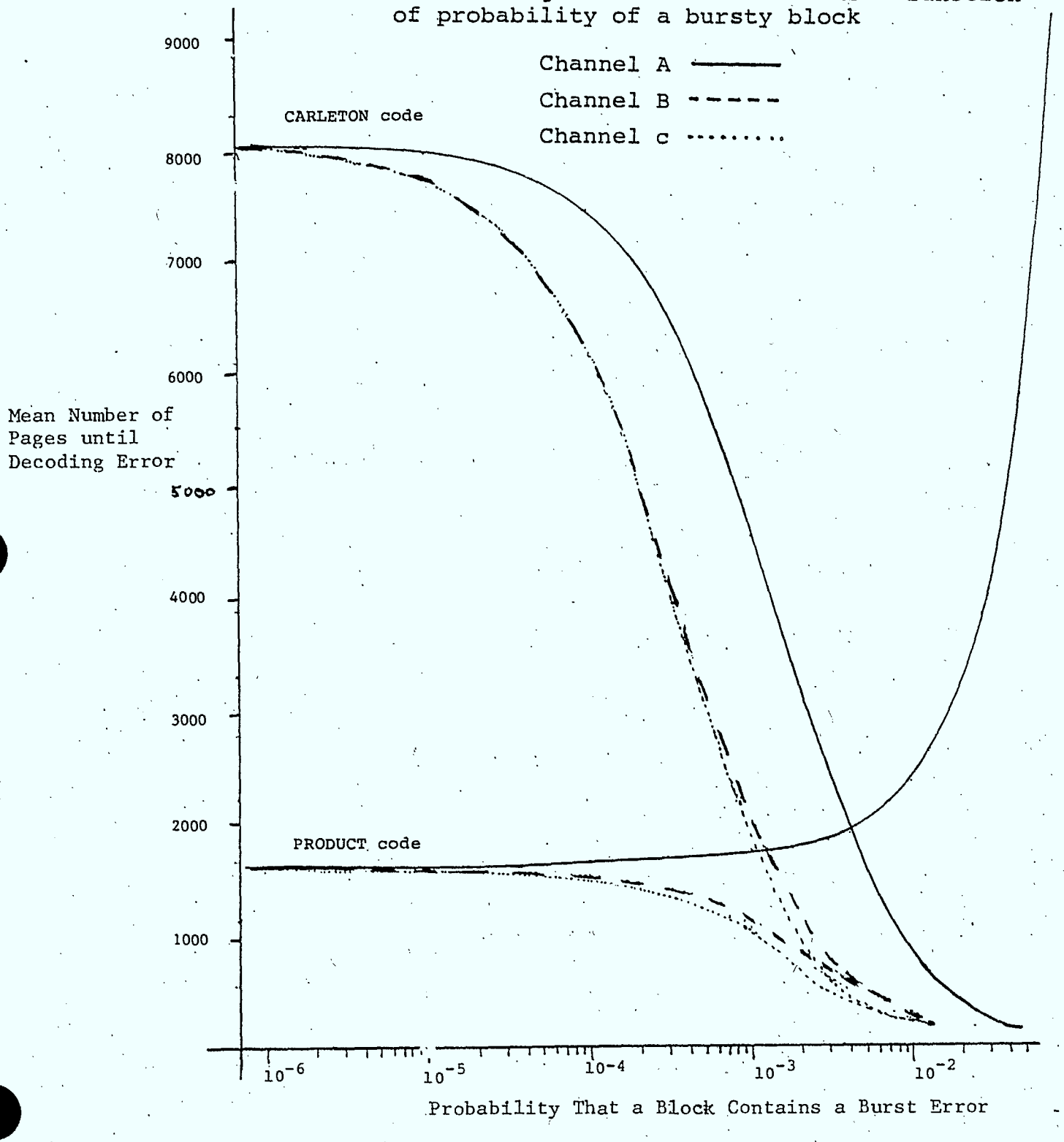


Figure 2.4.2

New Channel Model : mean numbr of pages to Decoding Error at BER=10^{-3} as a function of probability of a bursty block



for the curves. Thus as P_B increases the mean number of pages, until decoding error declines from the white gaussian values (for say $P_B < 10^{-5}$) to a matter of a few hundred when P_B is 10^{-3} to 10^{-2}.

The behavior of the codes is easy to explain. When P_B is small the channel approximates white gaussian closely. When P_B is large more errors occur in bursts and fewer as independent events. Thus the superiority of CARLETON code is lost in the roughly equivalent abilities of the two codes with bursts. In the case of Channel A we have a curious anomally. This channel never has a burst of errors which outwits PRODUCT code. Hence as P_B increases PRODUCT code makes fewer decoding errors on this channel.

This discussion is made simply to indicate the flexibility of the model and the intimate interdependence of performance with error pattern frequencies. This channel was developed to give a more accurate model of burst channel performance but has the additional virtue that it would be straight forward to fit this model to field data. It is ready and waiting if any such data every appear.

2.5 Decoding

The software decoding of both PRODUCT and CARLETON codes was discussed in [2]. As the algorithms given there were implemented for the shorter 25 byte data blocks we have included here in an appendix implementations of the software decoding algorithm for the length 28 bytes case of CARLETON code. Again these are written in 6809 assembler. The time to decode a single data block with this algorithm is at most 1254 machine cycles (msec. at 1.29 M/hz), for CARLETON. Again this algorithm uses 511 bytes of look-up table storage. The decoding time for CARLETON code by the algorithm given here is .97 msec.

Hardware decoding of both PRODUCT and CARLETON code is straightforward. We describe below hardware decoders for these codes to a reasonably detailed level. Of course the actual form of decoding uses in any particular videotex terminal will depend very much on the overall architecture of the terminal. Hence there isn't much point in refining the design completely. The hardware decoder is developed only for enough to show that its complexity is not extreme and implementation can be handled in various ways.

2.5.1 A Hardware Decoder for Carleton Code

This decoder will correct a single error or declares a decoding failure in 448 clock cycles. These cycles are divided into an input phase of 224 cycles and an equal output phase. The decoder is working in a bit-serial mode with the suffix byte last.

During input the incoming bits are counted off into bytes and the parity of each byte is checked. The bytes are counted from 0 to 27 and when a parity failure in a byte is detected we record the byte number in the set of 5 flip-flops. Simultaneously we count the number of byte failures by passing a 1 down a 2 stage shift-register. This allows us to identify whether there were 0,1 or more than 1 byte parity failures. At the same time a feedback shift register is calculating the algebraic syndrome (by multiplying by a root α of $X^7 + X^3 + 1$).

After 224 clock cycles the entire received string has been clocked into the data buffer. We now clock it out for 224 cycles. First the swtich A is opened to preserve the number of the last byte to show a parity failure. The five adders then compare this byte number with the number of the byte now leaving the data buffer. The NOR gate identifies a match.

The algebraic syndrome register continues to cycle during the

output phase. Two logic circuits decide whether the contents of this register is non-zero and whether it is the representation of $\alpha^{223} = (0,0,0,1,1,1,1)$. This last condition means that a single error in the bit now leaving the circuit would have given the same syndrome. Then if the byte number matches the last one with a parity failure and there was only one byte with a parity failure then we correct the bit.

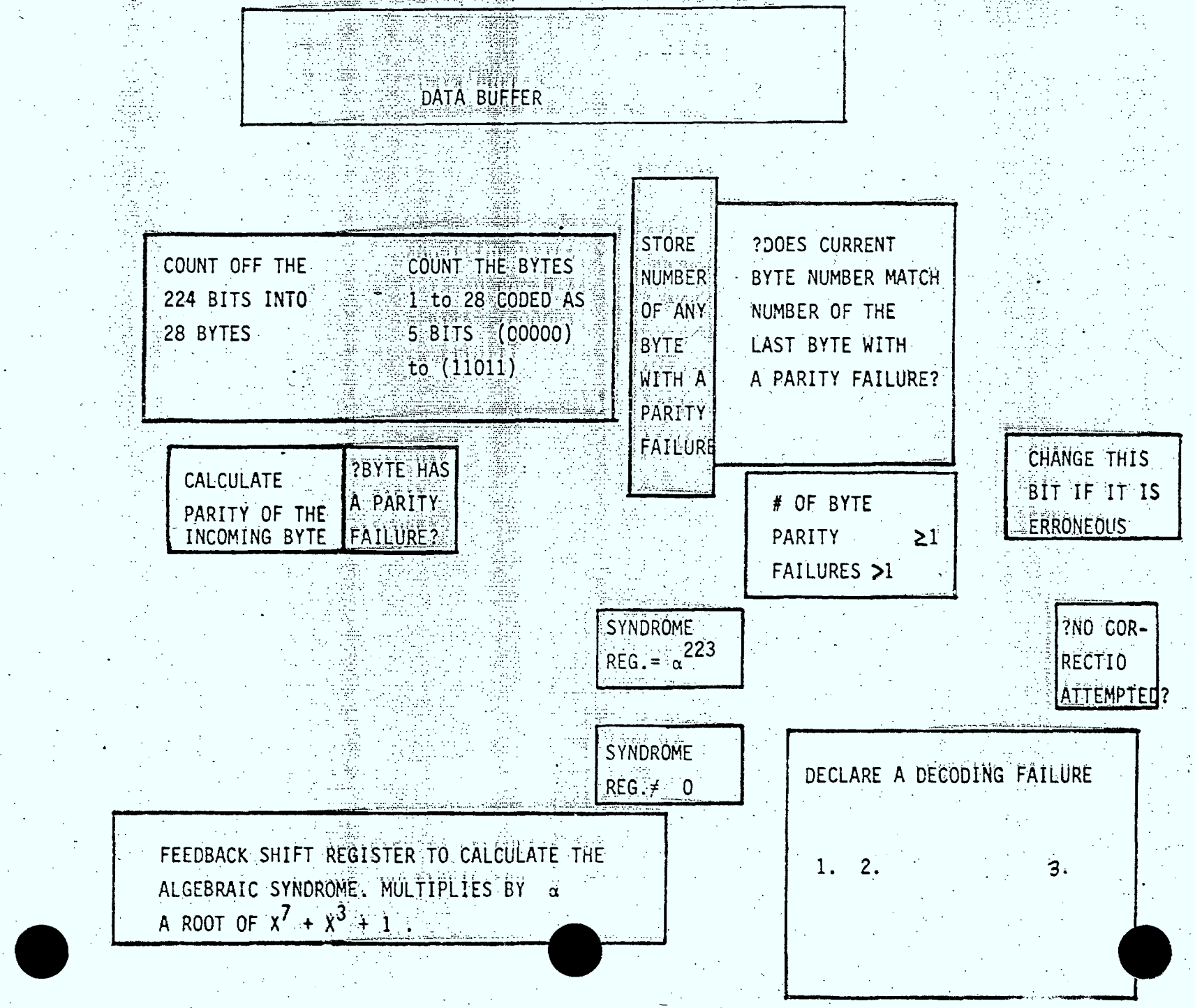
Decoding failure is declared if any of the following occur,

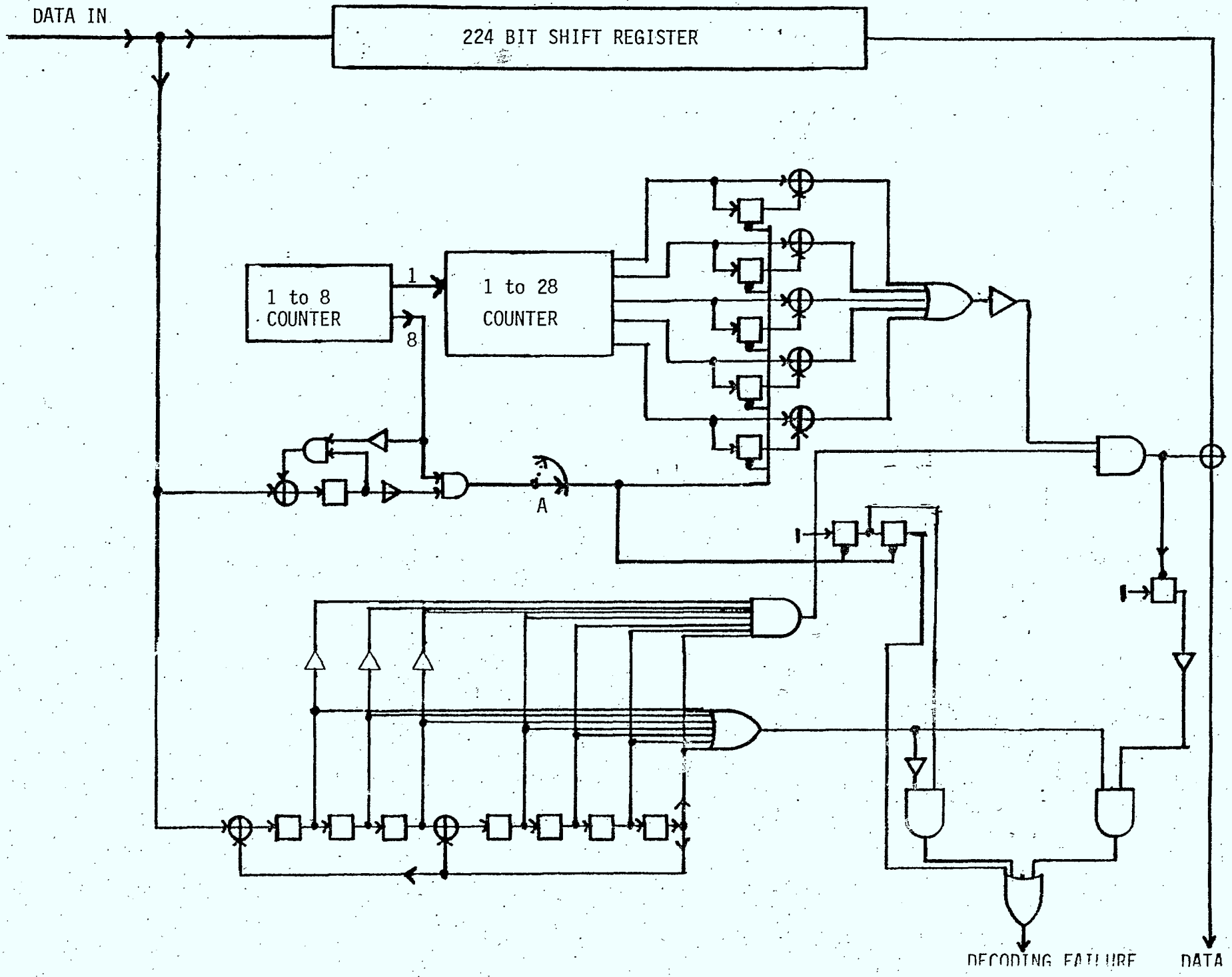
1. at least two bytes have a parity failure;
2. the algebraic syndrome is zero and at least one byte has a parity failure;
3. the algebraic syndrome is non-zero and there were no corrections attempted.

This last condition is unknown until the last bit is out so decoding failure is decided only at the end of the 224 cycles.

After the 224 bits are closekd out of the circuit the switch A is closed and all the registers are reset appropriately. (The 5 flip-flops must be reset to a string which is not a byte number.) The details of clocking and reset are not worked out.

(For the flip-flops a dot . marks the clock input.)


DATA BUFFER
COUNT OFF THE 224 BITS INTO 28 BYTES
COUNT THE BYTES 1 to 28 CODED AS 5 BITS (00000) to (11011)
STORE NUMBER OF ANY BYTE WITH A PARITY FAILURE
?DOES CURRENT BYTE NUMBER MATCH NUMBER OF THE LAST BYTE WITH A PARITY FAILURE?
CHANGE THIS BIT IF IT IS ERRONEOUS
CALCULATE PARITY OF THE INCOMING BYTE
?BYTE HAS A PARITY FAILURE?
OF BYTE PARITY FAILURES ≥1
SYNDROME REG.= α^{223}
?NO CORRECTIO ATTEMPTED?
SYNDROME REG.≠ 0
DECLARE A DECODING FAILURE
1. 2. 3.
FEEDBACK SHIFT REGISTER TO CALCULATE THE ALGEBRAIC SYNDROME. MULTIPLIES BY α A ROOT OF $X^7 + X^3 + 1$.

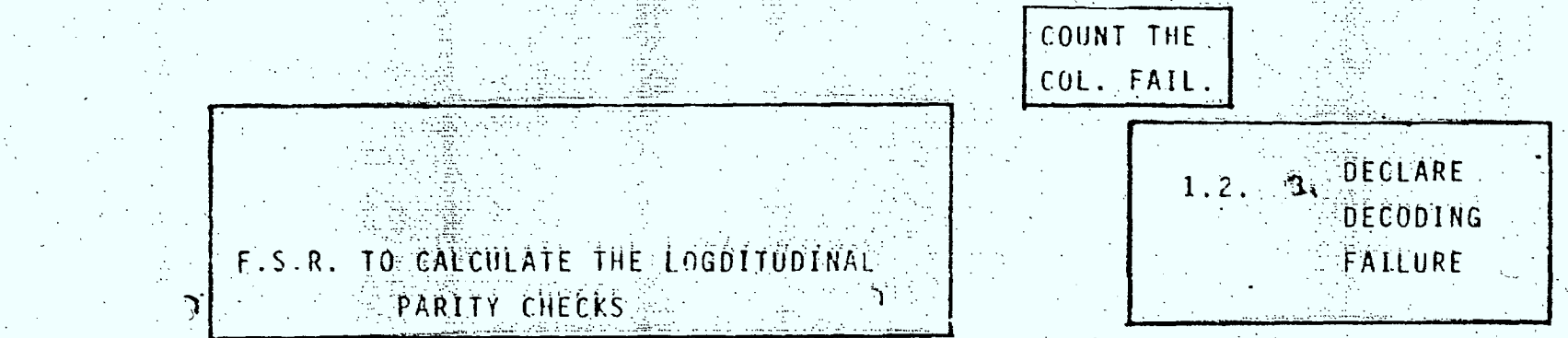

DATA IN
224 BIT SHIFT REGISTER
1 to 8
COUNTER
1
8
1 to 28
COUNTER
A
DECODING FAILURE
DATA OUT

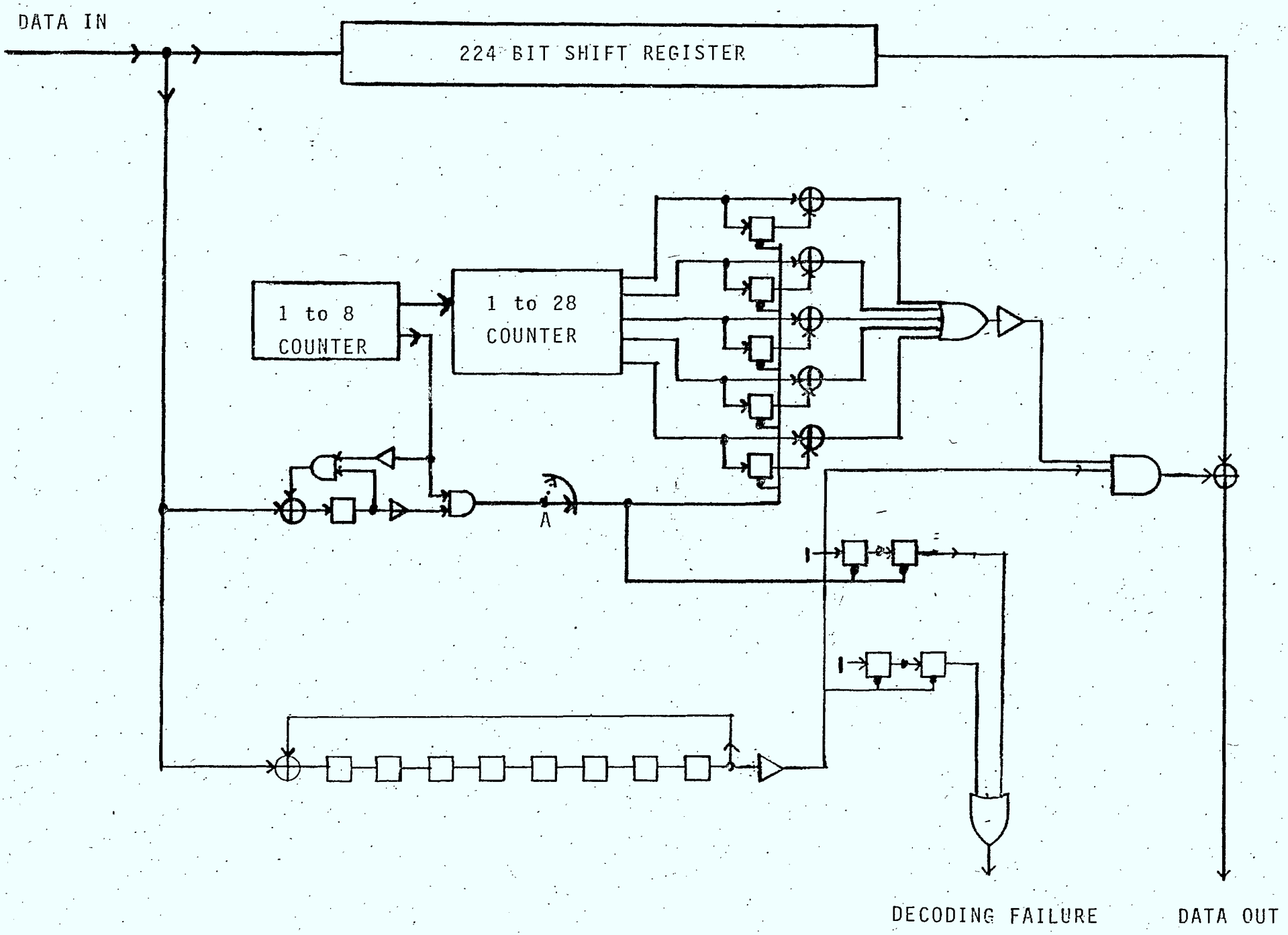
2.5.2 A Hardware Decoder for Product Code

This decoder is similar to the one for Carleton code. The layover highlights the differences. The algebraic syndrome is replaced by a feedback shift register which calculates the column checks. A 2 stage shift register counts the number of failures (i.e. 0's) in this set of 8 column checks.

Decoding failure is declared if,

1. there are at least 2 column failures;
2. there are at least 2 row (i.e. byte failures).


F.S.R. TO CALCULATE THE LOGDITUDINAL PARITY CHECKS
COUNT THE COL. FAIL.
1.2. 3. DECLARE DECODING FAILURE


DATA IN
224 BIT SHIFT REGISTER
1 to 8
COUNTER
1 to 28
COUNTER
A
DECODING FAILURE
DATA OUT

Appendix 2.A Definition of Carleton Code.

In addition to the odd-parity checks on the 27 data bytes a single suffix byte is added as follows. Each successive byte is added mod 2 (exclusive-or) to an accumulator $(a_1, \ldots, a_8)$. After each addition the accumulator is transformed according to the rules (mod 2):

$$a_1' = a_1 + a_2 + a_6 + a_7 + a_8$$

$$a_2' = a_1 + a_3 + a_6$$

$$a_3' = a_2 + a_3 + a_4 + a_7$$

$$a_4' = a_1 + a_4 + a_5 + a_8$$

$$a_5' = a_5 + a_7 + a_8$$

$$a_6' = a_2 + a_7$$

$$a_7' = a_1 + a_3 + a_4 + a_8$$

$$a_8' = a_1 + a_5 + a_6 + a_7 + a_8 .$$

After 27 bytes have been processed the accumulator contains the check byte.

Appendix 2.B A Software Encoder for Carleton Code.

This encoder implements the method of page . It is written in 6809 Assembler code.

```
X points to the first message                    LDX    BYTE1
byte.  A  is set to zero.                        CLRA
Each successive byte is exclusive-or'd  BYTELOOP EORA   ,X
with the A register.
The A register is them multiplied                LDB    #1
by α  eight times.  The iterations
are counted by shifting a one across
the B register.  The contents of A       XALPHA  ASLA
are shifted once to the left.  If a
one appears in the carry bit  C  then            ,
A contained (1,....) a number using
α^7  in its representation.  Thus if             BCC    SKIP
C = 1  after the shift we add  α^8  onto         EORA   #155
A where α^8 = α^7 + α^4 + α^3 + α + 1 ≡ (10011011).  SKIP  ASLB
In decimal this byte represents 155.             BCC    XALPHA
```

```
After 27 bytes have been processed      LEAX    ,X+
we leave the loop                       CMPX    BYTE1 + 28
                                        BNE     BYTELOOP

A which now contains the check byte     STA     ,X
is stored in the next byte below
the data bytes in memory.

Return                                  RTS
```

APPENDIX 2.C IMPLEMENTATION OF THE DECODER FOR CARLETON CODE (LENGTH = 28 BYTES)

MEMORY ALLOCATION

This routine uses 3 bytes of RAM:

PERROR — Is two consecutive bytes that will store the location of the first faulty byte that we find. We use two bytes, as we must store the index register 'Y' here, but since 'Y' only takes on values between 1 and 28, the first byte will always be zero. So, to see if anything has been stored in 'PERROR', we will always look at the contents of PERROR+1.

ERFLAG — Is the decoding failure flag. It is set to zero at the start of the routine and will remain zero if we decode correctly. However, if we have a decoding failure then ERFLAG receives a non-zero value. This is unnecessary if a bit in the condition code register is used as an error flag.

This routine also uses 511 bytes of tables in ROM:

QSYND — Is a table of length 256 - which has the quasi-syndrome for any given message byte. The table is stored in the following format:
For an entry in the table corresponding to a given byte. The first bit of the entry is the complement of the parity of the corresponding message byte while the last seven bits are the actual quasi-syndrome of the byte. This allows us to do the syndrome calculations and the parity checking of a message byte simultaneously.

MUL8 — Is a table of length 128 which is used to multiply a given partial syndrome by α^8. An entry in this table corresponding to a given byte is just the product of this byte and α^8, in the finite field. We only need a table of length 128, as the partial syndromes are all 7 bits long, which gives us a maximum of 128 different partial syndromes and 128 different locations in the table.

LOG — Is a table of logarithms in the given field. For a given byte B the corresponding entry in the table is the number U such that $\alpha^U = B$. We use this table to find the error position in our received message from a given syndrome.

Finally, the array 'message' is the received message, which is assumed to be 28 bytes long and to have the following format: An information byte in message looks like

$$X_{i+7}, X_{i+6}, \ldots, X_{i+1} X_i$$

with the leftmost bit being the most significant bit of the byte. Thus the first byte of message will go-from X_{223} to X_{216}, while the last byte comprises X_7 through X_0.

Assembler Program

```
TOPBYTE  EQU   #28
         LDY   #TOPBYTE
         CLRA
         CLR   PERROR
         CLR   PERROR + 1
         CLR   ERFLAG
```

TOPBYTE — Is the number of bytes in the received message. Currently, we assume a message length of 28 bytes.

Y — Is the internal index register of the 6809 microprocessor that points to the byte of the received message that we are currently examining. Since we are working from the last byte of our message down to the first, Y will initially have the value TOPBYTE.

A — Is the internal register of the 6809 that contains the partial syndrome during this main loop. It has an initial value of zero, as we initially have no syndrome.

PERROR — Is the first of two memory locations that will be used to store the locations of any bytes that have odd parity and thus an odd number of errors. Initially, PERROR is also zero, as we have not yet encountered any parity failures.

The following loop, from the statement labelled 'OLOOP' to the statement labelled 'GOON', is repeated once for each byte in the received message, starting with the last byte and going downwards to the first.

```
OLOOP   LDX   #MUL8
        LDA   A,X
```

We start by multiplying our current value of the partial syndrome by α^8. This is done by looking up the product in the table 'MUL8'.
Remember that 'A', the partial syndrome, is initially zero, and thus this multiplication has no effect on the first iteration of the loop.

Now, we add the quasi-syndrome from our current byte of message to the current partial syndrome that is stored in 'A'. Since we are working mod 2, we do this addition using the 'Exclusive-Or' Instructions of the microprocessor. Also, to find the quasi-syndrome corresponding to our current byte of received message, we look up the value in the table named 'QSYND'.

```
        LDB   MESSAGE-1,X
        LDX   #PSYND
        EORA  B,X
        BITA  #$80
        BEQ   GOON
```

Now, the entries in the table 'QSYND' are structured as follows: The first (most significant) bit of an entry in 'QSYND' corresponding to a given byte, is the parity of that byte, while the last seven digits of the table entry are the actual quasi-syndrome. Thus, we can calculate the parity of the current byte as well as the up-to-date partial syndrome in one operation.

So, in order to look at the parity of our current byte, we have to look at the leftmost bit of register 'A', which is our current partial syndrome. If the leftmost bit is set, then the current byte is of odd parity, and there is an odd number of errors in this byte.

If we reach this section, then the parity of the current byte was even so we had an odd number of errors. So we would like to store the location of this current byte, presently held in register 'Y', at PERROR.

```
        TST     PERROR+1
        BNE     FAIL

        STY     PERROR
        ANDA    #$7F
```

However, if PERROR is currently non-zero, then this is the second byte containing a parity error. Thus there are at least two errors in the received message, and we have no hope of performing a correction. In this case we branch to the location 'FAIL', which indicates a decoding failure. Otherwise, this is our first-parity failure, and we could still correct our error, and finish correctly. So we store the location of the current byte, 'Y', in location 'PERROR'.

Now we decrement register 'Y' by 1. If 'Y' is not zero, it will point to the next byte to be processed. In this case we branch back to the beginning of the loop. Otherwise, if Y is zero, we have finished the parity checking and syndrome calculation section of the program, and we can now go on to attempt an error correction.

```
GOON    LEAY    -1,Y
        BNE     OLOOP
```

Now we have generated the syndrome, and we have the location of any one byte that had a parity failure stored. We would obviously like to know if we could correct any errors that have crept in to the received message. To answer this, we need to consider four main cases:

```
        TSTA
        BEQ     NOSYND
        TST     PERROR+1
        BEQ
```

Case 1: PERROR=0 and A=0
We assume that we received the message correctly, and pass to the end of the program.

Case 2: PERROR≠0 and A=0
In this case there are at least two errors in the received message, and we cannot possibly decode correctly. So, we immediately report a decoding failure by branching to the label 'FAIL'.

Case 3: PSYND=0 and A≠0
Again, we have at least two errors, and we report a decoding failure.

Case 4: PSYND≠0 and A≠0
Here we have at least a chance of correcting the error. There are two possible positions in the received message where an error would cause a syndrome identical to 'A'. If either of these positions lies within the byte numbered 'PERROR' then we assume that we only had a single error at that position, and we correct it. However, if neither of these two positions is in the byte at location 'PERROR', we again fail.

This is case 4.
We would like to try to correct our one error, if there is only one. Our first step is to find the two possible error positions corresponding to our calculated syndrome. The first of these error positions is the number U, such that α^U =A (the calculated syndrome.) We may find such a U by looking it up in a logarithm table for the given field. The other pos-

```
        LDX     #LOG-1
        LDB     A,X
        TFR     B,A
        LSRA
        LSRA
        LSRA
        INCA
        CMPA    PERROR+1
        BEQ     CORERR
```

sible error position is simply U + 127, since our primitive element, α, has order 127.

We can calculate U very simply, by looking up the value of U appropriate to the syndrome in a "LOG" Table for the finite field. We then proceed to store this, the first possible error position in both registers 'A' and 'B'. Now we would like to see if this error position falls within the byte pointed to by 'PERROR'. To do this, we simply divide register 'A' by eight, which will give us a byte position between 0 and 24, then add one. Since the 6809 has no divide instruction, we shift register 'A' three bits to the right, which has the same effect as dividing by eight. Once we have computed this result, we compare it to 'PERROR', and if the two are the same, we proceed to the label 'CORERR' to attempt to correct the error at byte 'PERROR'.

If we reach the following section of code, it means that our first error position did not fall within the byte pointed to by 'PERROR'. However, there is still our second possible error position to try. This position is 127 bits later than the first. Our value for the first error position is stored in register 'B': We move the contents of this register into register 'A', add 135, and divide by eight, using three right shifts as above. Adding 135 has a dual purpose: We wish to add 127 to get the next possible error position, and eight more to avoid having to increment 'A', as was necessary in the last section.

```
TFR     B,A
ADDA    #$87
LSRA
LSRA
LSRA
CMPA    PERROR+1
BNE     FAIL
DECB
```

We should now have the byte position of this, our second possible error position. We compare this with 'PERROR'. If the two are equal, then this second error position is assumed to be the right one and we continue on to corerr where we correct the assumed error. Otherwise, neither of the potential error positions fell within the byte with odd parity. In this case, there are at least two errors in the received message, and again we must report a decoding failure.

Here, we have decided that there is exactly one error in the received message, at the position given by the contents of register 'B', in the byte of the received message numbered 'PERROR'. We would now like to correct this error. Since we know which byte the error falls in, to correct it we need only create a byte which is all zero except for a one in the position corresponding to the faulty position in the incorrect byte and 'EXCLUSIVE-OR' this byte with the faulty byte. The zeros in this correction byte will not affect the correct digits in the faulty byte while the one will change the faulty bit. This position within the faulty byte is given by the last three digits of register 'B' which are, in effect, the remainder when 'B' is divided by eight. So, our correction byte is created by placing a '1' into the leftmost bit of register 'A' and shifting it to the right the number of times given by seven less the last three digits of 'B'. Then, 'EOR'ing this byte with our faulty byte corrects the error.

```
CORERR   COMB
         ANDB   #7
         INCB
         LDX    PERROR
         LDA    #128
```

This next section is the loop that actually creates the correction byte. Register 'A' starts out with a one in the leftmost position (Most significant bit) and is shifted to the right the number of times given by register B less one. Note that the test in this loop is at the beginning, to take into account the case where the faulty bit occupies the leftmost position in the faulty byte, and no shift is required.

```
CORLOOP   DECB
          BEQ   ECRLOOP
          LSRA
          BRA   CORLOOP
```

Thus we have created our correction byte in register 'A'. We now use it to correct the error. Since register 'X' now has the location of the faulty byte, we use 'X' as an index to exclusive-or the faculty byte with the correction byte, then to store it back again.

```
ECRLOOP   EORA  MESSAGE-1,X
          STA   MESSAGE-1,X
          BRA   OK
```

The following are cases 1 and 2 from above. If we get to here, then register 'A', the syndrome is zero. Hence, if we did not observe any bytes with odd parity in the received message then we assume correct reception; and terminate. Otherwise, we did find a faulty byte and to produce a zero syndrome there must have been at least three errors and we have no hope of correcting. So we announce a decoding failure by branching to the label 'FAIL'.

```
NOSYND    TST   PERROR+1
          BNE   FAIL
```

If we reach this label, then we have had what we assume is a successful decoding, and we return to the calling program.

```
OK        RTS
```

If we arrive at this section, then we have a decoding failure. We have this if we are sure that we have two or more errors in the received message. Since a zero value 'ERFLAG', the decoding failure flag, indicates a correct decoding, we set 'ERFLAG' to be a non-zero number, before returning to the calling program.

```
FAIL      COM   ERFLAG
          RTS
```

Appendix 2.D Parameters of the Channels

The burst error channels used in the performance analysis of the codes were described briefly in Section 2.3 . These are the channels used in the earlier study [2]. This Appendix gives more details. The new burst channel model of Section 2.4 is not discussed any further here.

Burst Channel #1 is a Gilbert type of model channel. This channel model has often been used, for example by Weldon[8].The channel is assumed to be in either a good state or a bad state. In the good state errors occur independently with frequency σ ; in the bad state every bit is in error. (This latter assumption makes the calculation feasible though a bad error rate of $\frac{1}{2}$ would be more reasonable.) Transitions between the good and bad state occur at each bit transmission with the probabilities shown in Table 2.D.1.The "good" error probability is then determined to give the desired overall bit error rate. The distribution

	good → bad	bad → good
10^{-3}	2.5×10^{-4}	.5
10^{-4}	3.0×10^{-5}	.5
10^{-5}	3.0×10^{-6}	.5
10^{-6}	3.0×10^{-7}	.5

Table 2.D.1 State Transition Probabilities for Burst Channel #1.

into types of the weight four codewords as given in Appendix 2.E is the information required to calculate the probability of decoding error.

The four remaining burst channels are defined as special cases of a general model. We assume that the lengths of the gaps between correctly received bits are independent and identically distributed. If X is the number of bits from one correctly received bit to the next then we set $f_m = \Pr\{X = m\}$ for $m \geq 1$. These channels only have memory of strings of consecutive errors. Once a bit is received correctly the disposition of further bits is governed solely by the distribution of X.

An alternate way of specifying such a channel is to define p_i the probability that a bit is in error given that the previous i-1 bits are erroneous. The parameters p_i are related to the parameters f_n by the formulas,

$$f_m = (1 - p_m) \prod_{i=1}^{m-1} p_i, \quad m > 1$$

$$f_1 = 1 - p_1 .$$

We assume that $\Sigma f_m = 1$ and hence that the bit error rate (BER) is given by

$$BER = 1 - 1/(\Sigma m f_m) .$$

For Burst Channel #2 we set $p_1 = P_c$ and $p_i = P_E$ if $i \geq 2$. Then $BER = P_C/(1+P_C - P_E)$. The result is a channel in which errors arise as a bitwise Markov chain. A bit is in error with probability P_C if the previous bit was in error and with probability P_E if it was erroneous. Table 2.D.2 presents the values used in the calculations. We take always $P_C = 100\ P_E$.

BER	P_C
10^{-3}	$.9099 \times 10^{-3}$
10^{-4}	$.9902 \times 10^{-4}$
10^{-5}	$.9990 \times 10^{-5}$
10^{-6}	$.9999 \times 10^{-6}$

Table 2.D.2 Parameter values for Burst Channel #2.

The gap lengths between correct bits on Burst Channel #3 have a Pareto distribution. For some $\alpha > 1$ and for $m \geq 1$ we have

$$f_m = m^{-\alpha} / (\sum_{k=1}^{\infty} k^{-\alpha}).$$

The values of α are chosen so that the channel has a preset bit error rate; see Table 2.D.3. The formula for bit error rate in this case is

$$BER = 1 - \left(\sum_{k=1}^{\infty} k^{-\alpha}\right) / \left(\sum_{k=1}^{\infty} k^{1-\alpha}\right) .$$

BER	α
10^{-3}	10.01656
10^{-4}	13.30094
10^{-5}	16.61307
10^{-6}	19.93246

Table 2.D.3 Parameter Values for Burst Channel #3.

Burst Channel #4 was designed to produce many consecutive double errors. Over a background of white Gaussian noise with an error rate p an additional process causing errors is superimposed. This second process causes isolated single errors to be followed by a second error with probability M. This corresponds to setting $p_i = p$ for $i \neq 2$ and $p_2 = p + (1-p)M$ in our general model. Table 2.D.4 gives the values selected for the parameters.

BER	p	M
10^{-3}	$.667 \times 10^{-3}$	.5
10^{-4}	$.6667 \times 10^{-4}$	.5
10^{-5}	$.6667 \times 10^{-5}$	.5
10^{-6}	$.6667 \times 10^{-6}$	.5

Table 2.D.4 Parameter Values for Burst Channel #4

Burst Channel #5 was designed to have longer runs of erroneous bits than the other channels. At a fixed bit error rate the effect of a longer burst is merely that there are many fewer random errors between the bursts. The codes tend to behave on such a channel as if it were a white Gaussian channel with smaller overall bit error rate. We take $f_m = p^m/m!(e^p - 1)$ for $m \geq 1$. This gives $BER = 1 - e(e^p - 1)/pe^p$. The values determined for the parameter are given in Table 2.D.5.

BER	p
10^{-3}	2×10^{-3}
10^{-4}	2×10^{-4}
10^{-5}	10^{-5}
10^{-6}	10^{-6}

Table 2.D.5 Parameter Values for Burst Channel #5 .

Appendix 2.E Parameters of the Codes

The analysis of the performance of the codes on bursty channels requires a detailed cataloging of the weight into types according to the degree of separation of their ones by zeroes. We use the integer 1 to denote a single one and use 2 to stand for two ones in a row. The single hyphen "-" represents a single intervening zero while a double hyphen "--" denotes at least two and perhaps more consecutive zeroes. Thus the vector (---00011010000100---) has type 2-1--1.

Type	PRODUCT	CARLETON
2--2	2646	72
2--1-1	0	257
2--1--1	27	683
1-1-1--1	2268	60
1-1--1--1	54	571
1--1--1--1	5589	511

Chapter 3. Example of a Half-page Code

3.1 Introduction: A Reed-Solomon code.

In addition to the use of error correction on individual data lines coding can be introduced into the Broadcast Telidon system at other levels on a discretionary basis. This chapter explores one such possibility. We consider a Reed-Solomon code which groups 8 data blocks together then adds a ninth line of check bytes (figure 3.1). Thus this code has 8 x 28 "information" bytes and 28 "check" bytes. (The prefix bytes are ignored.) The symbols of this code are bytes rather than bits. In fact any pattern of 14 byte-errors can be corrected. These byte errors can come from 14 independent bit errors scattered in 14 separate bytes or from a series of 98 consecutive bit errors. This code gives extremely good performance at the cost of an increase in decoding overhead and in decoding time.

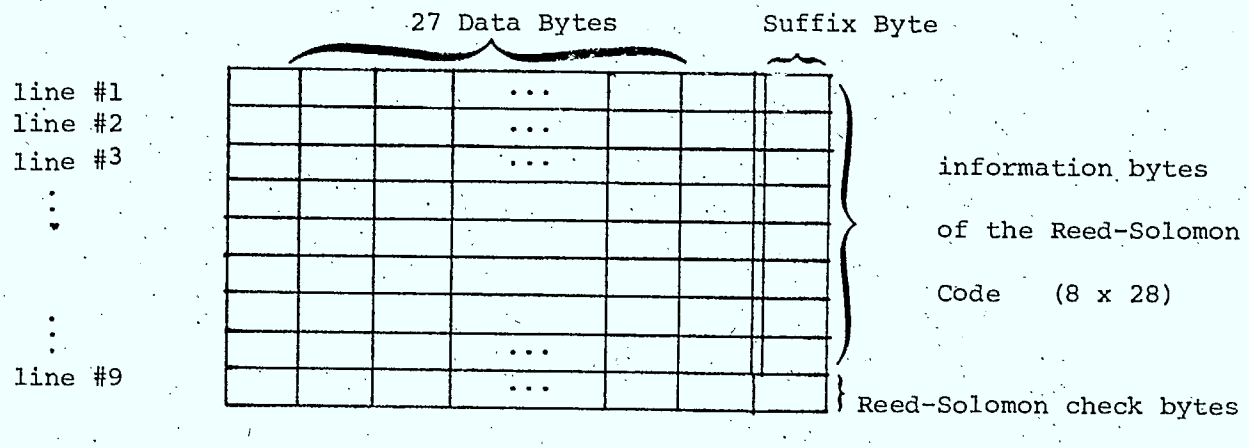


FIGURE 3.1 A Codeword of the Reed-Solomon Code

3.2 Performance.

We have used as a performance measure the expected length of a run of trouble free pages. We therefore make no distinction between decoding failure and decoding error. Any outcome of a decoding other than correct decoding we will call a decoding fault. We calculate the probability of a decoding fault. From this the mean number of pages until a decoding fault occurs is calculated. The results are plotted in Figure 3.2.

The data block code might or might not have been used prior to a Reed-Solomon decoding. Since the bytes of the data blocks all have known parity the data block code will never corrupt a good byte in an attempt to correct errors in that data block. Thus the data block code cannot make the job of the Reed-Solomon code any more difficult and it may make it easier.

Not making any distinction between decoding failures and decoding errors has two simplifying consequences. First, since we are only interested in the number of corrupt bytes in a 9 block unit a channel with independent errors will tend to be the worst. A bursty channel will have errors occuring in clusters. Thus at a fixed bit error rate, a bursty channel will have more clean pages and fewer corrupt bytes. The second simplification is that if we are

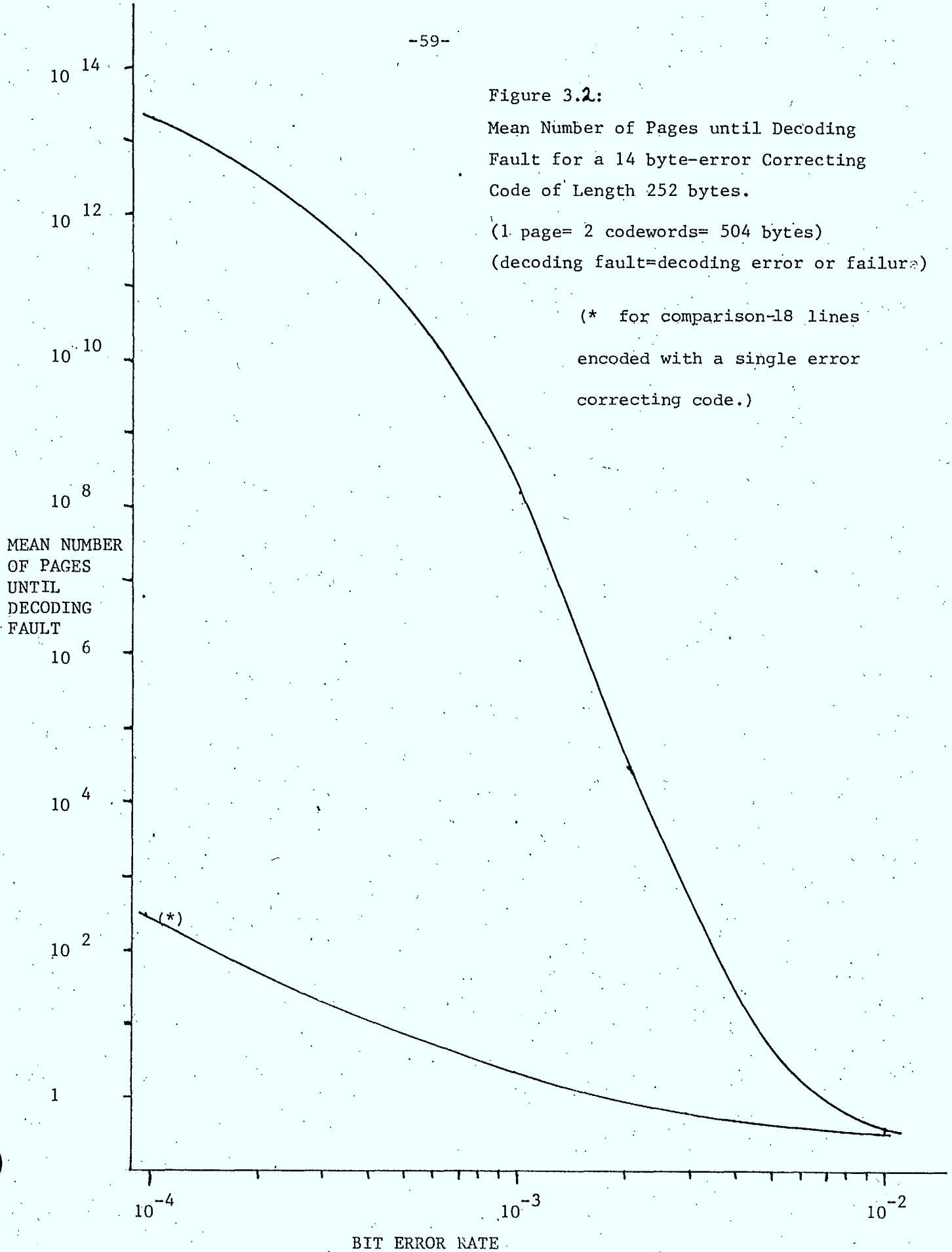


Figure 3.2:
Mean Number of Pages until Decoding Fault for a 14 byte-error Correcting Code of Length 252 bytes.
(1 page= 2 codewords= 504 bytes)
(decoding fault=decoding error or failure)

(* for comparison-18 lines encoded with a single error correcting code.)

going to calculated the probability of a decoding fault we are essentially calculating the probability of correct decoding. Thus the Reed-Solomon code can be replaced by an arbitrary 14 byte-error correcting code. The precise structure of the code is not required (and would be very difficult to calculated in any case).

Assume that errors occur independently in the channel with probability p. Then the probability that a byte is received correctly is $b_c = q^8$ where $q = 1-p$. The probability that the byte contains an error is then $b_e = 1-q^8$. Ignoring any effect due to corrections made at the data block level the probability of correct-decoding for the Reed-Solomon code is

$$P_{CD} = \sum_{i=0}^{14} \binom{252}{i} b_e^i b_c^{252-i} .$$

Then the probability of decoding fault is $P_F = 1-P_{CD}$ and the mean number of pages (= 2 codewords) until a decoding fault can be taken as $1/(2P_F)$.

We have included in Figure 3.2 the corresponding expected numbers of pages until decoding fault for a data block code used without a Reed-Solomon code (that is with single bit-error corrections line by line). Of course in our earlier studies we used scarcity of decoding errors as the prime indicator of good performance. These represent

only a small fraction of the decdoing faults. To calculate the probability of decoding error for the Reed-Solomon code would be difficult and not very illuminating (since decoding faults in general are so rare). An attempt at reconciling the two assessment measures is made in Table 3.1.

Bit Error Rate	Mean Number of Pages Until Decoding Fault: Data Block Code	Decoding Error: Carleton Code	Decoding Fault Reed-solomon Code
10^{-3}	2.8	8.0×10^{3}	1.7×10^{8}
10^{-4}	226	7.0×10^{6}	2.0×10^{13}

TABLE 3.1 Comparison of Performance

3.3 Decoding.

Only a preliminary study was made concerning the decoding of the Reed-Solomon code. Much has been written on this subject. See for example [3] and the references cited there. A decoding algorithm of the Massey type was implemented in microprocessor software (6502). Decoding required on the order of one second and used 1/2 K of the look-up tables for finite field arithmetic.

Chapter 4. The System as a Whole

The final stage in our analysis of error-correction coding for the Telidon system was an analysis of the performance of the overall system under various assumptions. Coding has been proposed or approved for the five byte prefix, the data blocks and for groups of data blocks. How do these codes inter-relate as regards performance?

We consider two arrangements for coding a sequence of nine data packets. The first Option A does not use a Reed-Solomon code while the second does.

Option A: -each prefix byte encoded with Hamming (8.4).
-each data block completed with a one-byte suffix to correct single errors.

Option B: -each prefix byte encoded with Hamming (8,4)

-coding on the data blocks optional

-the 9 data blocks as a whole form a codeword for a 14 byte-error correcting code (Reed-Solomon for example).

We assume that errors arise independently. We use as a measure of performance the expected number of (18 data packet) pages until decoding fault. In other words the number calculated is the expected number of pages in a problem-free run. Faults include both decoding errors and decoding failures with the former being a very small fraction on the channel used. The advantage of using the

probability of a decoding fault over using decoding error is that independent errors are the worse case for the former and generally the best case for the latter. A bursty channel has its errors packed into fewer bytes so there are longer runs of correct bytes. Moreover it would be very difficult to calculate the probability of decoding error for the 14-byte error correcting (Reed-Solomon) code even in the simplest case of independent errors.

Table 4.1 presents the expected number of pages until a decoding fault for Options A and B at various bit error rates. We include for comparison the same parameter calculated with the prefix code assumed to work perfectly (i.e. no faults).

ERROR RATE	OPTION A Prefix & Block codes	OPTION B Prefix & Reed Solomon Codes	DATA BLOCK alone	REED_SOLOMON alone
.01	.50	.56	.5	.6
.003	.66	41	.66	420
.001	2.8	399	2.8	1.7×10^{8}
.0003	26	4414	26	$> 10^{13}$
.0001	225	39692	225	$> 10^{13}$

TABLE 4.1 Expected Number of Pages until Decoding Fault

What we can quickly deduce is that in Option A the faults are mostly coming from the data block code while in Option B they are mostly coming from the prefix. If we examine the probabilities of faults arising in the prefices, data block code and Reed Solomon code this becomes more clear.

BIT ERROR RATE	PROBABILITY OF DECODING FAULT IN... 9 PREFICES	9 DATA BLOCKS WITH SINGLE ERROR CORR.	9 DATA BLOCKS WITH 14 BYTE-ERROR CORR.
.001	1.14×10^{-1}	0.9999933_{-1}	0.8829
.003	1.11×10^{-2}	7.58×10^{-1}	1.17×10^{-3}
.001	1.25×10^{-3}	1.78×10^{-1}	2.98×10^{-8}
.0003	1.13×10^{-4}	1.92×10^{-2}	$< 10^{-14}$
.001	1.26×10^{-5}	2.21×10^{-3}	$< 10^{-14}$

TABLE 4.2 Probability of at least one decoding fault in various parts of the system

Therefore most of the power of the 14 byte error-correcting code is lost since the prefix code is slowing the system down anyway. On the other hand, without this long code the single error correcting code on a data block is the limiting factor. We conclude that in the light of the decoding complexity of the Reed-Solomon code future

work should replace this Reed-Solomon code with simpler codes that correct fewer bytes, that can be decoded faster and which are closer to the prefix code in probability of a decoding fault.

References

1. "Television Broadcast Videotex", Broadcast Specification No. 14, Issue 1, Provisional, Telecommunication Regulatory Service,Department of Communications, Canada ,June 19, 1981.

2. Leroux, B., M. Moore, B. Mortimer, L. Oattes and T. Ritchford, "A Study of the Use of Error-correcting codes in the Canadian Broadcast Telidon System", Progress Report, DSS Contract No. OSU81-00095, Department of Communications, Canada, August 1981.

3. Allard, P.E., V.K. Bhargava, and G.E. Seguin, "Realization, Economic and Performance Analysis of Error-correcting Codes and ARQ Systems for Broadcast Telidon and other Videotex Transmission", DSS Contract OSU80-00133, Final Report, Department of Communications, Canada, June 1981.

4. Sablatash, M. and J.R. Storey, "Determination of Throughputs, Efficiencies and Optimal Block Lengths for an Error-correcting Scheme for the Canadian Broadcast Telidon System", Can. Elec. Eng. J. (1979), 25-39.

5. Mortimer, B.C., "A Description of the Carleton Code", DSS Contract No. OSU81-00095, Progress Report, Department of Communications, Canada, September 1981.

6. Gilbert, E.N., "Capacity of a Burst Noise Channel", Bell Sys. Tech. J., 39(1960), 1253-****

7. Sussman, S.M., "Analysis of the Pareto model for error statistics and telephone circuits", IEEE Trans. Comm. Sys., CS-11(1963), 213-221.

8. Weldon, E.J., "Error Control on High-Speed Satellite Channels", Int. Comm. Conf. Record, (1981).