ERROR CODING FOR ENGINEERS

THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE

ERROR CODING FOR ENGINEERS

A. Houghton
Synectic Systems, Ltd., United Kingdom

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

Houghton, A.
Error coding for engineers / A. Houghton.
p cm. (The Kluwer international series in engineering and computer science; SECS 641)
Includes bibliographical references and index.
ISBN 978-1-4613-5589-2 ISBN 978-1-4615-1509-8 (eBook)
DOI 10.1007/978-1-4615-1509-8
1. Signal processing. 2. Error-correcting codes (Information theory) I. Title. II. Series.

TK5102.9 .H69 2000
005.72 dc21

2001038560

Table of Contents

Preface	ix
1. Introduction	1
1.1 Messages Need Space	1
1.2 The Hamming Bound	4
1.3 The Gilbert Bound	6
1.4 Where Do Errors Come From?	7
1.5 A Brief History of Error Coding	12
1.6 Summary	13
2. A Little Maths	15
2.1 Polynomials and Finite Fields	16
2.2 Manipulating Field Elements	19
2.3 Summary	24
3. Error Detection	25
3.1 The Horizontal and Vertical Parity Check	25
3.2 The Cyclic Redundancy Check	27
3.3 Longhand Calculation of the CRC	28
3.4 Performance	31
3.5 Hardware Implementation	32
3.6 Table-Based Calculation of the CRC	35
3.7 Discussion	39
4. Error Correction by Parity	41
4.1 Correcting a Single Bit	43
4.2 Extending the Message Size	45

5. Error Correction Using the CRC 49
5.1 A Hardware Error Locator 52
5.2 Multiple Bit Errors 53
5.3 Expurgated Codes 54
5.4 The Perfect Golay Code 56
5.5 Fire Codes 63

6. Reed-Muller Codes 67
6.1 Constructing a Generator Matrix For RM Codes 67
6.2 Encoding with the Hadamard Matrix 74
6.3 Discussion 77

7. Reed-Solomon Codes 79
7.1 Introduction to the Time Domain 79
7.2 Calculation of the Check Symbols for One Error 80
7.3 Correcting Two Symbols 83
7.4 Error Correction in the Frequency Domain 88
7.5 Recursive Extension 91
7.6 The Berlekamp-Massey Algorithm 97
7.7 The Forney Algorithm 100
7.8 Mixed Domain Error Coding 101
7.9 Higher Dimensional Data Structures 107
7.10 Discussion 118

8. Augmenting Error Coding 119
8.1 Erasure 119
8.2 Punctured Codes 121
8.3 Interleaving 125
8.4 Error Forecasting 130
8.5 Discussion 131

9. Convolutional Coding 133
9.1 Error Correction with Convolutional Codes 135
9.2 Finding the Correct Path 136
9.3 Decoding with Soft Decisions 138
9.4 Performance of Convolutional Codes 140
9.5 Concatenated Codes 141
9.6 Iterative Decoding 142
9.7 Turbo Coding 143
9.8 Discussion 144

10. Hardware 147
10.1 Reducing Elements 147
10.2 Multiplication 149
10.3 Division 153
10.4 Logs and Exponentials 158
10.5 Reciprocal 160
10.6 Discussion 164

11. Bit Error Rates 165
11.1 The Gaussian Normal Function 165
11.2 Estimating the Bit Error Rate 166
11.3 Applications 172
11.4 Discussion 176

12. Exercises 177
12.1 Parity Exercises 177
12.2 CRC Exercises 178
12.3 Finite Field Algebra 180
12.4 Convolutional Codes 183
12.5 Other Codes 183
12.6 Solutions to Parity Exercises 184
12.7 Solutions to CRC Exercises 186
12.8 Solutions to Finite Field Algebra 192
12.9 Solutions to Convolutional Coding 202
12.10 Solutions to Other Codes 203
12.11 Closing Remarks 208
12.12 Bibliography 208

Appendix A Primitive Polynomials 211
Appendix B The Golay Code 219
Appendix C Solving for Two Errors 223
Appendix D Solving some Key Equations 229
Appendix F Software Library 233

Index 245

Preface

Error coding is the art of encoding messages so that errors can be detected and, if appropriate, corrected after transmission or storage. A full appreciation of error coding requires a good background in whole number maths, symbols and abstract ideas. However, the mechanics of error coding are often quite easy to grasp with the assistance of a few good examples. This book is about the mechanics. In other words, if you're interested in implementing error coding schemes but have no idea what a finite field is, then this book may help you. The material covered starts with simple coding schemes which are often rather inefficient and, as certain concepts are established, progresses to more sophisticated techniques.

Error coding is a bit like the suspension in a car. Mostly it's unseen, but the thing would be virtually useless without it. In truth, error coding underpins nearly all modern digital systems and without it, they simply couldn't work. Probably like some car suspensions, the elegance of error coding schemes is often amazing. While it's a bit early to talk about things like *efficiency*, two schemes of similar efficiency might yield vastly different performance and, in this way, error coding is rather 'holistic'; it can't be treated properly outside of the context for which it is required. Well, more of this later. For now, if you're interested in whole number maths (which is actually really good fun), have a problem which requires error coding, are an undergraduate studying DSP or communications, but you don't have a formal background in

maths, then this could be the book for you. Principally, you will require a working knowledge of binary; the rest can be picked up along the way.

Chapter 1

INTRODUCTION

The chapter deals with an overview of error coding, not especially in terms of what's been achieved over the years or what is the current state of the subject, but in pragmatic terms of what's underneath it that makes it work. The maths which surrounds error coding serves two key purposes; one, it shows us what can theoretically be achieved by error coding and what we might reasonable have to do to get there and, two, it provides mechanisms for implementing practical coding schemes. Both of these aspects are, without doubt daunting for anyone whose principal background is not maths yet, for the engineer who needs to implement error coding, many of the practical aspects of error coding are tractable.

1.1 Messages Need Space

Space is at the heart of all error coding so it seems like a good place to start. If I transmit an 8-bit number to you by some means, outside of any context, you can have absolutely no idea whether or not what you receive is what I sent. At best, if you were very clever and knew something about the quality of the *channel* which I used to send the byte (an 8-bit number) to you, you could work out how likely it is that the number is correct. In fact, quite a lot of low-speed communication works on the basis that the channel is good enough that most of the time there are no errors. The reason that you

A. Houghton, *Error Coding for Engineers*

cannot be certain that what you have received is what I sent is simply that there is no space between possible messages. An 8-bit number can have any value between and including 0 and 255. I could have sent any one of these 256 numbers so whatever number you receive **might** have been what I sent.

You've probably heard of the parity bit in RS232 communications. If not, it works like this. When you transmit a byte, you count up the number of 1s in it (its weight) and add a ninth bit to it such that the total number of 1s is even (or odd; it doesn't matter which so long as both ends of the channel agree.) So we're now transmitting 8 bits of *data* or *information* using 9 bits of *bandwidth*. The recipient receives a 9-bit number which, of course, can have 512 values (or 2^n where n is the number of bits). However, the recipient also knows that only 256 of the 512 values will ever be transmitted because there are only eight data bits. If the recipient gets a message which is one of the 256 messages that will never be transmitted, then they **know** that an error has occurred. In this particular case, the message would have suffered a *parity error*.

In essence, what the parity bit does is to introduce some space between messages. If a single bit changes in the original 8-bit message, it will look like another 8-bit message. If a single bit changes in the 9-bit, parity-encoded message, it will look like an invalid or non-existent message. To change the parity-encoded message into another valid message will require at least 2 bits to be flipped. Hopefully this is really obvious and you're wondering why I mentioned it. However space, in one form or another, is how all error coding works. The key to error coding is finding efficient ways of creating this space in messages and, in the event of errors, working out how to correct them.

Space between messages is measured in terms of *Hamming distance* (d). The Hamming distance between two messages is a count of the minimum number of bits that must be changed to convert one message into the other. Error codes have a special measure called d_{min}. d_{min} is the minimum number of bits that must be changed to make one valid message look like another valid message. This, ultimately, determines the ability of the code to detect or correct errors. Every bit-error that occurs increases the Hamming distance between the original and corrupted messages by one. If t errors are to be detected then

$$d_{min} > t. \tag{1.1}$$

In the case of the simple 8-bit message with no parity, $d_{min} = 1$, so $t = 0$. However, with the addition of a parity bit, d_{min} increases to two, so $t = 1$, i.e.

we can reliably detect a single-bit error. Should we want to be able to **correct** messages, then (1.1) is modified to (1.2), below.

$$d_{min} > 2t. \tag{1.2}$$

Clearly parity, as it has been presented above, cannot correct errors. So where does (1.2) come from? Error correction works by finding the nearest valid message (in terms of d) from the corrupted message. Using the simple case of parity checking, $d_{min} = 2$. A single-bit error will change a valid message into an invalid message which has a d of 1 from it. However, because d_{min} is only two, there will be other messages which are also only one bit away from the invalid message. In fact, any one of the nine bits could be changed in the invalid message to create a valid message. The problem is that there are several messages equally close to the corrupted message and we have no means to choose between them. If d_{min} was three, however, (i.e. $> 2t$) the original message would, as before, be one bit away from the invalid message, but no other valid message could be nearer than two bits away. In this case, there is a nearest candidate.

A good way to visualize this is to draw 2^n circles on a piece of paper (where there are n message bits) and fill in 2^k of them (where k bits are data or information) as evenly spread over the page as possible. The circles represent all possible messages, while those that are filled in are the subset of valid messages. Figure 1.1 is my attempt at this.

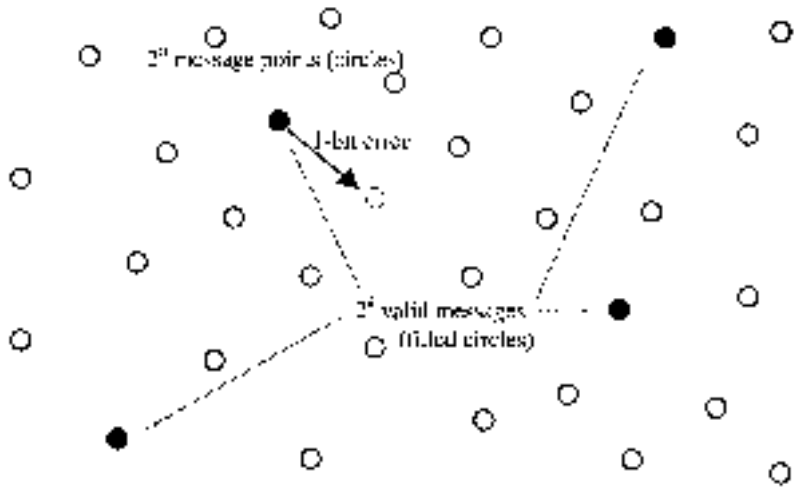

Figure 1.1 Valid data points within the total message space.

Starting at a valid message, introduce an error by moving away to the nearest circle. Each time you add an error bit, move further away from the starting message. Eventually, the correct (starting) message will no longer be the nearest filled in circle. When this happens, the capacity of the code has been exceeded and an attempt at correction will make matters worse. This illustration is actually the projection of an n-dimensional problem onto a 2-dimensional space (your paper) and so is rather approximate. It none-the-less describes quite well the principle.

1.2 The Hamming Bound

The use of n and k, above, leads to the idea of (n, k) or (n, k, t) codes, another way in which error codes are often defined. n is the total number of message bits, so there are 2^n possible messages for the code, while k defines how many of those bits are *data* or *information* bits. So out of the 2^n messages, 2^k will be valid. The job of the error code is to spread the 2^k valid messages evenly across the 2^n message space maximizing d_{min}. A question you might reasonably ask is, is there a relationship between t, n and k? Another way of putting this might be to wonder how many bits of *redundancy* $(n - k)$ must be added to a message to create a t-error correcting code. There is a pragmatic and a mathematical way of considering this question. It's useful to start at the mathematical end since, if you can acquire a broad understanding of what's involved in error coding right at the start, it all makes more sense later on. Here goes then:

n, k and t are a little intertwined and it's easiest to choose three values and see if they can be made to satisfy certain constraints. For an example, consider the possibility of a (7, 4, 1) code. Could such a thing exist? There are 16 valid messages (2^4) in a pool of 128 possible messages (2^7) and we want to correct 1-bit errors. Each valid message must be surrounded by a sea of invalid messages for which $d <= t$. For correction to be possible, no other valid messages are allowed in store these unless, for them, $d > t$. So in this example, each of the 16 messages must have seven surrounding messages of $d = 1$ that are unique to them to permit correction. Each valid message requires 1 (itself) + 7 ($d = 1$ surrounding) messages. We need, therefore, 8 messages per valid message and there are 16 valid messages giving $8 \times 16 = 128$ messages in total. In this case there are exactly 128 available so, in principle at least, this code could exist. This limit is called the *Hamming bound* and codes (of which there are a few), which satisfy exactly this bound

are called *perfect codes*. For most codes 2^n is somewhat greater than the Hamming bound.

In general terms, an n-bit message will have

$$\frac{n!}{d!(n-d)!} \tag{1.3}$$

messages that are d bits distance from it. This is simply the number of different ways that you can change d bits in n. So, for a t error correcting code, the Hamming bound is found from

$$2^k + 2^k \times \sum_{d=1}^{t} \frac{n!}{d!(n-d)!} \leq 2^n$$

which simplifies to

$$\sum_{d=0}^{t} \frac{n!}{d!(n-d)!} \leq 2^{n-k} \tag{1.4}$$

If you can visualize n dimensional space, then each message represents a point in this space. Around each point, there is a surface of other points with radius 1 bit. Around that surface there is another of radius 2 bits and so forth up to n bits. (1.3) tells us how many points exist in each of these surfaces. Some codes which satisfy exactly the Hamming bound include (3, 1, 1), (5, 1, 2), (7, 4, 1), (7, 1, 3), (9, 1, 4), (11, 1, 5), (13, 1, 6), (15, 11, 1), (15, 1, 7), (17, 1, 8), (23, 12, 3), (90, 78, 2). Most of these contain only one data bit so the two valid messages (2^1) might be all zeros and all ones. Codes with the form $(2^m - 1, 2^m - m - 1, 1)$ such as (15, 11, 1) are known as *Hamming codes* and we'll see how these are constructed later. The code (23, 12, 3) is called the *Golay code* and, according to Golay, there is no solution to the code (90, 78, 2).

In some ways perfect codes are the most efficient codes since there are absolutely no wasted points in the 2^n message space. However, this has its drawbacks. With perfect codes all received messages (whether valid or in error), will decode to a solution. There are no invalid messages more than t-bits away from a valid message. This means that if more that t bit-errors occur, there is no way of telling. The Golay code, which will correct up to three bit-errors, is usually extended to a (24, 12, 3) code by adding an overall parity bit. While it can still only correct three errors it can now detect the presence of four.

1.3 The Gilbert Bound

The Hamming bound is one extreme end of the (n, k, t) relationship, packing the most capability into the least redundancy. At the other end is something called the *Gilbert bound*. This bound gives the smallest size of the message space (2^n) that absolutely guarantees that there will be a t-error correcting code for k data bits. It's trivial to construct working codes at this end of the (n, k, t) relationship although that doesn't mean they're any good. The Gilbert bound works on the following argument. If we want a t-error correcting code then we choose at random from the 2^n message space a valid code. The message and all messages within $2t$ bits distance of it are deleted. The next message is chosen from the remaining pool and all messages within $2t$ of it are deleted, and so forth. Each valid message thus requires

$$\sum_{d=0}^{2t} \frac{n!}{d!(n-d)!}$$

messages out of the total pool. If there are k data bits then a total of

$$2^k \times \sum_{d=0}^{2t} \frac{n!}{d!(n-d)!}$$

messages are needed so the Gilbert bound is generated from (1.5)

$$\sum_{d=0}^{2t} \frac{n!}{d!(n-d)!} \leq 2^{n-k} \tag{1.5}$$

Most error codes sit somewhere between these bounds which means that often, even though the capacity to correct errors may have been exceeded, the code can still detect unrecoverable errors. In this instance the received error message will be more than t from any valid code so no correction will be attempted.

So what about the pragmatic approach? During decoding the extra bits added to a message for error coding give rise to a *syndrome*, typically of the same size as the redundancy. The syndrome is simply a number which is usually 0 if no errors have occurred. If t errors are to be corrected in a message of n bits then the syndrome must be able to identify uniquely the t bit-error locations. So in n bits, there will be

$$\sum_{d=0}^{t} \frac{n!}{d!(n-d)!}$$

possible error combinations (including the no-error case). The syndrome must have sufficient size to be able to identify all of these error combinations. If the syndrome is the same size as the redundancy ($n - k$), then to satisfy this requirement gives

$$\sum_{d=0}^{t} \frac{n!}{d!(n-d)!} \leq 2^{n-k}$$

which is, of course, the Hamming bound. Another way of viewing this is to consider the special case of messages where $n = 2^i - 1$ (i is some integer). At least i bits will be needed to specify the location of one bit in the message. For example, if the message is 127 bits then a 7-bit number is required to point to any bit location in the message (plus the all-zero/no error result). So for t errors, at least $t \times i$ check bits will be needed. We can compare this result with the (15, 11, 1) code. To find one error in 15 bits needs $1 \times 4 = 4$ check bits which there are. If you try this approach with the Golay code, you find that 15 check bits should be needed as opposed to the 11 that there actually are. See if you can work out why this is and repeat the calculation for the (90, 78, 2) code.

A few more basic measures can be derived from (n, k, t) codes. *Redundancy* is the name given to the extra bits (r) added to a message to create space in it. The amount of redundancy added gives rise to a measure called the *coding rate* R which is the ratio of the number of data bits k, divided by the total bits $k + r\ (= n)$. In percent, R gives the code *efficiency*.

1.4 Where do Errors Come From?

Errors arise from a number of sources and the principal source which is anticipated should, at least in part, determine the code that is used. There are two main types of error which are caused by very different physical processes. First there are *random* errors. Any communications medium (apart perhaps from superconductors which, as yet, are not ubiquitous), is subject to noise processes. These may be internal thermal and partition noise sources as electrons jostle around each other, or they may be the cumulative

actions of external electromagnetic interference being picked up *en route*. This kind of noise gives rise to the background hiss in your hi-fi system or the fuzzy haze on the TV screen in areas with poor reception. Noise like this typically has what is called a *Gaussian PDF* (probability density function). So travelling down the communications cable is the data (1s and 0s) plus noise. When the data (plus noise) reaches its destination, a receiver circuit has to decide whether the data is 1 or 0 (crudely speaking). The noise content adds a little uncertainty into this process and leads ultimately to the addition of random errors.

The great thing about random errors is that they can be precisely modelled. You can predict exactly how likely an error is, how many will occur each hour and so forth. Unfortunately, you can never predict which bit it'll be. Taking a very simple example, suppose that a data 1 is represented by +1 volt and a data 0 by −1 volt on a piece of wire. The receiver will decide on the current state of the signal by comparing it with 0 volts. Figure 1.2 illustrates the example.

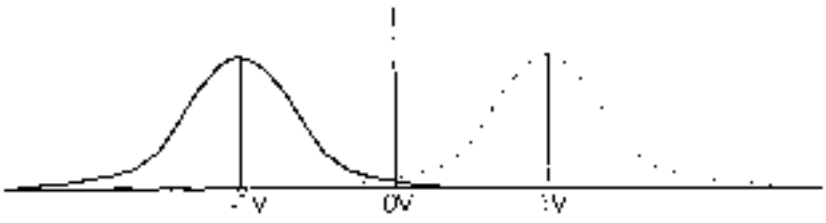

Figure 1.2 Gaussian PDF.

The curves about the ±1 volt markers are the Gaussian function (also called the *normal* function) which has the form in (1.6), below.

$$\frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\bar{x})^2/2\sigma^2} \tag{1.6}$$

σ is the standard deviation of the function, while $\bar{x}$ is the mean. The curve describes in terms of probability, what the received signal will be. The total area under the curve is 1 so the received signal **must** have some value. In this example, we could calculate how likely a 0 would be of being misinterpreted as a 1 by measuring the area under the solid curve that falls to

the right of the 0 volt decision threshold. Suppose it was 0.001, or 1/1000. This means that one in one thousand 0s will be read as a 1. The same would be true for 1s being misread as 0s. So all in all, one in five hundred bits received will be in error giving a *bit error rate* (or *BER*) of 0.002. The BER can be reduced by increasing the distance between the signals. Because of the shape of the PDF curve, doubling the 0/1 voltages to ±2 volts would give a dramatic error reduction.

Channels are often characterised graphically by showing BER on a logarithmic axis against a quantity called E_b/N_0. This is a normalized measure of the amount of energy used to signal each bit. You can think of it like the separation of the 0s and 1s in Figure 1.2, while the noise processes which impinge themselves on the signal control the width (or σ) of the Gaussian PDF. The greater E_b/N_0, the smaller the BER. E_b/N_0 can be thought of in a very similar way to *SNR* or *signal to noise ratio*. Figure 1.3 show an example of bit error rate against signal strength

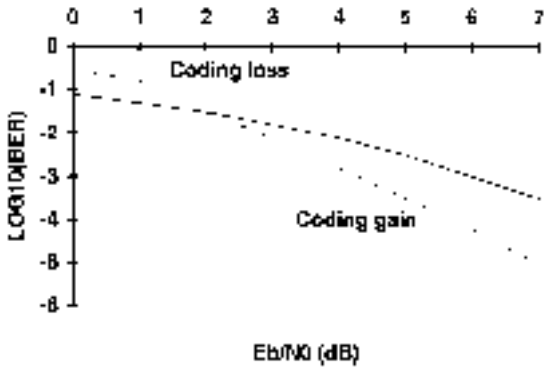

Figure 1.3 BER against E_b/N_0.

The solid line represents an uncoded signal whereas the dotted line is the same channel, but augmented by error coding. This gives rise to an enormously important idea called *coding gain*. At very low powers (and consequently high error rates), the addition of error coding bits to the message burdens it such that an E_b/N_0 of Figure 1.3 in the coded channel, gives the same performance as the uncoded signal with an E_b/N_0 of 0. The reason is that to get the same number of data bits through the channel when coded, more total bits must be sent and so more errors. Because the errors rates are high, the coding is inadequate to offset them (it is exceeded), so

we're actually worse off. However, at signal powers of 2, the coded and uncoded channels have similar performance while above 2, the coded channel has a better performance. The early part of the graph represents coding loss, while the latter, coding gain. So why is this important?

If a particular system requires a BER of no greater than one error in 10^6, then an uncoded channel in this example will need an E_b/N_0 of 5. However, the coded channel can provide this performance with an E_b/N_0 of only 4, so the code is said to give a coding gain of 1dB (E_b/N_0). Put this in terms of a TV relay satellite and it starts to make good economics. Error coding can reduce the required transmitter power or increase its range dramatically which has an major impact on the weight and lifetime of the satellite. Approximately, when error coding is added to a signal, the two distributions in Figure 1.2 get closer together since, for the same transmitter power, more bits are being sent. Normally, this would increase the BER. However, the effect of adding error coding reduces the widths (σ) of the distributions such that the BER actually decreases.

Before leaving random errors, there is another aspect to them which has been usefully exploited in some modern coding schemes. When a random error occurs, the nature of the Gaussian PDF means that the signal is increasingly likely to be near to the decision boundary (0 volts in this example). Rather than make a straight-forward 1/0 decision, each bit can be described by its probability of being a 1 or a 0 (a *soft decision*). Figure 1.4 shows the typical *soft* output.

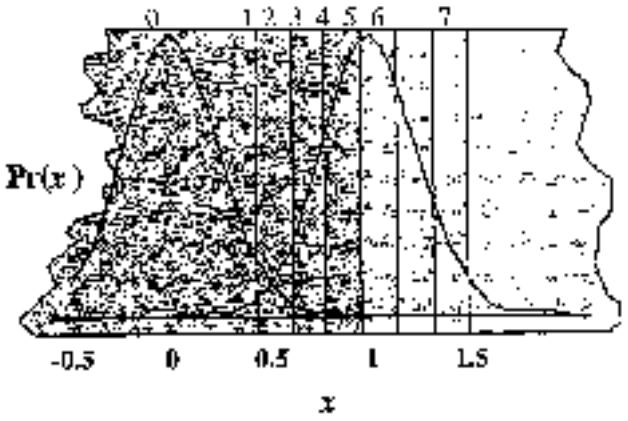

Figure 1.4 Probability distribution with soft boundaries.

A random error that turns a 0 into a 1 is statistically likely to have a low probability of being a 1. In simple terms, the signal undergues an analogue to digital conversion rather than *slicing* so that instead of outputting a 0 or 1, the decoder might output a value between 0 and 7. A good 0 would be 0, while a good 1 would be 7. A weak 0 would be 3, while a weak 1 would be 4. Overall (although not exclusively), random errors will give rise to weak 1s and 0s. If the decoder detects an error, the most likely suspects will be the weaker bits. In essence, what this process does is to increase the resolution of d from whole bits to fractions of bits. This is called *soft decision decoding* and, although not directly relevant to all codes, it helps error coding to approach its theoretical limits.

The second main type of error is called a *burst error*. A burst error is characterized by $b-2$ bits which may or may not be in error, sandwiched in-between two bits that definitely are in error, and preceded by at least b error-free bits. Errors of this kind typically come from electromagnetic transients caused by heavy electrical machinery switching. Deep scratches on the surface of a CD or other media defects might also cause burst errors. These are unpredictable in every way and will normally be handled very differently from random errors. An everyday example of the source of burst errors could be the clicking picked up on the radio when a badly suppressed car engine is running nearby. Returning to the example of a damaged or imperfect CD, many successive bits may be lost in one go. An error code capable of handling this kind of data loss would be excessively large and complex so other processes may be used in conjunction with error coding including *interleaving*.

If you're on of those people who read the technical specs. on your hi-tech appliances in order to compare notes with friends, you'll doubtless have noticed that your CD player boasts "*cross-interleaved Reed-Solomon coding*" (they all do!) Cross-interleaving is a relatively simple process which *decorrelates* or *spreads* errors out. A media defect can give rise to large gouts of lost data so it is necessary to divide the error in order to conquer it. In approximate terms, data is read into a memory row by row and each row is error coded. It is then written onto the media in a column wise basis. Upon reading, the data is arranged back into the memory column-wise, ready to be decoded row-wise. In the event of a large error, even if a whole column is lost, there will still be only one error in each row, which can be easily corrected. Devising efficient spreading algorithms is quite an art in itself and we'll visit this later. Question: Will interleaving help when combatting random errors?

1.5 A Brief History of Error Coding

Modern error coding, while based on maths that was discovered (in some cases) centuries ago, began in the late forties with most of the major innovations occurring over the fifties, sixties and seventies. In 1948 Claude Shannon proved that, if information was transmitted at less than the channel capacity then, it was possible to use the excess bandwidth to transmit error correcting codes to increase arbitrarily the integrity of the message. The race was now on to find out how to do it.

During the early fifties Hamming codes appeared, representing the first practical error correction codes for single-bit errors. Golay codes also appeared at this time allowing the correction of up to three bits. By the mid-fifties Reed–Muller codes had appeared and, by the late fifties, Reed–Solomon codes were born. These last codes were symbol, rather than bit, based and underpin the majority of modern *block* coding. With block codes, encoding and decoding operate on fixed sized blocks of data. Of necessity, the compilation of data into blocks introduces constraints on the message size, and latency into the transmission process, precluding their use in certain applications.

It is important to remember that, while the codes and their algebraic solutions now existed, digital processing was not what it is today. Since Reed–Solomon codes had been shown to be optimal, the search was shifted somewhat towards faster and simpler ways of implementing the coding and decoding processes to facilitate practical systems. To this end, during the mid-sixties, techniques like the Forney and Berlekamp-Massey algorithms appeared. These reduced the processing burdens of decoding to fractions of what had been previously required, heralding modern coding practice as it is today.

Almost in parallel with this sequence of events and discoveries was the introduction in the mid-fifties, by Elias, of convolutional codes as a means of introducing redundancy into a message. These codes differed from the others in that data did not need to be formatted into specific quantities prior to encoding or decoding. Throughout the sixties various methods were created for decoding convolutional codes and this continued until 1967 when Viterbi produced his famous algorithm for decoding convolutional codes which was subsequently shown to be a maximum-likelihood decoding algorithm (i.e., the best you could get).

The eighties contributed in two key ways to error coding, one of which was the application of convolutional coders to modulation, creating today's reliable, high-speed modems. Here the error coding and modulation

functions ceased to be separate processes but were integrated into a single system. The mathematical space generated in messages by error codes was translated into modulation space. The second was the application of so-called algebraic geometry to Reed–Solomon coding. This step created a class of codes called Quasi-Maximum Distance Separable codes, based on elliptic curves. Today, much attention is being given to Turbo codes. These are a class of iteratively decoded convolutional codes which appear to be approaching the theoretical limits of what error coding can achieve.

Clearly error coding as we know it started life in the fifties but, curiously, the first example of both the need for and the application of error coding could go back as far as 1400BC. In 1890, the Russian mathematician Dr Ivan Panin found a rigorous coding scheme that permeates the length and breadth of the bible. The bible's authors must have felt the information was important enough to warrant protection of some sort and, presumably with this in mind, built a very complex heptadic (based on sevens) structure into it. Since both the Old Testament (in Hebrew) and the New Testament (in Greek) share characters which are both letters and numbers, each word also has a numerical value. Apparently, only a simple test is required to see if a book complies with the code and, where two manuscripts are found to differ, it is easy to see which is the more accurate (according to the code).

Coding of this nature is asymmetric. While it is simple to decode, the encoding process is unknown to date and hasn't been reproduced on any significant scale. Panin, however, used the code to create his own version of parts of the bible, correcting any deviations from the heptadic structure.

1.6 Summary

While the introduction is in fairly broad brush-strokes, the ideas covered are central to error coding. Space, in one form or another, is what makes error coding possible. The Hamming and Gilbert boundaries link the amount of space required to the message size and error correcting capability. The maths that I included at this point needs a little thought. If equations like (1.3) are new to you, work through a few simple examples with small numbers that are manageable. Take, for example, four red snooker balls and three blue. See how many different ways you can arrange the blues between the reds then compare your results with

$$\frac{7!}{3!4!} = 35.$$

You've actually calculated the number of different ways that you can change three bits in a total of seven. Take any 7-bit message and there are 35 other messages that are 3 bits distance from it. At this point, you might even be able to construct an error code using the Gilbert bound coupled with a random code selection and deletion process.

Error types and sources have been considered, leading into processes which can augment error coding like soft-decision decoding and interleaving. When all these ideas are brought together, some spectacular results emerge. Messages that come back from deep space probes can be buried deeply in noise and yet, with error coding, the data are still recoverable. Digital TV places tight constraints on acceptable BER and yet, with error coding, it's possible without enormous transmitter power. CDs and especially DAT (Digital Audio Tape) players simply wouldn't work without error coding. Systems like modems combine error coding with signal modulation to provide communications almost at the limit of what is theoretically possible. So what we need now are some practical schemes for implementing error coding and decoding.

Chapter 2

A LITTLE MATHS

This chapter introduces some of the maths that is required in order to perform certain block coding schemes. Treatment is from a practical, rather than theoretical, point of view, always with hardware implementation in mind. No attempt is made to prove the ideas considered here since there are plenty of excellent texts on the subject and (mathematicians look away now) generally in engineering, "if it works twice, it's a law". No one who has ever used maths for anything practical can have failed to notice how amazing it is that diverse approaches to problems converge upon common answers. Logic, of course, predicts this and this is also why mathematicians go to great lengths to prove that "it wasn't a fluke and it'll work more than twice".

Whole number maths, which underpins much error coding, is one of those amazing subjects which opens up a whole vista of possibilities for engineers. Imagine being able to perform all sorts of complex operations using only a finite set of numbers, and never having to worry about fractions! Hopefully this chapter will be fun, rather than onerous and, if you've never heard of finite fields are, you're in for a pleasant surprise.

A. Houghton, *Error Coding for Engineers*