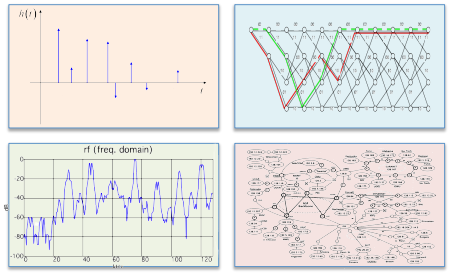


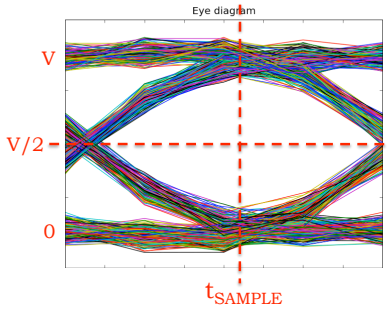
INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

6.02 Spring 2009
Lecture #7

- Dealing with errors through channel coding
- Hamming distance & error detection
- Parity, Checksums, CRC
- Error correction using multiple parity bits
- (n,k) block codes, interleaving

There's good news and bad news…

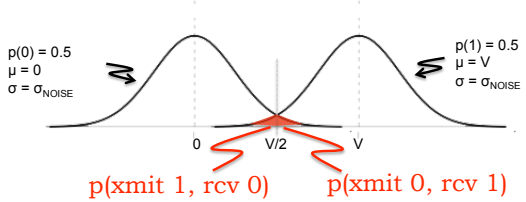


The good news: Our digital signaling scheme usually allows us to recover the original signal despite small amplitude errors introduced by inter-symbol interference and noise. An example of the digital abstraction doing its job!

The bad news: larger amplitude errors (hopefully infrequent) that change the signal irretrievably. These show up as bit errors in our digital data stream.

Bit Errors

Assuming a Gaussian PDF for noise and no only 1 bit of inter-symbol interference, samples at t_{SAMPLE} have the following PDF:



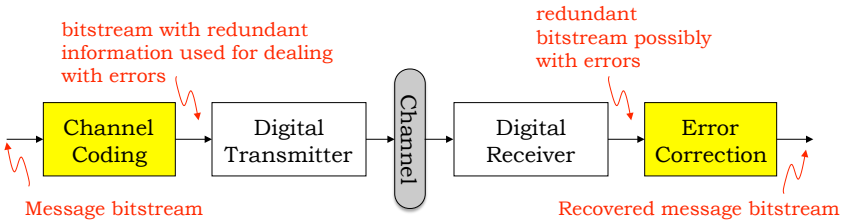
We can estimate the bit-error rate (BER) using Φ, the unit normal cumulative distribution function:

$$BER = (0.5)\left[1-\Phi\left[\frac{V/2-0}{\sigma_{NOISE}}\right]\right]+(0.5)\Phi\left[\frac{V/2-V}{\sigma_{NOISE}}\right]=\Phi\left[\frac{-V/2}{\sigma_{NOISE}}\right]$$

For a given σ_{NOISE}, if you want a smaller BER, you need a larger V!

Channel coding

Our plan to deal with bit errors:



We'll add redundant information to the transmitted bit stream (a process called channel coding) so that we can detect errors at the receiver. Ideally we'd like to correct commonly occurring errors, e.g., error bursts of bounded length. Otherwise, we should detect uncorrectable errors and use, say, retransmission to deal with the problem.

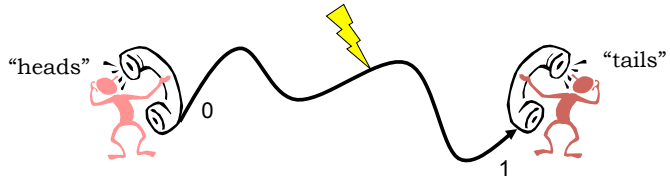
Error detection and correction

Suppose we wanted to reliably transmit the result of a single coin flip:

Heads: "0" Tails: "1"

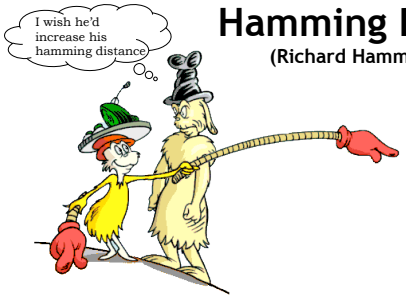
This is a prototype of the "bit" coin for the new information economy. Value = 12.5¢

Further suppose that during transmission a single-bit error occurs, i.e., a single "0" is turned into a "1" or a "1" is turned into a "0".



Hamming Distance

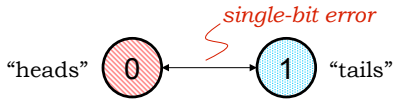
(Richard Hamming, 1950)



HAMMING DISTANCE: The number of digit positions in which the corresponding digits of two encodings of the same length are different

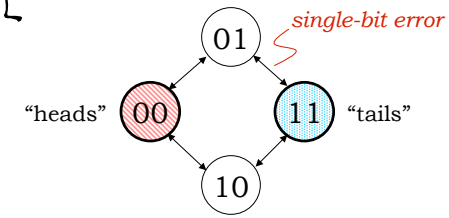
The Hamming distance between a valid binary code word and the same code word with single-bit error is 1.

The problem with our simple encoding is that the two valid code words ("0" and "1") also have a Hamming distance of 1. So a single-bit error changes a valid code word into another valid code word…



Error Detection

What we need is an encoding where a single-bit error doesn't produce another valid code word.



If D is the minimum Hamming distance between code words, we can detect up to (D-1)-bit errors

We can add single-bit error detection to any length code word by adding a *parity bit* chosen to guarantee the Hamming distance between any two valid code words is at least 2. In the diagram above, we're using "even parity" where the added bit is chosen to make the total number of 1's in the code word even.

Parity check

- A parity bit can be added to any length message and is chosen to make the total number of "1" bits even (aka "even parity").
- To check for a single-bit error (actually any odd number of errors), count the number of "1"s in the received word and if it's odd, there's been an error.

0 1 1 0 0 1 0 1 0 0 1 1 → original word with parity
0 1 1 0 0 0 0 1 0 0 1 1 → single-bit error (detected)
0 1 1 0 0 0 1 1 0 0 1 1 → 2-bit error (not detected)

- One can "count" by summing the bits in the word modulo 2 (which is equivalent to XOR'ing the bits together).

Checksums

- Simple checksum
  - Add up all the message units and send along sum
  - Easy for two errors to offset one another
    - Some 0 bit changed to 1; 1 bit in same position in another message unit changed 0… sum is unchanged
- Weighted checksum
  - Add up all the message units, each weighted by its index in the message, send along sum
  - Still too easy for two errors to offset one another
- Both! Adler-32
  - A = (1 + sum of message units) mod 65521
  - B = (sum of A_i after each message unit) mod 65521
    = $(n + n*m_1 + (n-1)*m_2 + \ldots + 1*m_n)$ mod 65521
  - Send 32-bit quantity (B << 16) + A
  - Cheaper in SW, but not quite as good as a CRC
  - Not good for short messages

Cyclical Redundancy Check

- Choose a "generator" bit string G of length r+1
  - G should always be of the form 1…..1, G = 11 is parity
- Using modulo 2 arithmetic, divide padded message bit string [M << r] by G, getting remainder R
  - $R_{len} \le r$
  - [M<<r] = Q*G + R, so ([M<<r] – R)/G has remainder 0
  - Send (M<<r)-R = $M_{m-1}M_{m-2}\ldots M_0R_{r-1}R_{r-2}\ldots R_0$ = T
  - Receiver calculates T/G and checks if remainder is 0, reports error if it's not
- CRC-16: G = 11000000000000101
  - Used in USB, many other links
  - Detects single- and double-bit errors, all odd numbers of errors, all errors with burst lengths < 16, and 99.984% of all other errors

CRC Example

M = 1101011011, G = 10011

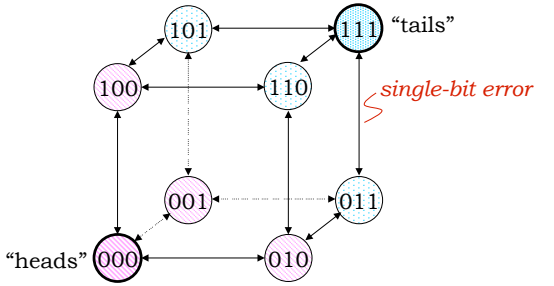
```
          1100001010
10011 | 11010110110000
        10011
         10011
         10011
          00001
          00000
           00010
           00000
            00101
            00000
             01011
             00000
              10110
              10011
               01010
               00000
                10100
                10011
                 01110
                 00000
                  1110
```

Message bits

Remainder bits

T = 11010110111110

Error Correction



If D is the minimum Hamming distance between code words, we can correct up to $\left\lfloor \frac{D-1}{2} \right\rfloor$ bit errors

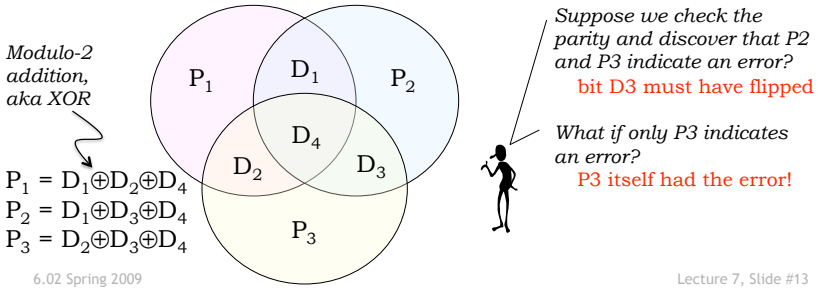
By increasing the Hamming distance between valid code words to 3, we guarantee that the sets of words produced by single-bit errors don't overlap. So if we detect an error, we can perform *error correction* since we can tell what the valid code was before the error happened.

- Can we safely detect double-bit errors while correcting 1-bit errors?
- Do we always need to triple the number of bits?

Error Correcting Codes (ECC)

Basic idea:

- Use multiple parity bits, each covering a subset of the data bits.
- No two message bits belong to exactly the same subsets, so a single-bit error will generate a unique set of parity check errors.



Checking the parity

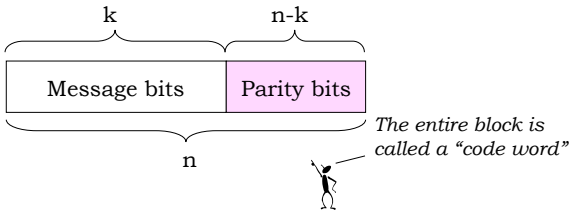
- Transmit: Compute the parity bits and send them along with the message bits
- Receive: After receiving the (possibly corrupted) message, compute a syndrome bit (E_i) for each parity bit. For the code on slide #13:

$$E_1 = D_1 \oplus D_2 \oplus D_4 \oplus P_1$$
$$E_2 = D_1 \oplus D_3 \oplus D_4 \oplus P_2$$
$$E_3 = D_2 \oplus D_3 \oplus D_4 \oplus P_3$$

- If all the E_i are zero: no errors!
- Otherwise the particular combination of the E_i can be used to figure out which bit to correct.

(n,k) Systematic Block Codes

- Split message into *k*-bit blocks
- Add (*n-k*) parity bits to each block, making each block *n* bits long.



If we want to correct single-bit errors, how many parity bits do we need?

How many parity bits?

- We have n-k parity bits and so can calculate n-k parity-check bits, which collectively can represent 2^{n-k} possibilities. To do single-bit error correction parity-check bits need to represent
  - When no error has occurred (1 possibility)
  - When exactly one of the n code word bits has an error; parity bits can get errors too! (n possibilities)
  - So we need $n+1 \le 2^{n-k}$ or

$$n \le 2^{n-k} - 1$$

- Hamming single-error correcting codes (SECC) have $n = 2^{n-k} - 1$
  - (7,4), (15,11), (31, 26)

The (7,4) Hamming SECC code is shown on slide 13

Which (n,k) code does one use?

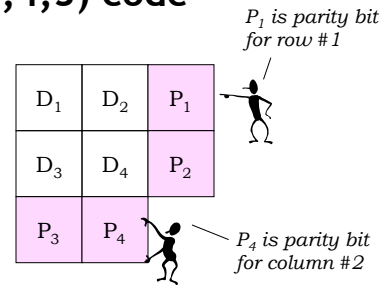
- The minimum Hamming distance *d* between code words determines how we can use code:
  - To detect D-bit errors: d > D
  - To detect and correct C-bit errors: d > 2C
  - Sometimes code names include min Hamming distance: (n,k,d)
- With a (n,k,3) code one can
  - Detect 1-bit and 2-bit errors OR
  - Detect and correct 1-bit errors (SEC)
- With a (n,k,4) code one can
  - Detect 1-bit, 2-bit and 3-bit errors OR
  - Detect and correct 1-bit errors, and detect 2-bit errors (SECDED)
- With a (n,k,d) code and for c < d/2 one can
  - Detect and correct c-bit errors, and
  - If c+1 < d/2, detect (c+1)-bit up to (d-c-1)-bit errors

Error-Correcting Codes

- To conserve bandwidth want to maximize a code's *code rate*, defined as k/n.
- Parity is a (n+1,n,2) code
  - Good code rate, but only 1-bit error detection
- Replicating each bit r times is a (r,1,r) code
  - Simple way to get great error correction; poor code rate
  - Handy for solving quiz problems!
- Hamming single-error correcting codes (SECC) are (n,n-p,3) where $n = 2^p-1$ for $p > 1$
  - Adding an overall parity bit makes the code (n+1,n-p,4)

A simple (8,4,3) code

Idea: start with rectangular array of data bits, add parity checks for each row and column. Single-bit error in data will show up as parity errors in a particular row and column, pinpointing the bit that has the error.



0 1 1
1 1 0
1 0

Parity for each row and column is correct ⇒ no errors

0 1 1
1 0 0
1 0

Parity check fails for row #2 and column #2 ⇒ bit D_4 is incorrect

0 1 1
1 1 1
1 0

Parity check only fails for row #2 ⇒ bit P_2 is incorrect

If we add an overall parity bit P_5, we get a (9,4,4) code!

Matrix notation for block codes

Task: given k-bit message, compute n-bit code word. We can use standard matrix arithmetic (modulo 2) to do the job. For example, here's how we would describe the (9,4,4) code from slide #20:

$$[D_1\ D_2\ D_3\ D_4] \bullet \begin{bmatrix} 1&0&0&0&1&0&1&0&1 \\ 0&1&0&0&1&0&0&1&1 \\ 0&0&1&0&0&1&1&0&1 \\ 0&0&0&1&0&1&0&1&1 \end{bmatrix} = [D_1\ D_2\ D_3\ D_4\ P_1\ P_2\ P_3\ P_4\ P_5]$$

1×k message vector

k×n generator matrix

1×n code word vector

The generator matrix has the form $\left[I_{k\times k} \mid P_{k\times(n-k)} \right]$

Syndrome Matrix

Task: given n-bit code word, compute (n-k) syndrome bits. Again we can use matrix multiply to do the job. Here's how to compute the syndrome bits for the (9,4,4) code:

$$[D_0\ D_1\ D_2\ D_3\ P_1\ P_2\ P_3\ P_4\ P_5] \bullet \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = [E_1\ E_2\ E_3\ E_4\ E_5]$$

1×n code word vector

n×(n-k) check matrix

1×(n-k) syndrome vector

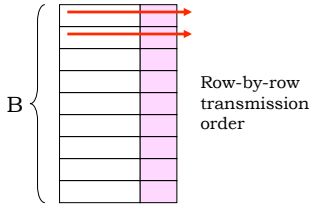
The check matrix has the form $\left[\frac{P_{k\times(n-k)}}{I^{**}_{(n-k)\times(n-k)}}\right]$

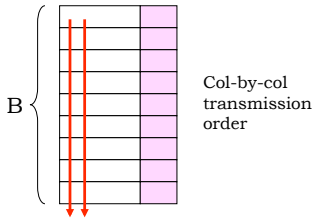
*** note the last column is all ones – the overall parity check covers parity bits too!*

Burst Errors

Correcting single-bit errors is nice, but in many situations errors come in bursts many bits long (e.g., damage to storage media, burst of interference on wireless channel, …). How does single-bit error correction help with that?

Well, can we think of a way to turn a B-bit error burst into B single-bit errors?





Problem: Bits from a particular codeword are transmitted sequentially, so a B-bit burst produces multi-bit errors.

Solution: interleave bits from B different codewords. Now a B-bit burst produces 1-bit errors in B different codewords.