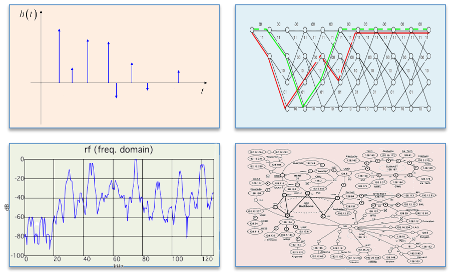
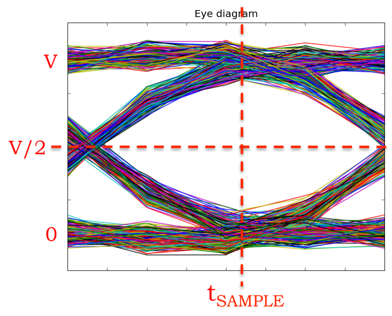


INTRODUCTION TO EECS II
DIGITAL COMMUNICATION SYSTEMS

6.02 Spring 2011
Lecture #8

• Coping with errors using packets
• Detecting errors: checksums, CRC
• Hamming distance & single error correction
• (n,k) block codes

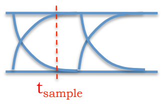
There's good news and bad news…



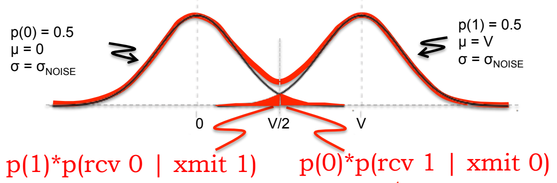
The good news: Our digital signaling scheme usually allows us to recover the original signal despite small amplitude errors introduced by inter-symbol interference and noise. An example of the digital abstraction doing its job!

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 only 1-bit of inter-symbol interference, samples at t_{SAMPLE} have the following PDF:



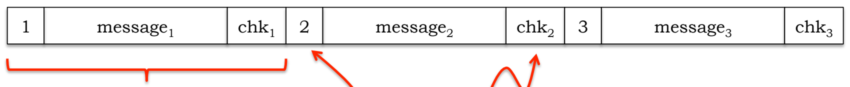
We can estimate the bit-error rate (BER) using Φ, the unit normal cumulative distribution function:

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

For a smaller BER, you need a smaller σ_{NOISE} or a larger V!

Dealing With Errors: Packets

To deal with errors, divide message into fixed-sized packets, which are transmitted one after another.



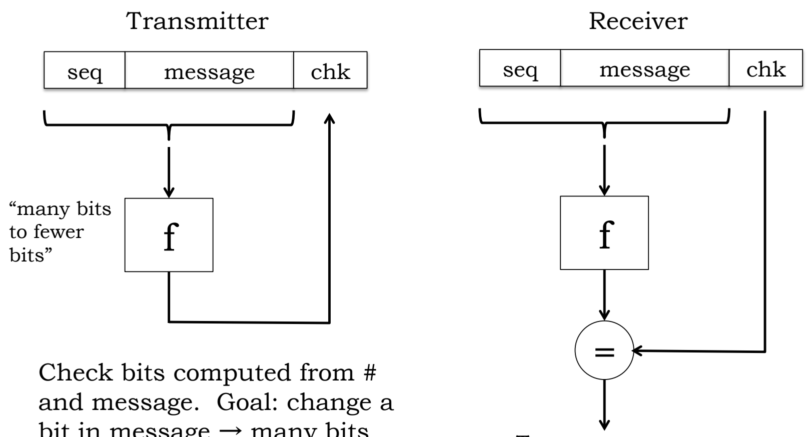
Packet = {#, message, chk}

Sequence number provides unique identifier for each packet.

Check bits are redundant information that lets receiver verify # and message. Failure? Ask for packet to be resent.

Packet size: Too small → #/chk overhead is large
Too big → p(error) is larger, more to resend

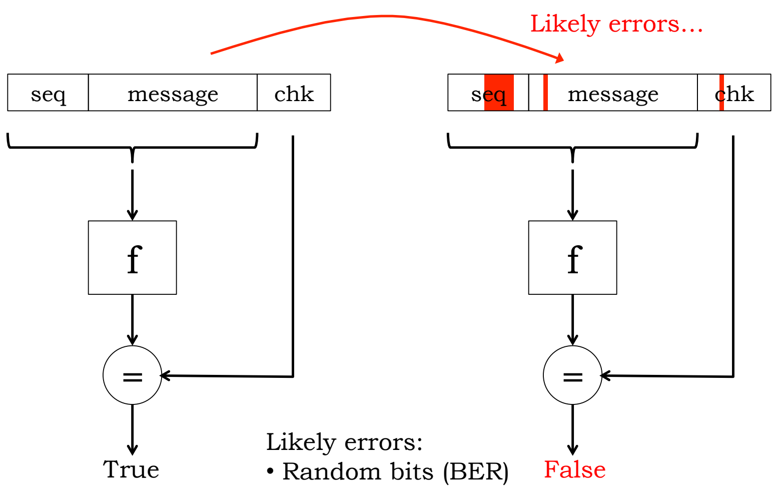
Check bits



Check bits computed from # and message. Goal: change a bit in message → many bits change in check bits.

True: no errors
False: errors

Detecting Errors



Likely errors:
• Random bits (BER)
• Error bursts

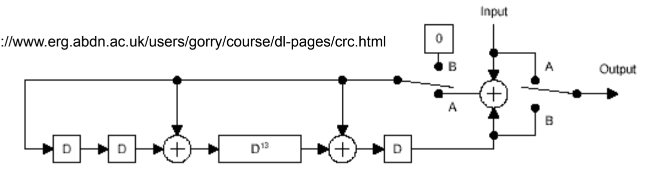
Checksums

- Simple checksum
  - Add up all the message units, send along sum
  - Easy for two errors to mask one another
    - Some 0 bit changed to a 1; 1 bit in same position in another message unit changed to a 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
  - Send 32-bit quantity (B<<16) + A
  - Good in software, not good for short messages

Cyclical Redundancy Check

Example: CRC-16

http://www.erg.abdn.ac.uk/users/gorry/course/dl-pages/crc.html



Sending: Initialize all D elements to 0. Set switch to position A, send message bit-by-bit. When complete, set switch to position B and send 16 more bits.

Receiving: Initialize all D elements to 0. Set switch to position A, receive message and CRC bit-by-bit. If correct, all D elements should be 0 after last bit has been processed.

CRC-16 detects all single- and double-bit errors, all odd numbers of errors, all errors with burst lengths < 16, and a large fraction ($1-2^{-16}$) of all other bursts.

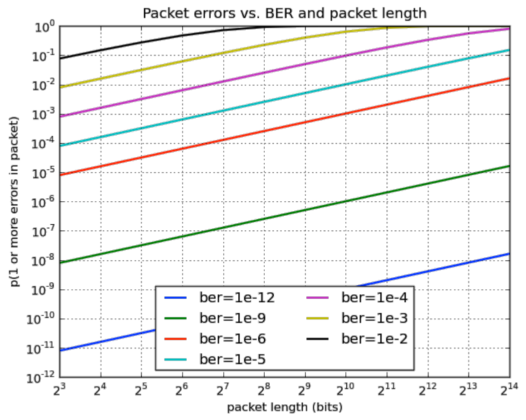
Approximate BER for common channels

Channel type	Bandwidth	BER
Telephone Landline	2 Mbits/sec	10^{-4} to 10^{-6}
Twisted pair (differential)	1 Gbits/sec	$\leq 10^{-7}$
Coaxial cable	100 Mbits/sec	$\leq 10^{-6}$
Fiber Optics	10 Tbits/sec	$\leq 10^{-9}$
Infrared	2 Mbits/sec	10^{-4} to 10^{-6}
3G cellular	1 Mbits/sec	10^{-4}

Source: Rahmani, et al, *Error Detection Capabilities of Automotive Technologies and Ethernet – A Comparative Study*, 2007 IEEE Intelligent Vehicles Symposium, p 674-679

How Frequent is Packet Retransmission?

p(1 or more errors) = 1 – p(no errors) = 1 – $(1 – \text{BER})^k$

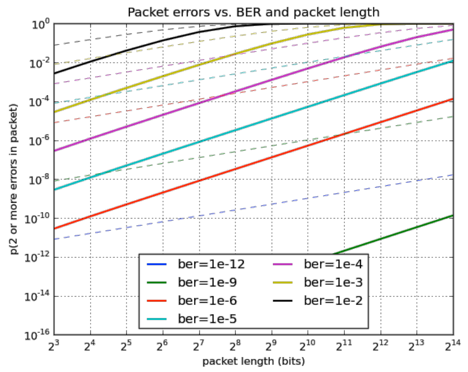


With 1kbyte packets and BER=1e-6, retransmit 1 every 100.

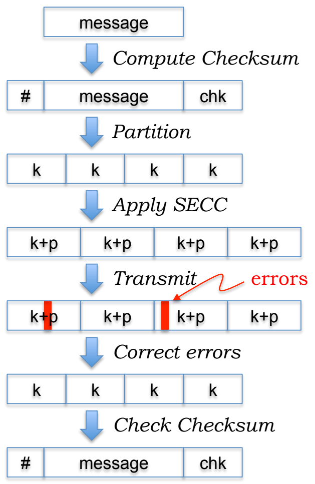
Implement Single Error Correction?

To reduce retransmission rate, suppose we invent a scheme that can correct single-bit errors and apply it to sub-blocks of the data packet (effectively reducing k). Does that help?

p(2 or more errors) = 1 – p(no errors) – p(exactly one error)
= 1 – $(1 – \text{BER})^k$ – k*BER*$(1\text{-BER})^{k-1}$



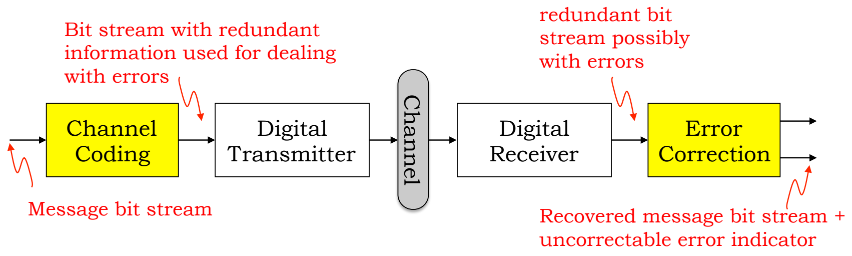
Digital Transmission using SECC



- Start with original message
- Add checksum to enable verification of error-free transmission
- Apply SECC, adding parity bits to each k-bit block of the message. Number of parity bits (p) depends on code:
  - Replication: p grows as O(k)
  - Rectangular: p grows as O(√k)
  - Hamming: p grows as O(log k)
- After xmit, correct errors
- Verify checksum, fails if undetected/uncorrectable error
- Deliver or discard message

Channel coding

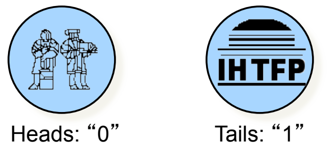
Our plan to deal with bit errors:


Bit stream with redundant information used for dealing with errors
redundant bit stream possibly with errors
Channel Coding
Digital Transmitter
Channel
Digital Receiver
Error Correction
Message bit stream
Recovered message bit stream + uncorrectable error indicator

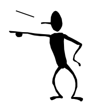
We'll add redundant information to the transmitted bit stream (a process called channel coding) so that we can detect errors at the receiver. Ideally we'd like to correct commonly occurring errors, e.g., error bursts of bounded length. Otherwise, we should detect uncorrectable errors and use, say, retransmission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

Error detection and correction

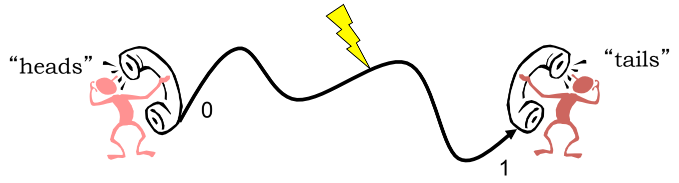
Suppose we wanted to reliably transmit the result of a single coin flip:



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

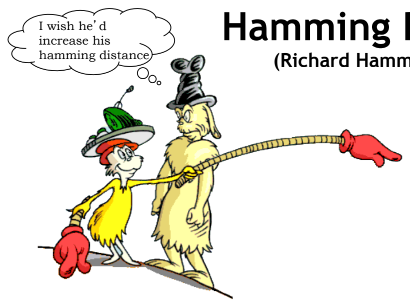


Further suppose that during transmission a single-bit error occurs, i.e., a single "0" is turned into a "1" or a "1" is turned into a "0".



Hamming Distance

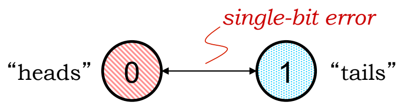
(Richard Hamming, 1950)



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

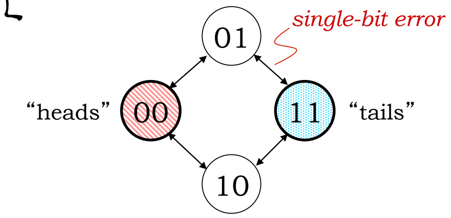
The Hamming distance between a valid binary code word and the same code word with single-bit error is 1.

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 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 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.

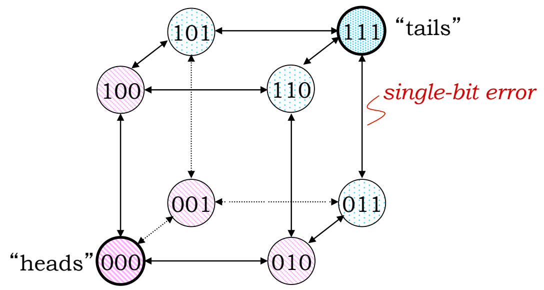
Parity check

- 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 message 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).

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?

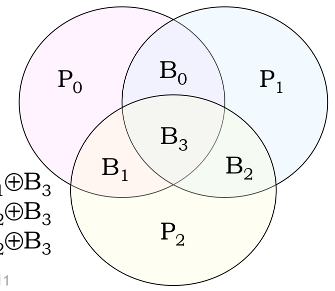
Single Error Correcting Codes (SECC)

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 error will generate a unique set of parity check errors.

Modulo-2 addition, aka XOR

$P_0 = B_0 \oplus B_1 \oplus B_3$
$P_1 = B_0 \oplus B_2 \oplus B_3$
$P_2 = B_1 \oplus B_2 \oplus B_3$



Suppose we check the parity and discover that P1 and P2 indicate an error?
bit B2 must have flipped

What if only P2 indicates an error?
P2 itself had the error!

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 previous slide:

$$E_0 = B_0 \oplus B_1 \oplus B_3 \oplus P_0$$
$$E_1 = B_0 \oplus B_2 \oplus B_3 \oplus P_1$$
$$E_2 = B_1 \oplus B_2 \oplus B_3 \oplus P_2$$

- 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.

Using the Syndrome to Correct Errors

Continuing example from previous slides: there are three syndrome bits, giving us a total of 8 encodings.

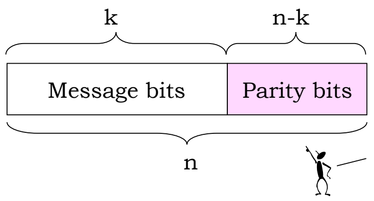
$E_2E_1E_0$	Single Error Correction
0 0 0	No errors
0 0 1	P0 has an error, flip to correct
0 1 0	P1 has an error, flip to correct
0 1 1	B0 has an error, flip to correct
1 0 0	P2 has an error, flip to correct
1 0 1	B1 has an error, flip to correct
1 1 0	B2 has an error, flip to correct
1 1 1	B3 has an error, flip to correct

What happens if there is more than one error?

The 8 encodings indicate the 8 possible correction actions: no errors, error in one of 4 data bits, error in one of 3 parity bits

(n,k,d) Systematic Block Codes

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

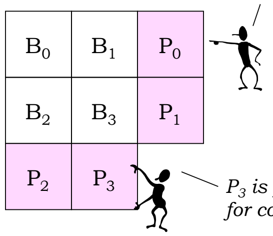


The entire block is called a "code word" and this is an (n,k) code.

- Often we'll use the notation (n,k,d) where d is the minimum Hamming distance between code words.
- The ratio k/n is called the *code rate* and is a measure of the code's overhead (always ≤ 1, larger is better).

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.



P_0 is parity bit for row #1

P_3 is parity bit for column #2

```
0 1 1        0 1 1        0 1 1
1 1 0        1 0 0        1 1 1
1 0          1 0          1 0
```

Parity for each row and column is correct ⇒ no errors

Parity check fails for row #2 and column #2 ⇒ bit B_3 is incorrect

Parity check only fails for row #2 ⇒ bit P_1 is incorrect

Can you verify this code has a Hamming distance of 3?

How many parity bits are needed?

- Suppose we want to do single-bit error correction
  - Need unique combination of syndrome bits for each possible single bit error + no errors
  - n-bit blocks → n possible single bit errors
  - Syndrome bits all zero → no errors
- Assume n-k parity bits (out of n total bits)
  - Hence there are n-k syndrome bits
  - $2^{n-k} - 1$ non-zero combinations of n-k syndrome bits
- So, at a minimum, we need $n \leq 2^{n-k} - 1$
  - Given k, use constraint to determine minimum n needed to ensure single error correction is possible
  - (n,k) Hamming SECC codes: (7,4) (15,11) (31,26)

The (7,4) Hamming SECC code is shown on slide 19, see the Notes for details on constructing the Hamming codes. The clever construction makes the syndrome bits into the index needing correction.