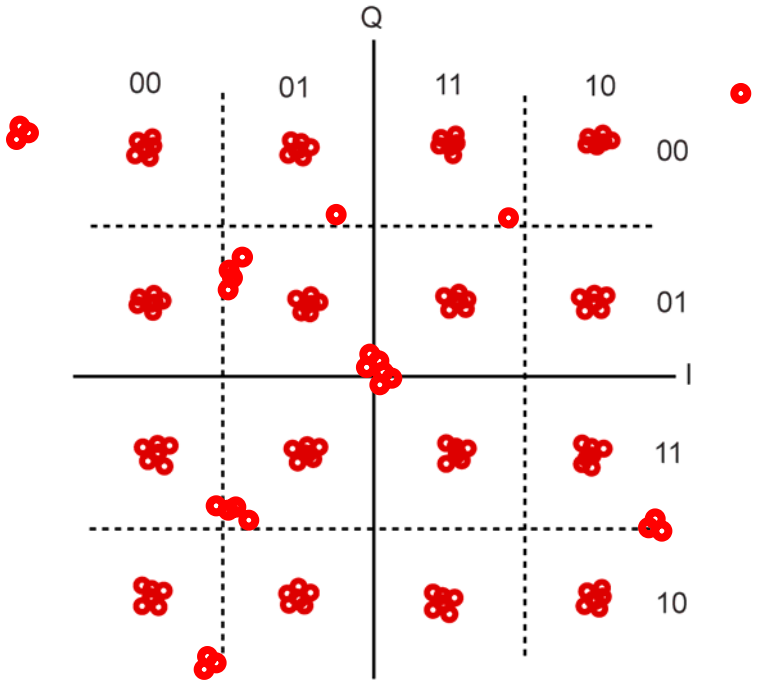
Detecting and Correcting Errors

- Codewords and Hamming Distance
- Error Detection: parity
- Single-bit Error Correction
- Burst Error Correction
- Framing

There's good news and bad news…

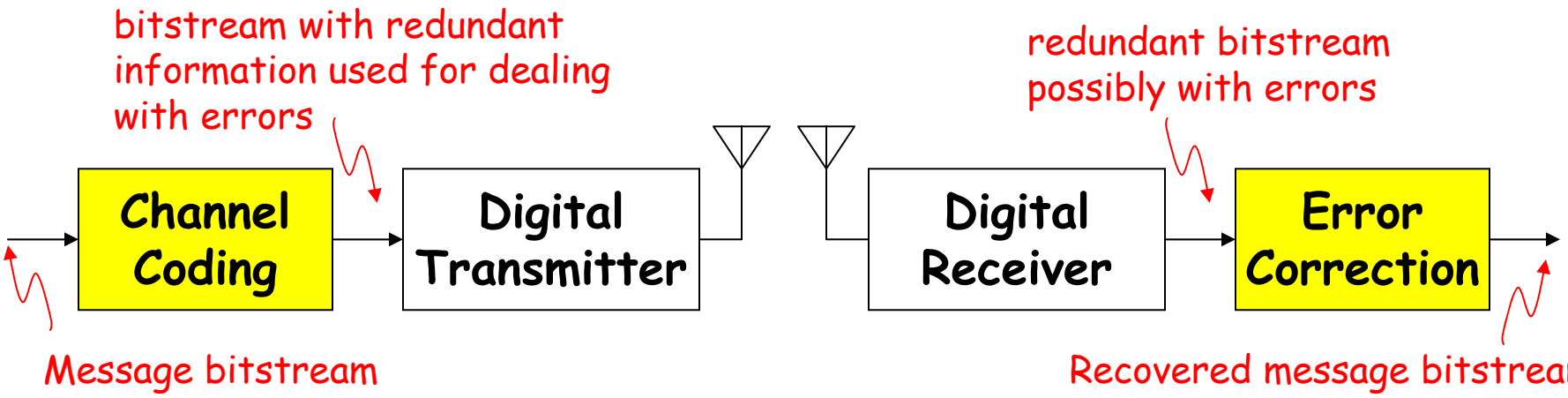


The good news: Our digital modulation scheme usually allows us to recover the original signal despite small amplitude errors introduced by the components and channel. An example of the digital abstraction doing its job!

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

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.

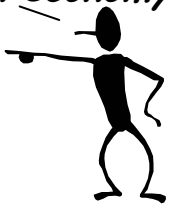
Error detection and correction

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

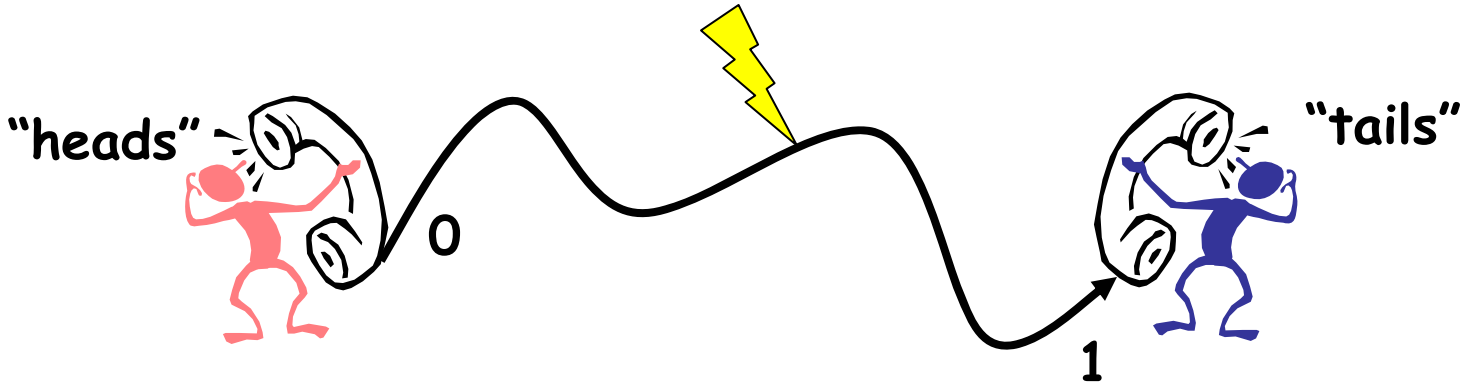
Heads: "0"

Tails: "1"

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

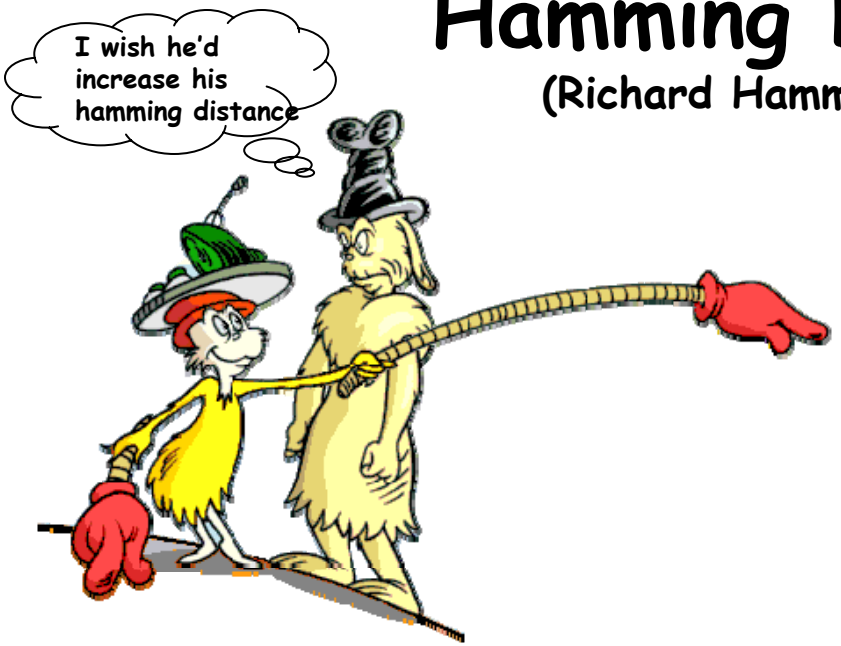


Further suppose that during transmission a single-bit error occurs, i.e., a single "0" is turned into a "1" or a "1" is turned into a "0".



Hamming Distance

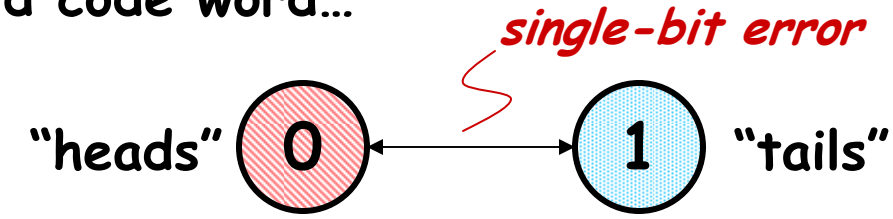
(Richard Hamming, 1950)



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

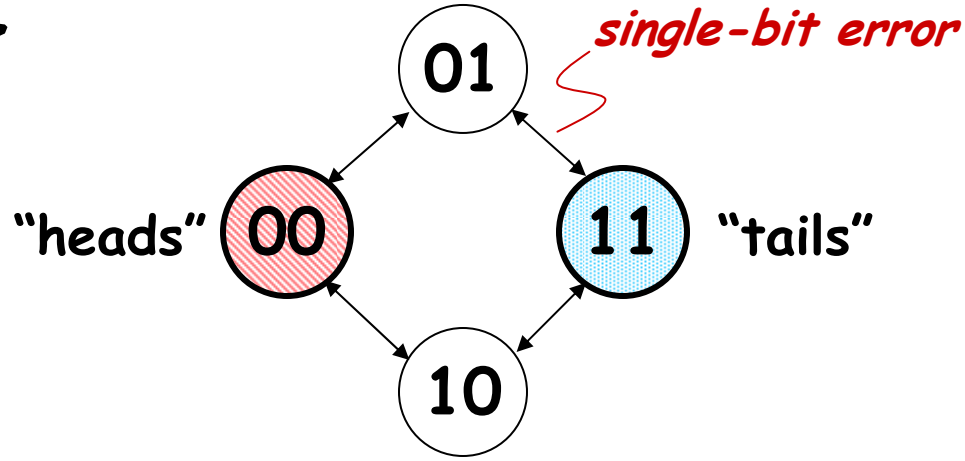
The Hamming distance between a valid binary code word and the same code word with single-bit error is 1.

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.

Can we correct detected errors? Not yet…

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.

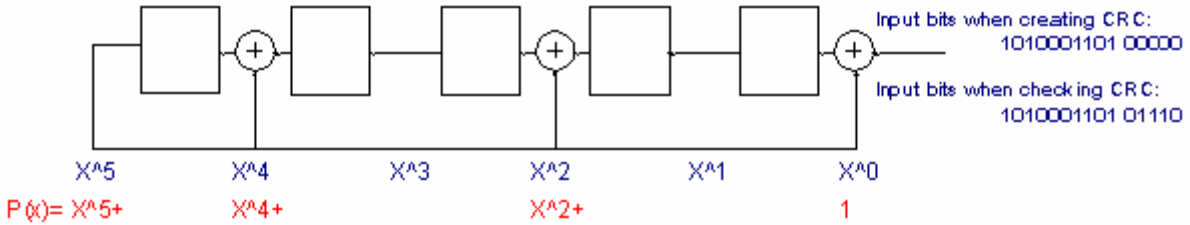
0 1 1 0 0 1 0 1 0 0 1 1 → original word with parity
0 1 1 0 0 **0** 0 1 0 0 1 1 → single-bit error (detected)
0 1 1 0 0 **0 1** 1 0 0 1 1 → 2-bit error (not detected)

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

Checksum, CRC

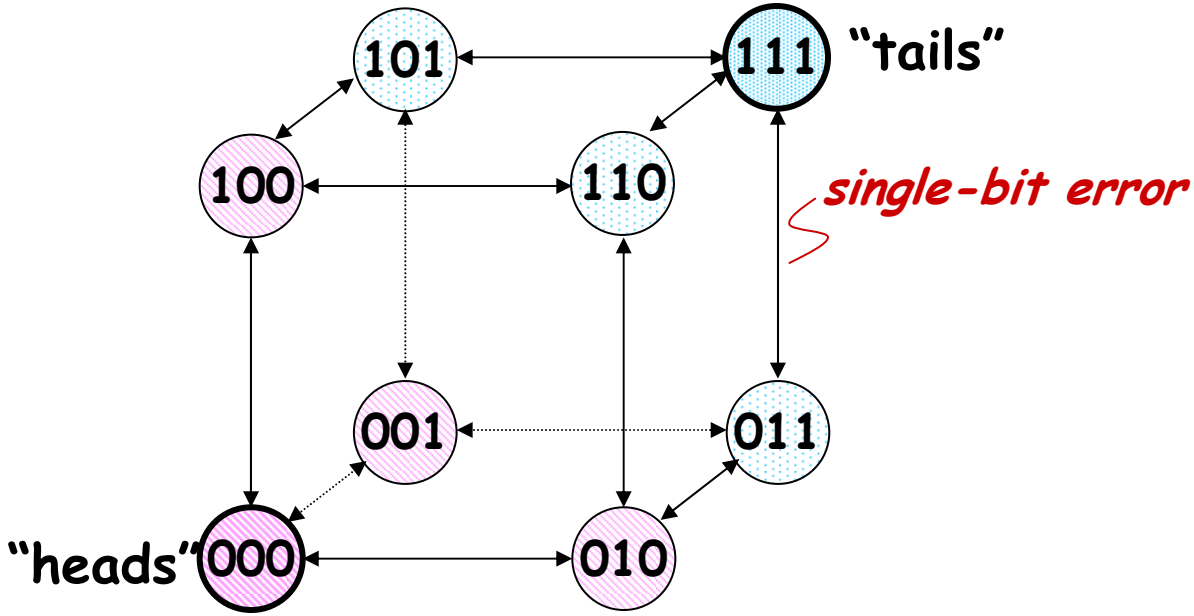
Other ways to detect errors:

- **Checksum:**
  - **Add up all the message units and send along sum**
  - **Adler-32: two 16-bit mod-65521 sums A and B, A is sum of all the bytes in the message, B is the sum of the A values after each addition.**
- **Cyclical redunancy check (CRC)**



The same circuit is used for both creating and checking the N-bit CRC. When creating the check bits, the circuit accepts the bits of the message and then an N-bit sequence of zeros. The final contents of the shift register (the check bits) will be appended to the message. When checking for errors, the circuit accepts the bits of the received message (message + check bits, perhaps corrupted by errors). After all bits have been processed, if the contents of the shift register is not zero, an error has been detected.

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

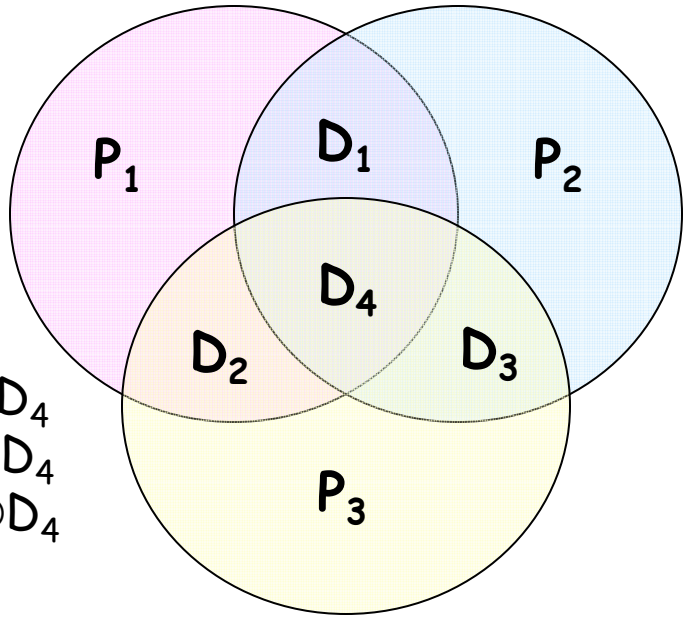
By increasing the Hamming distance between valid code words to 3, we guarantee that the sets of words produced by single-bit errors don't overlap. So if we detect an error, we can perform *error correction* since we can tell what the valid code was before the error happened.

- 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:

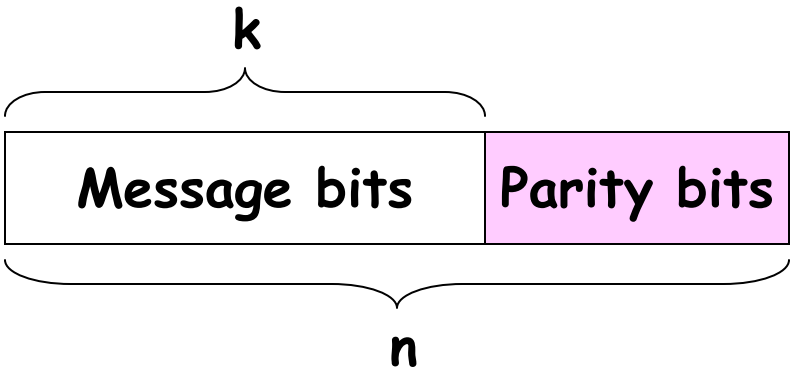
- Use multiple parity bits, each covering a subset of the data bits.
- The subsets overlap, ie, each data bit belongs to multiple subsets
- No two data bits belong to exactly the same subsets, so a single-bit error will generate a unique set of parity check errors



$P_1 = D_1 \oplus D_2 \oplus D_4$
$P_2 = D_1 \oplus D_3 \oplus D_4$
$P_3 = D_2 \oplus D_3 \oplus D_4$

Suppose we check the parity and discover that P2 and P3 indicate an error?
bit D3 must have flipped

What if only P3 indicates an error?
P3 itself had the error!



The entire block is called a "codeword"

(n,k) Block Codes

- Split message into k-bit blocks
- Add (n-k) parity bits to each block, making each block n bits long. How many parity bits do we need to correct single-bit errors?
  - Need enough combinations of parity bit values to identify which of the n bits has the error (remember that parity bits can get errors too!), or to indicate no error at all, so $2^{n-k} \geq n+1$ or, equivalently, $2^{n-k} > n$
  - Multiply both sides by 2^k and divide by n: $2^n/n > 2^k$
  - Most efficient (i.e., we use all possible encodings of the parity bits) when $2^{n-k} - 1 = n$
    - (7,4), (15,11), (31, 26) Hamming codes

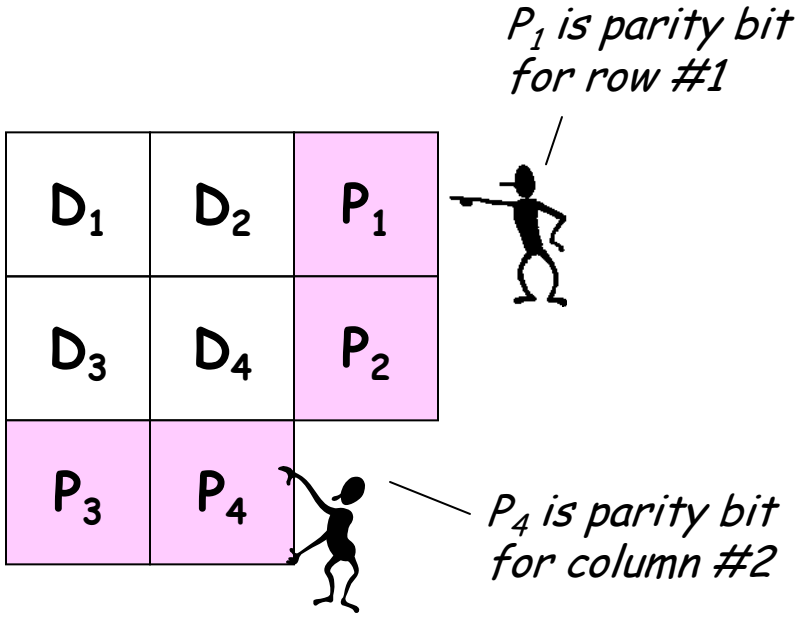
This code is shown on the previous slide

What (n,k) code does one use?

- The minimum Hamming distance d between codewords determines how we can use code:
  - To detect E-bit errors: D > E
  - To correct E-bit errors: D > 2E
  - So to correct 1-bit errors or detect 2-bit errors we need d ≥ 3. To do *both*, we need d ≥ 4 in order to avoid double-bit errors being interpreted as correctable single-bit errors.
  - Sometimes code names include min Hamming distance: (n,k,d)
- To conserve bandwidth want to maximize a code's *code rate*, defined as k/n.
- Parity is a (n+1,n,2) code
  - Efficient, but only 1-bit error detection
- Replicating each bit r times is a (r,1,r) code
  - Simple way to get great error correction, but inefficient

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.



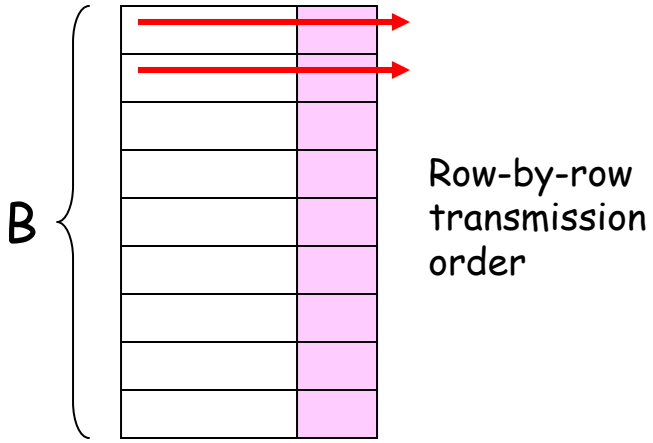
0 1 1	0 1 1	0 1 1
1 1 0	1 0 0	1 1 1
1 0	1 0	1 0
Parity for each row and column is correct ⇒ no errors	Parity check fails for row #2 and column #2 ⇒ bit D_4 is incorrect	Parity check only fails for row #2 ⇒ bit P_2 is incorrect

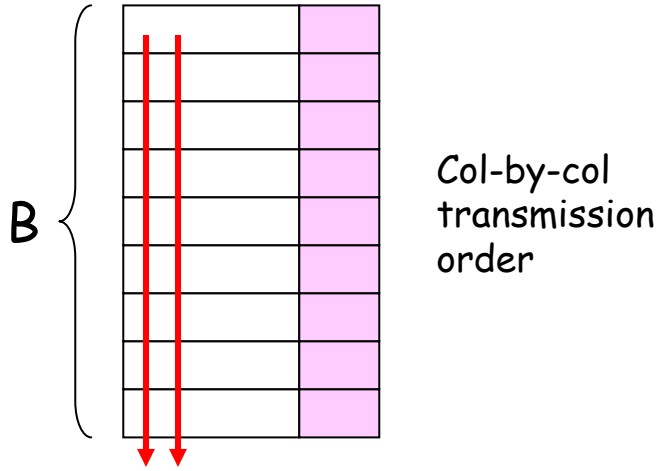
If we add an overall parity bit P_5, we get a (9,4,4) code!

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?

Burst Errors

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.

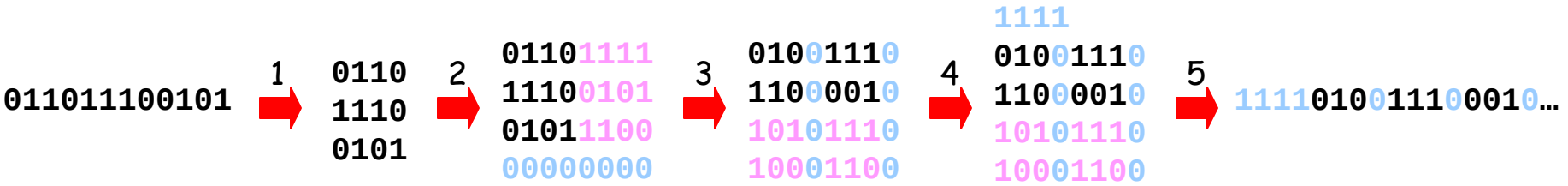
Solution: interleave bits from B different codewords. Now a B-bit burst produces 1-bit errors in B different codewords.

Framing

- Looking at a received bit stream, how do we know where a block of interleaved codewords begins?
- Physical indication (transmitter turns on, beginning of disk sector, separate control channel)
- Place a unique bit pattern (sync) in the bit stream to mark start of a block
  - Frame = sync pattern + interleaved code word block
  - Search for sync pattern in bit stream to find start of frame
  - Bit pattern can't appear elsewhere in frame (otherwise our search will get confused), so have to make sure no combination of message bits can accidentally generate the sync pattern (can be tricky…)
  - Sync pattern can't be protected by ECC, so errors may cause us to lose a frame every now and then, a problem that will need to be addressed at some higher level of the communication protocol.

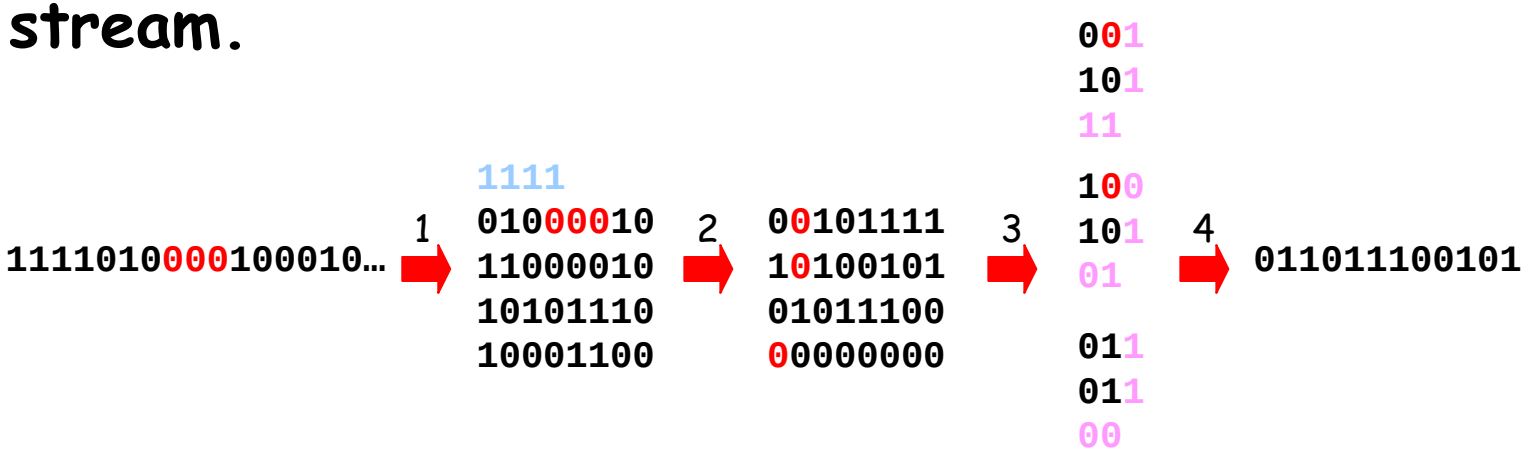
Summary: example channel coding steps

1. Break message stream into k-bit blocks.
2. Add redundant info in the form of (n-k) parity bits to form n-bit codeword. Goal: choose parity bits so we can correct single-bit errors, detect double-bit errors.
3. Interleave bits from a group of B codewords to protect against B-bit burst errors.
4. Add unique pattern of bits to start of each interleaved codeword block so receiver can tell how to extract blocks from received bitstream.
5. Send new (longer) bitstream to transmitter.



Summary: example error correction steps

1. Search through received bit stream for sync pattern, extract interleaved codeword block
2. De-interleave the bits to form B n-bit codewords
3. Check parity bits in each code word to see if an error has occurred. If there's a single-bit error, correct it.
4. Extract k message bits from each corrected codeword and concatenate to form message stream.



Summary

- To detect D-bit errors: Hamming distance > D
- To correct D-bit errors: Hamming distance > 2D
- (n,k,d) codes have code rate of k/n
- For our purposes, we want to correct single-bit errors *and* detect double bit errors, so d = 4
- Handle B-bit burst errors by interleaving B codewords
- Add sync pattern to interleaved codeword block so receiver can find start of block.
- Use checksum/CRC to detect uncorrected errors in message

SLIDES FOR FRIDAY

After reviewing post lab questions

In search of a better code

- Problem: information about a particular message unit (bit, byte, ..) is captured in just a few locations, ie, the message unit and some number of parity units. So a small but unfortunate set of errors might wipe out all the locations where that info resides, causing us to lose the original message unit.

- Potential Solution: figure out a way to spread the info in each message unit throughout all the codewords in a block. Require only some fraction good codewords to recover the original message.

Spreading the wealth…

- Idea: oversampled polynomials. Let

$$P(x) = m_0 + m_1 x + m_2 x^2 + \ldots + m_{k-1} x^{k-1}$$

where $m_0, m_1, \ldots, m_{k-1}$ are the k message units to be encoded. Transmit value of polynomial at n different predetermined points $v_0, v_1, \ldots, v_{n-1}$:

$$P(v_0), P(v_1), P(v_2), \ldots, P(v_{n-1})$$

Use any k of the received values to construct a linear system of k equations which can then be solved for k unknowns $m_0, m_1, \ldots, m_{k-1}$.

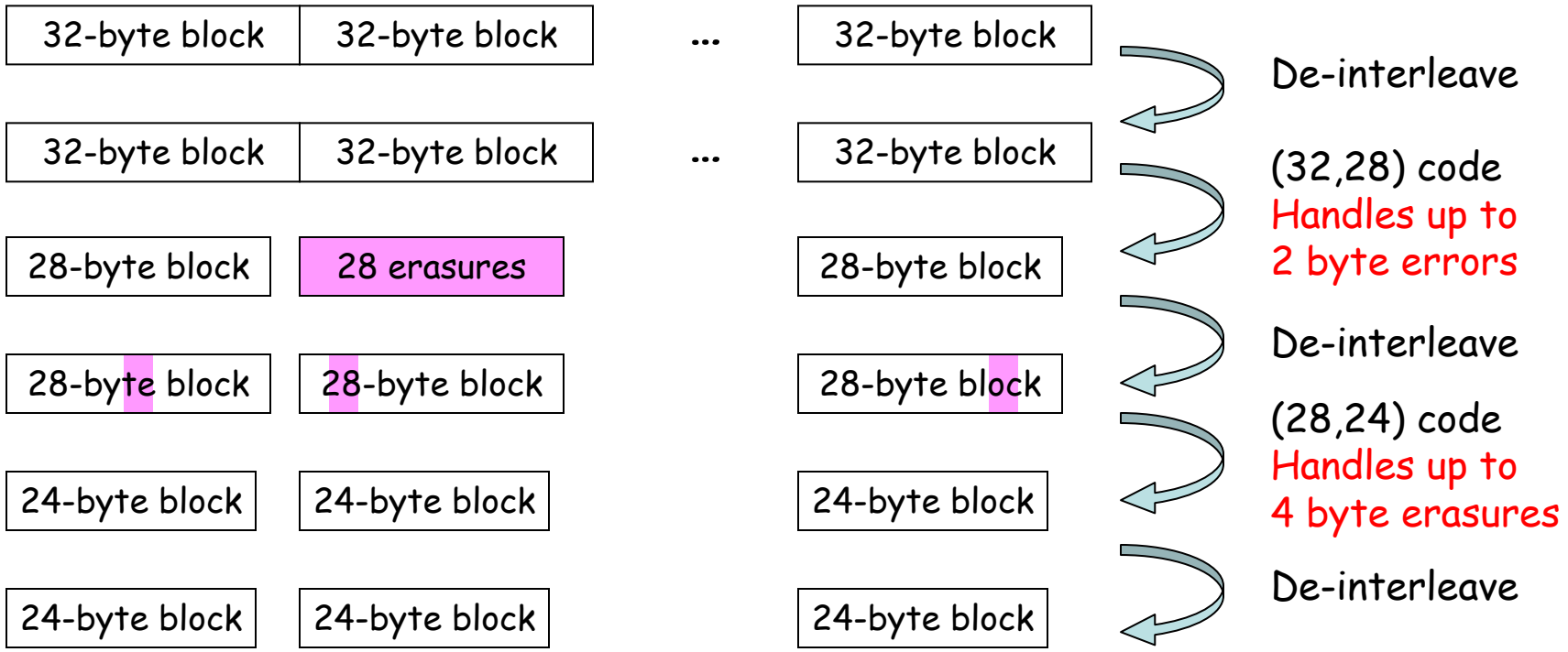
Each transmitted value contains info about all m_i.

Use for error correction

- If one of the $P(v_i)$ is received incorrectly, if it's used to solve for the m_i, we'll get the wrong result.
- So try all possible (n choose k) subsets of values and use each subset to solve for m_i. Choose solution set that gets the majority of votes.
  - No winner? Uncorrectable error… throw away block.
- If a particular received value is known to be erroneous (an "erasure"), don't use it all: (n,k) code can correct n-k erasures since we only need k equations to solve for the k unknowns.
- (n,k) code can correct up to (n-k)/2 errors since we need enough good values to ensure that the correct solution set gets a majority of the votes.

Example: CD error correction

- Reed-Solomon is an implementation of the over-sampled polynomial idea.
- On CD: two concatenated R-S codes



Result: correct up to 3500-bit error bursts (2.4mm on CD surface)