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(\sqrt{k})
  – Hamming: p grows as O(\log k)
• After xmit, correct errors
• Verify checksum, fails if undetected/uncorrectable error
• Deliver or discard message

How many parity bits to use?

• 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 ≤ 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 17 of Lecture #6

(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).
Error-Correcting Codes

- 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!
  - Number of parity bits grows linearly with size of message
- “Rectangular” codes with row/column parity
  - Easy to visualize how multiple parity bits can be used to triangulate location of 1-bit error
  - Number of parity bits grows as square root of message size
- Hamming single error correcting codes (SECC) are \((n,n-p,3)\)
  where \(n = 2^p - 1\) for \(p > 1\)
  - See Wikipedia article for details
  - Number of parity bits grows as \(\log_2\) of message size

Noise models

- Gaussian noise
  - Equal chance of noise at each sample
  - Gaussian PDF: low probability of large amplitude
  - Good for modeling total effect of many small, random noise sources
- Impulse noise
  - Infrequent bursts of high-amplitude noise, e.g., on a wireless channel
  - Some number of consecutive bits lost, bounded by some burst length \(B\)
  - Single-bit error correction seems like it’s useless for dealing with impulse noise… or is it???

Dealing with 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 code word are transmitted sequentially, so a \(B\)-bit burst produces multi-bit errors.

Solution: interleave bits from \(B\) different code words. Now a \(B\)-bit burst produces 1-bit errors in \(B\) different code words.

Interleaving

Interleaving message

Compute CRC

Partition

Apply ECC

Interleave

B-way interleaved block

Transmit

Receive

B-way interleaved block

Deinterleave

Correct errors

Check CRC

message

crc

k

k

k

k

k+p

k+p

k+p

k+p

message

crc

Deliver or discard
Framing

• The receiver needs to know
  – the beginning of the B-way interleaved block in order to do deinterleaving
  – the beginning of each ECC block in order to do error correction.
  – Since the interleaved block is made up of B ECC blocks, knowing where the interleaved block begins automatically supplies the necessary start info for the ECC blocks

• 8b10b encoding provides what we need! Here’s what gets transmitted
  – Prefix to help train clock recovery (alternating 0s/1s, …)
  – 8b10b sync symbol
  – Packet data: B ECC blocks recoded as 8b10b symbols
    (after 8b10b decoding and error correction we get [{#,data,chk}])
  – Suffix to ensure transmitter doesn’t cutoff prematurely, receiver has time to process last packet before starting search for beginning of next packet
  – On some channels: idle time (no transmission)

Our Recipe (so far)

• Transmit
  – Packetize: split message into fixed-size blocks, add sequence numbers, checksum
  – SECC: split [{#,data,chk}] into k-bit blocks, add parity bits to create n-bit code words with min Hamming distance of 3, B-way interleaving
  – 8b10b encoding: provide synchronization info to locate start of packet and sufficient transitions for clock recovery
  – Convert each bit into samples_per_bit voltage samples

• Receive
  – Perform clock recovery using transitions, derive bit stream from voltage samples
  – 8b10b decoding: locate sync, decode
  – SECC: deinterleave to spread out burst errors, perform error correction on n-bit blocks producing k-bit blocks
  – Packetize: verify checksum and discard faulty packets. Keep track of received sequence numbers, ask for retransmit of missing packets. Reassemble packets into original message.

Remaining agenda items

• With B ECC blocks per message, we can correct somewhere between 1 and B errors depending on where in the message they occur.
  – Can we make an ECC that corrects up to B errors without any constraints where errors occur?
  – Yes! Reed-Solomon codes, discussed next

• Framing is necessary, but the sync itself can’t be protected by an ECC scheme that requires framing.
  – This makes life hard for channels with higher BERs
  – Is there an error correction scheme that works on unframed bit streams?
  – Yes! Convolutional codes: encoding and the clever decoding scheme will be discussed next week.

In search of a better code

• Problem: information about a particular message unit (bit, byte, ..) is captured in just a few locations, i.e., 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 code words in a block. Require only some fraction good code words to recover the original message.
Thought experiment...

- Suppose you had two 8-bit values to communicate: A, B
- We’d like an encoding scheme where each transmitted value included information about both A and B
  - How about sending \( y = Ax + B \) for various values of \( x \)?
  - Standardize on a particular sequence for \( x \), known to both the transmitter and receiver. That way, we don’t have to actually send the \( x \)'s – the receiver will know what they are. For example, \( x = 1, 2, 3, 4, ... \)
  - How many values do you need to solve for A and B?
  - We’ll send extra to provide for recovery from errors...

Spreading the wealth...

- Generalize this 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 \( m_i \).
- Note that using integer arithmetic, the \( P(v) \) values are numerically greater than the \( m_i \) and so require more bits to represent than the \( m_i \). In general the encoded message would require a lot more bits to send than the original message!

Example

- Suppose you received four values from the transmitter \( y = 73, 249, 321, 393 \), corresponding to \( x = 1, 2, 3 \) and 4
  - 4 Eqns: \( A \cdot 1 + B = 73 \), \( A \cdot 2 + B = 249 \), \( A \cdot 3 + B = 321 \), \( A \cdot 4 + B = 393 \)
- We need two of these equations to solve for \( A \) and \( B \); there are six possible choices for which two to use
- Take each pair and solve for \( A \) and \( B \)
  - Majority rules: \( A = 72, B = 105 \)
    - The received value 73 had an error
    - If no errors: all six solutions for \( A \) and \( B \) would have matched

Solving for the \( m_i \)

- Solving \( k \) linearly independent equations for the \( k \) unknowns \( [i.e., \text{the } m_i] \):
  \[
  \begin{bmatrix}
  1 & v_0 & v_0^2 & \ldots & v_0^{k-1} \\
  1 & v_1 & v_1^2 & \ldots & v_1^{k-1} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  1 & v_{k-1} & v_{k-1}^2 & \ldots & v_{k-1}^{k-1}
  \end{bmatrix}
  \begin{bmatrix}
  m_0 \\
  m_1 \\
  \vdots \\
  m_{k-1}
  \end{bmatrix}
  =
  \begin{bmatrix}
  P(v_0) \\
  P(v_1) \\
  \vdots \\
  P(v_{k-1})
  \end{bmatrix}
  \]
- Solving a set of linear equations using Gaussian Elimination (multiplying rows, switching rows, adding multiples of rows to other rows) requires add, subtract, multiply and divide operations.
- These operations (in particular division) are only well defined over fields, e.g., rational numbers, real numbers, complex numbers -- not at all convenient to implement in hardware.
Finite Fields to the Rescue

- Reed’s & Solomon’s idea: do all the arithmetic using a finite field (also called a Galois field). If the \( m_i \) have \( B \) bits, then use a finite field with order \( 2^B \) so that there will be a field element corresponding to each possible value for \( m_i \).

- For example with \( B = 2 \), here are the tables for the various arithmetic operations for a finite field with 4 elements. Note that every operation yields an element in the field, i.e., the result is the same size as the operands.

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>*</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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.

- (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.
  - R-S (255,223) code can correct up to 16 symbol errors; good for error bursts: 16 consecutive symbols = 128 bits!

How many values to send?

- Note that in a Galois field of order \( 2^B \) there are at most \( 2^B \) unique values \( v \) we can use to generate the \( P(v) \)
  - if we send more than \( 2^B \) values, some of the equations we might use when solving for the \( m_i \) will not be linearly independent and we won’t have enough information to find a unique solution for the \( m_i \).
  - Sending \( P(0) \) isn’t very interesting (only involves \( m_0 \))

- Reed-Solomon codes use \( n = 2^B - 1 \) (n is the number of \( P(v) \) values we generate and send).
  - For many applications \( B = 8 \), so \( n = 255 \)
  - A popular R-S code is (255,223), i.e., a code block consisting of 223 8-bit data bytes + 32 check bytes

Erasures are special

- If a particular received value is known to be erroneous (an “erasure”), don’t use it all!
  - How to tell when received value is erroneous? Sometimes there’s channel information, e.g., carrier disappears.
  - See next slide for clever idea based on concatenated R-S codes

- (n,k) R-S code can correct n-k erasures since we only need k equations to solve for the k unknowns.

- Any combination of E errors and S erasures can be corrected so long as \( 2E + S \leq n - k \).
Example: CD error correction

- On a CD: two concatenated R-S codes

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