Today’s Lecture

- Homework 2 Help
  - Q&A
- Cleanup – Lecture 4
  - Frames
- Physical Layer
  - Error Correction
  - Chapter 2.1-2.3

Homework 2

- Debugging
  - printf is your friend
  - ssize_t is the same as an int
  - Looping

Thursday – Chat about Project 1
Encoding

- Signals propagate over a physical medium
  - modulate electromagnetic waves
  - e.g., vary voltage
- Encode binary data onto signals
  - e.g., 0 as low signal and 1 as high signal
  - known as Non-Return to zero (NRZ)

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problem: Consecutive 1s or 0s

- Low signal (0) may be interpreted as no signal
- Long strings of 0s or 1s lead to baseline wander
- Unable to recover clock

Alternative Encodings

- Non-return to Zero Inverted (NRZI)
  - Transition from current signal to encode a one
  - Stay at current signal to encode a zero
  - Solves the problem of consecutive ones
- Manchester
  - Transmit XOR of the NRZ encoded data and the clock
  - Only 50% efficient (bit rate = 1/2 baud rate)
Encodings (cont)

- **4B/5B**
  - every 4 bits of data encoded in a 5-bit code
  - 5-bit codes selected to have no more than one leading 0 and no more than two trailing 0s
  - thus, never get more than three consecutive 0s
  - resulting 5-bit codes are transmitted using NRZI
  - achieves 80% efficiency
### 4B/5B

- Never get more than 3 zeros

<table>
<thead>
<tr>
<th>Code</th>
<th>Valid codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>11110</td>
</tr>
<tr>
<td>0001</td>
<td>01001</td>
</tr>
<tr>
<td>0010</td>
<td>10100</td>
</tr>
<tr>
<td>0011</td>
<td>10101</td>
</tr>
<tr>
<td>0100</td>
<td>01010</td>
</tr>
<tr>
<td>0101</td>
<td>01011</td>
</tr>
<tr>
<td>0110</td>
<td>01100</td>
</tr>
<tr>
<td>0111</td>
<td>01101</td>
</tr>
<tr>
<td>1000</td>
<td>10010</td>
</tr>
<tr>
<td>1001</td>
<td>10011</td>
</tr>
<tr>
<td>1010</td>
<td>11000</td>
</tr>
<tr>
<td>1011</td>
<td>11001</td>
</tr>
<tr>
<td>1100</td>
<td>11010</td>
</tr>
<tr>
<td>1101</td>
<td>11011</td>
</tr>
<tr>
<td>1110</td>
<td>11110</td>
</tr>
</tbody>
</table>

### Framing

- Break sequence of bits into a frame
- Typically implemented by network adaptor

![Figure 9.7](image)

**Figure 9.7** Illustration of framing on a synchronous transmission system

### Approaches

- Sentinel-based

```
<table>
<thead>
<tr>
<th>8</th>
<th>8</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Header</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Body</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Flag</td>
<td>Address</td>
<td>Control</td>
<td>Protocol</td>
</tr>
</tbody>
</table>
```

-start frame with special pattern: 01111110
Approaches

• Counter-based
  – include payload length in header
  – e.g., DDCMP
  
  - Count | Header | Body | CRC

  – problem: count field corrupted
  – solution: catch when CRC fails

Approaches

• Bit-oriented: HDLC
  – uses 01111110 for beginning and end, also sent during idle times for synchronization
  – bit stuffing: when 5 consecutive 1s have been transmitted, sender inserts 0

Approaches

• Clock-based (SONET)
  – each frame is 125us long
  – e.g., SONET: Synchronous Optical Network
  – STS-n (STS-1 = 51.84 Mbps)
Error Detection

Outline
  Parity
  Checksum
  CRC

Error Detection

• Bit Error Rate (BER)
  – Bits will occasionally get “flipped”
  – Wired Networks
    • $10^{-10}$ to $10^{-14}$
  – Wireless Networks
    • $10^{-3}$ to $10^{-6}$

• Burstiness
  – Are bit errors spread uniformly?
  – Are they grouped together?

Both have a 20% loss rate, which is worse?
2-Dimensional Parity

<table>
<thead>
<tr>
<th>Parity bits</th>
<th>Data</th>
<th>Party byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010011</td>
<td>11010010</td>
<td>10111110</td>
</tr>
<tr>
<td>01010011</td>
<td>00011110</td>
<td>01100001</td>
</tr>
<tr>
<td>01010011</td>
<td>11101110</td>
<td></td>
</tr>
</tbody>
</table>

Force to even or odd number of ones

Force to odd or even as a summary

Internet Checksum Algorithm

- View message as a sequence of 16-bit integers; sum using 16-bit ones-complement arithmetic; take ones-complement of the result.

```c
u_short
dcksum(u_short *buf, int count)
{
    register u_long sum = 0;
    while (count--)
    {
        sum += *buf++;
        if (sum & 0xFFFF0000)
            /* carry occurred, so wrap around */
            sum &= 0xFFFF;
            sum++;
    }
    return ~(sum & 0xFFFF);
}
```

Really useful?

Cyclic Redundancy Check

- Add \( k \) bits of redundant data to an \( n \)-bit message
  
  \( \text{want } k << n \)
  
  \( \text{e.g., } k = 32 \text{ and } n = 12,000 \) (1500 bytes)

- Represent \( n \)-bit message as \( n-1 \) degree polynomial
  
  \( \text{e.g., MSG=10011010 as } M(x) = x^7 + x^4 + x^3 + x^1 \)

- Let \( k \) be the degree of some divisor polynomial
  
  \( \text{e.g., } C(x) = x^5 + x^4 + 1 \)
CRC (cont)

- Transmit polynomial $P(x)$ that is evenly divisible by $C(x)$
  - shift left $k$ bits, i.e., $M(x)x^k$
  - subtract remainder of $M(x)x^k / C(x)$ from $M(x)x^k$
- Receiver polynomial $P(x) + E(x)$
  - $E(x)$ is zero implies no errors
- Divide $(P(x) + E(x))$ by $C(x)$; remainder zero if:
  - $E(x)$ was zero (no error), or
  - $E(x)$ is exactly divisible by $C(x)$

Selecting $C(x)$

- All single-bit errors, as long as the $x^k$ and $x^0$ terms have non-zero coefficients.
- All double-bit errors, as long as $C(x)$ contains a factor with at least three terms
- Any odd number of errors, as long as $C(x)$ contains the factor $(x + 1)$
- Any ‘burst’ error (i.e., sequence of consecutive error bits) for which the length of the burst is less than $k$ bits.
- Most burst errors of larger than $k$ bits can also be detected
- See Table 2.5 on page 96 for common $C(x)$

Hardware Implementation

Most hardware using a shift register
Very easy to make it go fast
Validate at “line speed”
Acknowledgements & Timeouts

How do we manage if we can’t recover?

Stop-and-Wait

- Problem: keeping the pipe full
- Example
  - 1.5Mbps link x 45ms RTT = 67.5Kb (8KB)
  - 1KB frames implies 1/8th link utilization

Stop-and-Wait
Sliding Window

- Allow multiple outstanding (un-ACKed) frames
- Upper bound on un-ACKed frames, called window

SW: Sender

- Assign sequence number to each frame (SeqNum)
- Maintain three state variables:
  - send window size (SWS)
  - last acknowledgment received (LAR)
  - last frame sent (LFS)
- Maintain invariant: LFS - LAR <= SWS
- Advance LAR when ACK arrives
- Buffer up to SWS frames

SW: Receiver

- Maintain three state variables
  - receive window size (RWS)
  - largest acceptable frame (LAF)
  - last frame received (LFR)
- Maintain invariant: LAF - LFR <= RWS
- Frame SeqNum arrives:
  - if LFR < SeqNum <= LFA accept
  - if SeqNum < = LFR or SeqNum > LFA discarded
- Send cumulative ACKs
Sequence Number Space

- **SeqNum** field is finite; sequence numbers wrap around
- Sequence number space must be larger than number of outstanding frames

**SWS \(\leq \) MaxSeqNum-1** is not sufficient
- suppose 3-bit **SeqNum** field (0..7)
  - SWS=MaxSeqNum=7
  - sender transmit frames 0..6
  - arrive successfully, but ACKs lost
  - sender retransmits 0.6
  - receiver expecting 7, 0..5, but receives second incarnation of 0..5

**SWS \(<\) (MaxSeqNum+1)/2** is correct rule

Intuitively, **SeqNum** “slides” between two halves of sequence number space

---

Quiz

- Open book
- Open note
- 10 minutes

- Hints
  - Pictures are good if appropriate
  - Write at least something