CS 3700
Networks and Distributed Systems

Lecture 3: Physical and Data Link
**Physical Layer**

- **Function:**
  - Get bits across a physical medium

- **Key challenge:**
  - How to represent bits in analog
  - Ideally, want high-bit rate
  - But, must avoid desynchronization

---

**Diagram:**

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical
Key challenge

- Digital computers
  - 0s and 1s
- Analog world
  - Amplitudes and frequencies
Assumptions

- We have two discrete signals, high and low, to encode 1 and 0
Assumptions

- We have two discrete signals, high and low, to encode 1 and 0
- Transmission is synchronous, i.e. there is a clock that controls signal sampling
Assumptions

- We have two discrete signals, high and low, to encode 1 and 0
- Transmission is synchronous, i.e. there is a clock that controls signal sampling
Assumptions

- We have two discrete signals, high and low, to encode 1 and 0
- Transmission is **synchronous**, i.e. there is a clock that controls signal sampling
- Amplitude and duration of signal must be significant
Non-Return to Zero (NRZ)

- 1 $\rightarrow$ high signal, 0 $\rightarrow$ low signal

```plaintext
0 0 1 0 1 0 1 1 0 0
```

NRZ

Clock
Non-Return to Zero (NRZ)

- 1 → high signal, 0 → low signal

Problem: long strings of 0 or 1 cause desynchronization
- How to distinguish lots of 0s from no signal?
- How to recover the clock during lots of 1s?
**Desynchronization**

- Problem: how to recover the clock during sequences of 0’s or 1’s?

![NRZ waveform](image-url)
Problem: how to recover the clock during sequences of 0’s or 1’s?

Transitions signify clock ticks
Desynchronization

- Problem: how to recover the clock during sequences of 0’s or 1’s?

Transitions signify clock ticks
Desynchronization

- Problem: how to recover the clock during sequences of 0’s or 1’s?

Transitions signify clock ticks
Problem: how to recover the clock during sequences of 0’s or 1’s?

Transitions signify clock ticks

Receiver misses a 1 due to skew
Non-Return to Zero Inverted (NRZI)

- 1 $\rightarrow$ make transition, 0 $\rightarrow$ remain the same

NRZI

Clock
Non-Return to Zero Inverted (NRZI)

- 1 → make transition, 0 → remain the same
- Solves the problem for sequences of 1s, but not 0s
4-bit / 5-bit (100 Mbps Ethernet)

<table>
<thead>
<tr>
<th>4-bit</th>
<th>5-bit</th>
<th>4-bit</th>
<th>5-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>11110</td>
<td>1000</td>
<td>10010</td>
</tr>
<tr>
<td>0001</td>
<td>01001</td>
<td>1001</td>
<td>10011</td>
</tr>
<tr>
<td>0010</td>
<td>10100</td>
<td>1010</td>
<td>10110</td>
</tr>
<tr>
<td>0011</td>
<td>10101</td>
<td>1011</td>
<td>10111</td>
</tr>
<tr>
<td>0100</td>
<td>01010</td>
<td>1100</td>
<td>11010</td>
</tr>
<tr>
<td>0101</td>
<td>01011</td>
<td>1101</td>
<td>11011</td>
</tr>
<tr>
<td>0110</td>
<td>01110</td>
<td>1110</td>
<td>11100</td>
</tr>
<tr>
<td>0111</td>
<td>01111</td>
<td>1111</td>
<td>11101</td>
</tr>
</tbody>
</table>
### 4-bit/5-bit (100 Mbps Ethernet)

- **Observation:** NRZI works as long as no sequences of 0
- **Idea:** encode all 4-bit sequences as 5-bit sequences with no more than one leading 0 and two trailing 0

<table>
<thead>
<tr>
<th>4-bit</th>
<th>5-bit</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>01110</td>
</tr>
<tr>
<td>0111</td>
<td>01111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4-bit</th>
<th>5-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10010</td>
</tr>
<tr>
<td>1001</td>
<td>10011</td>
</tr>
<tr>
<td>1010</td>
<td>10110</td>
</tr>
<tr>
<td>1011</td>
<td>10111</td>
</tr>
<tr>
<td>1100</td>
<td>11010</td>
</tr>
<tr>
<td>1101</td>
<td>11011</td>
</tr>
<tr>
<td>1110</td>
<td>11100</td>
</tr>
<tr>
<td>1111</td>
<td>11101</td>
</tr>
</tbody>
</table>

- **Tradeoff:** efficiency drops to 80%
4-bit/5-bit (100 Mbps Ethernet)

- Observation: NRZI works as long as no sequences of 0
- Idea: encode all 4-bit sequences as 5-bit sequences with no more than one leading 0 and two trailing 0

<table>
<thead>
<tr>
<th>4-bit</th>
<th>5-bit</th>
<th>4-bit</th>
<th>5-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>11110</td>
<td>1000</td>
<td>10010</td>
</tr>
<tr>
<td>0001</td>
<td>01001</td>
<td>1001</td>
<td>10011</td>
</tr>
<tr>
<td>0010</td>
<td>10100</td>
<td>1010</td>
<td>10110</td>
</tr>
<tr>
<td>0011</td>
<td>10101</td>
<td>1011</td>
<td>10111</td>
</tr>
<tr>
<td>0100</td>
<td>01010</td>
<td>1100</td>
<td>11010</td>
</tr>
<tr>
<td>0101</td>
<td>01011</td>
<td>1101</td>
<td>11011</td>
</tr>
<tr>
<td>0110</td>
<td>01110</td>
<td>1110</td>
<td>11100</td>
</tr>
<tr>
<td>0111</td>
<td>01111</td>
<td>1111</td>
<td>11110</td>
</tr>
</tbody>
</table>

- Tradeoff: efficiency drops to 80%
Manchester

- 1 → high-to-low, 0 → low-to-high
Manchester

- 1 $\rightarrow$ high-to-low, 0 $\rightarrow$ low-to-high

- **Good:** Solves clock skew (every bit is a transition)
- **Bad:** Halves throughput (two clock cycles per bit)
Physical layer is the lowest, so...
- We tend not to worry about where to place functionality
- There aren’t other layers that could interfere
- We tend to care about it only when things go wrong

Physical layer characteristics are still fundamentally important to building reliable Internet systems
- Insulated media vs wireless
- Packet vs. circuit switched media
Data Link Layer

- **Function:**
  - Send blocks of data (frames) between physical devices
  - Regulate access to the physical media

- **Key challenge:**
  - How to delineate frames?
  - How to detect errors?
  - How to perform media access control (MAC)?
  - How to recover from and avoid collisions?
Outline

- Framing
- Error Checking and Reliability
Framing

- Physical layer determines how bits are encoded
- Next step, how to encode blocks of data
Framing

- Physical layer determines how bits are encoded
- Next step, how to encode blocks of data
  - Packet switched networks
  - Each packet includes routing information
  - Data boundaries must be known so headers can be read
Framing

- Physical layer determines how bits are encoded
- Next step, how to encode blocks of data
  - Packet switched networks
  - Each packet includes routing information
  - Data boundaries must be known so headers can be read
- Types of framing
  - Byte oriented protocols
  - Bit oriented protocols
  - Clock based protocols
Add **START** and **END** sentinels to the data
Byte Oriented: Sentinel Approach

- Add **START** and **END** sentinels to the data
Byte Oriented: Sentinel Approach

- Add **START** and **END** sentinels to the data
- Problem: what if **END** appears in the data?
- Add **START** and **END** sentinels to the data
- Problem: what if **END** appears in the data?
  - Add a special **DLE** (Data Link Escape) character before **END**
Add **START** and **END** sentinels to the data

- Problem: what if **END** appears in the data?
  - Add a special **DLE** (Data Link Escape) character before **END**
  - What if **DLE** appears in the data? Add **DLE** before it.
Add **START** and **END** sentinels to the data

Problem: what if **END** appears in the data?
- Add a special **DLE** (Data Link Escape) character before **END**
- What if **DLE** appears in the data? Add **DLE** before it.

Similar to escape sequences in C
- `printf("You must \"escape\" quotes in strings");`
- `printf("You must \\escape\\ forward slashes as well");`

Used by Point-to-Point protocol, e.g. modem, DSL, cellular
Byte Oriented: Byte Counting

132

Data
Byte Oriented: Byte Counting

- **Sender:** insert length of the data in bytes at the beginning of each frame
- **Receiver:** extract the length and read that many bytes

![Data](image)
Bit Oriented: Bit Stuffing

Data
Bit Oriented: Bit Stuffing

- Add sentinels to the start and end of data
- Both sentinels are the same
- Example: 01111110 in High-level Data Link Protocol (HDLC)
Bit Oriented: Bit Stuffing

- Add sentinels to the start and end of data
  - Both sentinels are the same
  - Example: 01111110 in High-level Data Link Protocol (HDLC)
- Sender: insert a 0 after each 11111 in data
  - Known as “bit stuffing”
- Receiver: after seeing 11111 in the data...
  - 111110 \(\rightarrow\) remove the 0 (it was stuffed)
  - 111111 \(\rightarrow\) look at one more bit
    - 1111110 \(\rightarrow\) end of frame
    - 1111111 \(\rightarrow\) error! Discard the frame
- Disadvantage: 20% overhead at worst
Clock-based Framing: SONET

- **Synchronous Optical Network**
  - Transmission over very fast optical links
  - STS-\(n\), e.g., STS-1: 51.84 Mbps, STS-768: 36.7 Gbps
- STS-1 frames based on fixed sized frames
  - \(9 \times 90 = 810\) bytes

Special start pattern

![Diagram showing 90 Columns and 9 Rows with Payload and Overhead sections.]
Clock-based Framing: SONET

- **Synchronous Optical Network**
  - Transmission over very fast optical links
  - STS-\(n\), e.g. STS-1: 51.84 Mbps, STS-768: 36.7 Gbps
- STS-1 frames based on fixed sized frames
  - \(9 \times 90 = 810\) bytes
- Physical layer details
  - Bits are encoded using NRZ
  - Payload is XORed with a special 127-bit pattern to avoid long sequences of 0 and 1
Framing

Error Checking and Reliability
The physical world is inherently noisy
- Interference from electrical cables
- Cross-talk from radio transmissions, microwave ovens
- Solar storms

How to detect bit-errors in transmissions?
How to recover from errors?
Naïve Error Detection

- Idea: send two copies of each frame
  - if (memcmp(frame1, frame2) != 0) { OH NOES, AN ERROR! }
- Why is this a bad idea?
Naïve Error Detection

- Idea: send two copies of each frame
  - if (memcmp(frame1, frame2) != 0) { OH NOES, AN ERROR! }

- Why is this a bad idea?
  - Extremely high overhead
  - Poor protection against errors
    - Twice the data means twice the chance for bit errors
Parity Bits

- Idea: add extra bits to keep the number of 1s even
- Example: 7-bit ASCII characters + 1 parity bit

0101001 1101001 1011110 0001110 0110100
Parity Bits

- Idea: add extra bits to keep the number of 1s even
- Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 1011110 0001110 0110100
Parity Bits

- Idea: add extra bits to keep the number of 1s even
- Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1
Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1

- Detects 1-bit errors and some 2-bit errors
Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

```
0101001 1 1101001 0 1011110 1 0001110 1 0110100 1
```

- Detects 1-bit errors and some 2-bit errors
Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

0101001 1 1101001 0 1011110 1 0001110 1 0110100 1

- Detects 1-bit errors and some 2-bit errors
Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

  0101001 1 1101001 0 1011110 1 0001110 1 0110100 1

- Detects 1-bit errors and some 2-bit errors
Parity Bits

- Idea: add extra bits to keep the number of 1s even
  - Example: 7-bit ASCII characters + 1 parity bit

```
0101001 1 1101001 0 1011110 1 0001110 1 0110100 1
```

- Detects 1-bit errors and some 2-bit errors
- Not reliable against bursty errors
## Two Dimensional Parity

<table>
<thead>
<tr>
<th>0101001</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101001</td>
</tr>
<tr>
<td>1011110</td>
</tr>
<tr>
<td>0001110</td>
</tr>
<tr>
<td>0110100</td>
</tr>
<tr>
<td>1011111</td>
</tr>
</tbody>
</table>
Two Dimensional Parity

<table>
<thead>
<tr>
<th></th>
<th>Parity bit for each row</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101001</td>
<td>1</td>
</tr>
<tr>
<td>1101001</td>
<td>0</td>
</tr>
<tr>
<td>1011110</td>
<td>1</td>
</tr>
<tr>
<td>0001110</td>
<td>1</td>
</tr>
<tr>
<td>0110100</td>
<td>1</td>
</tr>
<tr>
<td>1011111</td>
<td>0</td>
</tr>
</tbody>
</table>
### Two Dimensional Parity

#### Parity bit for each row

| 0101001 | 1 |
| 1101001 | 0 |
| 1011110 | 1 |
| 0001110 | 1 |
| 0110100 | 1 |
| 1011111 | 0 |

#### Parity bit for each column

**1111011**
## Two Dimensional Parity

<table>
<thead>
<tr>
<th>Bit String</th>
<th>Parity Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101001</td>
<td>1</td>
</tr>
<tr>
<td>1101001</td>
<td>0</td>
</tr>
<tr>
<td>1011110</td>
<td>1</td>
</tr>
<tr>
<td>0001110</td>
<td>1</td>
</tr>
<tr>
<td>0110100</td>
<td>1</td>
</tr>
<tr>
<td>1011111</td>
<td>0</td>
</tr>
</tbody>
</table>

Parity bit for each row:

Parity bit for each column:

Parity bit for the parity byte:

1111011 0
Two Dimensional Parity

- Can detect all 1-, 2-, and 3-bit errors, some 4-bit errors

<table>
<thead>
<tr>
<th></th>
<th>0101001</th>
<th>1101001</th>
<th>1011110</th>
<th>0001110</th>
<th>0110100</th>
<th>1011111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity bit for each row</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Parity bit for each column</td>
<td>1111011</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parity bit for the parity byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
### Two Dimensional Parity

- Can detect all 1-, 2-, and 3-bit errors, some 4-bit errors
- 14% overhead

<table>
<thead>
<tr>
<th></th>
<th>0101001</th>
<th>1101001</th>
<th>1011110</th>
<th>0001110</th>
<th>0110100</th>
<th>1011111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity bit for each row</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Parity bit for each column</td>
<td>1111011</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parity bit for the parity byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
## Two Dimensional Parity Examples

<table>
<thead>
<tr>
<th>Binary Sequence</th>
<th>Parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101001</td>
<td>1</td>
</tr>
<tr>
<td>1101001</td>
<td>0</td>
</tr>
<tr>
<td>1011110</td>
<td>1</td>
</tr>
<tr>
<td>0011110</td>
<td>1</td>
</tr>
<tr>
<td>0110100</td>
<td>1</td>
</tr>
<tr>
<td>1011111</td>
<td>0</td>
</tr>
<tr>
<td>1111011</td>
<td>0</td>
</tr>
</tbody>
</table>
Two Dimensional Parity Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101001</td>
<td>1</td>
</tr>
<tr>
<td>1101001</td>
<td>0</td>
</tr>
<tr>
<td>1011110</td>
<td>1</td>
</tr>
<tr>
<td>0011110</td>
<td>1</td>
</tr>
<tr>
<td>0110100</td>
<td>1</td>
</tr>
<tr>
<td>1011111</td>
<td>0</td>
</tr>
<tr>
<td>1111011</td>
<td>0</td>
</tr>
</tbody>
</table>

Odd number of 1s

Odd Number of 1s
Two Dimensional Parity Examples

Odd Number of 1s

Odd number of 1s
### Two Dimensional Parity Examples

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0111001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10111001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11011011</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01110101</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01011011</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00110111</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01101001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11011111</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Odd number of 1s**

**Odd number of 1s**
Two Dimensional Parity Examples

0101001 1
1110000 0
1011110 1
0011111 1
0110100 1
1011111 0
1111011 0
Checksums

- **Idea:**
  - Add up the bytes in the data
  - Include the sum in the frame

- Use ones-complement arithmetic
- Lower overhead than parity: 16 bits per frame
- But, not resilient to errors
  - Why?
- Used in UDP, TCP, and IP
Checksums

- **Idea:**
  - Add up the bytes in the data
  - Include the sum in the frame

- Use ones-complement arithmetic
- Lower overhead than parity: 16 bits per frame
- But, not resilient to errors
  - Why? \[ 0101001 + 1101001 = 10010010 \]
- Used in UDP, TCP, and IP
Check sums

- Idea:
  - Add up the bytes in the data
  - Include the sum in the frame

- Use ones-complement arithmetic
- Lower overhead than parity: 16 bits per frame
- But, not resilient to errors
  - Why?
    - $1\overline{1}01001 + 0\overline{1}01001 = 10010010$
- Used in UDP, TCP, and IP
Cyclic Redundancy Check (CRC)

- Uses field theory to compute a semi-unique value for a given message
- Much better performance than previous approaches
  - Fixed size overhead per frame (usually 32-bits)
  - Quick to implement in hardware
  - Only 1 in $2^{32}$ chance of missing an error with 32-bit CRC
- Details are in the book/on Wikipedia
What About Reliability?

- How does a sender know that a frame was received?
  - What if it has errors?
  - What if it never arrives at all?
What About Reliability?

- How does a sender know that a frame was received?
  - What if it has errors?
  - What if it never arrives at all?
What About Reliability?

- How does a sender know that a frame was received?
  - What if it has errors?
  - What if it never arrives at all?
What About Reliability?

- How does a sender know that a frame was received?
  - What if it has errors?
  - What if it never arrives at all?
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
- Problems?

Sender

Receiver

Frame

ACK

Frame

Timeout
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
- Problems?
  - Utilization
  - Can only have one frame in flight at any time
Stop and Wait

- Simplest form of reliability
- Example: Bluetooth
- Problems?
  - Utilization
  - Can only have one frame in flight at any time
- 10Gbps link and 10ms delay
  - Need 100 Mbit to fill the pipe
  - Assume packets are 1500B
  - $1500B \times 8 \text{bit} / (2 \times 10 \text{ms}) = 600 \text{Kbps}$
  - Utilization is 0.006%
Sliding Window

- Allow multiple outstanding, un-ACKed frames
- Number of un-ACKed frames is called the **window**
Sliding Window

- Allow multiple outstanding, un-ACKed frames
- Number of un-ACKed frames is called the window

- Made famous by TCP
  - We’ll look at this in more detail later
Should We Error Check in the Data Link?

- Recall the End-to-End Argument
- Cons:
  - Error free transmission cannot be guaranteed
  - Not all applications want this functionality
  - Error checking adds CPU and packet size overhead
  - Error recovery requires buffering
Should We Error Check in the Data Link?

Recall the End-to-End Argument

Cons:
- Error free transmission cannot be guaranteed
- Not all applications want this functionality
- Error checking adds CPU and packet size overhead
- Error recovery requires buffering

Pros:
- Potentially better performance than app-level error checking
Should We Error Check in the Data Link?

- Recall the End-to-End Argument
- Cons:
  - Error free transmission cannot be guaranteed
  - Not all applications want this functionality
  - Error checking adds CPU and packet size overhead
  - Error recovery requires buffering
- Pros:
  - Potentially better performance than app-level error checking
- Data link error checking in practice
  - Most useful over lossy links
  - Wifi, cellular, satellite