Last time: Physical link layer

- **Tasks**
  - Encode binary data from source node into signals that physical links carry
  - Signal is decoded back into binary data at receiving node
  - Work performed by network adapter at sender and receiver

- **Synchronous encoding algorithms**
  - NRZ, NRZI, Manchester, 4B/5B
Today: Data-link layer

- Framing (2.3)
- Error detection (2.4)
Recall: (Data) Link Layer

- **Framing**
  - Break stream of bits up into discrete chunks

- **Error handling**
  - Detect and/or correct errors in received frames

- **Media access**
  - Arbitrate which nodes can send frames at any point in time
  - Not always necessary; e.g. point-to-point duplex links

- **Multiplexing**
  - Determine appropriate destination for a given frame
  - Also not always required; again, point-to-point
Framing

- Break down a stream of bits into smaller, digestible chunks called frames

- Allows the physical media to be shared
  - Multiple senders and/or receivers can time multiplex the link
  - Each frame can be separately addressed

- Provides manageable unit for error handling
  - Easy to determine whether something went wrong
  - And perhaps even to fix it if desired
What’s a Frame?

- Wraps payload up with some additional information
  - Header usually contains addressing information
  - Maybe includes a trailer (w/checksum—to be explained)
- Basic unit of reception
  - Link either delivers entire frame payload, or none of it
  - Typically some maximum transmission unit (MTU)
- Some link layers require absence of frames as well
  - I.e., minimum gaps between frames
Identifying Frames

- First task is to delineate frames
  - Receiver needs to know when a frame starts and ends
  - Otherwise, errors from misinterpretation of data stream

- Several different alternatives
  - Fixed length (bits) frames
  - Explicitly delimited frames
    - Length-based framing
    - Sentinel-based framing
  - Fixed duration (seconds) frames
Fixed-Length Frames

- Easy to manage for receiver
  - Well understood buffering requirements

- Introduces inefficiencies for variable length payloads
  - May waste space (padding) for small payloads
  - Larger payloads need to be fragmented across many frames

- Requires explicit design tradeoff
  - ATM uses 53-byte frames (cells)
  - Aside: why 53 bytes?
Length-Based Framing

- To avoid overhead, we’d like variable length frames
  - Each frame declares how long it is
  - E.g. DECNet DDCMP

- Issues?
  - What if you decode it wrong?
    » Remember, need to decode while receiving
  - Still need to identify the frame beginning correctly…
Sentinel-based Framing

- Idea: mark start/end of frame with special "marker"
  - Byte pattern, bit pattern, signal pattern

- But… must make sure marker doesn’t appear in data

- Two solutions
  - Special non-data physical-layer symbol (e.g., 00000 in 4B/5B)
    - Impact on efficiency (can’t use symbol for data) and utility of code (now can have long strings of 000’s sometimes)
  - Stuffing
    - Dynamically remove market bit patterns from data stream
    - Receiver “unstuff” data stream to reconstruct original data
Bit-level Stuffing

- Avoid sentinel bit pattern in payload data
  - Commonly, sentinel is bit pattern $01111110$ (0x7E)
  - Invented for SDLC/HDLC, now standard pattern

- Sender: any time five ones appear in outgoing data, insert a zero, resulting in 01111101

- Receiver: any time five ones appear, removes next zero
  - If there is no zero, there will either be six ones (sentinel) or
    - It declares an error condition!
  - Note bit pattern that cannot appear is $01111111$ (0x7F)

- What’s bad case?
Byte Stuffing

- Same as bit stuffing, except at byte (character) level
  - Generally have two different flags, STX and ETX
  - Found in PPP, DDCMP, BISYNC, etc.

- Need to stuff if either appears in the payload
  - Prefix with another special character, DLE (data-link escape)
  - New problem: what if DLE appears?

- Stuff DLE with DLE!
  - Could be as bad as 50% efficient to send all DLEs
Consistent-Overhead BS

- Control expansion of payload size due to stuffing
  - Important for low-bandwidth links or fixed-sized buffers
- Idea is to use **0x00** as a sentinel, and replace all zeros in data stream with *distance* to next 0x00.
  - Break frame up into runs without zeros, encode by prepending each run with length (including length byte)
  - Pretend frame ends in 0x00. Max run is 254; if no zeros prepend with 255 (0xFF)

```
45 00 00 2C 4C 79 00 00 40 06 4F 37
```
```
02 45 01 04 2C 4C 79 01 05 40 06 4F 37 00
```
Consistent-Overhead Byte Stuffing (COBS)

- Sentinel based framing
- Run length encoding applied to byte stuffing
  - Add implied 0 to end of frame
  - Each 0 is replaced with \((\text{number of bytes to next 0}) + 1\)
  - What if no 0 within 255 bytes? – 255 value indicates 254 bytes followed by no zero
  - Worst case – no 0’s in packet – 1/254 overhead
- Appropriate for very low-bandwidth links

<table>
<thead>
<tr>
<th>Code</th>
<th>Followed by</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>(not applicable)</td>
<td>(not allowed)</td>
</tr>
<tr>
<td>0x01</td>
<td>No data bytes</td>
<td>A single zero byte</td>
</tr>
<tr>
<td>(n)</td>
<td>((n-1)) data bytes</td>
<td>Data followed by 0</td>
</tr>
<tr>
<td>0xFF</td>
<td>254 data bytes</td>
<td>Data, no following 0</td>
</tr>
</tbody>
</table>
Clock-Based Framing

- So far, we’ve based framing on what’s on the wire
  - Any bit errors may throw off our framing
  - What happens with missed flag? Spurious flag?

- An alternative is to base framing on external clock
  - Kind of like Phy-layer signaling: sample at specific intervals
  - This is what SONET does, among others

- Significant engineering tradeoffs
  - No extra bits needed in the data stream itself, but…
  - Need tight clock synchronization between sender and receiver
SONET

- Synchronous Optical NETwork
  - Engineering goal to reduce delay and buffering
- All frames take same amount of time
  - Independent of bit rate!
- Each frame starts with signal bits
  - Can synch clock just like PLL—look for periodic signal bits
  - No need to stuff; signal pattern is unlikely, so won’t be periodic in data
- Keep sync within frames with transitions
  - Encoded using NRZ, but
  - Data is XORed with special 127-bit pattern
  - Creates lots of transitions, makes signal pattern unlikely
SONET Frame

- Every STS frame is 125 us long
- Supports multiple bit rates in same network
- STS-1 is base (slowest) speed: 51.84 Mbps
  - Frame contains 9 rows of 90 bytes each (810 bytes)
  - First 3 bytes of each row are header
    » 2-byte sync pattern, one byte for “flags”
Multiplexed SONET Links

- SONET actually defines networking functionality
  - Conflates layers; we’ll talk more in future lectures
  - Thinks about how to move frames between links
- Higher-speed links are multiples of STS-1 frames
  - E.g., STS-3 is three times as fast as STS-1
- Frames are byte-wise interleaved
  - Ensures pace of embedded STS-1 frames remains same
Synchronization…

Not too difficult to synchronize clocks such that first byte of all incoming flows arrives just before sending first 3 bytes of outgoing flow.
Synchronization…

... but now try to synchronize this network’s clocks
Framing: When Things Go Wrong

- May misinterpret frame boundaries
  - Length corrupted
  - Sentinel corrupted
  - Clock drift confuses frame boundaries

- Data in frame may be corrupted
  - Bit errors from noise, hardware failures, software errors

- In general, need to make sure we don’t accept bad data
  - Error detection (and perhaps correction)
Error Handling

- Error handling through redundancy
  - Adding extra bits to the frame to check for errors

- Hamming Distance
  - When we can detect
  - When we can correct

- Simple schemes: parity, voting, 2d-parity
- Checksum
- Cyclic Remainder Check (CRC)
Error Detection

- Implemented at many layers (link-layer today)
  - D = Data, EDC = Error Detection Code (redundancy)
Basic Idea

- The problem is data itself is not self-verifying
  - Every string of bits is potentially legitimate
  - Hence, any errors/changes in a set of bits are equally legit

- The solution is to reduce the set of potential bitstrings
  - Not every string of bits is allowable
  - Receipt of a disallowed string of bits means the original bits were garbled in transit

- Key question: which bitstrings are allowed?
Codewords

- Let’s start simple, and consider fixed-length bitstrings
  - Reduce our discussion to $n$-bit substrings
  - E.g., 7-bits at a time, or 4 bits at a time (4B/5B)
  - Or even a frame at a time
- We call an allowable sequence of $n$ bits a codeword
  - Not all strings of $n$ bits are codewords!
  - The remaining $n$-bit strings are “space” between codewords
- We’re going to encode data in terms of codewords (just like 4B/5B)
  - Non-codewords indicate an error (just like 4B/5B)
- How many codewords with how much space between them?
Hamming Distance

- Distance between legal codewords
  - Measured in terms of number of bit flips

- Efficient codes are of uniform Hamming Distance
  - All codewords are equidistant from their neighbors
2d+1 Hamming Distance

- Can **detect** up to $2d$ bit flips
  - The next codeword is always $2d+1$ bit flips away
  - Any fewer is guaranteed to land in the middle
- Can **correct** up to $d$ bit flips
  - We just move to the closest codeword
  - Unfortunately, no way to tell how many bit flips
    » E.g., 1, or $(2d+1)-1$?

![Diagram showing Hamming Distance]

27
Simple Embedding: Parity

- Add extra bit to ensure odd(even) number of ones
  - Can detect any single bit flip (hamming distance 2)
  - Code is 66% efficient (need three bits to encode two)

  » Note: Even parity is simply XOR
Simple Correction: Voting

- Simply send each bit \( n \) times (3 in this example)
  - Majority voting
  - Can detect any 2 bit flips and correct any 1 flip (\( d=1 \))
- Straightforward duplication is extremely inefficient
  - We can be much smarter about this
Two-Dimensional Parity

- Start with normal parity
  - \( n \) data bits, 1 one parity bit
- Do the same across rows
  - \( m \) data bytes, 1 parity byte
- Can detect up to 3 bit errors
  - Even most 4-bit errors
- Can correct any 1 bit error
  - Why?
Per-Frame Error Detection Codes

- Want to add an error detection code per frame
  - Frame is unit of transmission; all or nothing.
  - Computed over the entire frame—including header! Why?
- Receiver *recomputes* EDC over frame and checks against received EDC value
  - If frame fails check, throw it away
- We *could* use error-correcting codes
  - But they are less efficient, and we *expect errors to be rare*
Checksums

- Simply sum up all of the data in the frame
  - Transmit that sum as the EDC

- Extremely lightweight
  - Easy to compute fast in hardware
  - Fragile: Hamming Distance of 2

- Also easy to modify if frame is modified in flight
  - Happens a lot to packets on the Internet

- IP packets include a 1’s compliment checksum
IP Checksum Example

- 1’s compliment of sum of *words* (not bytes)
  - Final 1’s compliment means all-zero frame is not valid

```c
u_short cksum(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);
}
```
Checksum in Hardware

- Compute checksum in Modulo-2 Arithmetic
  - Addition/subtraction is simply XOR operation
  - Equivalent to vertical parity computation

- Need only a word-length shift register and XOR gate
  - Assuming data arrives serially
  - All registers are initially 0
Checksum Example

01010011110100101011110100011101011010011011111011110110
Checksum Example

0101001111010010101110100111010011011111011110110

0 0 0 0 0 0 0 0 + 0101...

Data

Parity Byte
Checksum Example

010100111101001010111101000111101011010011011111011110110

Data ➫ [0]

+ 1010…
Checksum Example

Data ↓ 01
Checksum Example

0101001111010010101111010001110101101001101111101110110

Data 010

+ 1001...
Checksum Example

Data 0101

01010011101001010111101000111010110100110111110111101101010011011111011110110 0011...
Checksum Example

Data $\uparrow \downarrow \ 01010011$

01010011110100101011110100011101011010011011111011110110

+ 1101...
Checksum Example

010100111101001010111101000111101011010011011111011011

1 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 1 1 1 0 1 0 1 1 0 1 1 1 0 1 0 1 1 0

Data

Parity Byte

0101001
1

1

1010...
Checksum Example

01010011110100101011110100011101011010011011111011110110...

Data
01010011

Parity Byte
11

10

0100...
Checksum Example

010100111101001010111101000111101011010011011111011110110101110110

1 0 0 0 0 0 0 1 + 1011...

Data

Parity Byte

01010011
11010010
10000001
Checksum Example

010100111101001010111101000111010101101001101111101110110

01010011 01010010 0111…

Data

Parity Byte

01010011
11010010
1
0
Checksum Example

010100111101001010111101000111101011010011011111011110110

1 1 1 1 0 1 1 0

Data
01010011
11010010
10111101
00011101
01101001
01110110

Parity Byte
11110110
From Sums to Remainders

- Checksums are easy to compute, but very fragile
  - In particular, burst errors are frequently undetected (yet common)
  - We’d rather have a scheme that “smears” parity

- Need to remain easy to implement in hardware
  - So far just shift registers and an XOR gate

- We’ll stick to Modulo-2 arithmetic
  - Multiplication and division are XOR-based as well
Cyclic Remainder Check (CRC)

- Also called Cyclic Redundancy Check
- Polynomial code
  - Treat packet bits as coefficients of n-bit polynomial
    - Message = 10011010
    - Generator polynomial
      \[= 1 \times x^7 + 0 \times x^6 + 0 \times x^5 + 1 \times x^4 + 1 \times x^3 + 0 \times x^2 + 1 \times x + 0\]
      \[= x^7 + x^4 + x^3 + x\]
  - Choose r+1 bit generator polynomial
    (well known – chosen in advance… handles burst errors of size r)
  - Add r bits to packet such that message is divisible by generator polynomial (these bits are the EDC)
  - Note: easy way to think of polynomial arithmetic mod 2
    - Multiplication: binary addition without carries
    - Division: binary subtraction without carries
- Better loss detection properties than checksums
Error Detection – CRC

- View data bits, $D$, as a binary number
- Choose $r+1$ bit pattern (generator), $G$
- Goal: choose $r$ CRC bits, $R$, such that
  - $<D,R>$ exactly divisible by $G$ (modulo 2)
  - Receiver knows $G$, divides $<D,R>$ by $G$. If non-zero remainder: error detected!
  - Can detect all burst errors less than $r+1$ bits
- Widely used in practice (Ethernet, FDDI, ATM)

\[
D \cdot 2^r \text{ XOR } R
\]
# Common Generator Polynomials

<table>
<thead>
<tr>
<th>Generator</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC-8</td>
<td>$x^8 + x^2 + x^1 + 1$</td>
</tr>
<tr>
<td>CRC-10</td>
<td>$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$</td>
</tr>
<tr>
<td>CRC-12</td>
<td>$x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$</td>
</tr>
<tr>
<td>CRC-16</td>
<td>$x^{16} + x^{15} + x^2 + 1$</td>
</tr>
<tr>
<td>CRC-CCITT</td>
<td>$x^{16} + x^{12} + x^5 + 1$</td>
</tr>
<tr>
<td>CRC-32</td>
<td>$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x^{1} + 1$</td>
</tr>
</tbody>
</table>
CRC Example Encoding

\[ x^3 + x^2 + 1 = 1101 \]
\[ x^7 + x^4 + x^3 + x = 10011010 \]

Message plus \( r \) zeros (*2^k)

Result:
Transmit message followed by remainder:

\[ 10011010101 \]
CRC Example Decoding

\[ x^3 + x^2 + 1 = 1101 \]
\[ x^{10} + x^7 + x^6 + x^4 + x^2 + 1 = 10011010101 \]

Generator

Received Message

r + 1 bit check sequence \( g \), equivalent to a degree-\( r \) polynomial

\[ D \mod g \]

Result:

CRC test is passed
CRC Example Failure

\[ \begin{align*}
    x^3 + x^2 + 1 &= 1101 \\
    x^{10} + x^7 + x^5 + x^4 + x^2 + 1 &= 10010110101
\end{align*} \]

Result:
CRC test failed
Summary

- Data Link Layer provides four basic services
  - Framing, multiplexing, error handling, and MAC

- Framing determines when payload starts/stops
  - Lots of different ways to do it, various efficiencies
    - Sentinels: increase size of packet, allow variable length frames
      - Stuffing
      - Clock-based

- Error detection
  - Add redundant bits to detect error
  - Strength of code depends on Hamming distance
  - Checksums & CRCs commonly used
    - CRC’s stronger, but somewhat more computational complexity
For Next Class

- Reliable transmission
  - Read 2.5 in P&D
- Geoff Voelker will be lecturing
- Reminder:
  - Homework #1 due at the beginning of class