From Signals to Packets

Bit Stream: 0 0 1 0 1 1 1 0 0 0 1

"Digital" Signal

Analog Signal

Link Layer: Implementation

- Implemented in "adapter"
  - E.g., PCMCIA card, Ethernet card
  - Typically includes: RAM, DSP chips, host bus interface, and link interface

Datalink Functions

- Framing: encapsulating a network layer datagram into a bit stream.
  - Add header, mark and detect frame boundaries
- Media access: controlling which frame should be sent over the link next.
- Error control: error detection and correction to deal with bit errors.
  - May also include other reliability support, e.g. retransmission
- Flow control: avoid that the sender outruns the receiver
- Hubbing, bridging: extend the size of the network
Outline

- Encoding and decoding
  - Translate between bits and digital signal
- Framing
  - Bit stream to packets
- Packet loss & corruption
  - Error detection
  - Flow control
  - Loss recovery

Why Encode?

- Seems obvious, why take time with this?

Why Do We Need Encoding?

- Keep receiver synchronized with sender.
- Create control symbols, in addition to regular data symbols.
  - E.g. start or end of frame, escape, ...
- Error detection or error corrections.
  - Some codes are illegal so receiver can detect certain classes of errors
  - Minor errors can be corrected by having multiple adjacent signals mapped to the same data symbol
- Encoding can be done one bit at a time or in multi-bit blocks, e.g., 4 or 8 bits.
- Encoding can be very complex, e.g. wireless.
Non-Return to Zero (NRZ)
- 1 → high signal; 0 → low signal
- Used by Synchronous Optical Network (SONET)
- Long sequences of 1’s or 0’s can cause problems:
  - Sensitive to clock skew, i.e. hard to recover clock
  - DC bias hard to detect – low and high detected by difference from average voltage

Non-Return to Zero Inverted (NRZI)
- 1 → make transition; 0 → signal stays the same
- Solves the problem for long sequences of 1’s, but not for 0’s.

Manchester Encoding
- Used by Ethernet
- 0=low to high transition, 1=high to low transition.
- Transition for every bit simplifies clock recovery
- DC balance has good electrical properties
- But you pay a price …
  - doubles the number of transitions – more spectrum!
  - Circuitry must run twice as fast

4B/5B Encoding
- Data coded as symbols of 5 line bits → 4 data bits, so 100 Mbps uses 125 MHz.
  - Uses less frequency space than Manchester encoding
  - Encoding ensures no more than 3 consecutive 0’s
  - Uses NRZI to encode resulting sequence
  - 16 data symbols, 8 control symbols
    - Data symbols: 4 data bits
    - Control symbols: idle, begin frame, etc.
  - Example: FDDI.
**4B/5B Encoding**

<table>
<thead>
<tr>
<th>Data</th>
<th>Code</th>
<th>Data</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>11110</td>
<td>0000</td>
<td>10010</td>
</tr>
<tr>
<td>0001</td>
<td>01001</td>
<td>0001</td>
<td>10011</td>
</tr>
<tr>
<td>0010</td>
<td>10100</td>
<td>0010</td>
<td>10110</td>
</tr>
<tr>
<td>0011</td>
<td>10101</td>
<td>0011</td>
<td>10111</td>
</tr>
<tr>
<td>0100</td>
<td>01010</td>
<td>0100</td>
<td>11010</td>
</tr>
<tr>
<td>0101</td>
<td>01011</td>
<td>0101</td>
<td>11011</td>
</tr>
<tr>
<td>0110</td>
<td>01110</td>
<td>0110</td>
<td>11100</td>
</tr>
<tr>
<td>0111</td>
<td>01111</td>
<td>0111</td>
<td>11101</td>
</tr>
</tbody>
</table>

From datalink

To modulator

**Other Encodings**

- 8B/10B: Fiber Channel and Gigabit Ethernet
- 64B/66B: 10 Gbit Ethernet (& 40 and 100 Gb/S)
- B8ZS: T1 signaling (bit stuffing)

**Things to Remember**

- Encoding necessary for clocking
- Lots of approaches
- Rule of thumb:
  - Little bandwidth \(\rightarrow\) complex encoding
  - Lots of bandwidth \(\rightarrow\) simple encoding

**From Signals to Packets**

- **Packet Transmission**
  - Sender
  - Receiver
- **Packets**
  - Header/Body
- **Bit Stream**
  - 0 0 1 0 1 1 0 0 0 1
- **“Digital” Signal**
- **Analog Signal**

**Outline**

- Encoding
  - Bits to digital signal
- Framing
  - Bit stream to packets
- Packet loss & corruption
  - Error detection
  - Flow control
  - Loss recovery
Framing

• How do we break up a stream of bits into frames?

01000101010111001010101011101110000001111010101110101010101101011010111001

Framing

• A link layer function, defining which bits have which function.
• Minimal functionality: mark the beginning and end of packets (or frames).
• Some techniques:
  • Out of band delimiters (e.g. 4B/5B control symbols)
  • Frame delimiter characters with character stuffing
  • Frame delimiter codes with bit stuffing
  • Synchronous transmission (e.g. SONET)
    • Boundaries are based on timing

Out-of-band: E.g., 802.5

• 802.5/token ring uses 4b/5b
• Start delim & end delim are “illegal” data codes

Delimiter Based

• SYN: sync character
• SOH: start of header
• STX: start of text
• ETX: end of text
• What happens when ETX is in Body?
Character and Bit Stuffing

- Mark frames with special character.
  - What happens when the user sends this character?
  - Use escape character when controls appear in data:
    - \*abc\*def \*abc\*def
      - Very common on serial lines, in editors, etc.
- Mark frames with special bit sequence
  - must ensure data containing this sequence can be transmitted
  - example: suppose 11111111 is a special sequence.
  - transmitter inserts a 0 when this appears in the data:
    - 11111111 \rightarrow 111111101
  - must stuff a zero any time seven 1s appear:
    - 11111110 \rightarrow 111111100
  - receiver unstuffs.

Ethernet Framing

- Preamble is 7 bytes of 10101010 (5 MHz square wave) followed by one byte of 10101011
- Allows receivers to recognize start of transmission after idle channel

Outline

- Encoding
  - Bits to digital signal
- Framing
  - Bit stream to packets
- Packet loss & corruption
  - Error detection
  - Flow control
  - Loss recovery

Error Coding

- Transmission process may introduce errors into a message.
  - Single bit errors versus burst errors
- Detection:
  - Requires a convention that some messages are invalid
  - Hence requires extra bits
  - An (n,k) code has codewords of n bits with k data bits and r = (n-k) redundant check bits
- Correction
  - Forward error correction: many related code words map to the same data word
  - Detect errors and retry transmission
**Error Detection**

- EDC = Error Detection and Correction bits (redundancy)
- D = Data protected by error checking, may include header fields
- Error detection not 100% reliable!
  - Protocol may miss some errors, but rarely
  - Larger EDC field yields better detection and correction

---

**Parity Checking**

**Single Bit Parity:**
Detect single bit errors

---

**Internet Checksum**

- Goal: detect “errors” (e.g., flipped bits) in transmitted segment

**Sender**
- Treat segment contents as sequence of 16-bit integers
- Checksum: addition (1’s complement sum) of segment contents
- Sender puts checksum value into checksum field in header

**Receiver**
- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless?

---

**Basic Concept: Hamming Distance**

- Hamming distance of two bit strings = number of bit positions in which they differ.
- If the valid words of a code have minimum Hamming distance D, then D-1 bit errors can be detected.
- If the valid words of a code have minimum Hamming distance D, then \([(D-1)/2]\) bit errors can be corrected.
Cyclic Redundancy Codes (CRC)

- Commonly used codes that have good error detection properties.
  - Can catch many error combinations with a small number of redundant bits
- Based on division of polynomials.
  - Errors can be viewed as adding terms to the polynomial
  - Should be unlikely that the division will still work
- Can be implemented very efficiently in hardware.
- Examples:
  - CRC-32: Ethernet
  - CRC-8, CRC-10, CRC-32: ATM

CRC: Basic idea

- Treat bit strings as polynomials:
  \[ X^4 + X^3 + X^1 + X^0 \]
- Sender and Receiver agree on a divisor polynomial of degree k
- Message of M bits \( \rightarrow \) send M+k bits
- No errors if M+k is divisible by divisor polynomial
- If you pick the right divisor you can:
  - Detect all 1 & 2-bit errors
  - Any odd number of errors
  - All Burst errors of less than k bits
  - Some burst errors \( \geq k \) bits

Outline

- Encoding
  - Bits to digital signal
- Framing
  - Bit stream to packets
- Packet loss & corruption
  - Error detection
  - Flow control
  - Loss recovery

Link Flow Control and Error Recovery

- Dealing with receiver overflow: flow control.
- Dealing with packet loss and corruption: error control.
- Meta-comment: these issues are relevant at many layers.
  - Link layer: sender and receiver attached to the same “wire”
  - End-to-end: transmission control protocol (TCP) - sender and receiver are the end points of a connection
- How can we implement flow control?
  - “You may send” (windows, stop-and-wait, etc.)
  - “Please shut up” (source quench, 802.3x pause frames, etc.)
  - Where are each of these appropriate?
A Naïve Protocol

- Sender simply sends to the receiver whenever it has packets.
- Potential problem: sender can outrun the receiver.
  - Receiver too slow, buffer overflow, ..
  - Not always a problem: receiver might be fast enough.

Adding Flow Control

- Stop and wait flow control: sender waits to send the next packet until the previous packet has been acknowledged by the receiver.
  - Receiver can pace the receiver

Drawback: Performance

- Stop and wait flow control results in poor throughput for long-delay paths: packet size/roundtrip-time.
- Solution: receiver provides sender with a window that it can fill with packets.
  - The window is backed up by buffer space on receiver
  - Receiver acknowledges the a packet every time a packet is consumed and a buffer is freed

Window Flow Control

- Max Throughput = \( \frac{1 \text{ pkt}}{\text{Roundtrip Time}} \)
**Bandwidth-Delay Product**

\[
\text{Max Throughput} = \frac{\text{Window Size}}{\text{Roundtrip Time}}
\]

**Error Recovery**

- Two forms of error recovery
  - Error Correcting Codes (ECC)
  - Automatic Repeat Request (ARQ)
- ECC
  - Send extra redundant data to help repair losses
- ARQ
  - Receiver sends acknowledgement (ACK) when it receives packet
  - Sender uses ACKs to identify and resend data that was lost
- Which should we use? Why? When?

**Stop and Wait**

- Simplest ARQ protocol
- Send a packet, stop and wait until acknowledgement arrives
- Will examine ARQ issues later in semester

**Recovering from Error**

- Timeouts
  - Packet lost
  - ACK lost
  - Early timeout
How to Recognize Retransmissions?

- Use sequence numbers
  - both packets and acks
- Sequence # in packet is finite \( \rightarrow \) How big should it be?
  - For stop and wait?
  - One bit – won’t send seq #1 until received ACK for seq #0

Implementation Issues with Window-based Protocol

- Window size: # of total outstanding packets that sender can send without acknowledged
- How big a sequence number do we need?
  - For m-bit sequence number: \( W_s = 2^m - 1 \)
  - Reason: if window could be \( 2^m \), then if the first packet in a window is lost, the receiver cannot not distinguish a retransmission from a new packet
- How to deal with sequence number wrap around?
  - Use unsigned arithmetic, modulo \( 2^m \)

What is Used in Practice?

- No flow or error control.
  - E.g. regular Ethernet, just uses CRC for error detection
- Flow control only
  - E.g. Gigabit Ethernet
- Flow and error control.
  - E.g. X.25 (older connection-based service at 64 Kbs that guarantees reliable in order delivery of data)
  - Flow and error control solutions also used in higher layer protocols
  - E.g., TCP for end-to-end flow and error control