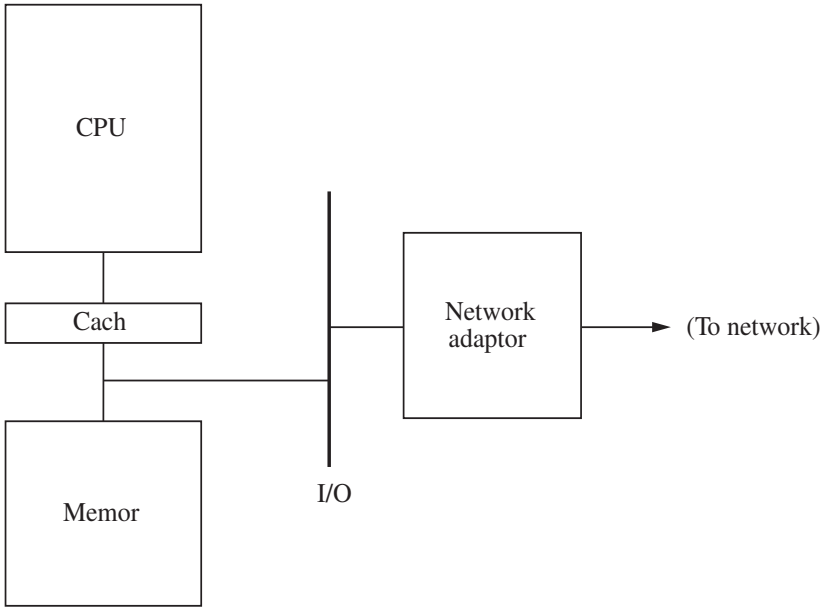
Copyright 1996. All rights reserved. Permission to reproduce this document for not for profit educational purposes is hereby granted. This document may not be reproduced for commercial purposes without the express written consent of the author.

Hardware Building Blocks

Nodes

Assume general-purpose (programmable) computers; e.g., workstations. Sometimes replaced with special-purpose hardware.



- Finite memory (implies limited buffer space)
- Connects to network via a *network adaptor*
- Fast processor, slow memory

Links

Sometimes you install your own

Category 5 twisted pair	10-100Mbps, 1
50-ohm coax (ThinNet)	10-100Mbps, 2
75-ohm coax (ThickNet)	10-100Mbps, 5
Multimode fiber	100Mbps, 2km
Single-mode fiber	100-2400Mbps

Sometimes leased from the phone company

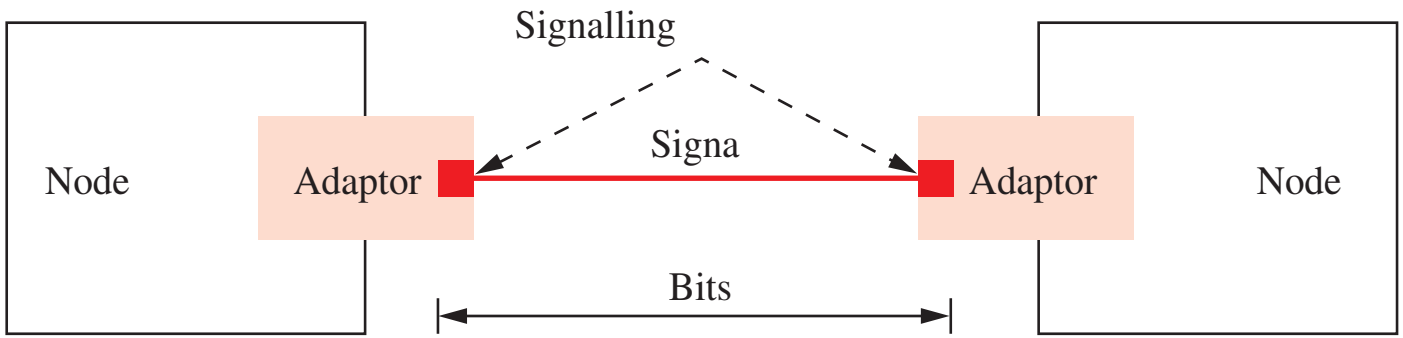
Service to ask for	Bandwidth you get
ISDN	64 Kbps
T1	1.544 Mbps
T3	44.736 Mbps
STS-1	51.840 Mbps
STS-3	155.250 Mbps
STS-12	622.080 Mbps
STS-24	1.244160 Gbps
STS-48	2.488320 Gbps

Encoding

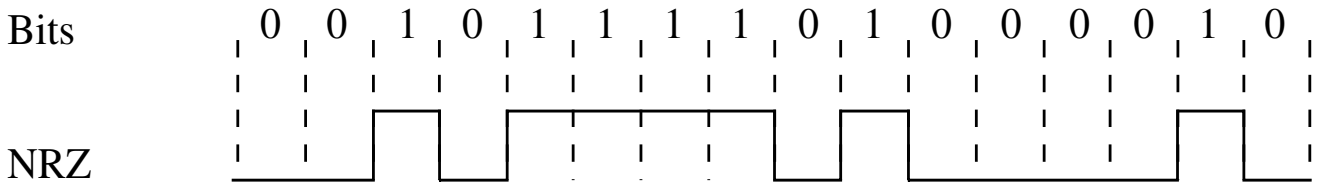
Overview

- Signals propagate over a physical medium.
- Digital signals
- Analog signals
- Data can be either digital or analog; we're interested in digital data.

Problem: Encode the binary data that the source node wants to send to the destination node into the signal that propogates over to the destination node into the signal that propogates over


Signalling
Node
Adaptor
Signa
Adaptor
Node
Bits

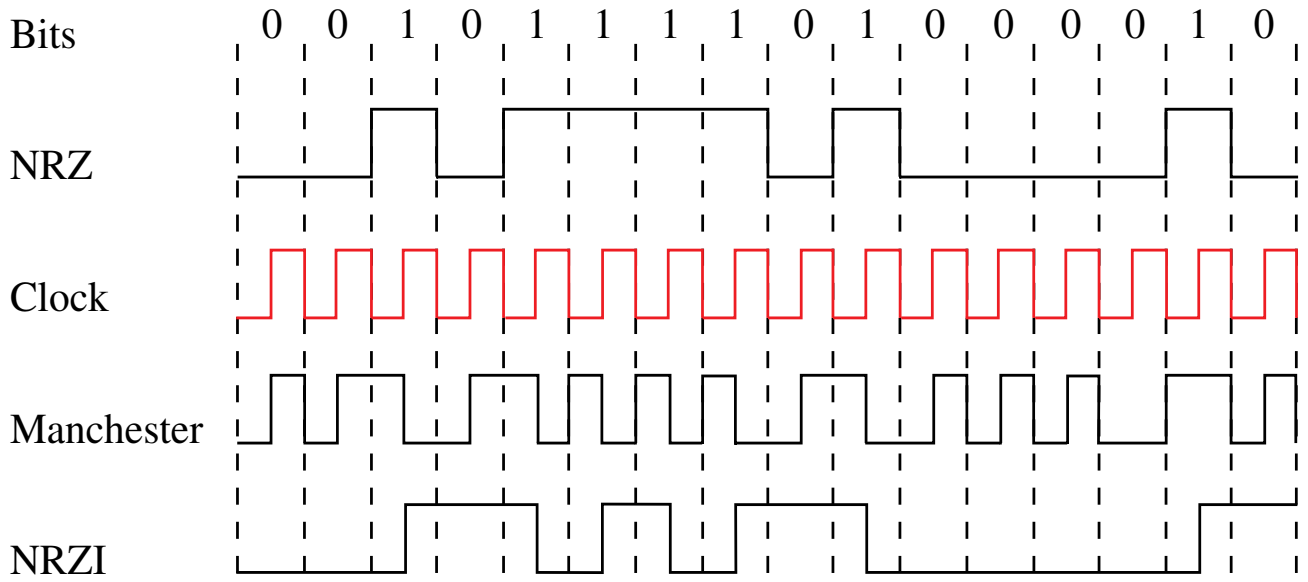
Non-Return to Zero (NRZ)



- Problem: Consecutive 1s or 0s
- Low signal (0) may be interpretted as no signal
- High signal (1) leads to baseline wander
- Unable to recover clock

NRZI and Manchester

- Non-return to Zero Inverted (NRZI): Make a transition from the current signal to encode a one, and stay at the current signal to encode a zero; solves the problem of consecutive ones.
- Manchester: Transmits the XOR of the NRZ encoded data and the clock; only 50% efficient.



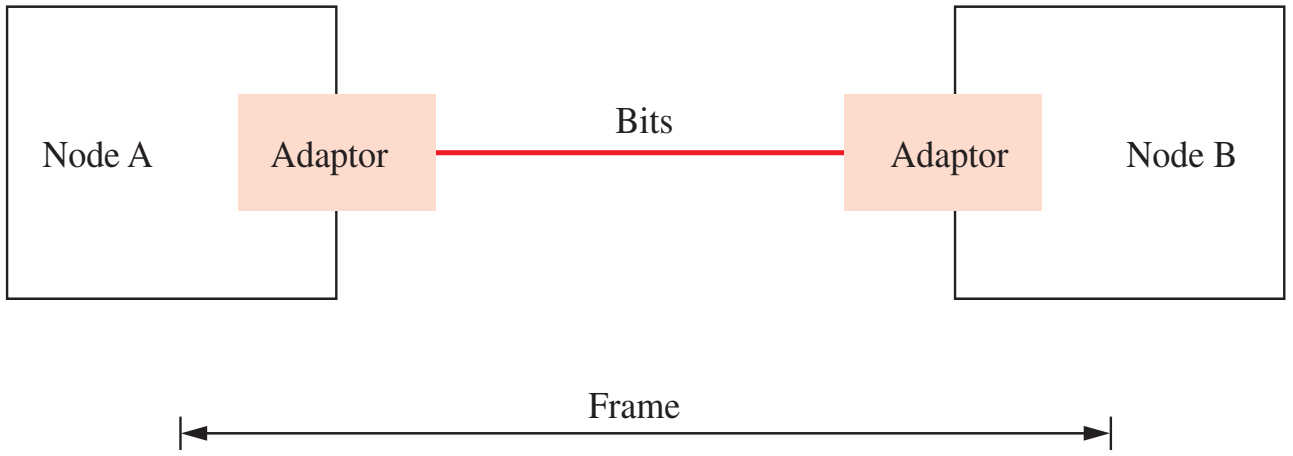
4B/5B

Idea: Every 4 bits of data is encoded in a 5-bit code, with the 5-bit codes selected to have no more than one leading 0 and no more than two trailing 0 (i.e., never get more than three consecutive 0s). Resulting 5-bit codes are then transmitted using the NRZI encoding. Achieves 80% efficiency.

Framing

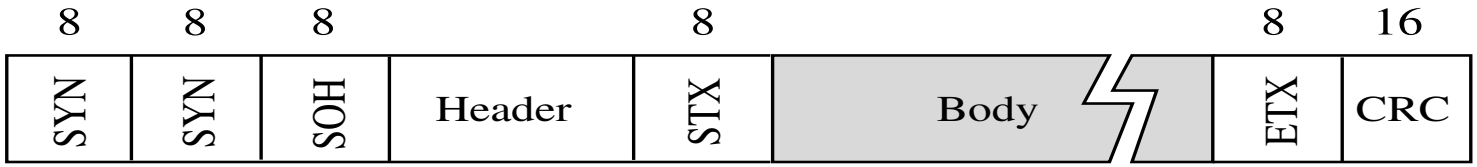
Overview

- Problem: Breaking sequence of bits into a frame
- Must determine first and last bit of the frame
- Typically implemented by network adaptor
- Adaptor fetches (deposits) frames out of (into) host memory

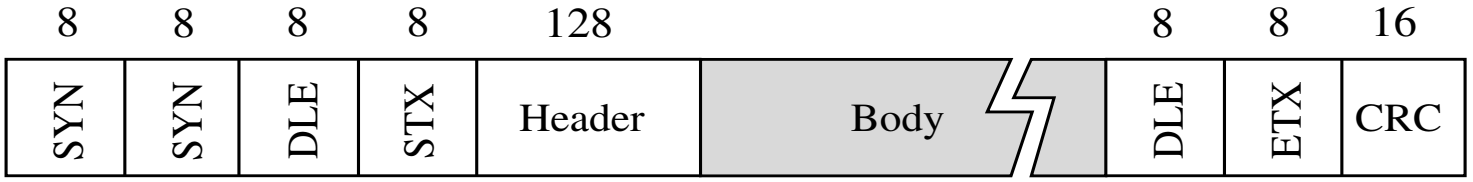


Byte-Oriented Protocols

- Sentinel Approach
  - BISYNC

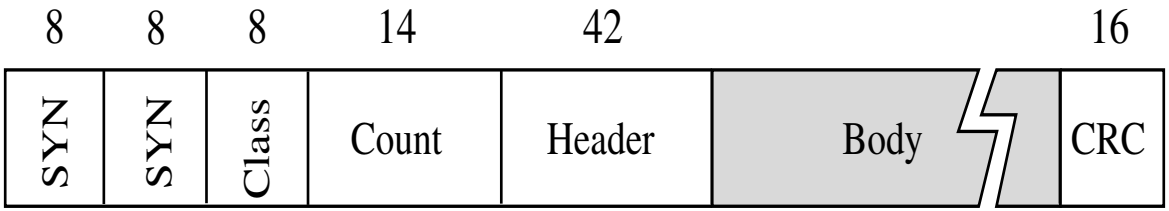


  - IMP-IMP



  - Problem: ETX character might appear in the data portion of the frame.
  - Solution: Escape the ETX character with a DLE character in BISYNC; escape the DLE character with a DLE character in IMP-IMP.

- Byte Counting Approach (DDCMP)

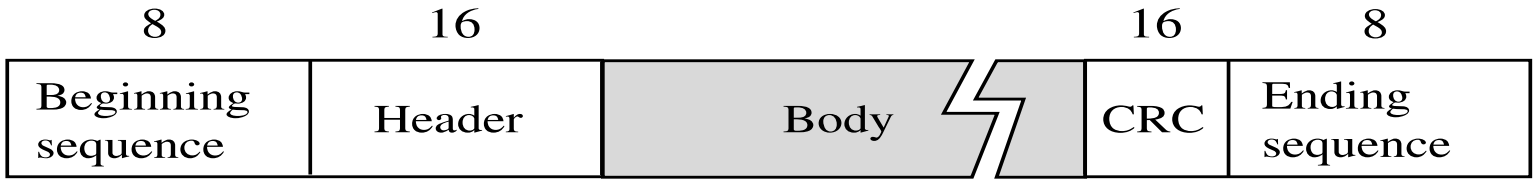


  - Problem: Count field is corrupted (framing error).
  - Solution: Catch when CRC fails.

Bit-Oriented Protocols

- HDLC: High-Level Data Link Control (also SDLC and PPP)

- Delineate frame with a special bit-sequence: 01111110

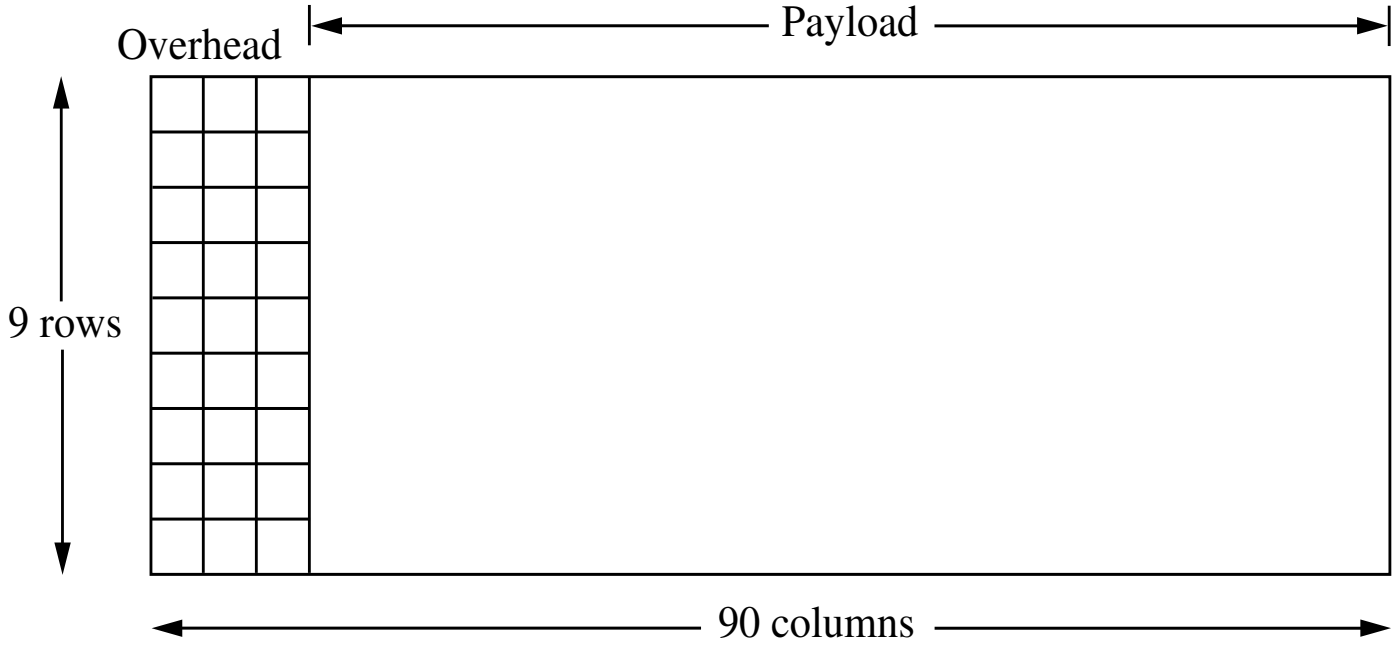


Bit Stuffing

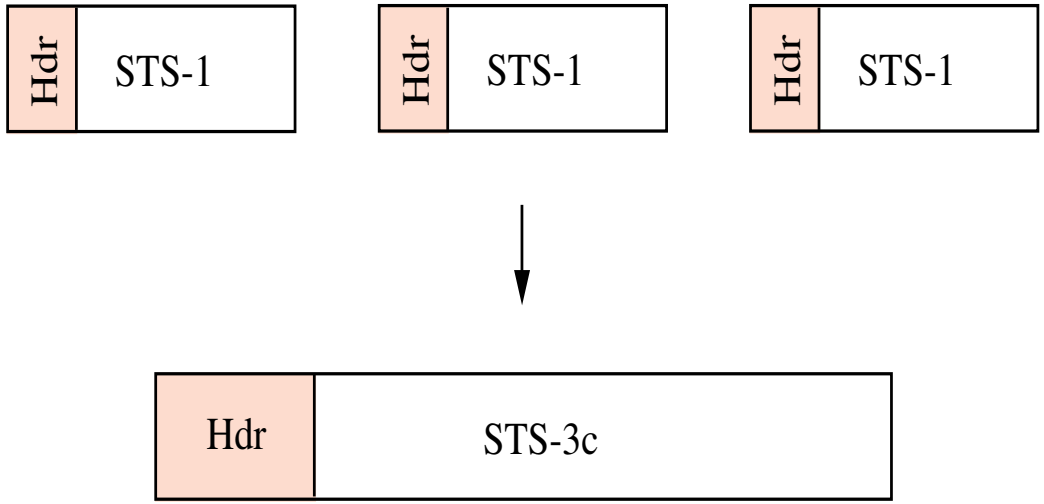
- Sender: any time five consecutive 1s have been transmitted from the body of the message, insert a 0.
- Receiver: should five consecutive 1s arrive, look at next bit(s):
  - if next bit is a 0: remove it
  - if next bits are 10: end-of-frame marker
  - if next bits are 11: error

Clock-Based Framing

- SONET: Synchronous Optical Network
- ITU standard for transmission over fiber
- STS-1 (51.84 Mbps)



- Byte-interleaved multiplexing



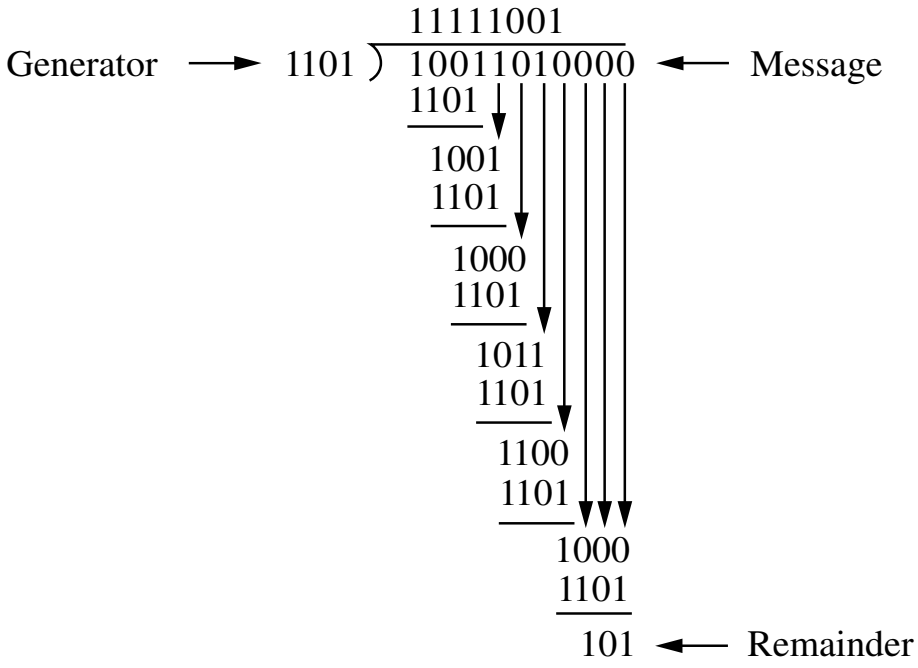
- Each frame is 125μs long.

Error Detection

Cyclic Redundancy Check

- Add *k* bits of redundant data to an *n*-bit message.
- Represent *n*-bit message as an *n-1* degree polynomial; e.g., MSG=10011010 corresponds to $M(x) = x^7 + x^4 + x^3 + x^1$.
- Let *k* be the degree of some divisor polynomial *C(x)*; e.g., $C(x) = x^3 + x^2 + 1$.
- Transmit polynomial *P(x)* that is evenly divisible by *C(x)*, and receive polynomial *P(x) + E(x)*; *E(x)*=0 implies no errors.
- Recipient divides *(P(x) + E(x))* by *C(x)*; the remainder will be zero in only two cases: *E(x)* was zero (i.e. there was no error), or *E(x)* is exactly divisible by *C(x)*. Choose *C(x)* to make second case extremely rare.

- Sender:
  - multiply *M(x)* by *xk*; for our example, we get $x^{10} + x^7 + x^6 + x^4$ (10011010000);
  - divide result by *C(x)* (1101);



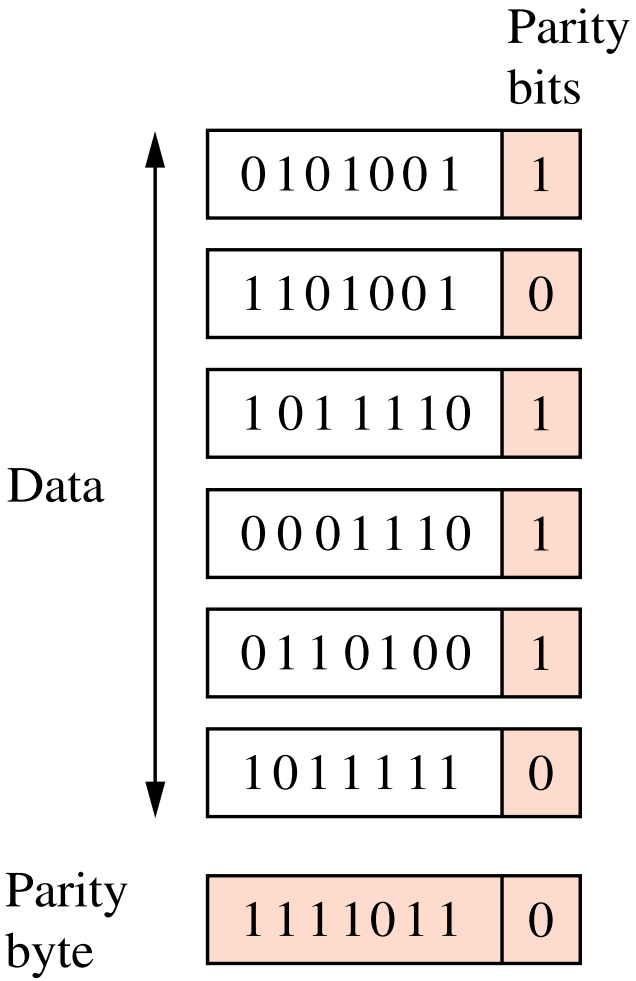
  - Send 10011010000 - 101 = 10011010101, since this must be exactly divisible by *C(x)*;

- Want to ensure that *C(x)* does not divide evenly into polynomial *E(x)*.
- All single-bit errors, as long as the *xk* and *x0* terms have non-zero coefficients.
- All double-bit errors, as long as *C(x)* has 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 errored 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.

- Common polynomials for *C(x)*:

CRC	*C(x)*
CRC-8	$x^8+x^2+x^1+1$
CRC-10	$x^{10}+x^9+x^5+x^4+x^1+1$
CRC-12	$x^{12}+x^{11}+x^3+x^2+x^1+1$
CRC-16	$x^{16}+x^{15}+x^2+1$
CRC-CCITT	$x^{16}+x^{12}+x^5+1$
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

Two-Dimensional Parity



Internet Checksum Algorithm

Idea: view message as a sequence of 16-bit integers. Add these integers together using 16-bit ones complement arithmetic, and then take the ones complement of the result. That 16-bit number is the checksum.

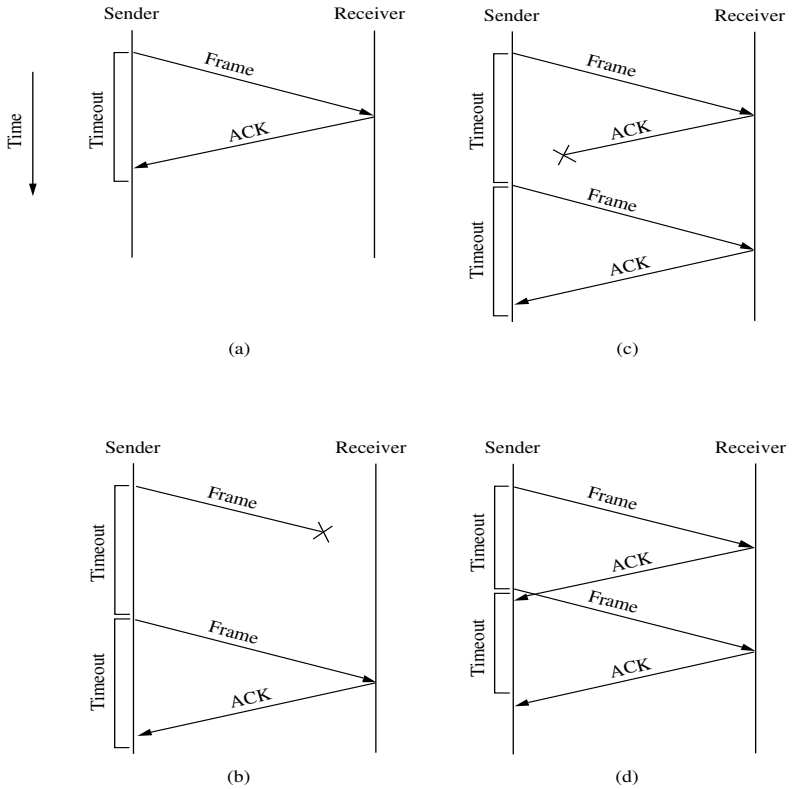
```
u_short
cksum(u_short *buf, int count)
{
   register u_long sum = 0;

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

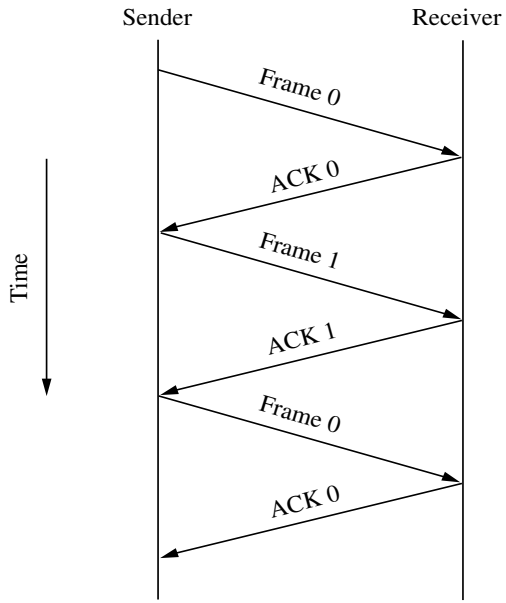
Reliable Transmission

Overview

- Recover from Corrupt Frames
  - Error Correction Codes (ECC); also called Forward Error Correction (FEC)
  - Acknowledgements and Timeouts; also called Automatic Repeat reQuest (ARQ)



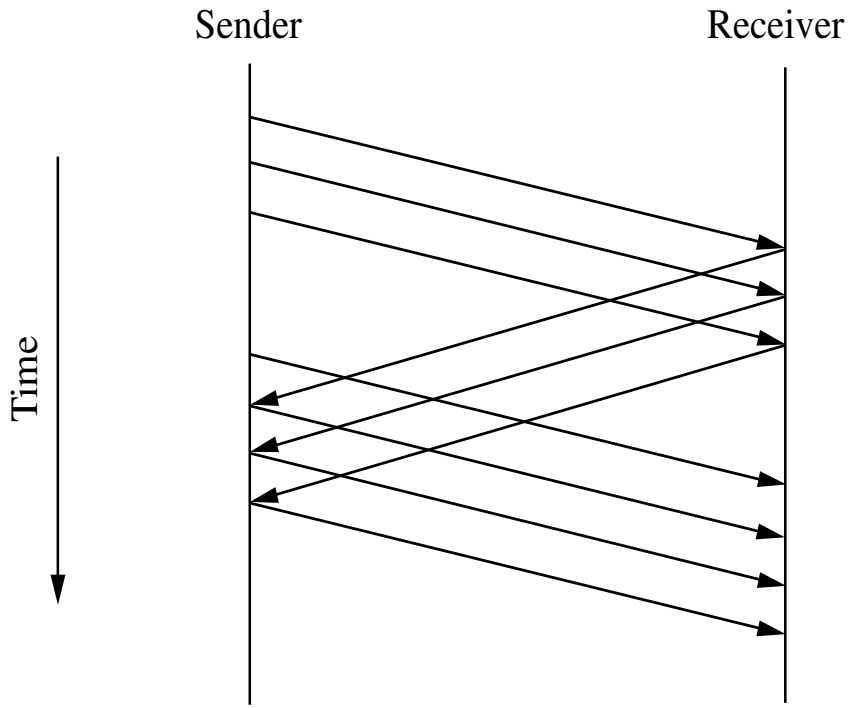
Stop-and-Wait



- Problem: Keeping the pipe full.
- Example: 1.5Mbps link x 45ms RTT = 67.5Kb (8KB). Assuming frame size of 1KB, stop-and-wait uses about one-eighth of the link's capacity. Want the sender to be able to transmit up to 8 frames before having to wait for an ACK.

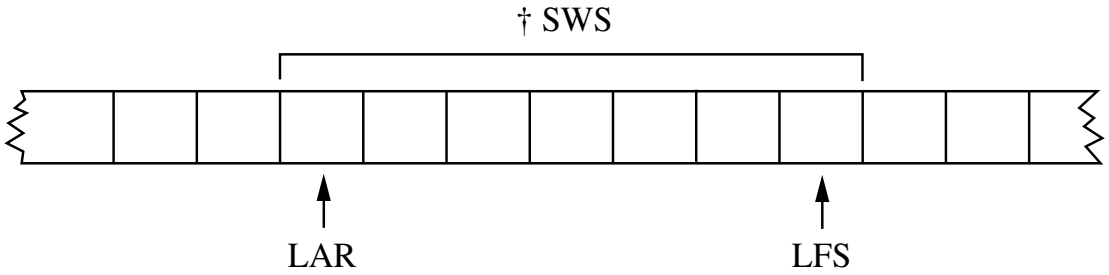
Sliding Window

Idea: Allow sender to transmit multiple frames before receiving an ACK, thereby keeping the pipe full. There is an upper limit on the number of outstanding (un-ACKed) frames allowed.



Sender:

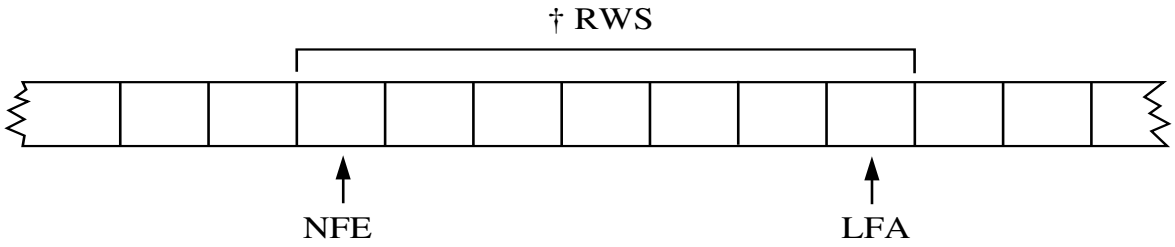
- Assign sequence number to each frame (**`SeqNum`**)
- Maintain three state variables:
  - send window size (**`SWS`**)
  - last acknowledgment received (**`LAR`**)
  - last frame sent (**`LFS`**)
- Maintain invariant: **`LFS - LAR + 1 <= SWS`**



- When **`ACK`** arrives, advance **`LAR`**, thereby opening window
- Buffer up to **`SWS`** frames

Receiver:

- Maintain three state variables:
  - receive window size (**`RWS`**)
  - last frame acceptable (**`LFA`**)
  - next frame expected (**`NFE`**)
- Maintain invariant: **`LFA - NFE + 1 <= RWS`**



- Frame **`SeqNum`** arrives:
  - if **`NFE <= SeqNum <= LFA`** ----> accept
  - if **`SeqNum < NFE`** or **`SeqNum > LFA`** ----> discarded
- Send cumulative ACK
- Variations:
  - Selective Acknowledgements (SAKS)
  - Negative Acknowledgements (NAKS)

Sequence Number Space

- **`SeqNum`** field is finite; sequence numbers wrap around
- Sequence number space must be larger than number of outstanding frames
- **`SWS <= MaxSeqNum-1`** is not sufficient
  - suppose 3-bit **`SeqNum`** field (**`0..7`**)
  - **`SWS=RWS=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

Concurrent Logical Channels

- Multiplex several logical channels over a single point-to-point link; run stop-and-wait on each logical channel.
- Maintain three bits of state for each channel:
  - boolean saying whether the channel is currently busy
  - sequence number for frames sent on this logical channel
  - next sequence number to expect on this logical channel
- ARPANET supported eight logical channels over each ground link (16 over each satellite link).

- Header for each frame included a 3-bit channel number and a 1-bit sequence number, for a total of 4 bits; same number of bits as the sliding window protocol requires to support up to eight outstanding frames on the link.
- Separates reliability from *flow control* and *frame order*.

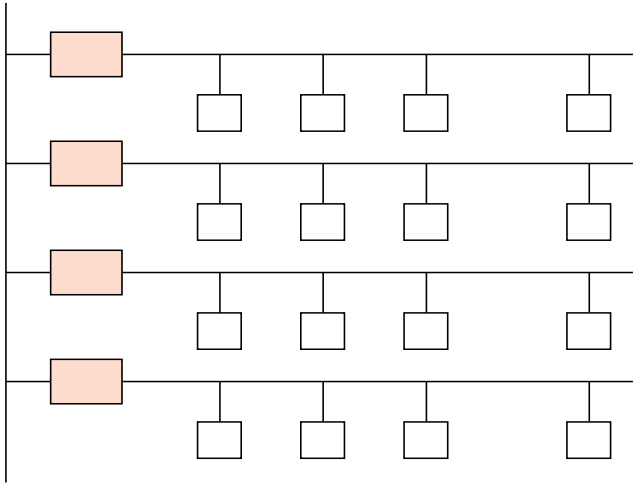
Ethernet

Overview

- History
  - Developed by Xerox PARC in mid-1970s
  - Roots in Aloha packet-radio network
  - Standardized by Xerox, DEC, and Intel in 1978
  - Similar to IEEE 802.3 standard
- CSMA/CD
  - carrier sense
  - multiple access
  - collision detection
- Bandwidth: 10Mbps and 100Mbps
- Problem: Distributed algorithm that provides fair access to a shared medium

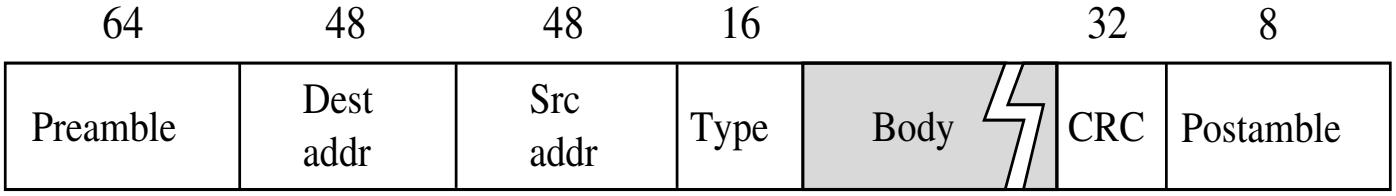
Physical Properties

- Classical Ethernet (thick-net)
  - maximum segment of 500m
  - transceiver taps at least 2.5m apart
  - connect multiple segments with repeaters
  - no more than 2 repeaters between any pair of nodes (1500m total)
  - maximum of 1024 hosts
  - also called 10Base5



- Alternative technologies
  - 10Base2 (thin-net): 200m; daisy-chain configuration
  - 10BaseT (twisted-pair): 100m; star configuration

Frame Format



Addresses:

- Unique, 48-bit unicast address assigned to each adaptor
- Example: `8:0:2b:e4:b1:2`
- Broadcast: all `1`s
- Multicast: first bit is `1`

Adaptor receives all frames; it accepts (passes to host):

- Frames addressed to its own unicast address
- Frames addressed to the broadcast address
- Frames addressed to any multicast address it has been programmed to accept
- All frames when in promiscuous mode

Transmitter Algorithm

If line is idle:

- Send immediately
- Upper bound message size of 1500 bytes
- Must wait 51μs between back-to-back frames

If line is busy:

- Wait until idle and transmit immediately
- Called *1-persistent* (*special case of p-persistent*)

If collision:

- jam for 512 bits, then stop transmitting frame
- minimum frame is 64 bytes (header + 46 bytes of data)
- delay and try again
  - 1st time: uniformly distributed between 0 and 51.2μs
  - 2nd time: uniformly distributed between 0 and 102.4μs
  - 3rd time: uniformly distributed between 0 and 204.8μs
  - give up after several tries (usually 16)
  - exponential backoff

Experiences

Observe in Practice

- 10-200 hosts (not 1024)
- Length shorter than 1500m (RTT closer to 5μ than 51μ)
- Packet length is bimodal
- High-level flow control and host performance limit load

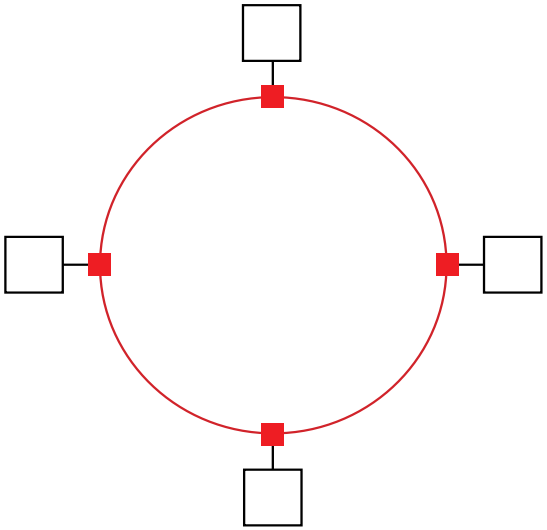
Recommendations

- Do not overload (30% utilization is about max)
- Implement controllers correctly
- Use large packets
- Get the rest of the system right (broadcast, retransmission)

FDDI

Overview

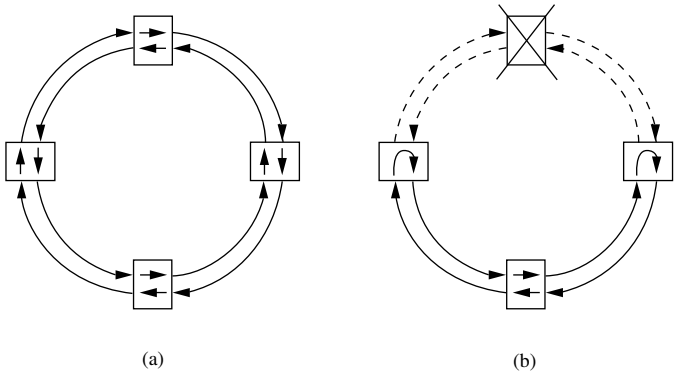
- Token Ring Networks
  - PRONET: 10Mbps and 80 Mbps rings
  - IBM: 4Mbps token ring
  - 16Mbps IEEE 802.5/token ring
  - 100Mbps Fiber Distributed Data Interface (FDDI)



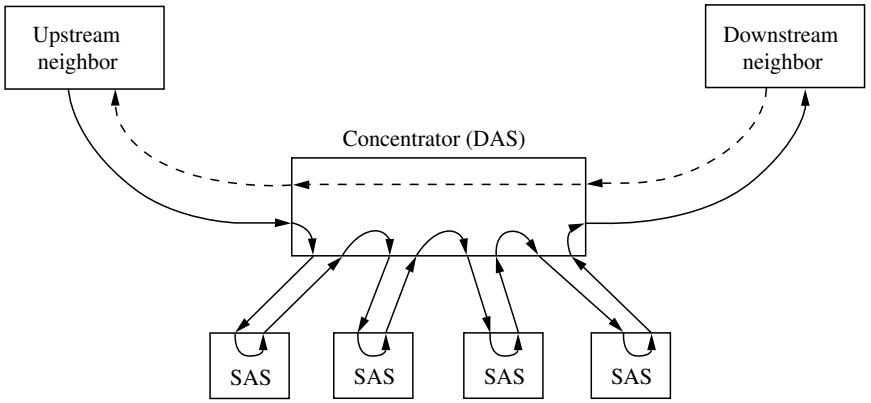
- Basic Idea
  - frames flow in one direction: upstream to downstream
  - special bit pattern (token) rotates around ring
  - must capture token before transmitting
  - release token after done transmitting
    - immediate release
    - delayed release
  - remove your frame when it comes back around
  - stations get round-robin service

Physical Properties of FDDI

Dual Ring Configuration



Single and Dual Attachment Stations



- Each station imposes a delay (e.g., 50ns)
- Maximum of 500 stations
- Upper limit of 100km (200km of fiber)
- Uses 4B/5B encoding
- Can be implemented over copper (CDDI)

Timed Token Algorithm

- Token Holding Time (THT): upper limit on how long a station can hold the token.
- Token Rotation Time (TRT): how long it takes the token to traverse the ring.

  TRT <= ActiveNodes x THT + RingLatency

- Target Token Rotation Time (TTRT): agreed-upon upper bound on TRT.

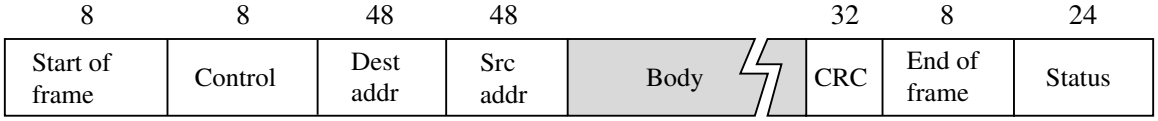
- Algorithm
  - each node measures TRT between successive arrivals of the token
  - if measured TRT > TTRT, then token is late so don't send data
  - if measured TRT < TTRT, then token is early so OK to send data
  - define two classes of traffic
    - synchronous data: can always send
    - asynchronous data: can send only if token is early
  - worse case: 2xTTRT between seeing token
- not possible to have back-to-back rotations that take 2xTTRT time

Token Maintenance

- Lost Token
  - no token when initializing ring
  - bit error corrupts token pattern
  - node holding token crashes
- Generating a Token (and agreeing on TTRT)
  - execute when join ring or suspect a failure
  - each node sends a special *claim frame* that includes the node's *bid* for the TTRT
  - when receive claim frame, update bid and forward
  - if your claim frame makes it all the way around the ring:
    - your bid was the lowest
    - everyone knows TTRT
    - you insert new token

- Monitoring for a Valid Token
  - should see valid transmission (frame or token) periodically
  - maximum gap = ring latency + max frame <= 2.5ms
  - set timer at 2.5ms and send claim frame if it fires

Frame Format



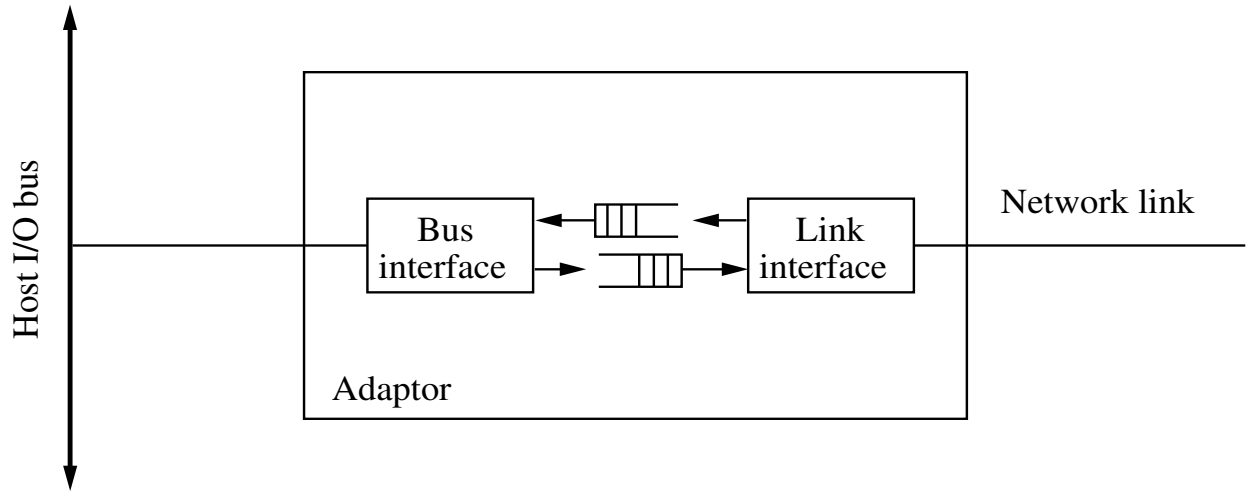
- Control Field
- 1st bit: asynchronous (0) versus synchronous (1) data
- 2nd bit: 16-bit (0) versus 48-bit (1) addresses
- last 6 bits: demux key (includes reserved patterns for token and claim frame)
- Status Field
  - from receiver back to sender
  - error in frame
  - recognized address
  - accepted frame (flow control)

Network Adaptors

Overview

Typically where data link functionality is implemented

- Framing
- Error Detection
- Media Access Control (MAC)



Host Perspective

Control Status Register (CSR)

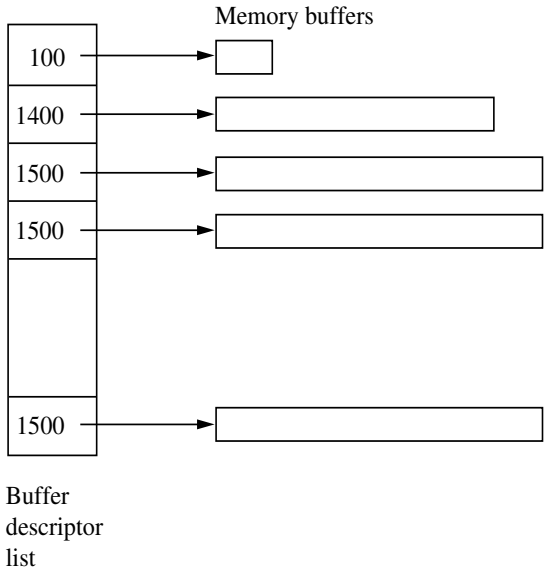
- Available at some memory address
- CPU can read and write
- CPU instructs Adaptor (e.g., transmit)
- Adaptor informs CPU (e.g., receive error)

Example

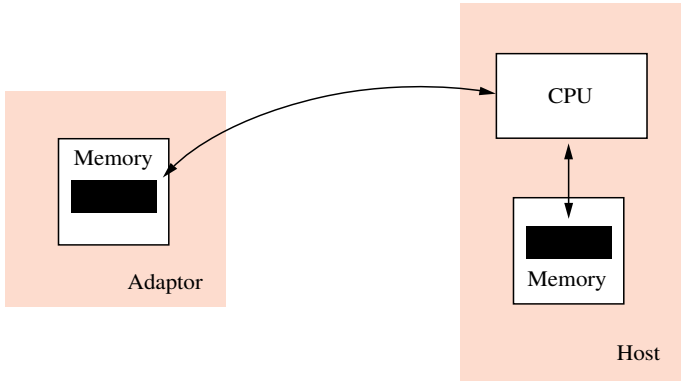
```
LE_RINT  0x0400  Received packet Interrupt (RC)
LE_TINT  0x0200  Transmitted packet Interrupt (RC)
LE_IDON  0x0100  Initialization Done (RC)
LE_IENA  0x0040  Interrupt Enable (RW)
LE_INIT  0x0001  Initialize (RW1)
```

Moving Frames Between Host and Adaptor

Direct Memory Access (DMA)



Programmed I/O (PIO)



Device Driver

Interrupt Handler

```
interrupt_handler()
{
   disable_interrupts();
   /* some error occurred */
   if (csr & LE_ERR)
   {
      print_and_clear_error();
   }
   /* transmit interrupt */
   if (csr & LE_TINT)
   {
      csr = LE_TINT | LE_INEA;
      semSignal(xmit_queue);
   }
   /* receive interrupt */
   if (csr & LE_RINT)
   {
      receive_interrupt();
   }
   enable_interrupts();
   return(0);
}
```

Transmit Routine:

```
transmit(Msg *msg)
{
   char *src, *dst;
   Context c;
   int len;

   semWait(xmit_queue);
   semWait(mutex);
   disable_interrupts();
   dst = next_xmit_buf();
   msgWalkInit(&c, msg);
   while ((src = msgWalk(&c, &len)) != 0)
       copy_data_to_lance(src, dst, len);
   msgWalkDone(&c);
   enable_interrupts();
   semSignal(mutex);
   return;
}
```

Receive Interrupt Routine

```
receive_interrupt()
{
   Msg  *msg,  *new_msg;
   char *buf;

   while (rdl = next_rcv_desc())
   {
      /* create process to handle this message */
      msg = rdl->msg;
      process_create(ethDemux, msg);

      /* msg eventually freed in ethDemux */
      /* now allocate a replacement */
      buf = msgConstructAllocate(new_msg, MTU);
      rdl->msg = new_msg;
      rdl->buf = buf;
      install_rcv_desc(rdl);
   }
   return;
}
```