EEEN20060 Communication Systems

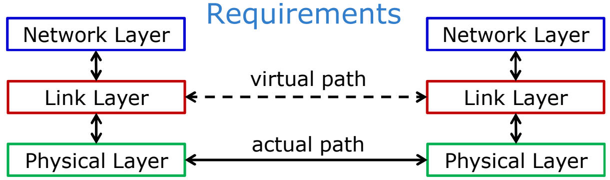
Link Layer Protocols – part 1

Brian Mulkeen

UCD School of Electrical, Electronic and Communications Engineering

Scoil na hInnealtóireachta Leictrí, Leictreonaí agus Cumarsáide UCD

Requirements



- Provide agreed service to network layer
  - using bit-transmission service of physical layer
- Organise stream of bits
  - identify start and end of groups of bits
- Make link reliable – no errors?
  - physical layer may introduce bit errors
- Make link reliable – flow control?
  - sender must slow down if receiver busy…

2

How to Eliminate Errors?

- First need to detect error(s) in a block of data
  - need extra information to do this…
- Option 1: arrange to re-transmit any bad block
  - need reverse path to do this – back to sender
  - will cause delay in data stream...
- Option 2: ignore any bad block
  - avoids using bad data, but no good data either
  - avoids delay in data stream...
- How big should the block of data be?
  - send entire file, then check?
  - send a few bits at a time?
  - will analyse this later…

3

Link Layer Frame

Header	Data Block (payload)	Trailer

- Need to work with blocks of data
  - optimum size range to be determined later
  - if given larger blocks, break them up
  - if given smaller blocks, group them
- Need to add extra information to each block
  - control info, to make link-layer protocol work
  - usually some before data block = header
  - and some after = trailer
- Header + data block + trailer = *frame*

4

Extra Information? Examples:

- Frame identification and structure
  - receiver must identify start and end of frame
  - may include size - how much data carried?
  - usually includes *sequence number…*
- Frame type
  - carrying data? control info for protocol? etc.
- Addresses – identify destination & source
  - who should receive? who sent?
- Error detection
  - extra information to allow errors to be detected
  - various methods – will look at some examples
  - bits added for error detection called *check bits*

5

Link Layer Protocol Categories

- Byte-oriented protocols – look at these now
  - assume all data in groups of 8 bits
    - could be text characters or arbitrary data
  - control information also in bytes
  - simpler to implement, but not as flexible
    - developed earlier, when most data was text
- Bit-oriented protocols – return to these later
  - data may be any number of bits
    - not necessarily groups of 8…
    - but often multiples of 8 bits – computer processors…
  - control information can also be any length
  - protocol works at bit level
    - different techniques for identifying frames, error detection, etc.

6

Question 1 – How to Detect Errors?

PPSN	1	2	3	4	5	6	7	T	
multiply	8	7	6	5	4	3	2		
add	8	14	18	20	20	18	14		= 112
modulo 23 (remainder after ÷ 23)									= 20 → T

- Error detection widely used in important data
  - PPS No., bank account, credit card number. . .
  - mainly to guard against human error!
  - in bar codes – possible mis-read by scanner
- Communication systems
  - binary data, usually received in order sent
  - common techniques: parity bits, checksum, cyclic redundancy check (CRC - later)

7

Detecting Bit Errors

count of 1s in group is always even

11000101 → 011000101

10100111 → 110100111

- Add extra bits to a group of data bits
  - according to some rule
- Check at receiver, see if rule still obeyed
- Called *error-detecting code*
  - bits added called *check bits*
- Even parity example – very simple code
  - can only detect one error in the group
  - note that error in a check bit also possible!

8

Error Bursts

TX 1 0 1 1 0 0 0 1 1 0 1 0 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 1 0 1

RX 1 0 1 1 0 0 0 1 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 0 1 0 0 1 0 1

- On some channels, errors occur in bursts
  - group of consecutive bits, many errors
  - not necessarily all in error
- Simple parity bit as above cannot cope
  - what is probability of detecting error?
- Often use interleaving of data
  - transmit bits in different order
    - damaged bits not consecutive in message
  - can also use interleaved parity check scheme

9

Interleaved Parity Check

- View bits in 2-D array
  - or sequence of bytes
  - calculate parity on columns
  - transmit row by row
  - check bits in last row
- Advantage of interleaving?
- Check bits usually at end
  - generated in hardware
    - often while data bits are being transmitted
  - checked in hardware
    - while bits being received...

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

10

Checksum 1

- View bits in 2-D array
  - or sequence of bytes
    - suits byte oriented protocol
  - each row/byte is a number
  - add all numbers, modulo *N*
    - remainder after divide by *N*
  - transmit result at end
- Example - bytes
  - sum 1203,
  - modulo 256 get 179
  - gives 8-bit checksum
    - 8 check bits, or one byte
- What does receiver do?

Binary view								Decimal
1	0	0	1	0	1	1	1	151
1	0	0	1	1	0	0	0	152
1	0	0	1	1	0	0	1	153
1	1	0	0	1	0	1	0	202
1	0	1	1	1	0	1	1	187
0	0	1	1	1	1	0	0	60
0	1	0	1	1	1	0	1	93
0	1	0	1	1	1	1	0	94
0	0	0	1	1	1	1	1	31
0	1	0	1	0	0	0	0	80
1	0	1	1	0	0	1	1	179

11

Checksum 1 at Receiver

- Errors have occurred
  - shown in red
- Receiver also calculates checksum, same way
  - add all numbers, modulo *N*
  - at end, compare result
    - should match received checksum
- Example
  - sum 1231
  - modulo 256 get 207
  - does not match - ERROR

1	0	0	1	0	1	1	1	151
1	0	0	1	1	0	0	0	152
1	0	0	1	1	0	0	1	153
1	1	1	0	1	0	1	0	234
1	0	1	1	1	0	1	1	187
0	0	1	1	1	1	0	0	60
0	1	0	1	1	0	0	1	89
0	1	0	1	1	1	1	0	94
0	0	0	1	1	1	1	1	31
0	1	0	1	0	0	0	0	80
1	1	0	0	1	1	1	1	207
1	0	1	1	0	0	1	1	179

12

Checksum 2

- Alternative
  - check bits are complement of sum
  - so entire block (including check bits) adds to 0
  - more work at tx, simpler task at rx…
- Example as before
  - sum modulo 256 = 179
  - transmit 256 – 179 = 77
    - subtraction modulo 256
  - receiver adds all bytes, modulo 256
  - result should be 0 (if no error)

1	0	0	1	0	1	1	1	151
1	0	0	1	1	0	0	0	152
1	0	0	1	1	0	0	1	153
1	1	0	0	1	0	1	0	202
1	0	1	1	1	0	1	1	187
0	0	1	1	1	1	0	0	60
0	1	0	1	1	1	0	1	93
0	1	0	1	1	1	1	0	94
0	0	0	1	1	1	1	1	31
0	1	0	1	0	0	0	0	80
0	1	0	0	1	1	0	1	77
								0

How Reliable is Checksum?

- Will detect any single bit error
  - no matter how large the block
- What about two bits in error?
- Error burst?
- Completely random data at receiver?
- Can improve by using larger checksum
  - e.g. modulo 65536, giving 16-bit result
  - even better if add 16-bit values to get it…
  - cost is increased overhead
- Other techniques for bit-oriented protocols…

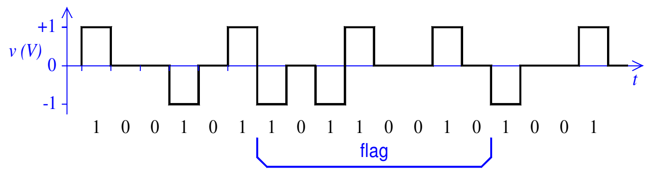
Question 2 – How to Recognise Frames

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

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

- Need to find start & end of every frame
  - receiver gets stream of bits from physical layer
    - sometimes divided into groups (bytes?)
- Possibilities?
  - use special signal at physical layer
    - e.g. violation of normal bit rules
  - count bits (or bytes)
  - use special marker bytes
    - only useful if already have bytes
    - common with byte-oriented protocols
  - use special marker bit sequence – flag
    - common in bit-oriented protocols

Physical Layer Example



- Recall AMI (alternate mark inversion)
  - 3-level signal, 0 → 0 V, 1 → ±1 V alternately
- Could allow violation of normal rule
  - to mark start of frame only
  - e.g. two consecutive 1 bits of same polarity
    - maybe followed by two of opposite polarity, to maintain zero DC component
- Mixes link-layer with physical layer. . .

Counting Example

- This example uses bytes…
- First byte of frame is no. data bytes
  - this is followed by those data bytes
  - then 2 bytes with check bits
- Byte after check bits must be start of next frame
  - and so on. . .
- Problems?
  - in practice, never used alone…

27
12
7
56
123
222
check
check
34
45
87

Each box is one byte – 8 bits
Value shown in decimal

17

Using Marker Bytes

- Define some specific byte values
  - use one to mark start of header
  - maybe end of header, if variable length
  - also mark start and/or end of trailer
    - need to know where frame ends…
- Suits byte-oriented protocol
  - often view each byte as representing a character, so *character-oriented* protocol
- Problems?
  - marker byte values may occur in data?
    - or elsewhere in header, or in check bits
  - need stream of bytes - how?

StartF
12
7
EndH
data
data
data
check
check
EndF
StartF
13

18

Character Coding

G =	1000111	71
a =	1100001	97

- Often need to transmit text
  - need to agree binary code for each character
  - this is presentation layer issue…
- ASCII – original 7-bit code, still common
  - American Standard Code for Information Interchange
  - with 7 bits, can represent 128 characters
  - 95 normal characters, printed or displayed
  - 33 control characters
    - some to control layout of text
    - others for use in Link Layer protocols
- Often extended to 8 bits
  - allows extra accented letters, symbols, etc.
  - moving to wider codes, for more languages

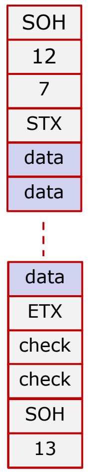
19

ASCII – decimal values

<table>
<tr><td>0</td><td>NUL</td><td>16</td><td>DLE</td><td>32</td><td>SP</td><td>48</td><td>0</td><td>64</td><td>@</td><td>80</td><td>P</td><td>96</td><td>`</td><td>112</td><td>p</td></tr>
<tr><td>1</td><td>SOH</td><td>17</td><td>DC1</td><td>33</td><td>!</td><td>49</td><td>1</td><td>65</td><td>A</td><td>81</td><td>Q</td><td>97</td><td>a</td><td>113</td><td>q</td></tr>
<tr><td>2</td><td>STX</td><td>18</td><td>DC2</td><td>34</td><td>"</td><td>50</td><td>2</td><td>66</td><td>B</td><td>82</td><td>R</td><td>98</td><td>b</td><td>114</td><td>r</td></tr>
<tr><td>3</td><td>ETX</td><td>19</td><td>DC3</td><td>35</td><td>#</td><td>51</td><td>3</td><td>67</td><td>C</td><td>83</td><td>S</td><td>99</td><td>c</td><td>115</td><td>s</td></tr>
<tr><td>4</td><td>EOT</td><td>20</td><td>DC4</td><td>36</td><td>$</td><td>52</td><td>4</td><td>68</td><td>D</td><td>84</td><td>T</td><td>100</td><td>d</td><td>116</td><td>t</td></tr>
<tr><td>5</td><td>ENQ</td><td>21</td><td>NAK</td><td>37</td><td>%</td><td>53</td><td>5</td><td>69</td><td>E</td><td>85</td><td>U</td><td>101</td><td>e</td><td>117</td><td>u</td></tr>
<tr><td>6</td><td>ACK</td><td>22</td><td>SYN</td><td>38</td><td>&</td><td>54</td><td>6</td><td>70</td><td>F</td><td>86</td><td>V</td><td>102</td><td>f</td><td>118</td><td>v</td></tr>
<tr><td>7</td><td>BEL</td><td>23</td><td>ETB</td><td>39</td><td>'</td><td>55</td><td>7</td><td>71</td><td>G</td><td>87</td><td>W</td><td>103</td><td>g</td><td>119</td><td>w</td></tr>
<tr><td>8</td><td>BS</td><td>24</td><td>CAN</td><td>40</td><td>(</td><td>56</td><td>8</td><td>72</td><td>H</td><td>88</td><td>X</td><td>104</td><td>h</td><td>120</td><td>x</td></tr>
<tr><td>9</td><td>TAB</td><td>25</td><td>EM</td><td>41</td><td>)</td><td>57</td><td>9</td><td>73</td><td>I</td><td>89</td><td>Y</td><td>105</td><td>i</td><td>121</td><td>y</td></tr>
<tr><td>10</td><td>LF</td><td>26</td><td>SUB</td><td>42</td><td>*</td><td>58</td><td>:</td><td>74</td><td>J</td><td>90</td><td>Z</td><td>106</td><td>j</td><td>122</td><td>z</td></tr>
<tr><td>11</td><td>VT</td><td>27</td><td>ESC</td><td>43</td><td>+</td><td>59</td><td>;</td><td>75</td><td>K</td><td>91</td><td>[</td><td>107</td><td>k</td><td>123</td><td>{</td></tr>
<tr><td>12</td><td>FF</td><td>28</td><td>FS</td><td>44</td><td>,</td><td>60</td><td><</td><td>76</td><td>L</td><td>92</td><td>\</td><td>108</td><td>l</td><td>124</td><td>|</td></tr>
<tr><td>13</td><td>CR</td><td>29</td><td>GS</td><td>45</td><td>-</td><td>61</td><td>=</td><td>77</td><td>M</td><td>93</td><td>]</td><td>109</td><td>m</td><td>125</td><td>}</td></tr>
<tr><td>14</td><td>SO</td><td>30</td><td>RS</td><td>46</td><td>.</td><td>62</td><td>></td><td>78</td><td>N</td><td>94</td><td>^</td><td>110</td><td>n</td><td>126</td><td>~</td></tr>
<tr><td>15</td><td>SI</td><td>31</td><td>US</td><td>47</td><td>/</td><td>63</td><td>?</td><td>79</td><td>O</td><td>95</td><td>_</td><td>111</td><td>o</td><td>127</td><td>DEL</td></tr>
</table>

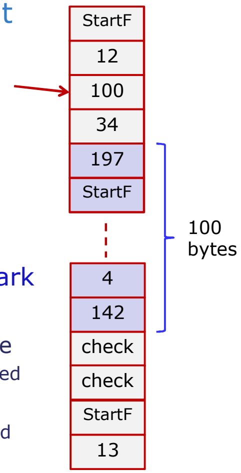
Marker Bytes Example

- ASCII control characters
  - some to mark parts of frame: e.g.
  - SOH = 1 = start of header
  - STX = 2 = start of text (payload)
  - ETX = 3 = end of text
  - EOT = 4 = end of transmission
- Easy if data does not use these values
  - e.g. simple text file
    - only visible characters and layout control
    - usually, all bytes < 128
- But how can we send arbitrary data?
  - e.g. send image as JPEG…?
    - could contain any byte value (8-bit pattern)



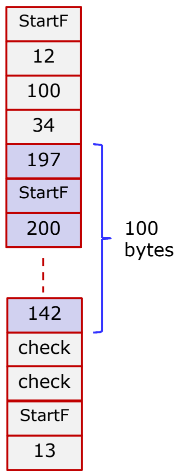
Arbitrary Data – Byte Count

- Header includes byte count
  - receiver ignores all bytes in data part of frame
  - until specified number of bytes received
  - so can send arbitrary data…
- Depends entirely on start mark
  - and byte count in header
  - in example, fixed header size
    - so no end header marker needed
  - usually fixed trailer size
    - so no end frame marker needed
    - often included as safety check



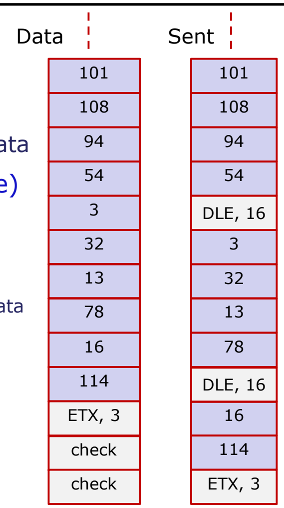
Byte Count Problems

- Start mark corrupted
  - miss entire frame?
  - find start mark in data?
  - could we make this more reliable?
- Byte count corrupted?
  - use error detection on header?
    - at least know something wrong…
    - could just send byte count twice…
- How to get started?
  - sometimes receiver starts when transmission already in progress



Arbitrary Data – Byte Stuff

- "Byte Stuffing"
  - allows any byte to occur in data
- Define *escape* character (byte)
  - e.g. ASCII DLE, code 16
    - Data Link Escape
  - meaning:
    - treat next character (byte) as data
- TX: *stuff* DLE into stream
  - during data part of frame
    - and most of header and trailer?
  - before any protocol byte
  - including before any DLE
- RX: ?



Arbitrary Data - Encoding

77 M	1 SOH	97 a

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

19 T	16 Q	5 F	33 h

- Encode data – using more bytes than needed
  - but restrict range of bytes in code
    - avoid bytes used in protocol
- Example – BASE64 encoding
  - often used for e-mail attachments
    - e-mail standards originally designed for text only…
  - encode three data bytes as four 6-bit groups
  - then map 6-bit groups to printable characters
    - A to Z, a to z, 0 to 9, +, /, in that order
    - so A represents 000000, B = 000001, / = 111111
  - only normal text characters in data, 7-bit ASCII
  - but 4 bytes sent for every 3 data bytes

25

Byte Synchronisation – Link Layer

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

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

- How to find bytes in a stream of bits?
- 1. Continuous transmission
  - send special SYNC byte between frames
    - or in any idle time, so always transmitting
    - ignored by receiver, but keeps receiver synchronised
  - binary pattern with no internal repetition
- 2. Burst transmission
  - send a few SYNC bytes before start of frame
  - allow receiver to synchronise with bytes
  - example sends LSB first, StartFrame = 1
  - uses ASCII SYN: 00010110 (decimal 22)

26

Byte Synchronisation – Physical Layer

- Physical layer recognises groups of bits
  - delivers stream of bytes to link layer
  - does not follow standard layer boundaries!!
- e.g. *Asynchronous* Transmission
  - asynchronous = not synchronous
    - no fixed timing signal
  - send group of bits (byte) whenever ready
    - designed to make it easy for receiver to recover timing
    - also to cope with sporadic data – human typing…
  - each group has start bit and stop bit for sync.
    - may also add parity bit for error detection
  - common at low speeds, e.g. PC serial port

27

Link Layer Protocol Design

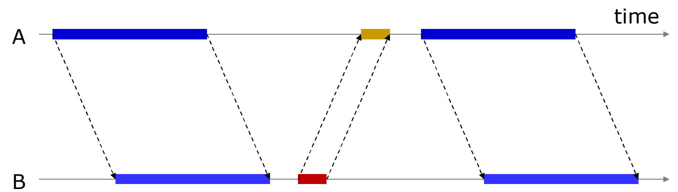
- Can now:
  - identify frames at receiver
  - detect errors in received frames
- Question 3
  - how to arrange re-transmission of bad frame?
  - frame has been received, but failed error check
  - must also cope if frame header damaged
    - maybe frame not recognised at receiver
- Ideas?
- To be precise, this is Logical Link Control
  - higher part of link layer protocol

28

Acknowledgements

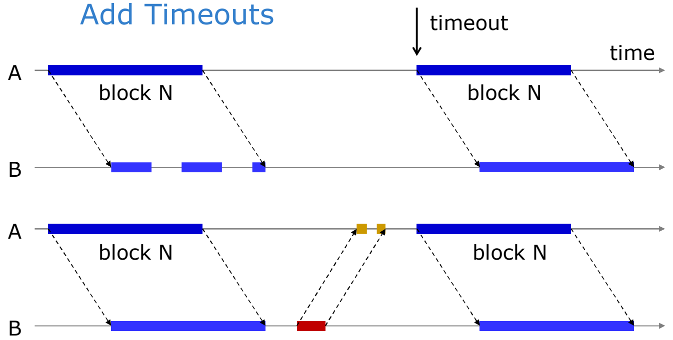
- Most designs use acknowledgement – ACK
  - sent by receiver, to confirm data block OK
  - may also mean "ready for more" – flow control
- Negative acknowledgement – NAK
  - indicates data block had errors
  - not always used – can just say nothing…
- Sent as short frame
  - byte-oriented protocol, could be single byte?
    - see ASCII table: ACK = 6, NAK = 21
    - but no protection against errors… other problems…
  - data transfer in both directions at same time?
    - can include ACK/NAK in header of data frame travelling in opposite direction

Stop and Wait Protocol



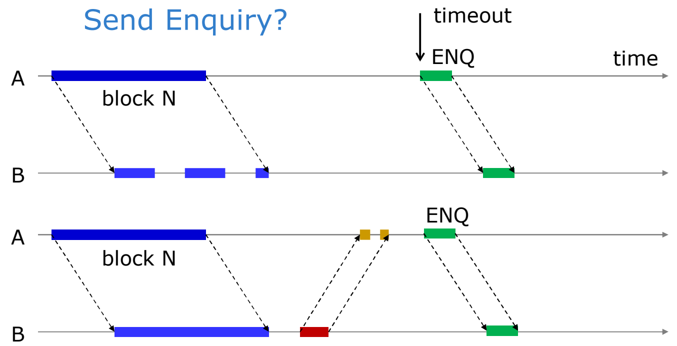
- Simplest way to solve the problem
  - sender sends block of data, wrapped in frame
  - then stops and waits for reply
  - receiver sends ACK or NAK, as appropriate
  - sender sends next block or re-sends last block
- What could go wrong?

Add Timeouts



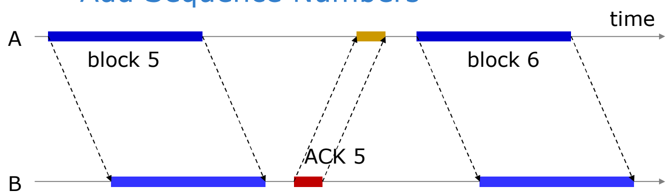
- Set maximum waiting time
  - if no reply from receiver, send data block again
  - (if time limit short, no need to send NAK…)
- What could go wrong?

Send Enquiry?



- Enquiry frame: repeat your last ACK/NAK
  - e.g. ASCII ENQ = 5
- What will receiver send back?
  - what will sender do then?

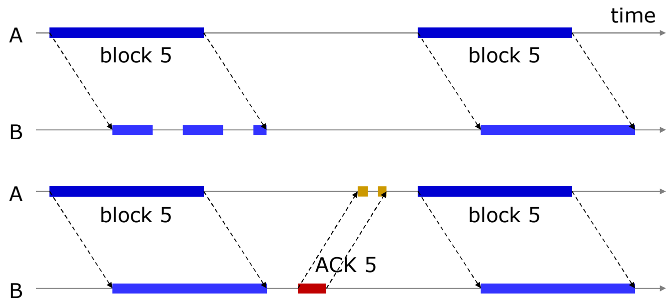
Add Sequence Numbers



- In data frames and acknowledgments
  - sequence number identifies block of data
  - increments as each new **block** sent
    - not for each frame
  - no ambiguity about which block ack'ed
  - receiver can reject duplicate blocks
- Works with enquiry or re-sending

33

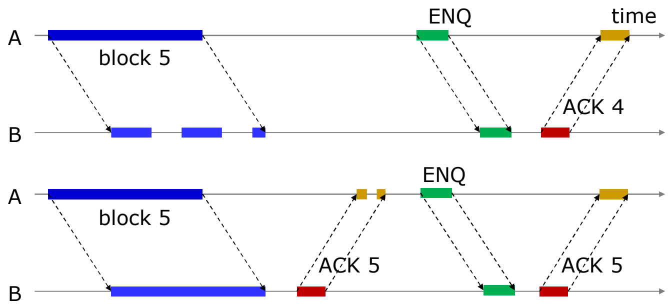
Example with Re-transmission



- What happens next?
  - how does receiver respond?
  - what will sender do then?

34

Example with Enquiry



- Finite number of bits, so numbers repeat
  - e.g. 4-bit seq. no. is modulo 16
  - what range of seq. no. do we need?

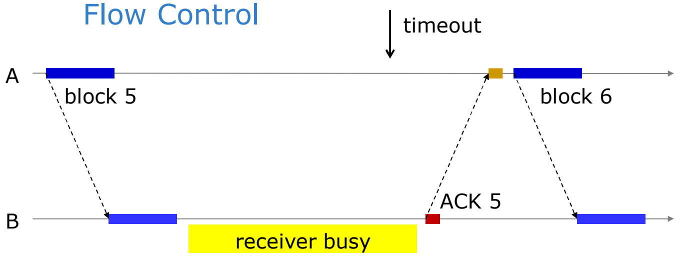
35

Sequence Numbers

- Relate to block of data, not to frame
  - re-transmission has same number as original
- What sequence number does NAK carry?
  - sequence number from bad frame received?
  - why not?
- Often seq. number of last good block received
  - or sequence number of next block expected
    - always one after last good block received
- Some protocols use same no. for ACK & NAK
  - both carry sequence number of last good block
  - or both use next block expected by receiver
  - then no need to distinguish ACK from NAK...

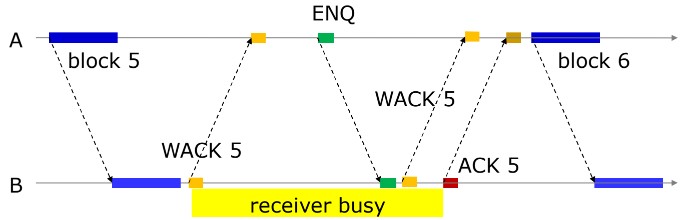
36

Flow Control



- What if receiver cannot keep up?
  - need to get sender to pause…
- Option 1 - delay sending ACK
  - now means "last OK and ready for next"
  - but if delay too long, sender will timeout
    - waste time on re-sending last data block

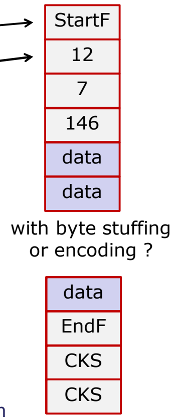
Flow Control



- Option 2 – third type of acknowledgement
  - e.g. WACK = wait acknowledge
  - means "last OK but not ready for next"
  - how to resume?
    - sender ENQ or receiver ACK?
  - other variations on this idea…

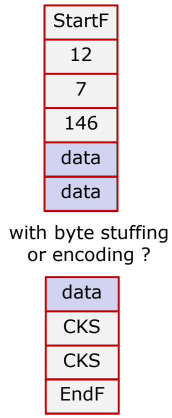
Example Frame Structure – Byte Oriented

- Header – fixed length
  - start of frame marker
  - maybe address, dest. & source
    - needed if multiple devices on link
    - dest. address first, can ignore unwanted
  - sequence number - 1 byte?
  - maybe control info – type of frame
    - data frame, ENQ, ACK, WACK, NAK…
  - maybe byte count
    - data bytes or total frame bytes?
    - helps with end-of-frame detection
    - allows RX to ignore marker bytes in data
  - maybe header checksum
    - need to detect errors in header
    - could include in frame checksum calculation



Example Frame Structure…

- Trailer – usually fixed length
  - marker byte for end of data?
  - or mark end of frame (as shown)
    - deduce end of data
    - either way, need protection against marker occurring in data, header or checksum…
  - or rely on byte count in header…
  - checksum (typically 1 or 2 bytes)
    - modulo 256 or modulo 65536
- End marker protection?
  - header: all bytes controlled by design
    - limit range to exclude end marker, stuff
    - or receiver ignores markers in first *N* bytes
  - data: count, byte stuff or encode
  - checksum: limit range, stuff, mark before



Example Response Frame…

- Responses – ACK, NAK, etc.
- Often just an empty frame
  - full header and trailer, but no data
  - type byte in header identifies function
  - must have sequence number
  - must have checksum – detect errors
- Alternative, for simple stop & wait
  - no type byte needed in header
    - all frames from A to B are data frames
    - all frames from B to A are responses
  - response is frame with 1 data byte (or none)
    - this byte indicates type of response, if required
  - normal header and trailer as above

StartF
24
7
146
CKS
CKS
EndF

41

Recall: Link Layer Protocol Categories

- Byte-oriented protocols
  - assume all data in groups of 8 bits
    - could be text characters or arbitrary data
  - control information also in bytes
  - simpler to implement, but not as flexible
    - developed earlier, when most data was text
- Bit-oriented protocols
  - data may be any number of bits
    - not necessarily groups of 8…
    - but often multiples of 8 bits – computer processors…
  - control information can also be any length
  - protocol works at bit level
    - different techniques for identifying frames, error detection, etc.

42

Frame Marker: Special Bit Sequence

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

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

- Example:
  - define flag as sequence of exactly 6 1s
  - to get this, must transmit 01111110
  - use flag to mark start and end of frame
  - sometimes shared between two frames
- Frame can have any number of bits
  - no need to be multiple of 8 (but often is)
  - finding flag also gives byte synchronisation...
- Use bit stuffing on contents of frame
  - prevent flag from occurring when not intended

43

Bit Stuffing

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

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

- At transmitter, inside frame (between flags)
  - whenever 5 consecutive 1s sent
  - stuff a 0 into the stream
- At receiver
  - see 5 1s followed by 0 remove the 0
  - see 6 1s followed by 0 FLAG
  - see 7 or more 1s abandon frame
- Exercise on received frame above
  - write down the content bits of the frame
    - interpret as bytes, LSB first

44

Error Detection: Cyclic Redundancy Check – CRC

- Add check bits to end of block of data bits
  - usually 16 or 32 check bits
  - on block of up to several thousand bits
- View entire sequence of bits as binary number
  - most significant bit sent first, check bits last
- Rule for check bits:
  - when entire sequence is *divided* by a special number, remainder must be zero
  - division uses modulo-2 arithmetic
    - 0 + 0 = 0, 1 + 0 = 1, 1 + 1 = 0 (no carry)
    - same for subtraction, same as exclusive-OR
  - like long division, but modulo-2 subtraction

45

CRC Example

- 5-bit divisor
- so 4-bit remainder

divisor, 1 at start & end

```
                  1 1 0 0 0 1 1 1 0 0 1 0 0 1
1 0 0 1 1 | 1 1 0 1 0 0 1 1 1 0 1 1 0 0 0 0 0 0
            1 0 0 1 1
              1 0 0 1 0
              1 0 0 1 1
                0 0 0 1 1
                0 0 0 0 0
                  0 0 1 1 1
                  0 0 0 0 0
                    0 1 1 1 1
                    0 0 0 0 0
                      1 1 1 1 0
                      1 0 0 1 1
                        1 1 0 1 1
                        1 0 0 1 1
                          1 0 0 0 1
                          1 0 0 1 1
                            0 0 1 0 0
```

"divides" if leftmost bits match

4 check bits, 0 at first

14 data bits, in blue

46

CRC Example

discard the quotient

```
                  1 1 0 0 0 1 1 1 0 0 1 0 0 1
1 0 0 1 1 | 1 1 0 1 0 0 1 1 1 0 1 1 0 0 0 0 0 0
            1 0 0 1 1

                            0 0 1 0 0
                            0 0 0 0 0
                              0 1 0 0 0
                              0 0 0 0 0
                                1 0 0 0 0
                                1 0 0 1 1
                                  0 0 1 1 0
                                  0 0 0 0 0
                                    0 1 1 0 0
                                    0 0 0 0 0
                                      1 1 0 0 0
                                      1 0 0 1 1
                                        1 0 1 1
```

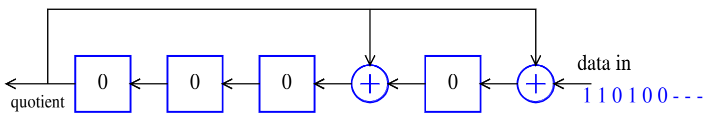
replace check bits with remainder

- transmit remainder as check bits
- at receiver, use same divisor to check
- remainder should be 0000

4-bit remainder

47

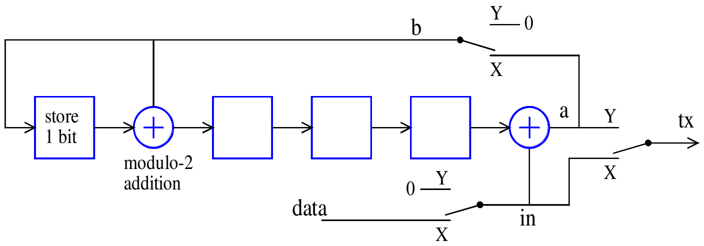
Division in Hardware



- storage element (flip-flop) holds one bit
  - all set to 0 initially
- modulo-2 adder is exclusive-OR gate (XOR)
  - positions match 1 bits in divisor (10011)
- data moves in from right to left
  - followed by 0s in check bit positions
- quotient available bit by bit on left
- after last bit, storage holds remainder

48

CRC Implemented in Hardware



- modified so check bits ready when needed
  - switches at X to send data, Y for check bits
  - calculates check bits during data transmission
  - ready to send after last data bit…
- at receiver, no switches
  - just check remainder = 0000 at end

49

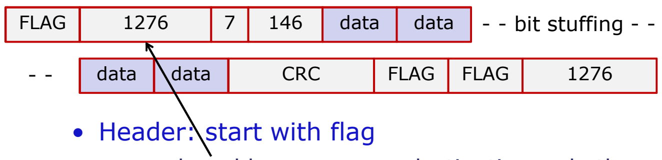
How Reliable is CRC?

- choose divisor carefully – *Generator Polynomial*
- 10011 is $G(x) = x^4 + x + 1$ where $x = 2$ here

- Analysis
  - let transmitted bits be number T
  - received bits $T + E$, where E is error pattern
  - divide by G to check, know $\frac{T}{G} = 0$, so $\frac{E}{G} = ?$
  - will detect any single error
  - can choose G to detect 2 errors in large block
    - e.g. 1100 0000 0000 0001 OK up to 32 768 bits
  - can choose G to detect any odd no. errors
  - will detect any burst of errors up to length of G
  - example: Ethernet and WiFi use 32 check bits
    - G = 1 0000 0100 1100 0001 0001 1101 1011 0111
    - blocks up to 12 000 bits

50

Example Frame Structure – Bit Oriented



- Header: start with flag
  - maybe address: source, destination or both
  - sequence number, other control information
- Trailer
  - CRC (typ. 16 or 32 bits)
  - flag (maybe share with start of next frame)
- Acknowledgement
  - flag, ACK or NAK signal (or other)
  - sequence number
  - CRC, flag

51