UCD School of Electrical, Electronic and Communications Engineering

EEEN20060 Communication Systems

Link Layer Protocol Design

Author: Fergal Lonergan **Student Number:** 13456938

Working with: Conor Burke

Collaborating with: We used our protocol on two computers.

Introduction

The aim of this assignment was to write a link-layer protocol, based on the examples seen in the lectures, in order to transfer a data file from one computer to another using a cable connected into both serial ports.

These files, images and text files, are stored on the hard drives of the computer in a particular sequence. They needed to be transferred in this exact sequence in order for the receiving computer to receive the correct data file and save it to its' hard-drive. For this we needed to design a protocol that not only recognised errors in data blocks, or frames, but to correct these errors in order for the receiver to obtain the correct text or image.

We were provided with application software for this assignment which dealt with the user interface and with the opening and closing of files etc. It relied on link layer functions that we had to design or manipulate in order for the data blocks to be sent and received in the correct order between the connected computers.

Our physical layer for this assignment was a short 5m cable between both ends of the serial ports. We were provided with functions that opened and closed the ports at both ends as well as functions that both send and receive bytes using these ports. Assuming the data travels at 2/3 the speed of light down through the cable we were able to calculate our ideal frame size and bitrate.

Link Layer Protocol Design

Link Layer

A link layers role in a communication system is to ensure a reliable link and transfer of data between the sender and the receiver. It organises bits into bytes and controls how they are transmitted along the physical layer. Usually it contains a form of error detection and it deals with these errors accordingly.

Stop and wait

We based our protocol on stop and wait. This is basically where the receiver and sender wait for an acknowledgement whether good or bad before proceeding with sending the data. It requires only one computer to send data along the channel at a time otherwise data gets corrupted

Protocol Details

Our protocol depended on our receiver always returning the last good data block. We decided on this methodology as we believed it would be the easiest to implement whilst ensuring a reliable protocol. It relies on the sender sending what it believes to be the next wanted block of data from the receiver and the receiver returning the last good data block it receives. All variations of this i.e. if there are errors in acknowledgements etc. dealt with below. We decided to make our acknowledgement frames look similar to our data frames in order to utilise those functions instead of writing new ones which we believed would be tedious and unnecessary. We also decided to do a checksum that protected both our data and sequence number in order to ensure that the data was transmitted correctly.

The sender begins by sending the first data block, it then waits to see if there is an acknowledgement from the receiver that it received a good data block i.e. block 0, if not it times out and resends block 0.
Once block zero has been received ‘correctly’, ie our checksums match however this can sometimes be fooled see later in report, the receiver returns an acknowledgement of block 0 to the sender who then proceeds to send block 1.
If the receiver’s acknowledgement is somehow damaged then the sender resends the previous data block when the timeout is reached. In our protocol we allowed for this with the receiving end checking the sequence number of the incoming data frame and reacting accordingly. i.e.
1. If the frame is the correct frame and good…..store frame and send positive acknowledgement
2. If the frame is the correct frame and bad…..discard and send negative acknowledgement
3. If the frame is the wrong frame and good…..discard and send negative acknowledgement

On the sending end:
1. If we receive a positive acknowledgement we increment the sequence number and send the next frame and wait for a response or until timeout.
2. If a sequence number is returned that is not expected we resend the frame we sent previously.
3. If we receive a negative acknowledgement we resend the frame.
We continue to do this until all frames have been sent ‘correctly’.

We used 2 sequence numbers, 0 and 1, as we only ever looked for the last good frame or the frame we were now attempting to send/receive. By this methodology we only needed the two sequence numbers and any more would be rendered redundant.

For our time limit we decided to wait 5secs on the receiving end as we thought that this would be plenty of time for a block of 100 bytes to be sent and received. It also has 6 attempts to do this so we believed that 30 seconds would be more than enough.

As the sender only needs to receive an acknowledgement that contains very few bytes we reduced its’ wait time to 1sec.

Frame Structure

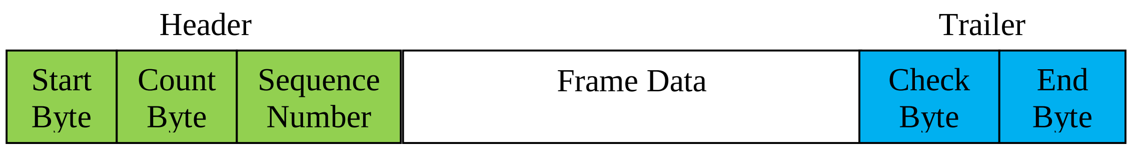
Our frame structure started with a startbyte of 206 and ended with an endbyte of 204. For this reason we made our modulo for our checksum 200 so that it was impossible for the computer to interpret a data byte as a startbyte or endbyte.

The startbyte beigns the header.
The second element in our frame is the number of data bytes in our frame so that the receiver can check that against what it receives. This will count up the total amount of bytes in the frame which can then be used as part of our checksum in order to ensure that the bytes are transmitted correctly.
This is followed by our sequence number allows both computers to know which data block is needed or being sent. Our sequence number is protected by our checksum as is our databytes.

Our databytes fill out the middle of our frame.

The second last value in our frame is our checkbyte. This is used by the receiving end for error detection.
The final byte in our trailer is the endbyte.

Our acknowledgement frames are the exact same however seeing as we need no databytes we do not include them as they would just waste time. We also do not need a modulo as the sequence number will always be either 1 or 0 or in previous versions 0-15 so a modulo of 200 is irrelevant. Also seeing as there are now only 5 bytes in our frame we do not need to calculate nBytes we can just enter 5.



Acknowledgement Frame

```
// First build the frame
    ackFrame[0] = STARTBYTE;
    ackFrame[1] = 5;        //send 5 bytes of data to make our build
frame to be compatible with our send ack frame.
    ackFrame[2] = seq;      //sequence number
    ackFrame[3] = seq+5;      //no modulo needed as sequence number
between 0 and 15. In checkframe this value will equal 0+seq. i.e. seq
    ackFrame[4] = ENDBYTE;
```

…

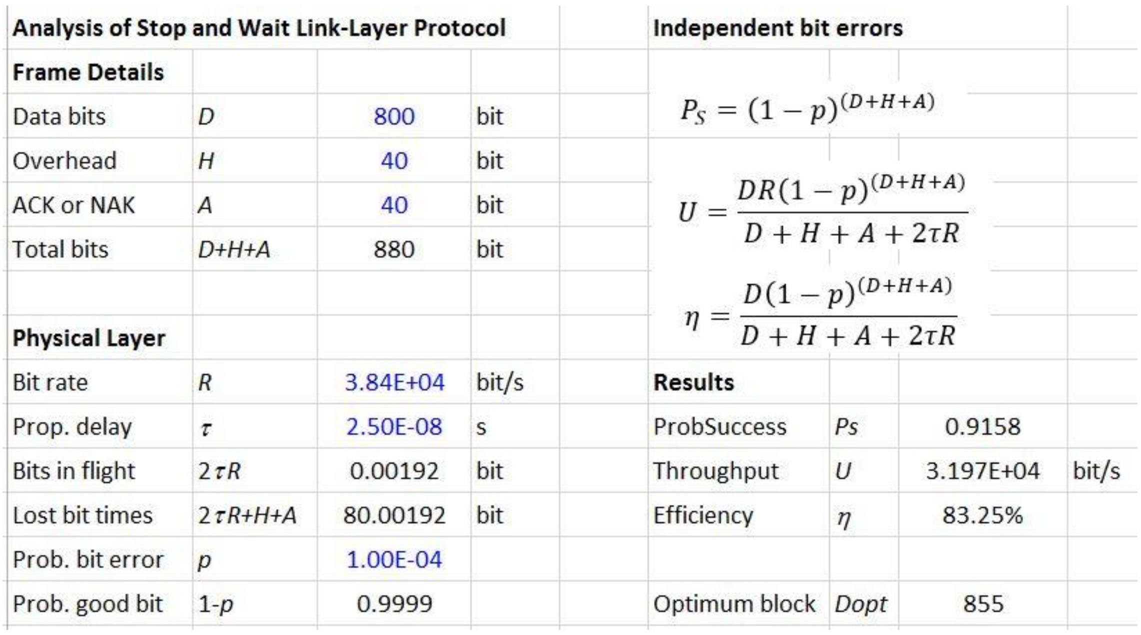
🟩 Start Byte	Data = 5	Sequence Number	Sequence Number **+ 5**	🟦 End Byte

Diagrams from Conor Burke

Frame Size

We had about 5 m of cable connecting the two computers, and assuming a signal propagation speed along our serial cable of 2×10^8 m/s, we calculated a propagation delay of 25ns. Our probability error was 10^{-4}. We then ran our numbers through the formulas discussed in class using the spreadsheet provided. We used a bit rate of 38400 bit/s and fund that our optimum block size is 855. Our probability of success then would equal to 0.9185. We used 100 bytes as our block size in the terminal outputs. Our optimum bytes would be 106 so we ran it for the second group of terminal outputs.

Our data frame is not limited to anything. It could take any value however we limited it ourselves to 100. Obviously increasing the size reduces the probability of success drastically, see screenshot below, however technically there is not an upper limit.



Analysis of Stop and Wait Link-Layer Protocol

Frame Details			
Data bits	*D*	800	bit
Overhead	*H*	40	bit
ACK or NAK	*A*	40	bit
Total bits	*D+H+A*	880	bit
Physical Layer			
Bit rate	*R*	3.84E+04	bit/s
Prop. delay	τ	2.50E-08	s
Bits in flight	$2\tau R$	0.00192	bit
Lost bit times	$2\tau R+H+A$	80.00192	bit
Prob. bit error	*p*	1.00E-04	
Prob. good bit	1-*p*	0.9999	

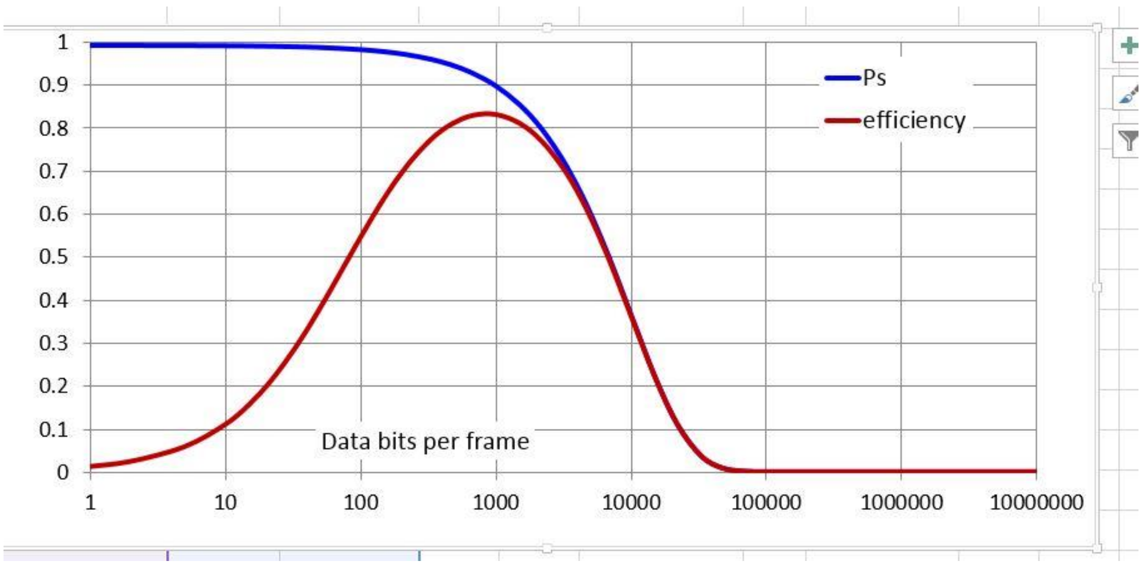
Independent bit errors

$$P_S = (1-p)^{(D+H+A)}$$

$$U = \frac{DR(1-p)^{(D+H+A)}}{D+H+A+2\tau R}$$

$$\eta = \frac{D(1-p)^{(D+H+A)}}{D+H+A+2\tau R}$$

Results			
ProbSuccess	*Ps*	0.9158	
Throughput	*U*	3.197E+04	bit/s
Efficiency	η	83.25%	
Optimum block	*Dopt*	855	



Protocol Implementation

Implementation

We were given functions to manipulate. These were LL_receive, LL_send, buildDataFrame, checkFrame, getFrame, processFrame and sendAck. All other parts of the program were given to us ie the physical layer functions and the application layer functions.

*LL_* The purpose of this function is to make a connection with the other computer through the serial ports.

LL_discon(): This function closes the connection made between the computers.

LL_connect

This functions checks the connection between both computers through the serial ports before sending any data. We #defined our BITRATE above this function.

LL_diconn

This function stops the connection between the computers.

LL_send

This function calls on the buildDataFrame, next and sendAck functions in order to complete its task. This function sends the sends our frame to the receiver and waits for either an acknowledgement or timeout. It also keeps track of the frame to which it is sending in order to know which frame is in line to be sent. It calls the next function to increment the sequence numbers and send next frame. It also keeps track of positive and negative acknowledgements both sent and received. It runs through different scenarios depending on whether the frame is good or bad and whether it is the expected sequence number. As long as the attempts counter doesn't exceed 6 it will continue to attempt to transmit each frame until the entire file is transmitted.

LL_receive

We needed to reduce the amount of time the receiver waited for the frame. We also had to design how the receiver would react to each scenario of whether the correct acknowledgement/frame was received or whether it was a wrong sequence number or bad frame. This function calls on the other functions checkFrame, processFrame and sendAck. These are explained in comments in code for each scenario. It also uses the lastSeqRx to increment the last good frame received.

buildDataFrame

This function builds the frame into an array of databytes. It has a checksum error detection built in which will be used by the receiver to check that the correct data has been transmitted. This is returned in nData.

getFrame

This function locates a frame within the stream of bytes along the physical layer by targeting the start and end markers. Once it sees the start marker it collects a stream of bytes until the end marker is also recognised or it is timed out.

checkFrame

This function checks the data of the frame received and compares it against the expected data. It checks for the start and end markers and then checks the data between them. If it returns the expected value the function returns 1 else it will return 0 and print an error message. If the bytes in the frame are not the same it prints an error with a location for the user to know where the error has occurred.

```
if (check != frameRx[nFrame-2])
         {
              printf("CFLL: Error! Received bytes not equal to sent 
bytes");
              return 0;
         }
```

processFrame

This function processes the received data bytes into an array called dataRx.

sendAck

We built our send_Ack like a data frame so that we could use the frame functions already defined. We also have two counters that count the negative and positive acknowledgements sent by the receiver.

next

This function increments the sequence numbers of the frames.

timeSet

Sets a time limit for each function.

timeUp

Checks to see if time limit has been exceeded.

printFrame

This is used if there is an error. It prints the first and last ten bytes of the frame.

Explain what these functions do, and why – probably function by function? Give details of the parts that you wrote and the changes that you made. You may include short sections of code to illustrate the descriptions – there is no need to include everything here, as the full linklayer.c file will be printed and attached. If you do include code, use a fixed-pitch font, as in the fragment below, to preserve indentation. Try to avoid having lines wrap at the right margin...

Linklayer2.h File

We changed some constants her to optimise our program. These were:

```
#define MOD_SEQNUM 2   // modulo for sequence numbers
#define TX_WAIT 1.0    // sender waiting time in seconds
#define RX_WAIT 5.0    // receiver wait time in seconds
```

We reduced the amount of sequence numbers as we thought the extra sequence numbers were unnecessary as we only need to know frame being sent and last good frame i.e. 2 different sequence numbers.
We also reduced the times significantly after recognising that our propagation delay was so short especially considering our cable was only 5m long. The times still give ample time for the receiver to receive all the data from the sender and vice versa.

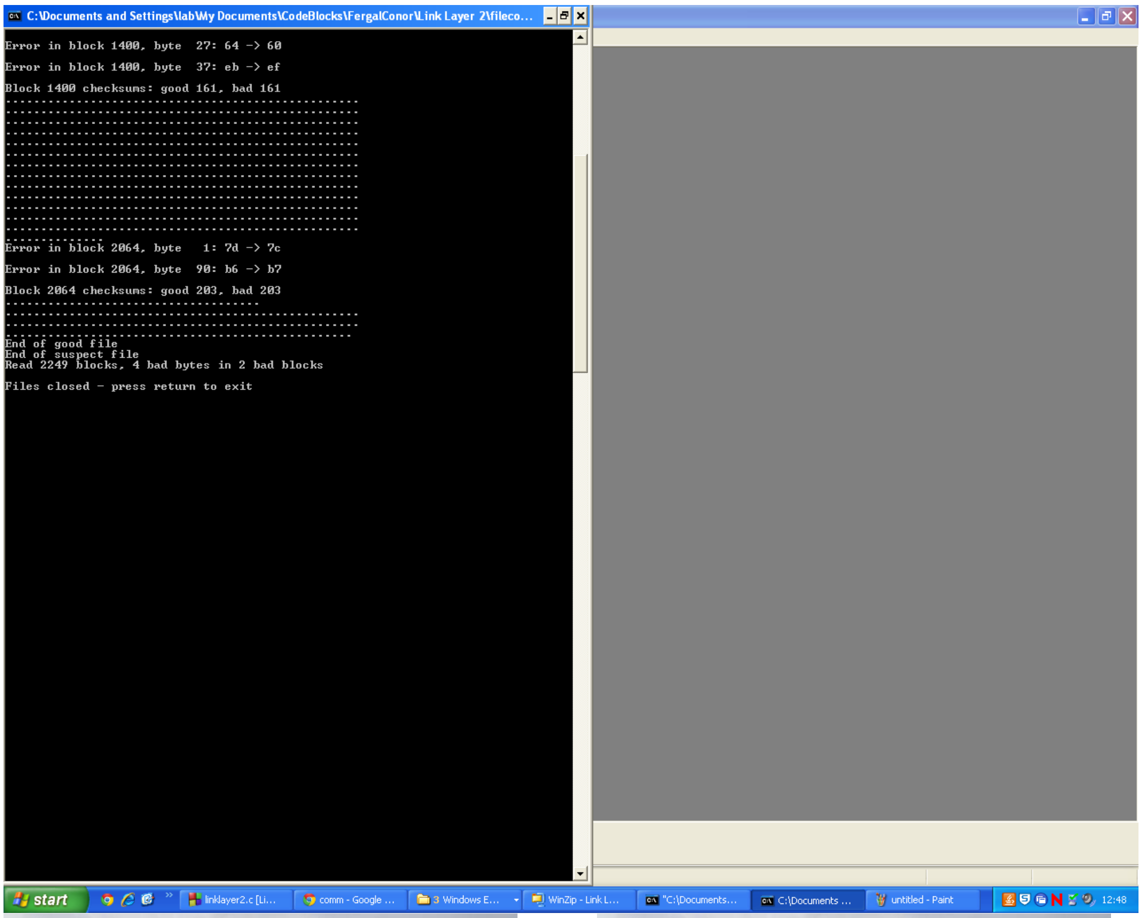
Testing

For testing we sent sample.txt, sample2.txt, led2s.jpg, crest.jpg and boat.jpg. All files were transferred correctly apart from boat.jpg which when sent we noticed a problem with our checksum protocol. We noticed that if one byte is above its value and another byte is below its value to an extent that this cancels out in the checksum so that they add up to the correct extent this can fool our checksum in believing that the data block has been transferred correctly. The probability of this happening are extremely minute. We calculated it to be around $8.9x10^{-6}$. We realised that if we split our checksum into a group of checksums, and summed each individual before summing the total number this would increase the probability of detecting an error. We also thought about multiplying each byte by a unique increasing prime number. Another solution would have been to introduce some byte stuffing into the program. We didn't implement these seeing as the probability was so minute.

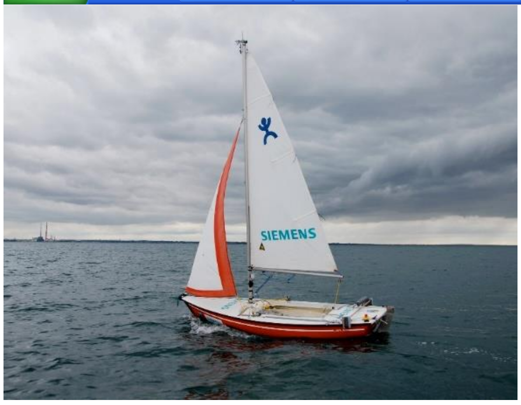
The execution time for our boat image to send was 128 s with about 120 of those being the time taken to send the data. The throughput is then 14666.67bit/s (220kb/120s). Our physical layer bitrate was 38400. Therefore our throughput is about half the speed of our bitrate which would make sense due to the fact that our program has to run our checksum and wait for acknowledgements etc. our block size was 100 bytes and our probability was 10^{-4}.

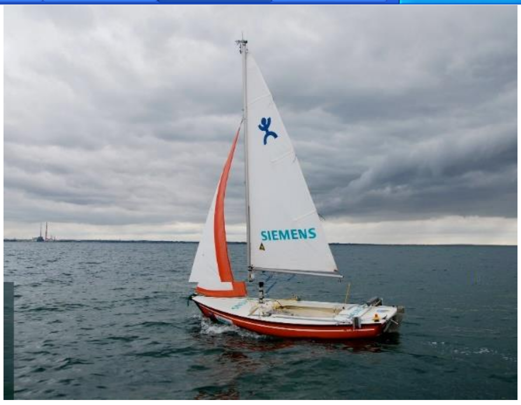
We also tried sending crest.jpg and disconnecting the cable before reconnecting to see how our protocol would react. The image was transmitted correctly.

In our programme if a data frame is resent when it shouldn't be this is seen as a positive ack on the sending side and a negative ack on the receiving side.



```
Error in block 1400, byte  27: 64 -> 60

Error in block 1400, byte  37: eb -> ef

Block 1400 checksums: good 161, bad 161
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..............
Error in block 2064, byte   1: 7d -> 7c

Error in block 2064, byte  90: b6 -> b7

Block 2064 checksums: good 203, bad 203
....................................
..................................................
..................................................
.................................................
End of good file
End of suspect file
Read 2249 blocks, 4 bad bytes in 2 bad blocks

Files closed - press return to exit
```





Sending end

LL: Sent frame 106 bytes, block 8, attempt 1

CDFLL Sum is 12   Check is 12LL: Response received, seq 8
LL: Sent frame 106 bytes, block 8, attempt 2

LL: Sent frame 106 bytes, block 9, attempt 1

Receiving end

```
PHY_get:  ####  Simulated error...  ####
LL: Got frame, 106 bytes, attempt 1

CDFLL    Sum is 9482   Check is 82CFLL: Error! Received bytes not equal to sent
bytesLL: Bad frame received
206 106   3 234  32 111 112 101 114  97 :  ╬j♥Ω opera
- - -
114 101 115 101 110 116 115  32 154 204 :  resents Ü╠
LL: Got frame, 106 bytes, attempt 2
```

As we can see in the image above when boat was transmitted, the bytes in two blocks cancel each other in the checksum, so the program thinks that it has transmitted correctly whereas we can see it hasn't. Dark pixels and the bottom of the photo has also shifted to the right.

```
Crest.jpg broken cable transmission

Link Layer Assignment - Application Program

Select debug or quiet mode: d

Select send or receive: s

Enter name of file to send with extension (name.ext): crest.jpg


Send: Opening crest.jpg for input
Send: Connecting...
LL: Connected
Send: Sending file name 10 bytes...
LL: Sent frame 16 bytes, block 0, attempt 1
Send: Read 100 bytes, sending...
LL: Sent frame 106 bytes, block 1, attempt 1
Send: Read 100 bytes, sending...
LL: Sent frame 106 bytes, block 2, attempt 1
Send: Read 100 bytes, sending...
........
.......
Send: Read 100 bytes, sending...
LL: Sent frame 106 bytes, block 7, attempt 1
LLGF: Timeout A with 0 bytes received
LL: Timeout waiting for response
LL: Sent frame 106 bytes, block 7, attempt 2
LLGF: Timeout A with 0 bytes received
LL: Timeout waiting for response
LL: Sent frame 106 bytes, block 7, attempt 3
LLGF: Timeout A with 0 bytes received
LL: Timeout waiting for response
```

```
LL: Sent frame 106 bytes, block 7, attempt 4
LLGF: Timeout A with 0 bytes received
LL: Timeout waiting for response
LL: Sent frame 106 bytes, block 7, attempt 5
.....
.....
LL: Sent frame 96 bytes, block 4, attempt 1
Send: End of input file after 5190 bytes
LL: Sent frame 6 bytes, block 5, attempt 1
Send: Disconnecting...
LL: Disconnected.  Sent 61 data frames
LL: Received 57 good and 0 bad frames, had 4 timeouts
LL: Sent 0 ACKs, 0 NAKs
LL: Received 54 ACKs, 3 NAKs

File sent!

Process returned 0 (0x0)   execution time : 47.781 s
        Press any key to continue.
```

Black above is where cable was disconnected before being reconnected.

Conclusion

Our link-layer protocol for the most part works as desired with the exception that sometimes, even if it is very rare, there is a possibility that our checksum may be fooled. Our program is perfect when sending text files and small image files but we recognised that there is a slight chance that there will be problems if we send large image files i.e. boat.jpg. We do know of different methods in which to solve our problem, outlined above, however we didn’t see the need for these considering the time constraints of the lab and as it left more topics for discussion in our reports.