UCD School of Electrical and Electronic Engineering

EEEN20060 Communication Systems

Link Layer Protocol Design

Final Report

Name: Ruairi Hogan **Student Number: 21432816**

Name: Alannah Mehigan **Student Number: 21343546**

Lab Session: Thursday 9:00 / 15:00 / Friday **Brightspace Group: 52**

By submitting this report through Brightspace, we certify that ALL of the following are true:

1. We have read the *UCD Plagiarism Policy* (available on Brightspace). We understand the definition of plagiarism and the consequences of plagiarism.
2. We recognise that any work that has been plagiarised (in whole or in part) may be subject to penalties, as outlined in the document above.
3. We have not plagiarised any part of this report. The work described was done by the team, and this report is all our own original work, except where otherwise acknowledged in the report.

Introduction

The aim of this assignment was to design and implement a link-layer protocol to allow reliable transfer of data from one computer to another, using the serial ports.

Application-layer software was provided for this assignment, which allowed us to test our link layer protocol by transferring files. A file is stored as a collection of bytes, in a particular order. So, the aim was to be able to transfer all those bytes, in the correct sequence, to another computer, where they could be stored in another file.

The application-layer software handled interaction with the user, opening files, etc. It used our link layer functions to transfer blocks of data bytes reliably to another computer, where another copy of the program was running.

The physical layer consisted of a cable connecting the serial ports on two computers.

The serial port hardware handles bit synchronisation and byte synchronisation, effectively providing a byte transfer service. A set of software functions was provided to open and close the serial ports and to send and receive bytes using these ports. The receive function also added simulated bit errors, to allow us to test the reliability of our link-layer protocol.

In the coded link layer files, a protocol was set up. Functions were added to the code for error detection using a checksum, adding stuff bytes to problematic bytes and finally implementing a stop and wait protocol.

The checksum was added as a check byte at the end of the frame before the stop byte. This was to ensure that on sending and receiving the data, the receiver can detect if the value of the data frame had changed in the process of sending it. The check byte put in the frame would equal the modulo (a set number – 253) when added with the sum of the data bytes.

To avoid error in transferring files that include the byte of value 210, 212 or 204. We included a stuff byte system that would put a 210 before a byte that had these values (the stuff byte, start byte and end byte respectively). On receipt of this data, the receiving end would then remove the stuff bytes and leave the original data that could then be processed.

To avoid the link layer sending a bad data frame and then sending another and causing a problem, a stop and wait protocol was added. The stop and wait protocol would send an ACK (acknowledgment) from the receiving end to the sending end upon receipt of a data frame. The ACK would inform the sender whether the frame just received was good or bad and the sender would respond accordingly. This dealt with the possibility of the sender sending a duplicate frame again, the sender sending a good frame with a wrong sequence number, or the sender sending a bad frame that didn't pass the error detection.

Link Layer Protocol Design

The role of the link layer is to connect the different layers of the data streams. For detection errors in frames, for detecting the incoming and sending frames and processing them. It makes sure that connections can communicate reliably.

Logical Link Control

We used a Stop and Wait protocol. The basic idea here is that the sender will send a frame to the receiver, then wait for a response, called an acknowledgement, and then choose what frame to send next depending on the response from the receiver.

The protocol first sends the frame in the physical layer from the sending end to the receiving end. The receiving end then gets the frame and processes it by checking if it's the correct sequence number and correct checksum and has a valid start and end byte. Then the receiving sends a response called an acknowledgment (ACK) to the sending end. The ACK that our receiving end sends back is a four bit frame made up of a start byte, sequence number, checksum and end byte. The ACK is positive or negative depending on the sequence number of the ACK frame. If the frame received is good is sends the sequence number of the next frame. If the frame received is bad, it sends the sequence number of the same frame again. It always sends the sequence number of the frame it wants to receive.

If the receiving end gets a duplicate data block or a block out of sequence, it takes both as a bad frame and sends the same sequence number again in the ACK frame. If there is no response from the receiver, the sender waits 1 seconds and then gets the error of a timeout, it does this up to five times and then exits the program. If the receiver gets no response, it does the same thing but waits 4 seconds between each timeout. We used sequence numbers of a range 0-15. We found this to be a good range so that we can see clearly the blocks when testing, if it was smaller it might not be as clear, but it's not too wide a range so were not dealing with unnecessarily large numbers.

We left the time limit for the receiving end to be 6 seconds but changed the sending end to 1 second. Through trial and error we found that the maximum throughput occurred when the sending time limit was 1 second. This is the time the sender waits for the ACK. We also chose this because the cables were only about 6m, and the propagation delay was small, so we only needed about 1 second to get a reply. If, for some reason, the receiver took longer to send the ACK, we still had 5 attempts at the sending end to receive one before it cuts out of the program, so this was enough time.

Error Detection

The error detection code that we chose was a checksum complement. This code would sum all the data bytes of the payload and the sequence number. It would then get the remainder of this number when divided by 253 and then subtract this number from 253.

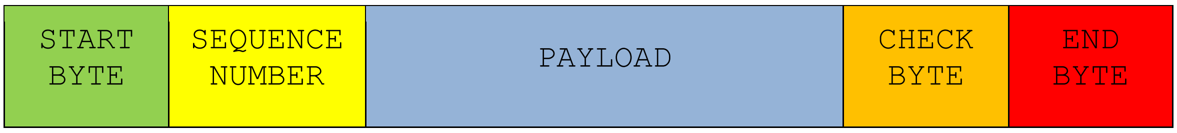
Then, on the receiving end, the code adds all the bytes and gets the received checksum from the second last index of the frame, adds these numbers and gets the remainder division of 253 to make sure that it's 0. If this sum was a perfect multiple of 253, then the frame was transferred correctly.

We could have used a parity byte to detect errors in the transfer of these bytes but we decided to choose a checksum complement because if the parity was an odd parity for example, if there were two errors in one byte where one bit goes from 1 to 0 and another does the opposite, then the parity will still think that there was the same amount of 1's in the byte. On the other hand, a checksum complement cannot detect an error if there are two errors in one data frame and there are both on the same place of significance and they also both change from opposite states (1 to 0 and the other 0 to 1). We found that this has a lot lower chance of missing an error in the frame so we decided to go for a checksum.

Frame Structure

…

<u>Data Frame Structure</u>



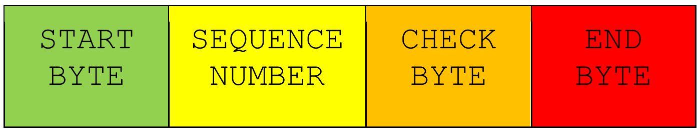
The diagram above shows what the structure of the data frame is. When this frame is sent, on the receiving end we have to implement some protocol so that the receiver can recognise the frame even if some of the bytes have the same value as marker bytes.

To solve this, we implemented a byte stuffing system. This would put a byte of value 210 in front of any byte in the frame that had the same value as a marker byte, but shouldn't (start byte, end byte and stuff byte).

When the program is building the data frame to be sent, it will iterate through the data and add a 210 in front of any data that has the same value as the marker byte. Then, on the receiving end the program will take out any 210 values, apart from when there is a 210 after a 210 as this would mean the second byte was a data byte.

We could have used a frame size byte that we put in the header of the frame that would contain the number of bytes that are in the frame and would therefore indicate to the receiver when to go onto the next frame. We didn't use this system because if the frame size byte developed an error then the receiver would not know how big the frame is and therefore start looking for a start byte too early or too late. We could have also used data encoding, which is when you encode the data in the frame onto a smaller frame and send that as well to determine the marker bytes in the frame. In the end we decided to use byte stuffing because it was easier to implement than data encoding and has efficacy.

<u>Acknowledgement Frame Structure</u>



This is diagram shows the structure of the ACK (acknowledgement frames). For these frames we didn't include a payload because the sequence number of the frame is the byte that gives the response positively or negatively. The sequence number of the ACK frame would tell the sender the sequence number of the data block that it wanted next.

Frame Size

Our data frame has 2 bytes in the header and trailer. The maximum number of data bytes in a frame is limited to 222 because our optimal data frame is 108 bytes, so if we add 4 bytes (marker bytes) we have a frame of 112 bytes, then 110 of those bytes are able to get a stuff byte put in front of them, but each can only get one stuff byte so we add 110 and get the maximum size a block can be as 222 bytes.

Acknowledgements are sent as short frames of 4 bytes.

Optimum Frame Size

We assumed that we would have about 6 m of cable connecting the two computers, with a signal propagation speed of 2×10^8 m/s. This would give a propagation delay of 30 ns.

Because $6(1/2x10^8) = 30$ ns

We assumed a probability of bit error of $8x10^{-5}$.

We tested at a physical layer bit rate of 38400 bit/s. As each byte is sent with a start bit and a stop bit, that corresponds to 3840 byte/s and an effective bit rate of ??? data bit/s.

These are found by (38400 bit/s) /10 = 3840 bytes/s

And 38400/1.25 = 30720 bit/s

Using these numbers, we found the optimum number of data bits per frame 863.

This is 108 bytes, and this is the size of the data block we actually used.

Protocol Implementation

Implementation

We were given functions to build the data frame, to send the data frame to receive the data frame, to process the data frame, to check for errors, to send acknowledgments and to check if bytes were marker bytes.

We had to modify these to suit our protocol design, this is how:

- **buildDataFrame():** This function was given to us to build the data frame. We had to modify it to add a checksum to our frame and to add stuff bytes to data that had the same value as the marker bytes. For the checksum we first initialised an integer variable called checksum to equal 0, we then added the value of the sequence number byte using this code:
  ```
  checksum += frameTX[SEQNUMPOS];
  ```

  Then, in the for loop that copies the data bytes into the frame, we added a line that also added the value to the checksum variable:
  ```
  checksum += dataTX[i]; // sum each data byte
  ```

  We also implemented the stuff byte system in this function. The code we wrote would iterate through the dataTX[] array and copy the elements to the frameTX[] array. In this for loop, we added an if statement that would check if each element had the same value as a marker byte, using a function called special() which we also coded (next function). If this particular byte was the same value, we would put a stuff byte in that index, and then in the next index put the data byte. This was the code:
  ```
  // Copy the data bytes into the frame, starting after the header
      for (i = 0; i < nDataTX; i++)      // step through the data array
      {
        checksum += dataTX[i]; // sum each data byte
        if(special(dataTX[i])==1)// if the byte in the frame is the same
  as the endbyte then we need to put a stuff byte before it
        {
  ```

```
          frameTX[nByte] = STUFFBYTE; // the index of the number of
bytes in the array gets the value of the stuff byte
          nByte++;
      }
      frameTX[nByte] = dataTX[i]; // the next index gets the value from
the data array

      nByte++;
    }
```

- **special()**: this was a function given to us that we had to code. It was to check if the integer given to it was the same value as the start byte, end byte or stuff byte (212, 204, 210 respectively. This was done by a simple If statement to determine if they were the same or not and then return TRUE or FALSE (1 or 0) correspondingly. This is the code:
  int special(byte_t b)

```
{
  if(b == ENDBYTE || b == STARTBYTE || b == STUFFBYTE) // if the byte in
the frame is the same as the endbyte then we need to put a stuff byte
before it
    {
      return TRUE; // the byte is a special byte
    }
  else{
      return FALSE;   // The byte isn't a special byte
  }

}
```

- **checkframe():** This was a function given to us to check the frame for errors, on the receiving end. On this frame, we had to write a code that would find the total sum of all the values in the payload and the sequence number, retrieve the checksum sent in the trailer of the frame, add them together and check if it was a perfect multiple of 253 (our modulo number). We did this by using a while loop with the condition that if the number of bytes went greater than the index where the checksum was it would cut out. In the while loop we added the bytes to an integer variable called total. Then we added the total to the checksum byte, that was in the second last index of the frame, and checked if it was a multiple of 253 using this code:

```
total += checksumRX;
if (total % MODULO != 0)
```

- **getFrame():** The getFrame function was given to us. This function got the frame from the physical layer and processed it. The code we had to write was to take the stuff bytes out of the frame before checking it with the checkFrame() function. To take these stuff bytes out of the frame, we used a while loop. This loop iterated through the frame and when it found a stuff byte, it would decrement the amount of bytes received so that on the next iteration the next byte would overwrite the stuff byte. This while loop also determined if there was an endbyte that wasn't stuffed and if it found it then it would exit the loop. This is the code:

```
do
    {
        bytesGot = PHY_get((frameRX + bytesRX), 1);  // get one byte at
a time
        if (bytesGot < 0) return bytesGot;  // check for problem and
give up
        else bytesRX += bytesGot;  // otherwise update the bytes
received count

        /* if statement to take out stuff bytes */
     // if the byte is an end byte and stuff flag is false
        if (frameRX[bytesRX-1] == ENDBYTE && stuff_flag == FALSE){
          marker_flag = TRUE; // set marker flag true
        }

        else if (stuff_flag == TRUE) // if stuff flag is true
        {
          stuff_flag = FALSE; // set it to false
        }

        else if (frameRX[bytesRX-1] == STUFFBYTE){ // stuff byte has
been found
          stuff_flag = TRUE; // set stuff flag to true
          bytesRX--;
// decrement number of bytes so next byte overwrites the stuff byte
        }

     }
    while (marker_flag == 0 && !timeUp(timerRX) && bytesRX < maxSize);
```

- **LL_send_LLC:** This was a function given to us to send the frame to the physical layer. It also waits to receive an ACK and then decides which frame to send next subsequently. The code we had to write here was fill in an if else statement that decided what to do when we got a negative ACK or positive ACK. We set the condition of the if statement to be that if the sequence number of the ACK received was the same as the next sequence number, it was a positive ACK, but if it was the same as the block just sent, it was a negative ACK. This was the if condition:
```
if ( seqAck == next(seqNumTX))
```

- **LL_receive_LLC:** This function was used to receive the frame from the physical layer and decide which ACK to send back to the sender. The code we had to write were to fill in if statements and decide which ACK to send. These were the two lines of code that we used depending on the if statement given to us (e.g. if the sequence number was the same as the last one or if the frame failed the checksum, etc.).
```
sendAck(NEGACK, expected); // send negative ACK to sender
sendAck(POSACK, next(expected)); // send positive ACK to sender
```

- **sendAck:** The sendAck function was given to us to build an Ack frame and send the Ack frame to the sending end. The code we had to write for it was building the frame. We had to put a start byte, and sequence number (which was given to us when calling the function in LL_receive_LLC), a checksum and an end byte into the frame. The size of the frame was 4 bytes and we used the same code as the buildDataFrame() function to find the checksum. However this time, the checksum was only on the sequence number in the Ack frame as there was no payload. Here is the code for adding the bytes to the Ack Frame.
```
// First build the frame
    ackFrame[0] = STARTBYTE;
    ackFrame[SEQNUMPOS] = seqNum;

// Adding the checksum to the frame
    ackFrame[HEADERSIZE] = checksum;

    // Adding the end byte
    ackFrame[HEADERSIZE+1] = ENDBYTE;

    // Getting the size of the Ackframe
    sizeAck = HEADERSIZE + TRAILERSIZE;
```

Testing

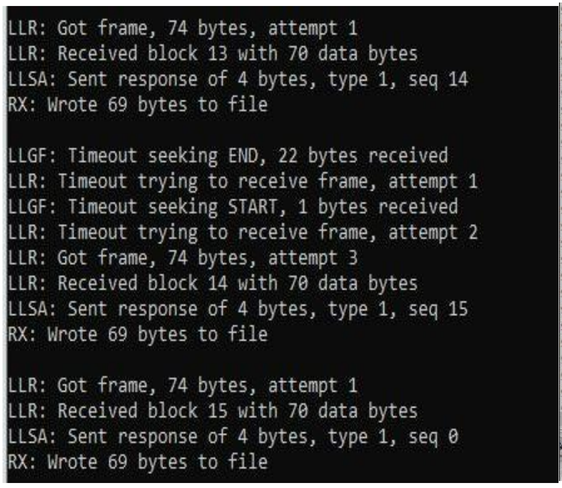
The throughput of our program was calculated using the time taken to send the boat.jpg file. We calculated it with 2 different values of optimal block sizes of 107 and 70 which resulted in two throughput values of 2555.5 bytes/s 2082.26 bytes/s respectively.

To ensure that the software was working as intended we conducted multiple different tests.

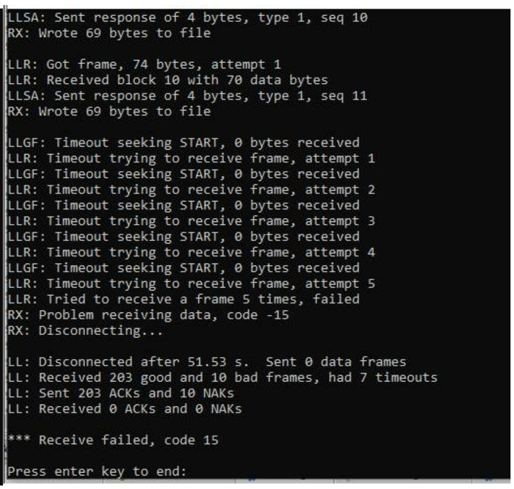
We began with running the code in debug mode so that if any problems arose, we could see exactly where they occurred and why.

First of all we checked the timeouts on the program. We did this two different ways, firstly by pausing one of the programs and letting the other one begin to timeout and then unpausing it to check that the other one continued to run after beginning to timeout. The second way we tested it was by disconnecting the cables and then reconnecting them after a few seconds. Both ways provided the same result, shown below (fig.1).

We then let the program completely timeout to make it disconnect and give up trying as also shown below (fig.2).



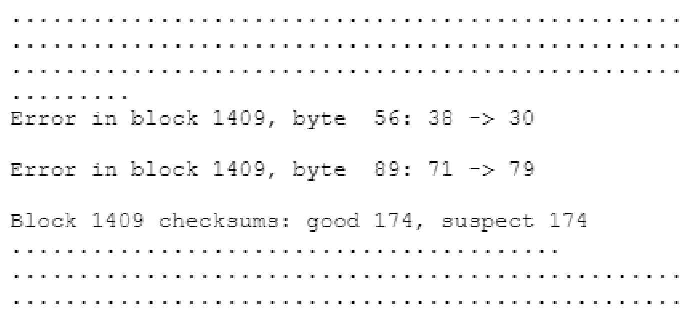
```
LLR: Got frame, 74 bytes, attempt 1
LLR: Received block 13 with 70 data bytes
LLSA: Sent response of 4 bytes, type 1, seq 14
RX: Wrote 69 bytes to file

LLGF: Timeout seeking END, 22 bytes received
LLR: Timeout trying to receive frame, attempt 1
LLGF: Timeout seeking START, 1 bytes received
LLR: Timeout trying to receive frame, attempt 2
LLR: Got frame, 74 bytes, attempt 3
LLR: Received block 14 with 70 data bytes
LLSA: Sent response of 4 bytes, type 1, seq 15
RX: Wrote 69 bytes to file

LLR: Got frame, 74 bytes, attempt 1
LLR: Received block 15 with 70 data bytes
LLSA: Sent response of 4 bytes, type 1, seq 0
RX: Wrote 69 bytes to file
```



```
LLSA: Sent response of 4 bytes, type 1, seq 10
RX: Wrote 69 bytes to file

LLR: Got frame, 74 bytes, attempt 1
LLR: Received block 10 with 70 data bytes
LLSA: Sent response of 4 bytes, type 1, seq 11
RX: Wrote 69 bytes to file

LLGF: Timeout seeking START, 0 bytes received
LLR: Timeout trying to receive frame, attempt 1
LLGF: Timeout seeking START, 0 bytes received
LLR: Timeout trying to receive frame, attempt 2
LLGF: Timeout seeking START, 0 bytes received
LLR: Timeout trying to receive frame, attempt 3
LLGF: Timeout seeking START, 0 bytes received
LLR: Timeout trying to receive frame, attempt 4
LLGF: Timeout seeking START, 0 bytes received
LLR: Timeout trying to receive frame, attempt 5
LLR: Tried to receive a frame 5 times, failed
RX: Problem receiving data, code -15
RX: Disconnecting...

LL: Disconnected after 51.53 s.  Sent 0 data frames
LL: Received 203 good and 10 bad frames, had 7 timeouts
LL: Sent 203 ACKs and 10 NAKs
LL: Received 0 ACKs and 0 NAKs

*** Receive failed, code 15

Press enter key to end:
```

Fig.1 Fig.2

One of the problems we encountered during the testing was an issue where an error would occur in the same block. It was increasing the value of one byte by 8 and decreasing another by 8 so the checksum was correct but it caused an error in the printed image. We discovered this problem using the file comparison program and realised it was just an error that couldn't be resolved within our program. Below is the printed error from the file comparison program.



```
............................................
............................................
............................................
.........
Error in block 1409, byte  56: 38 -> 30

Error in block 1409, byte  89: 71 -> 79

Block 1409 checksums: good 174, suspect 174
....................................
............................................
............................................
```

Conclusion

We have succeeded in designing and implementing a link-layer protocol. It has some limitations, such as the problem we encountered during the testing where two simulated errors were created in the same block but didn’t result in a checksum calculation error so you couldn’t debug it from the code. However, it meets the requirements of the assignment, and all other testing carried out was successful and every file evaluated was reprinted correctly.