Homework 2

Tejas Kamtam

305749402

Question 1 - PPP Byte Stuffing

First, to make visualizing the bytes easier, we can convert the binary representations to hex:

```
Flag (F)    = 0b01111110 = 0x7e
Escape (E) = 0b01111101 = 0x7d
Mask (M)    = 0b00100000 = 0x20

XOR(F,M)    = 0b01011110 = 0x5e # when F occurs replace w/: 0x7d5e
XOR(E,M)    = 0b01011101 = 0x5d # when E occurs replace w/: 0x7d5d
```

Q1 - Resulting Frame

Given sender datagram:

```
source_datagram = 0b01111101_01111101_01111110_01111110 = 0x7d_7d_7e_7e
```

PPP Byte stuffing results in the modified frame:

```
PP_frame_struct = <F>_<E^M>_<E^M>_<F^M>_<F^M>_<F>
PPP_frame = 0x7e_7d5d_7d5d_7d5e_7d5e_7e
```

```
=
0b01111110_01111101_01011101_01111101_01011101_01111101_01011110_01111101_01011110_01111110
```

Q2 - Byte vs Bit Stuffing

Byte stuffing seems to be easier for software than bit stuffing because byte stuffing allows us to work with hex and mod 2 arithmetic at word/byte level. Most memory and data in software are stored as hex, and trying to manipulate a single bit within a byte can be expensive compared to running operations on hex.

Q3 - Overhead

Considering only the flag byte and not possible escape bytes, HDLC has a $1/32$ chance of the flag appearing in any 5-bit pattern in the datagram. On the other hand, PPP has a $1/256$ chance of the flag appearing in any 1-byte pattern in the datagram. This is roughly a 0.4% overhead for bit stuffing, given uniformly random bytes in the datagram.

Q4 - Worst Case

In HDLC, we only bit stuff 0s after every sequence of five 1s. Because the flag is six 1s contained in 2 0s, there are no escape characters as false flags cannot occur with stuffing every 5-bits. So for every 5-bit sequence of 1s, we stuff at most 1 bits. This is an overhead of $1/5 = 20\%$. This holds for any sequence of groups of five 1s, e.g., 10 1s requires 2 stuffed 0s which is also a 20% overhead. This is the worst case → when all 5-bit groups in the datagram are 1s.

In PPP, we escape false flags with byte stuffing. This effectively doubles the number of bits for each occurrence of the false flag (the same is true for every occurrence of a false escape). In the worst case, all bytes are either flags or escapes, in which case we double the overall length of the datagram by escaping. This results in a worst case overhead of 100%.

Question 2 - Error Recovery

Q1 - STATUS Packets

The `STATUS` packet is required to verify that the receiver has received all packets. Suppose there is no `STATUS` packet sent. In the given example, suppose the receiver receives packet 1, loses packet 2, and loses packet 3. The receiver doesn't know that the sequence has ended without a `STATUS` packet, so does not send a `NACK`. This means the receiver assumes only 1 packet was intended and the sender assumes the receiver has received all 3 packets because no `NACK` was sent preemptively.
Thus, a `STATUS` packet is required to signify the end of a sequence of packets $\longleftrightarrow$ to tell the sender how many packets the receiver has successfully received.

Q2 - Timers

The source of ambiguity in the given error recovery example is how the sender can be informed about how many packets the receiver has successfully received. In the example, lost/erroneous packets in the middle of the sequence are already successfully handled as the receiver can distinguish packet ids and send an immediate `NACK` (with last received packet ID) when a future packet is detected e.g., 1 received, 2 lost, 3 received. The only

issue now, is ensuring the last packet is received. The only way to do so is by sending a `STATUS` packet (due to the argument in Q1). Therefore, implementing a timer will ensure that a `STATUS` packet is sent at the end of the packet sequence which will ultimately notify the sender of how many packets of the sequence the receiver has successfully received (the core issue). The sender can then decide to retransmit packets. So, the timer ensure the `STATUS` is sent, and the `STATUS` ensures all packets have been received.

Q3 - Timer Conditions

The first criteria for the timer is that the timer must expire after the last packet (suppose this is packet no. 3) is sent. In this case there are 2 outcomes, either the receiver has already received the last packet, then it will correctly `NACK 3`, or the packet is in transmit, in which case the receiver will `NACK 2` and the timer must be restarted and the sender must again send `SEQ 3`, wait for the timer to expire, send status, then receive the correct `NACK 3`.
The second criteria is that the timer must be restarted whenever a preemptive `NACK` is received. This indicates a lost packet, so the sender must retransmit all packets from the "Nack-ed" packet. In this situation, the new (restarted) timer must again expire after the last packet in the sequence.

Q4 - Loss Latency

Suppose the round-trip time is given as $RTT = 2 \times OWT$ where OWT is the one-way time and the `STATUS TIMER` is ST. The process for sending a single packet `D` that is first lost is as follows:

1. Start timer
2. Send `D` : $+OWT$
3. Timer ends : $+ST$
4. `STATUS` sent : $+OWT$
5. `NACK 0` received : $+OWT$
6. Start timer
7. Retransmit `D` : $+OWT$
8. Timer ends : $+ST$
9. `STATUS` sent : $+OWT$
10. `NACK D` received : $+OWT$

Now, if we assume that $ST >> OWT$, we get the following total latency for the example:

$$\text{Latency} = ST + OWT + OWT + ST + OWT + OWT = 2ST + 2RTT$$

Question 3 - CRC

Q1 - Debugging

Firstly, there is an error in the `compute_crc` function where the added padding in the line:

```
padded_message = message_bin + '0' * (generator_length)
```

This line adds `generator_length` padding but should be adding only `generator_length-1` padding.
Now the next error is in the binary division function `bin_div`. Specifically, the following line:

```
temp = bin(int(temp, 2) ^ int(divisor, 2))
[2:].zfill(divisor_length)
```

This line uses `zfill` to prepend 0s until the binary value is of length `divisor_length`. This is not what we want to happen as what we want to do is pull down values from the dividend down and append them to the remainder if there are not enough bits to perform binary division again (given there are still bits to the right in the dividend). The next error, also in the `bin_div` function is the following lines:

```
temp = temp[1:]
if len(dividend) > divisor_length:
        temp += dividend[divisor_length]
        dividend = dividend[1:]
```

These lines, firstly disregard the MSB of the remainder, which we want to keep for the next iteration of division (especially so in the case where the remainder is the same length as the divisor). Secondly, this code only brings down a single value to the right of the remainder from the dividend. Instead, we should be bringing down as many values from the dividend as possible until the number of bits in the modified remainder is as large as the divisor (given there are bits to the right in the dividend available to "pull down"). This should make much more sense in the images below.

Q2 - Incorrect CRC

The incorrect CRC printed is `011`.

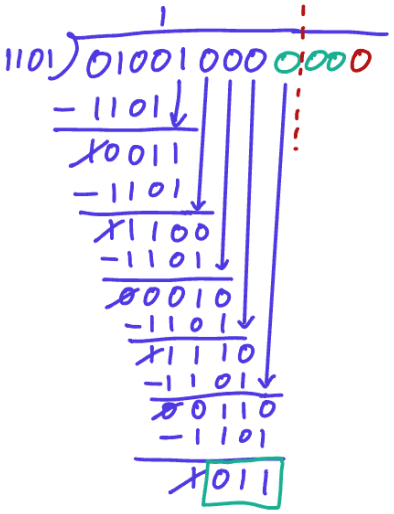
Incorrect Impl:

Sender:

'H' → 01001000 Gen → 1101

Compute_crc ("H", 0b1101):

message_bin = "01001000"
generator_bin = "1101"
padded_msg = "010010000000"
bin_div ("010010000000", "1101"):
temp = "0100"
While len(dividend) ≥ 4:
it0: temp = (0100 ^ 1101)[2:].zfill(4) = 1001
temp = 1001[1:] = 001



Q3 - Correct CRC

The correct `CRC` should be `001` as we show it is validated by the receiver. Note, here we also pad only with the

correct/required number of bits: `len(generator)-1`.

Correct: