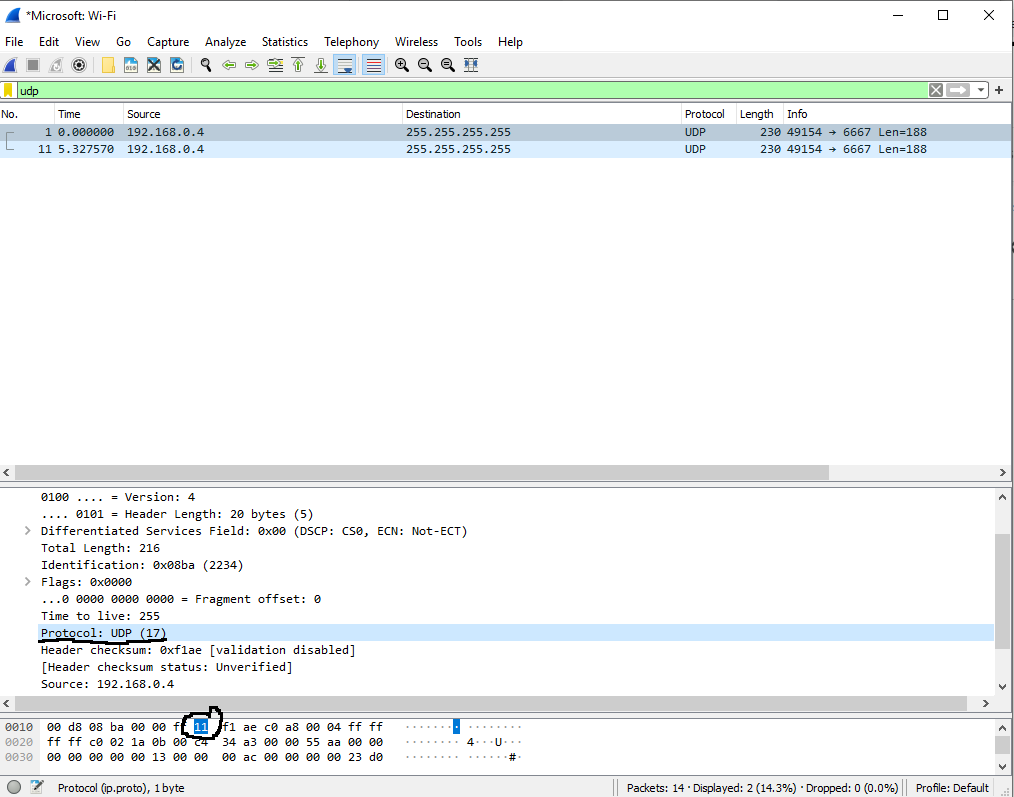
**Assignment 3**

**Question 1: Understanding UDP Protocol**

**#1-A:**

*What is the protocol number for UDP? Give your answer in both hexadecimal and decimal notation?*

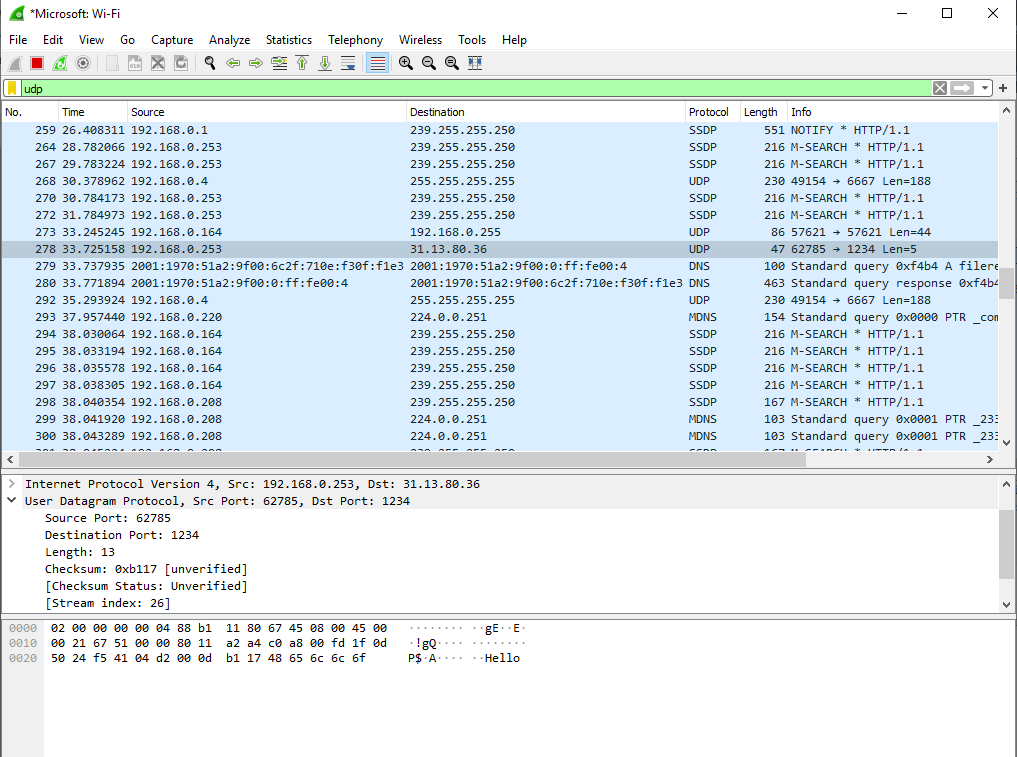
****

As clearly seen in the above screenshot, UDP’s protocol number is 17 as a decimal, and 0x11 in hexadecimal notation. I have highlighted these fields in the wireshark packet contents body above.

**#1-B:**

To calculate UDP checksum, we first must know that in addition to its own header, UDP checksum uses a pseudo header, consisting of the original source IP, destination IP, reserved (0000 0000), protocol (0x11), and the length from the UDP header. This get’s added with the actual UDP header, consisting of a source port, destination port, length, and actual data.

The following screenshot illustrates the UDP packet I used for this question. It has as it’s data the string “Hello.”

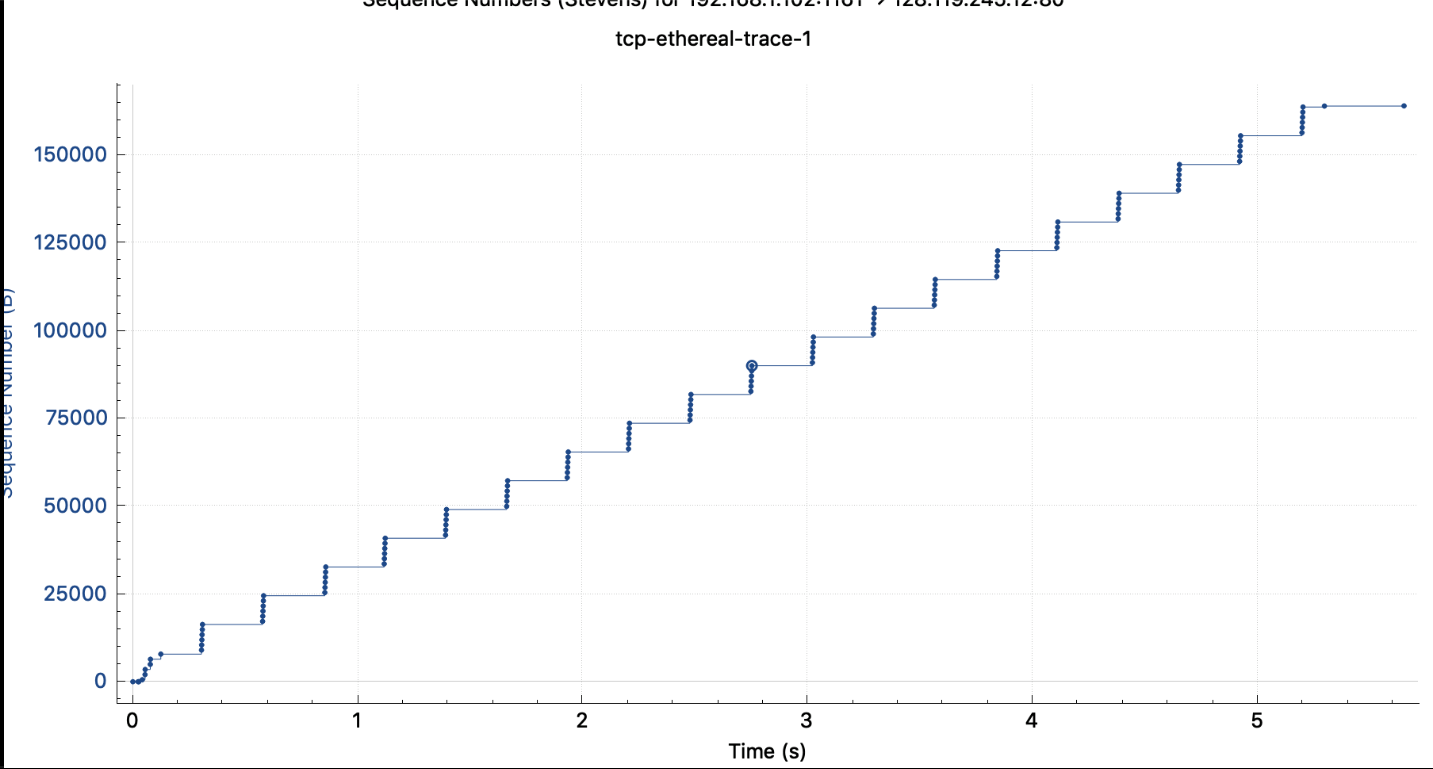


|  |  |  |
| --- | --- | --- |
| **Byte #** | **Data/Content (in hex)** | **Current rolling sum** |
| **PSEUDO HEADER** | **PSEUDO HEADER** | 0 |
| 26-29 (Source IP) | C0A8  00FD | C1A5 |
| 30-33 (Destination IP) | 1F0D  5024 | 130D6 -> 30D7 (after overflow addition) |
| 23 (UDP Protocol) plus reserve | 0011 | 30E8 |
| 38-39 (UDP Length) | 000D | 30F5 |
| **UDP HEADER** | **UDP HEADER** | 30F5 |
| 34-35 (UDP Source port) | F1AA | 1229F -> 22A0 (after overflow addition) |
| 36-37 (UDP Destination port) | 04D2 | 2772 |
| 38-39 (UDP Length) | 000D | 277F |
| 42-46 (UDP Data) | 4865  6c6c  6f00 | 4B51 |
| **TAKE ONE’S COMPLEMENT** |  | **FFFF – 4B51 = B4AE (Calculated Checksum)** |
| 40-41 (Actual Checksum) | 0xB4AE | **CORRECT!!!!** |

Comparing my calculated checksum with the actual wireshark captured checksum, we can verify that my solution was correct.

**Question 2: Understanding TCP Protocol**

**#2-A:**

****

The TCP slowstart phase begtins when the connection is initializesd (When the HTTP POST segment is sent out). Although the identification of the TCP slowstart phase and congestion avoidance phase depends on the value of the congestion window size of the TCP sender, we unfortunately cannot obtain the exact value of the congestion window size directly from the Time Sequence Graph. However, we can estimate the lower bound of this value by the amount of outstanding data, as this represents the amount of data without acknowledgement. Also, we know that the TCP window is constrained by the receiver window size and the receiver buffer can act as the upper bound of the TCP window size. Since the receiver buffer isn’t the bottleneck in this trace, we can use the lower bound of the TCP window size.

However, despite this, we cannot determine the exact end of the slow start phase and the start of the congestion avoidance phase for this trace. The major reason behind this is that the TCP sender is not pushing enough throughput to enter the congestion state. This means that before the end of the slow start phase, the application is already stopping transmission temporally. In the above graph, however, we can see the estimated congestion avoidance phase represented by the flatline, when the sequence number isn’t increasing with regards to time.

In the text, the idealized behaviour of TCP assumes that TCP senders are extremely aggressive in sending data. Since traffic may congest the network layer, TCP senders ideally should follow the AIMD algorithm to detect packet loss, and should drop their sending window size. However, in practice, TCP behaviour also depends greatly on the application. For example, in a web application, some web objects are extremely small, and thus, before the end of the slow start phase, the transmission is already over, thus suffering from the long delays of the slow start phase of TCP.

**#2-B:**

Bytes 46-47 indicate the flags that are set in the TCP header, corresponding to which type of TCP segment is being sent. \*Note: only half of byte 46 is used for the Flags, the other half is part of the Header Length: which is 32 Bytes (8).

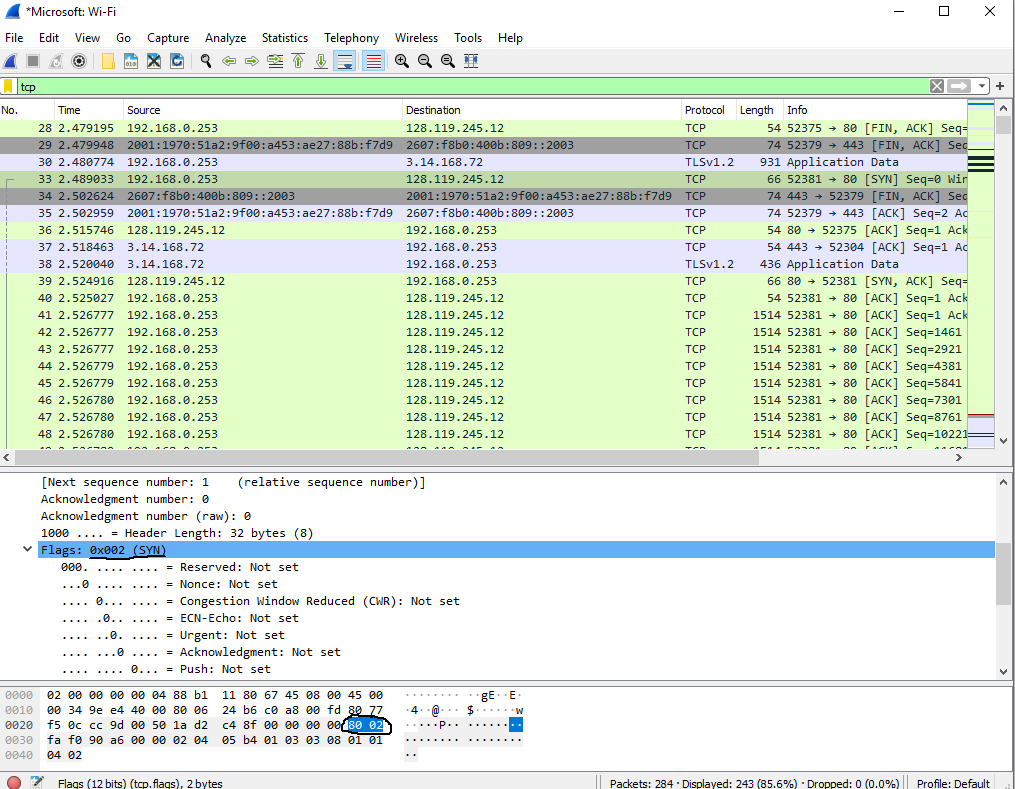
**#2-C:**

First, I will add the table:

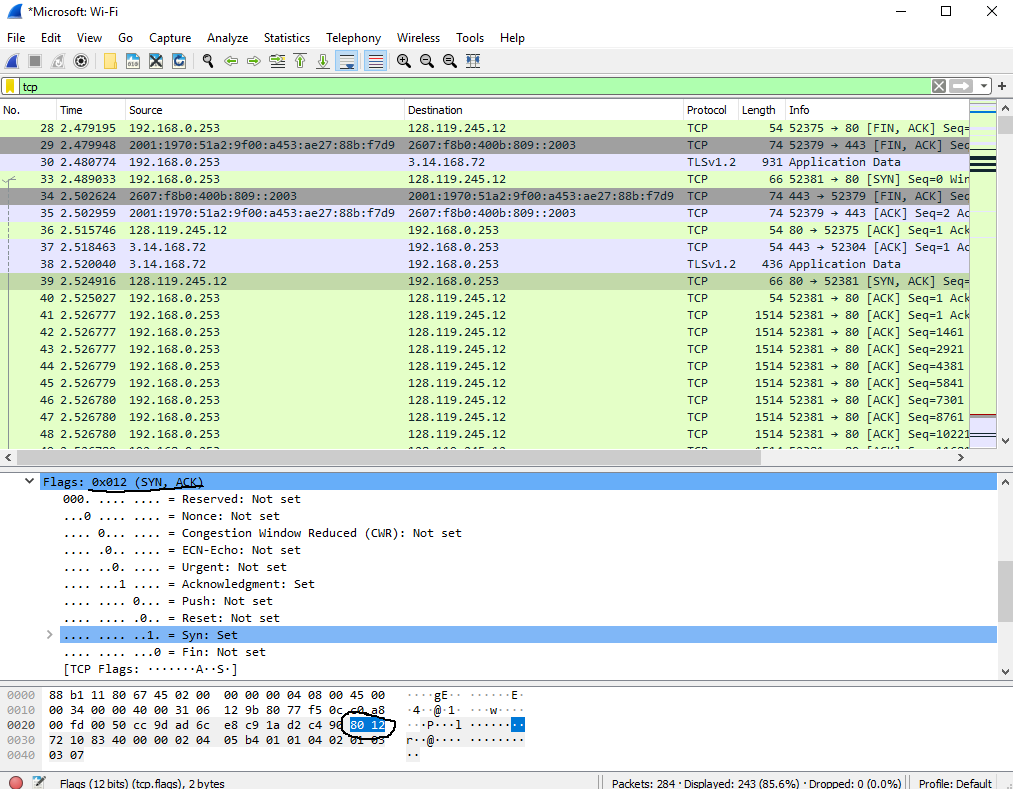
|  |  |  |
| --- | --- | --- |
| **Segment Type** | **Data/Content of Byte #46-47 (Hex)** | **Data/Content of Byte #46-47 (Binary)** |
| **SYN** | 0x002 | 0000 0000 0010 |
| **SYN-ACK** | 0x012 | 0000 0001 0010 |
| **ACK** | 0x010 | 0000 0001 0000 |
| **DATA** | 0x010 | 0000 0001 0000 |
| **FIN-ACK** | 0x011 | 0000 0001 0001 |

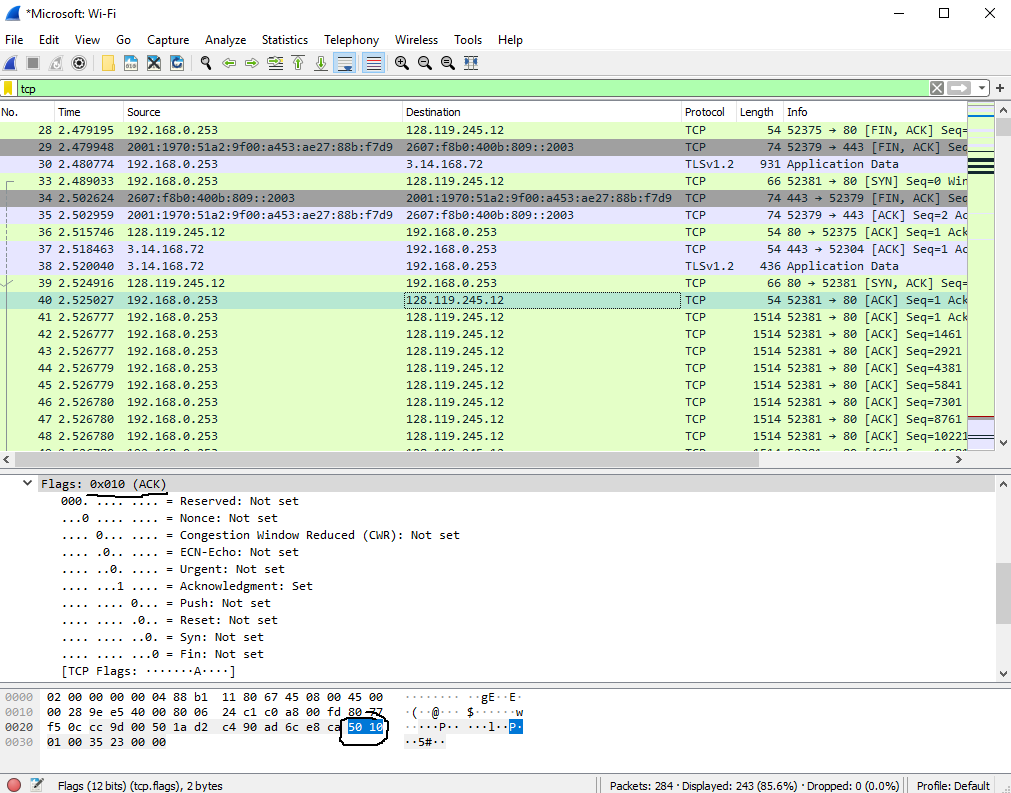
Next, I will add the screenshots:

**SYN:**

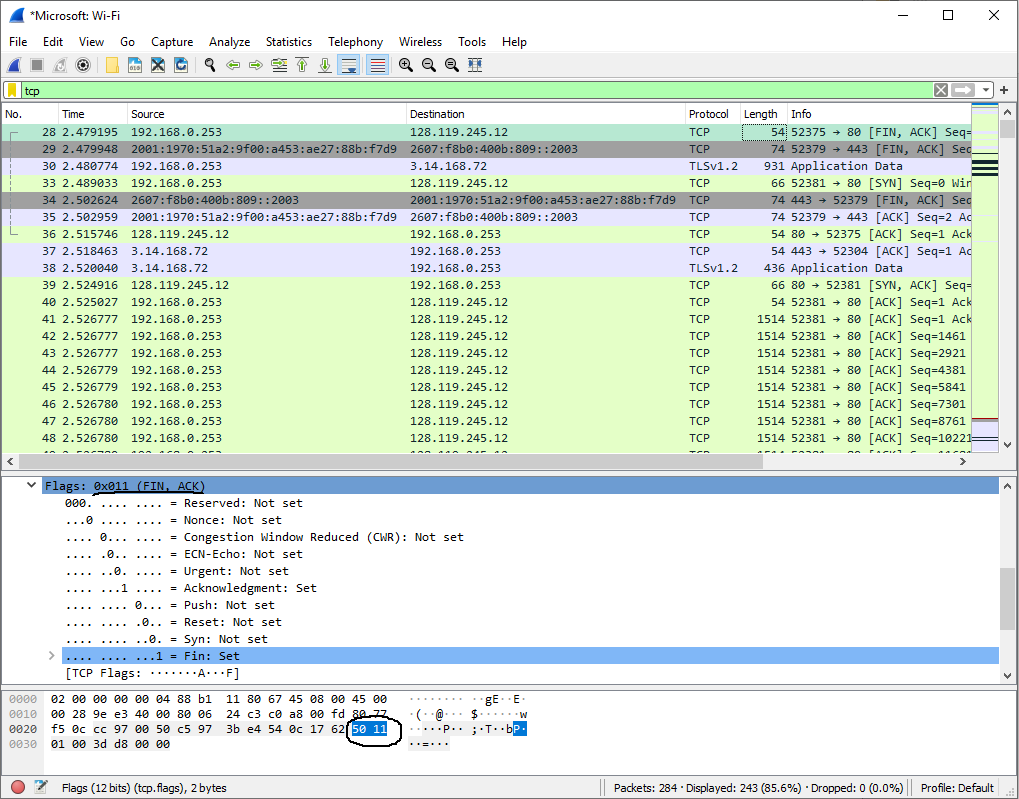
****

**SYN-ACK:**

****

**ACK:**

**FIN-ACK:**

****

**#2-D:**

Similar to UDP, to calculate TCP checksum, we first must know that in addition to its own header, TCP checksum uses a pseudo header, consisting of the original source IP, destination IP, reserved (0000 0000), protocol (0x11), and the length from the TCP header.

|  |  |  |
| --- | --- | --- |
| **Byte #** | **Data/Content (in hex)** | **Current rolling sum** |
| **PSEUDO HEADER** | **PSEUDO HEADER** | 0 |
| 26-29 (Source IP) | C0A8  00FD | C1A5 |
| 30-33 (Destination IP) | 82D3  1035 | 154AD -> 54AE After Overflow addition |
| 23 (TCP Protocol) plus reserve | 0006 | 54B4 |
| 16-17 (TCP Length) | 0029 | 54DD |
| **UDP HEADER** | **UDP HEADER** | 54DD |
| 34-35 (Source Port) | EC59 | 14136 -> 4137 After Overflow Addition |
| 36-37 (Destination Port) | 01BB | 42F2 |
| 38-41 (Sequence Number) | 737D  1FF8 | D667 |
| 42-45 (Acknowledgement Number) | E3B4  7538 | 2F55 |
| 46-47 (Header Length + Flags) | 5010 | 7F65 |
| 48-49 (Window Size Value) | 00FF | 8064 |
| 52-53 (Urgent Pointer) | 0000 | 8064 |
| 54 (Data) | 0000 | 8064 |
| **TAKE ONE’S COMPLEMENT** |  | **FFFF – 8064 = 7FAF Calculated Checksum** |
| **50-51 (Actual Checksum)** |  | **7FAF CORRECT!!!!** |