WES 237A: Introduction to Embedded System Design (Winter 2024)
Lab 4: Network Communication
Due: 2/19/2024 11:59pm

Ricardo Lizarraga, rlizarraga0@gmail.com *619.252.4157*	**PID: A69028483** https://github.com/RiLizarraga/WES237A_Lab4

In order to report and reflect on your WES 237A labs, please complete this Post-Lab report by the end of the weekend by submitting the following 2 parts:

- Upload your lab 4 report composed by a single PDF that includes your in-lab answers to the bolded questions in the Google Doc Lab and your Jupyter Notebook code. You could either scan your written copy, or simply type your answer in this Google Doc. **However, please make sure your responses are readable.**
- Answer two short essay-like questions on your Lab experience.

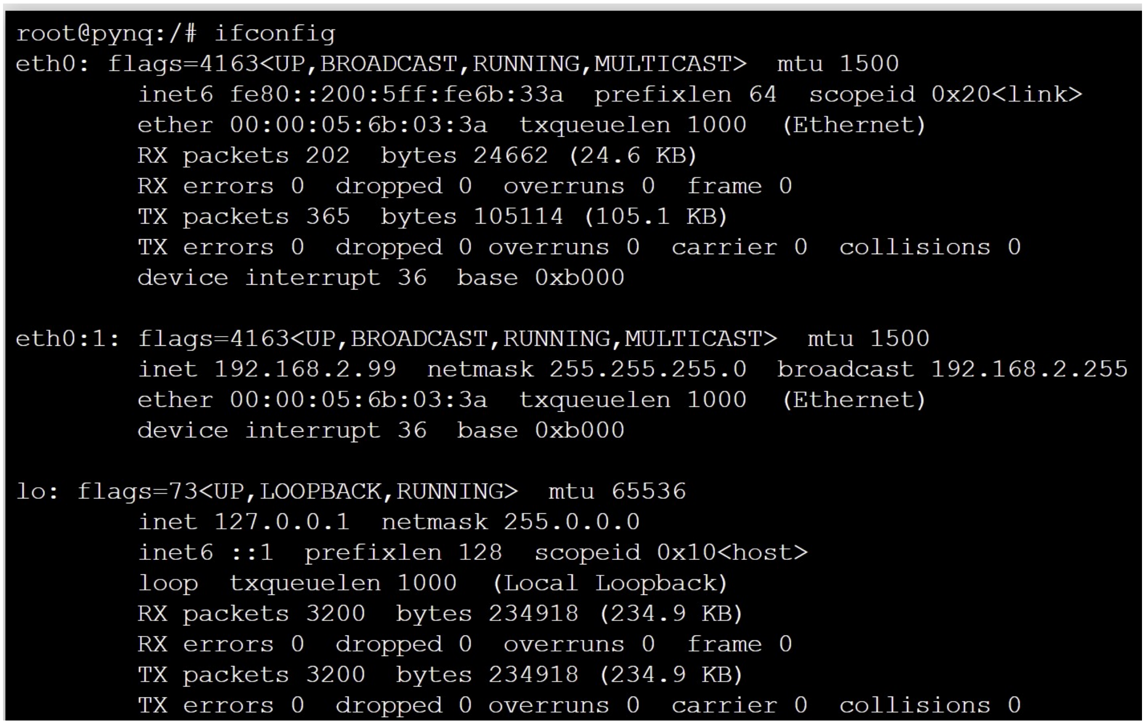
All responses should be submitted to Canvas. Please also be sure to push your code to your git repo as well.

Locating IP Addresses of Devices in your Network

1. Open a serial connection to your PYNQ board (see Lab3 if you forgot)

```
xilinx@pynq:~$
```

2. Connect the PYNQ board to the network switch over ethernet.
3. Run '$ ifconfig'. This is the *Interface Configuration* command and will tell you the different interfaces on your PYNQ board.



```
root@pynq:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::200:5ff:fe6b:33a  prefixlen 64  scopeid 0x20<link>
        ether 00:00:05:6b:03:3a  txqueuelen 1000  (Ethernet)
        RX packets 202  bytes 24662 (24.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 365  bytes 105114 (105.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 36  base 0xb000

eth0:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.99  netmask 255.255.255.0  broadcast 192.168.2.255
        ether 00:00:05:6b:03:3a  txqueuelen 1000  (Ethernet)
        device interrupt 36  base 0xb000

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 3200  bytes 234918 (234.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3200  bytes 234918 (234.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

a. **How many ipv4 addresses are assigned to the board? What is the ipv4 address assigned to the 'eth0' or ethernet interface? What is the netmask of this address?**

i. Note: `eth0: 1` or `usb0` is a virtual interface through the USB cable. This assigns your board an IP address over USB. This is a static IP address so you can always reach your board from this IP address over USB.

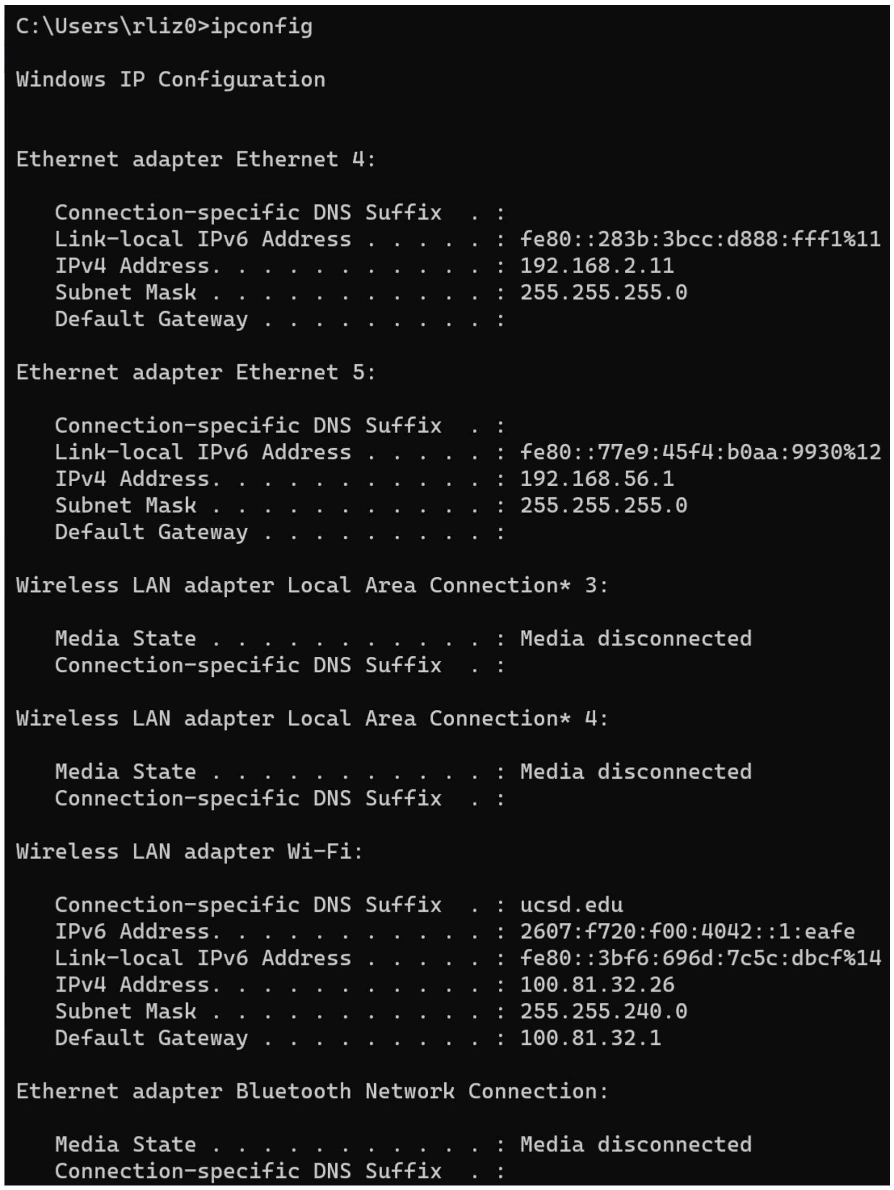
Physical H/W interface	**Ip address & type**
Eth0	inet6 fe80::200:5ff:fe6b:33a
	192.168.2.99 {IP4} **netmask** 255.255.255.0
	127.0.0.1 {local loop}

4. Use a lab computer or your personal computer with either of the following setups **{SKIPPED per TA instruction}**
   a. Connected to the *WES237A_Private* wifi network (passwd: X!I!nxWes237A)
   b. Connected directly to the network switch through ethernet cable
5. Open a command prompt and run `$ **ipconfig**` on windows and `$ ifconfig` on MAC/linux (it may take a second to connect so wait a minute and then run the command)
   a. **How many ipv4 addresses are assigned to this machine?**

Total 3:
192.168.2.11
192.168.56.1
100.81.32.26

   b. **What ipv4 address has the same netmask as the PYNQ board?**

192.168.2.11 {my PC} has the same netmask as my pynq board 255.255.255.0



```
C:\Users\rliz0>ipconfig

Windows IP Configuration


Ethernet adapter Ethernet 4:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::283b:3bcc:d888:fff1%11
   IPv4 Address. . . . . . . . . . . : 192.168.2.11
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :

Ethernet adapter Ethernet 5:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::77e9:45f4:b0aa:9930%12
   IPv4 Address. . . . . . . . . . . : 192.168.56.1
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :

Wireless LAN adapter Local Area Connection* 3:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 4:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . : ucsd.edu
   IPv6 Address. . . . . . . . . . . : 2607:f720:f00:4042::1:eafe
   Link-local IPv6 Address . . . . . : fe80::3bf6:696d:7c5c:dbcf%14
   IPv4 Address. . . . . . . . . . . : 100.81.32.26
   Subnet Mask . . . . . . . . . . . : 255.255.240.0
   Default Gateway . . . . . . . . . : 100.81.32.1

Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :
```

6. Right now, your local machine and your PYNQ board form a network! However, we're more interested in networking two PYNQ boards together rather than your local machine and your PYNQ board. Luckily, every device hooked up to the switch, is assigned an IP address on this network. That means we can communicate with any other board in the class. **Below, compile all the IP addresses of the PYNQ boards in your group.**

Only one PYNQ board (we didn't connect to the class wifi)

7. To access your PYNQ board jupyter notebooks, go to <PYNQ-IP>:9090

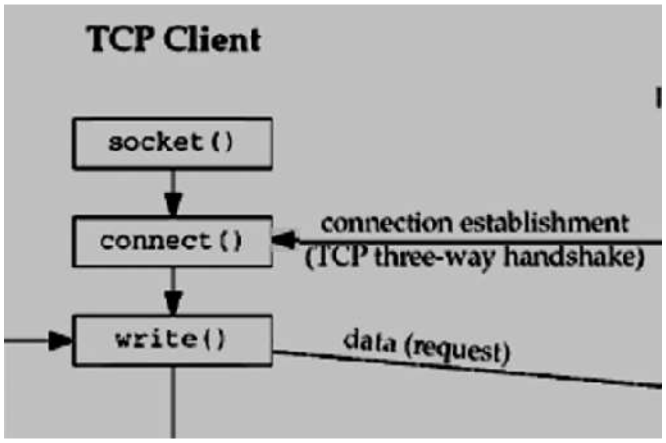
PYNQ-PYNQ Communication with Python

1. Here we're going to implement basic message sending functionality in python from one PYNQ board to another.
2. Download 'sockets_example.ipynb'
3. Go through and complete the code. Answer the following questions.
    a. **What does `socket.SOCK_STREAM` mean (hint: search the documentation link in the notebook)?**

The socket.SOCK_STREAM constant is used to specify that a socket should use the TCP protocol. TCP is a connection-oriented protocol, which means that a reliable connection is established between the two parties before data is transmitted. This makes TCP ideal for applications such as web browsing and file transfer, where it is important to ensure that all data is received correctly.
In contrast, the socket.SOCK_DGRAM constant is used to specify that a socket should use the UDP protocol. UDP is a connectionless protocol, which means that data is sent without establishing a connection first. This makes UDP ideal for applications such as streaming video and audio, where it is more important to minimize latency than to ensure that all data is received correctly.

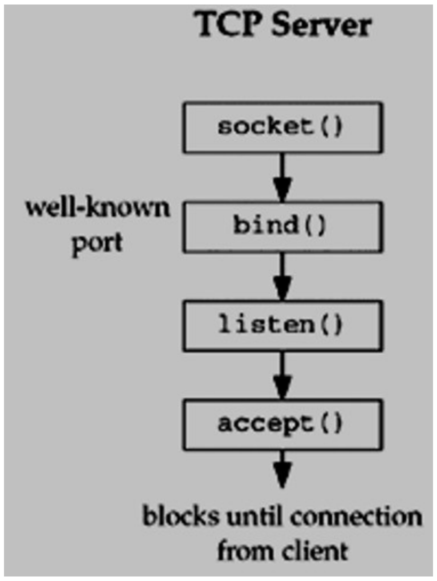
   b. **What is the order of operations for starting a client socket and sending a message?**

Server(s) is listening for client(s) to connect, after the connections has been accepted by the Server, then the sending & receiving information can start



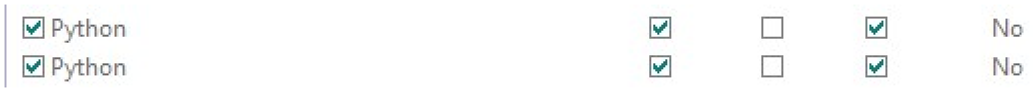
   c. **What is the order of operations for starting a server socket and receiving a message?**

The Server Listening has to be ready, before accepting client connection, then after connection is done, messages can be send and received



Wireshark

1. On your local machine (or lab machine), install Wireshark
2. Open the Firewall and Network Protection
3. Click 'Allow an app through firewall'
4. Click 'Change Settings'
5. Scroll down to 'Python'
6. Select all 'Python' applications and all 'Public' boxes for each 'Python'

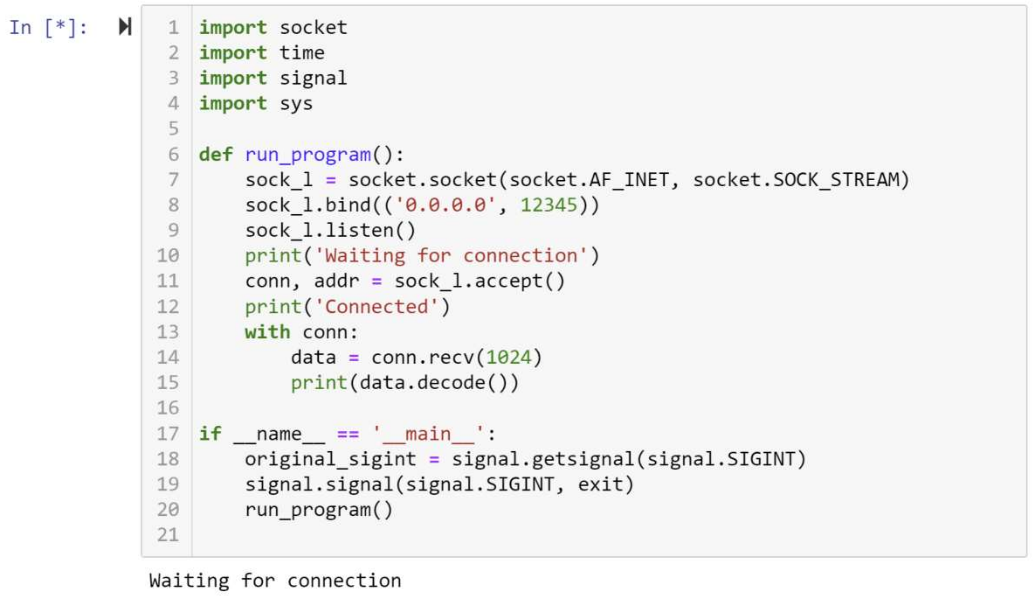


✓ Python	✓	☐	✓	No
✓ Python	✓	☐	✓	No

7. Open the program `IDLE (Python 3.7 64-bit)`
8. Click File->New File and paste the following code **(Check for tab v space errors when copying and pasting)**

```
import socket
import time
import signal
import sys

def run_program():
      sock_l = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
      sock_l.bind(('0.0.0.0', 12345))
      sock_l.listen()
      print('Waiting for connection')
      conn, addr = sock_l.accept()
      print('Connected')
      with conn:
      data = conn.recv(1024)
      print(data.decode())

if __name__ == '__main__':
      original_sigint = signal.getsignal(signal.SIGINT)
      signal.signal(signal.SIGINT, exit)
      run_program()
```

9. Save the file, then select 'Run -> Run Module'. This is a slight variation to your server. It is waiting on port 12345 on the local lab machine.

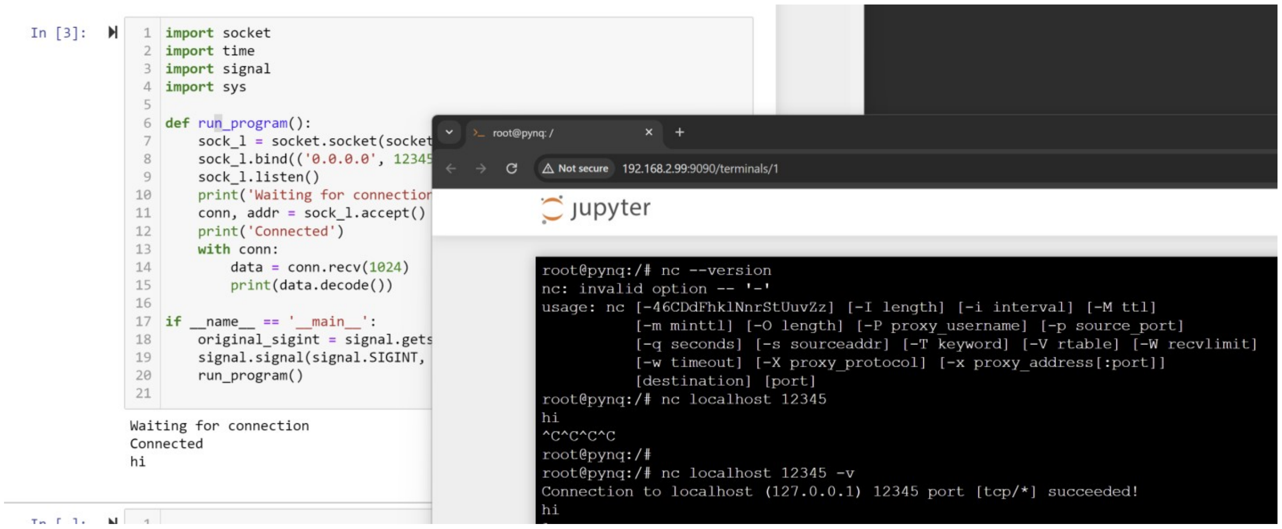


In [*]:

```
import socket
import time
import signal
import sys

def run_program():
    sock_l = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock_l.bind(('0.0.0.0', 12345))
    sock_l.listen()
    print('Waiting for connection')
    conn, addr = sock_l.accept()
    print('Connected')
    with conn:
        data = conn.recv(1024)
        print(data.decode())

if __name__ == '__main__':
    original_sigint = signal.getsignal(signal.SIGINT)
    signal.signal(signal.SIGINT, exit)
    run_program()

```

```
Waiting for connection
```

10. From your PYNQ board, connect your client to
    a. Ip: local lab IP
    b. Port: 12345



```
In [3]:
import socket
import time
import signal
import sys

def run_program():
    sock_l = socket.socket(socket
    sock_l.bind(('0.0.0.0', 12345
    sock_l.listen()
    print('Waiting for connection
    conn, addr = sock_l.accept()
    print('Connected')
    with conn:
        data = conn.recv(1024)
        print(data.decode())

if __name__ == '__main__':
    original_sigint = signal.gets
    signal.signal(signal.SIGINT,
    run_program()

Waiting for connection
Connected
hi
```

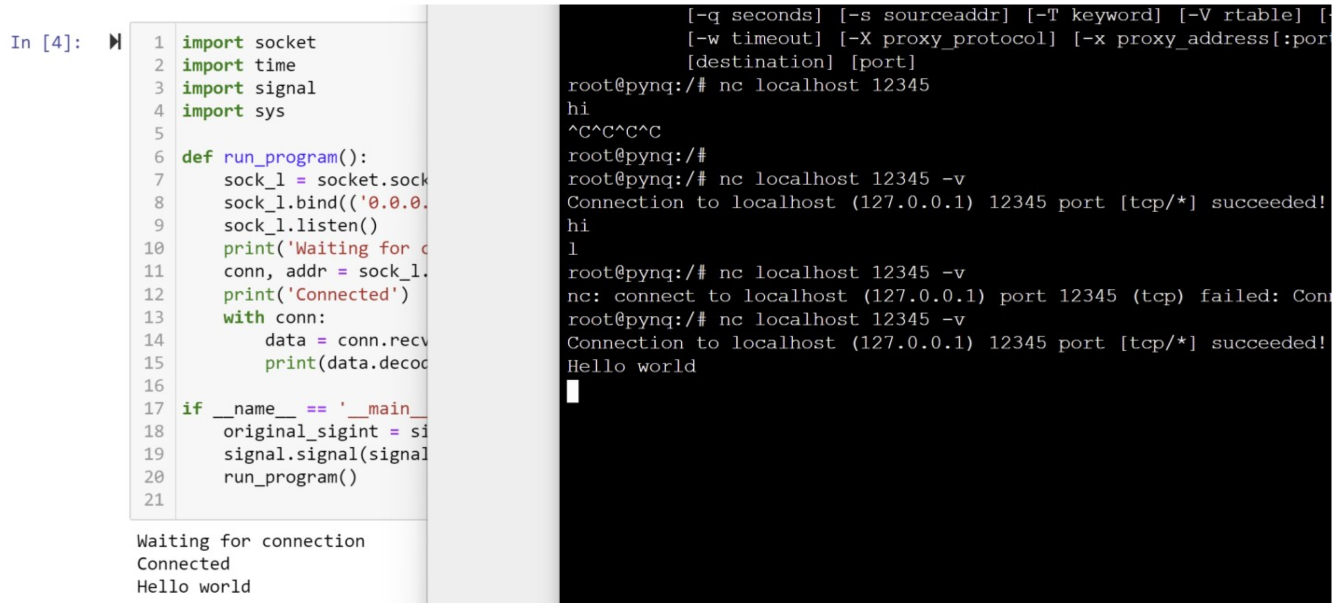
root@pynq: /

Not secure 192.168.2.99:9090/terminals/1

jupyter

```
root@pynq:/# nc --version
nc: invalid option -- '-'
usage: nc [-46CDdFhklNnrStUuvZz] [-I length] [-i interval] [-M ttl]
          [-m minttl] [-O length] [-P proxy_username] [-p source_port]
          [-q seconds] [-s sourceaddr] [-T keyword] [-V rtable] [-W recvlimit]
          [-w timeout] [-X proxy_protocol] [-x proxy_address[:port]]
          [destination] [port]
root@pynq:/# nc localhost 12345
hi
^C^C^C^C
root@pynq:/#
root@pynq:/# nc localhost 12345 -v
Connection to localhost (127.0.0.1) 12345 port [tcp/*] succeeded!
hi
```

11. Send the message “Hello world\n”
12. You should see it displayed in the Python terminal

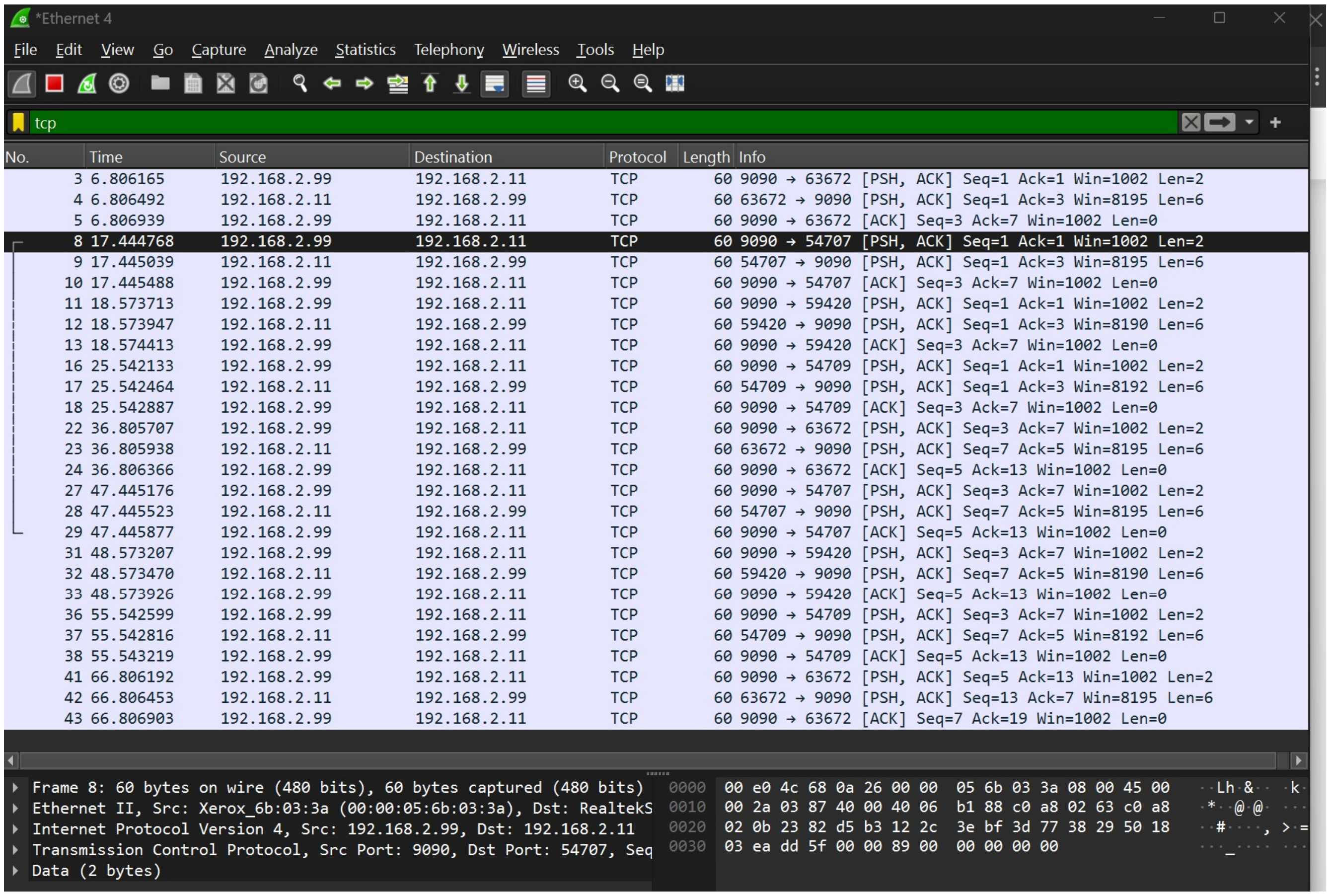


```
In [4]:
import socket
import time
import signal
import sys

def run_program():
    sock_l = socket.sock
    sock_l.bind(('0.0.0.
    sock_l.listen()
    print('Waiting for c
    conn, addr = sock_l.
    print('Connected')
    with conn:
        data = conn.recv
        print(data.decod

if __name__ == '__main_
    original_sigint = si
    signal.signal(signal
    run_program()

Waiting for connection
Connected
Hello world
```

```
          [-q seconds] [-s sourceaddr] [-T keyword] [-V rtable] [-
          [-w timeout] [-X proxy_protocol] [-x proxy_address[:por
          [destination] [port]
root@pynq:/# nc localhost 12345
hi
^C^C^C^C
root@pynq:/#
root@pynq:/# nc localhost 12345 -v
Connection to localhost (127.0.0.1) 12345 port [tcp/*] succeeded!
hi
l
root@pynq:/# nc localhost 12345 -v
nc: connect to localhost (127.0.0.1) port 12345 (tcp) failed: Con
root@pynq:/# nc localhost 12345 -v
Connection to localhost (127.0.0.1) 12345 port [tcp/*] succeeded!
Hello world
```

13. Now open Wireshark
14. Double click ‘Wi-Fi’ or ‘Ethernet’ depending on how you connected to the network. You’re now capturing a trace of the network which is only between your machine and the PYNQ board through the router. Look at a few of the traces. Notice which are between your PYNQ board and the local machine (check the IP addresses) and which involve the router. There will only be a difference if you also connected the PYNQ board directly to your local machine.

15. Where it says ‘Apply a display filter’ at the top, type ‘tcp’


*Ethernet 4
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp

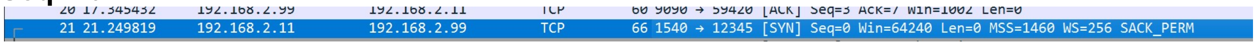
```
No.   Time        Source         Destination    Protocol Length Info
    3 6.806165    192.168.2.99   192.168.2.11   TCP        60 9090 → 63672 [PSH, ACK] Seq=1 Ack=1 Win=1002 Len=2
    4 6.806492    192.168.2.11   192.168.2.99   TCP        60 63672 → 9090 [PSH, ACK] Seq=1 Ack=3 Win=8195 Len=6
    5 6.806939    192.168.2.99   192.168.2.11   TCP        60 9090 → 63672 [ACK] Seq=3 Ack=7 Win=1002 Len=0
    8 17.444768   192.168.2.99   192.168.2.11   TCP        60 9090 → 54707 [PSH, ACK] Seq=1 Ack=1 Win=1002 Len=2
    9 17.445039   192.168.2.11   192.168.2.99   TCP        60 54707 → 9090 [PSH, ACK] Seq=1 Ack=3 Win=8195 Len=6
   10 17.445488   192.168.2.99   192.168.2.11   TCP        60 9090 → 54707 [ACK] Seq=3 Ack=7 Win=1002 Len=0
   11 18.573713   192.168.2.99   192.168.2.11   TCP        60 9090 → 59420 [PSH, ACK] Seq=1 Ack=1 Win=1002 Len=2
   12 18.573947   192.168.2.11   192.168.2.99   TCP        60 59420 → 9090 [PSH, ACK] Seq=1 Ack=3 Win=8190 Len=6
   13 18.574413   192.168.2.99   192.168.2.11   TCP        60 9090 → 59420 [ACK] Seq=3 Ack=7 Win=1002 Len=0
   16 25.542133   192.168.2.99   192.168.2.11   TCP        60 9090 → 54709 [PSH, ACK] Seq=1 Ack=1 Win=1002 Len=2
   17 25.542464   192.168.2.11   192.168.2.99   TCP        60 54709 → 9090 [PSH, ACK] Seq=1 Ack=3 Win=8192 Len=6
   18 25.542887   192.168.2.99   192.168.2.11   TCP        60 9090 → 54709 [ACK] Seq=3 Ack=7 Win=1002 Len=0
   22 36.805707   192.168.2.99   192.168.2.11   TCP        60 9090 → 63672 [PSH, ACK] Seq=3 Ack=7 Win=1002 Len=2
   23 36.805938   192.168.2.11   192.168.2.99   TCP        60 63672 → 9090 [PSH, ACK] Seq=7 Ack=5 Win=8195 Len=6
   24 36.806366   192.168.2.99   192.168.2.11   TCP        60 9090 → 63672 [ACK] Seq=5 Ack=13 Win=1002 Len=0
   27 47.445176   192.168.2.99   192.168.2.11   TCP        60 9090 → 54707 [PSH, ACK] Seq=3 Ack=7 Win=1002 Len=2
   28 47.445523   192.168.2.11   192.168.2.99   TCP        60 54707 → 9090 [PSH, ACK] Seq=7 Ack=5 Win=8195 Len=6
   29 47.445877   192.168.2.99   192.168.2.11   TCP        60 9090 → 54707 [ACK] Seq=5 Ack=13 Win=1002 Len=0
   31 48.573207   192.168.2.99   192.168.2.11   TCP        60 9090 → 59420 [PSH, ACK] Seq=3 Ack=7 Win=1002 Len=2
   32 48.573470   192.168.2.11   192.168.2.99   TCP        60 59420 → 9090 [PSH, ACK] Seq=7 Ack=5 Win=8190 Len=6
   33 48.573926   192.168.2.99   192.168.2.11   TCP        60 9090 → 59420 [ACK] Seq=5 Ack=13 Win=1002 Len=0
   36 55.542599   192.168.2.99   192.168.2.11   TCP        60 9090 → 54709 [PSH, ACK] Seq=3 Ack=7 Win=1002 Len=2
   37 55.542816   192.168.2.11   192.168.2.99   TCP        60 54709 → 9090 [PSH, ACK] Seq=7 Ack=5 Win=8192 Len=6
   38 55.543219   192.168.2.99   192.168.2.11   TCP        60 9090 → 54709 [ACK] Seq=5 Ack=13 Win=1002 Len=0
   41 66.806192   192.168.2.99   192.168.2.11   TCP        60 9090 → 63672 [PSH, ACK] Seq=5 Ack=13 Win=1002 Len=2
   42 66.806453   192.168.2.11   192.168.2.99   TCP        60 63672 → 9090 [PSH, ACK] Seq=13 Ack=7 Win=8195 Len=6
   43 66.806903   192.168.2.99   192.168.2.11   TCP        60 9090 → 63672 [ACK] Seq=7 Ack=19 Win=1002 Len=0

▸ Frame 8: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)       0000  00 e0 4c 68 0a 26 00 00  05 6b 03 3a 08 00 45 00   ··Lh·&··  ·k·
▸ Ethernet II, Src: Xerox_6b:03:3a (00:00:05:6b:03:3a), Dst: RealtekS      0010  00 2a 03 87 40 00 40 06  b1 88 c0 a8 02 63 c0 a8   ·*··@·@·  ···
▸ Internet Protocol Version 4, Src: 192.168.2.99, Dst: 192.168.2.11        0020  02 0b 23 82 d5 b3 12 2c  3e bf 3d 77 38 29 50 18   ··#····,  >·=
▸ Transmission Control Protocol, Src Port: 9090, Dst Port: 54707, Seq      0030  03 ea dd 5f 00 00 89 00  00 00 00 00               ···_····  ···
▸ Data (2 bytes)
```

16. Repeat steps 9-11

17. Check the packet trace for any changes

a. **What is the sequence number of the TCP SYN segment that is used to initiate the TCP connection between the PYNQ board and the local machine?**

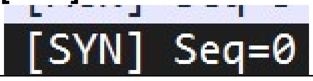
18. Seq = 0

19. 

```
   20 17.345432   192.168.2.99   192.168.2.11   TCP   60 9090 → 59420 [ACK] Seq=3 Ack=7 Win=1002 Len=0
   21 21.249819   192.168.2.11   192.168.2.99   TCP   66 1540 → 12345 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
```

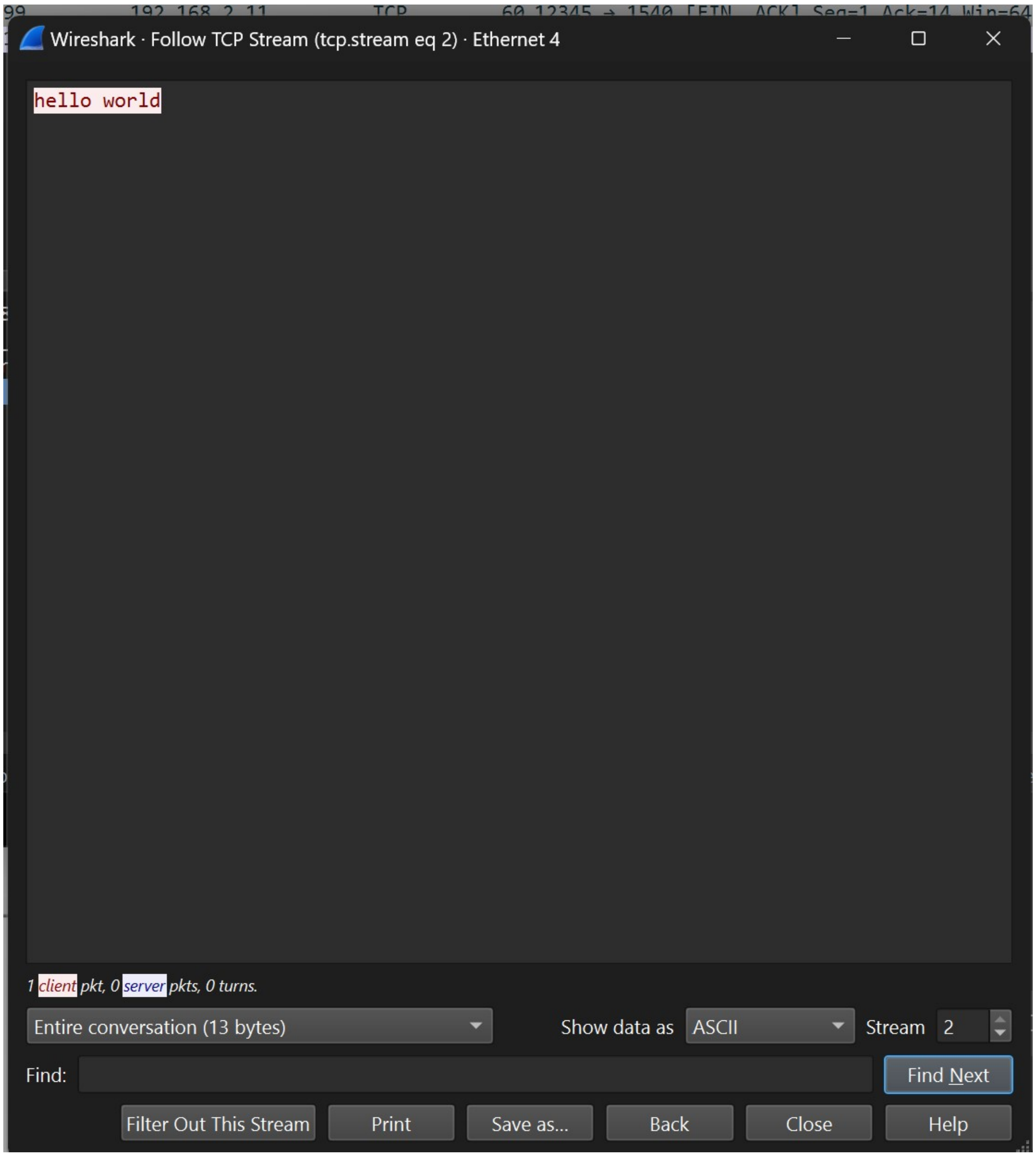
a. **What is it in the segment that identifies the segment as a SYN segment?**

[SYN]



```
[SYN] Seq=0
```

b. Right click this trace and select ‘Follow->TCP Stream’

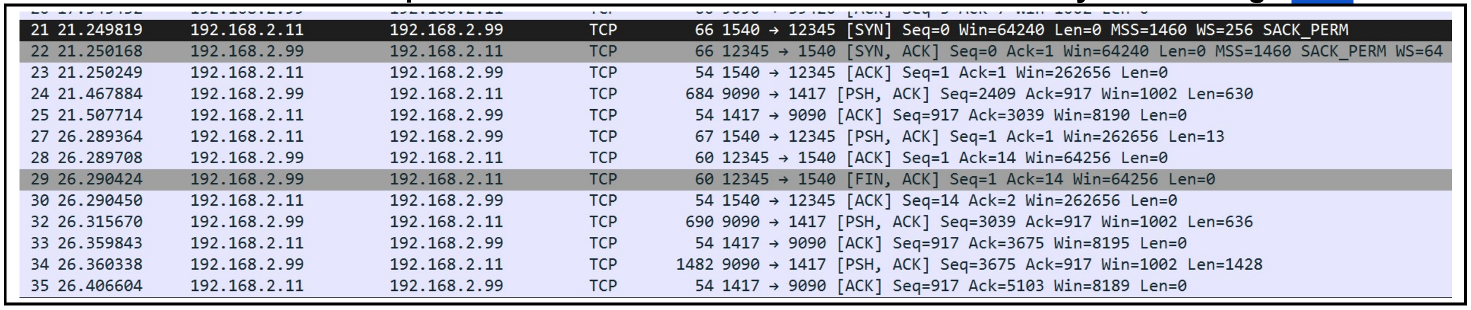

Wireshark · Follow TCP Stream (tcp.stream eq 2) · Ethernet 4

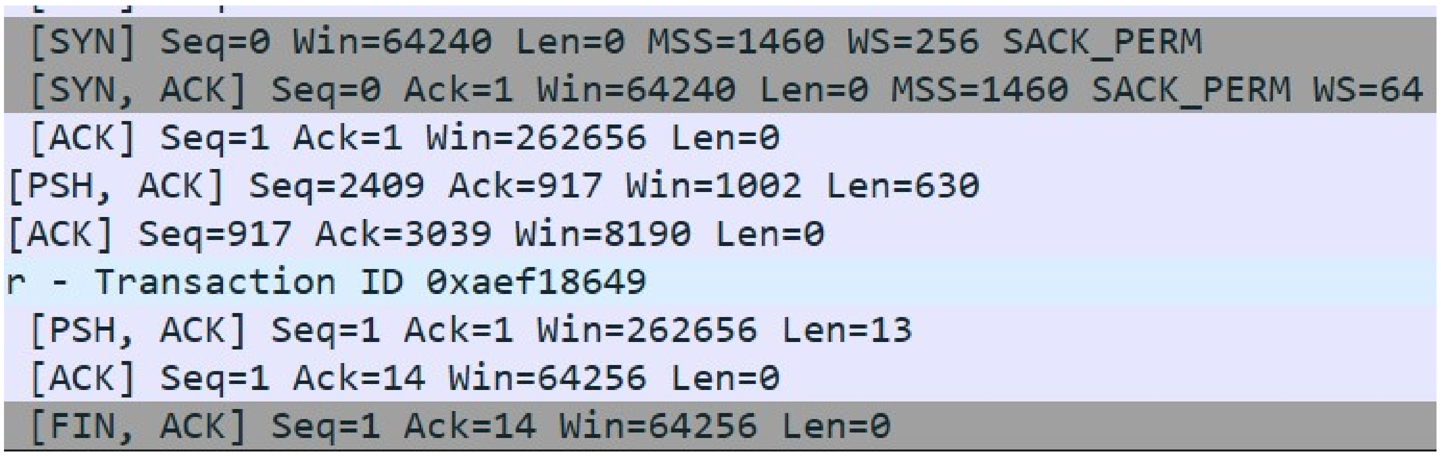
```
hello world
```

1 client pkt, 0 server pkts, 0 turns.

Entire conversation (13 bytes) · Show data as ASCII · Stream 2

c. **Repeat a few times with different messages. Describe what's happening in the 5-10 steps of the TCP sequence for this communication. You can refresh your TCP flags here.**


```
21 21.249819   192.168.2.11   192.168.2.99   TCP     66 1540 → 12345 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
22 21.250168   192.168.2.99   192.168.2.11   TCP     66 12345 → 1540 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM WS=64
23 21.250249   192.168.2.11   192.168.2.99   TCP     54 1540 → 12345 [ACK] Seq=1 Ack=1 Win=262656 Len=0
24 21.467884   192.168.2.99   192.168.2.11   TCP    684 9090 → 1417 [PSH, ACK] Seq=2409 Ack=917 Win=1002 Len=630
25 21.507714   192.168.2.11   192.168.2.99   TCP     54 1417 → 9090 [ACK] Seq=917 Ack=3039 Win=8190 Len=0
27 26.289364   192.168.2.11   192.168.2.99   TCP     67 1540 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=262656 Len=13
28 26.289708   192.168.2.99   192.168.2.11   TCP     60 12345 → 1540 [ACK] Seq=1 Ack=14 Win=64256 Len=0
29 26.290424   192.168.2.99   192.168.2.11   TCP     60 12345 → 1540 [FIN, ACK] Seq=1 Ack=14 Win=64256 Len=0
30 26.290450   192.168.2.11   192.168.2.99   TCP     54 1540 → 12345 [ACK] Seq=14 Ack=2 Win=262656 Len=0
32 26.315670   192.168.2.99   192.168.2.11   TCP    690 9090 → 1417 [PSH, ACK] Seq=3039 Ack=917 Win=1002 Len=636
33 26.359843   192.168.2.11   192.168.2.99   TCP     54 1417 → 9090 [ACK] Seq=917 Ack=3675 Win=8195 Len=0
34 26.360338   192.168.2.99   192.168.2.11   TCP   1482 9090 → 1417 [PSH, ACK] Seq=3675 Ack=917 Win=1002 Len=1428
35 26.406604   192.168.2.11   192.168.2.99   TCP     54 1417 → 9090 [ACK] Seq=917 Ack=5103 Win=8189 Len=0
```



```
 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM WS=64
 [ACK] Seq=1 Ack=1 Win=262656 Len=0
[PSH, ACK] Seq=2409 Ack=917 Win=1002 Len=630
[ACK] Seq=917 Ack=3039 Win=8190 Len=0
r - Transaction ID 0xaef18649
 [PSH, ACK] Seq=1 Ack=1 Win=262656 Len=13
 [ACK] Seq=1 Ack=14 Win=64256 Len=0
 [FIN, ACK] Seq=1 Ack=14 Win=64256 Len=0
```

• [SYN] Len =0	• Initiate connection
• [SYN, ACK] Len =0	• Packet(s) confirmation received
• [ACK] Len =0	• Packet(s) confirmation received
• [PSH, ACK] Len =0	• Incoming un-buffered data0
• [ACK] Len =0	• Packet(s) confirmation received
• [PSH, ACK] Len =13	• Len =13 "Hello world \n"
• [ACK] Len =0	• Packet(s) confirmation received
• [FIN, ACK] Len =0	• Both the sender & receiver send the FIN packets to gracefully terminate the connection

https://www.howtouselinux.com/post/tcp-flags

TCP Flags List

- **SYN** Packets that are used to initiate a connection.
- **ACK** Packets that are used to confirm that the data packets have been received, also used to confirm the initiation request and tear down requests
- **RST** Signify the connection is down or maybe the service is not accepting the requests
- **FIN** Indicate that the connection is being torn down. Both the sender and receiver send the FIN packets to gracefully terminate the connection
- **PSH** Indicate that the incoming data should be passed on directly to the application instead of getting buffered
- **URG** Indicate that the data that the packet is carrying should be processed immediately by the TCP stack. It can be used to provide out-of-band data transfer, such as signaling that a message is urgent and should be delivered before other data.

Here are the numbers which match with the corresponding TCP flags.

Flag	Decimal Value
URG	32
ACK	16
PSH	8
RST	4
SYN	2
FIN	1

In [ ]:
```
import time
from pynq.overlays.base import BaseOverlay
import socket

base = BaseOverlay("base.bit")
btns = base.btns_gpio
leds = base.leds
```

Sockets

This notebook has both a client and a server functionality. One PYNQ board in the group will be the client and SENDS the message. Another PYNQ board will be the server and RECEIVES the message.

In [ ]:
```

```

Server

Here, we'll build the server code to LISTEN for a message from a specific PYNQ board.

When we send/receive messages, we need to pieces of information which will tell us where to send the information. First, we need the IP address of our friend. Second, we need to chose a port to listen on. For an analogy, Alice expects her friend, Bob, to deliver a package to our back door. With this information, ALICE (server ip address) can wait at the BACK DOOR (port) for BOB (client ip address) to deliver the package.

Format of the information ipv4 address: ###.###.###.### (no need for leading zeros if the number is less than three digits) port: ##### (it could be 4 or 5 digits long, but must be >1024)

Use the socket documentation (Section 18.1.3) to find the appropriate functions https://python.readthedocs.io/en/latest/library/socket.html

In [ ]:
```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# TODO:
# 1: Bind the socket to the pynq board <CLIENT-IP> at port <LISTENING-PORT>
# 2: Accept connections
# 3: Receive bytes from the connection
# 4: Print the received message
```

Client

Now, we can implement the CLIENT code.

Back to the analogy, now we're interested in delivering a package to our friend's back door. This means BOB (client ip address) is delivering a package to ALICE (server ip address) at her BACK DOOR (port)

Remember to start the server before running the client code

In [ ]:
```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# TODO:
# 1: Connect the socket (sock) to the <SERVER-IP> and choosen port <LISTENING-PORT>
# 2: Send the message "Hello world!\n"
# 3: Close the socket
```

On your server, you should see the message and then the server will shutdown! When we close a socket, both the client and the server are disconnected from the port.

Instead, change the function above to send 5 messages before closing.

The pseudocode looks like this

- connect the socket
- for i in range(5)
  - msg = input("Message to send: ")
  - send the message (msg)
- close the socket

In [ ]:
```
```

In [1]:
```
import socket
import time
import signal
import sys

def run_program():
    sock_l = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock_l.bind(('0.0.0.0', 12345))
    sock_l.listen()
    print('Waiting for connection')
    conn, addr = sock_l.accept()
    print('Connected')
    with conn:
        data = conn.recv(1024)
        print(data.decode())

if __name__ == '__main__':
    original_sigint = signal.getsignal(signal.SIGINT)
    signal.signal(signal.SIGINT, exit)
    run_program()
```

```
Waiting for connection
Connected
hello world
```

In [ ]:
```
```