Back to the Stone Age

A basic networking stack

Team 2: Eric Moss, Sarah Strickman, Will Merges

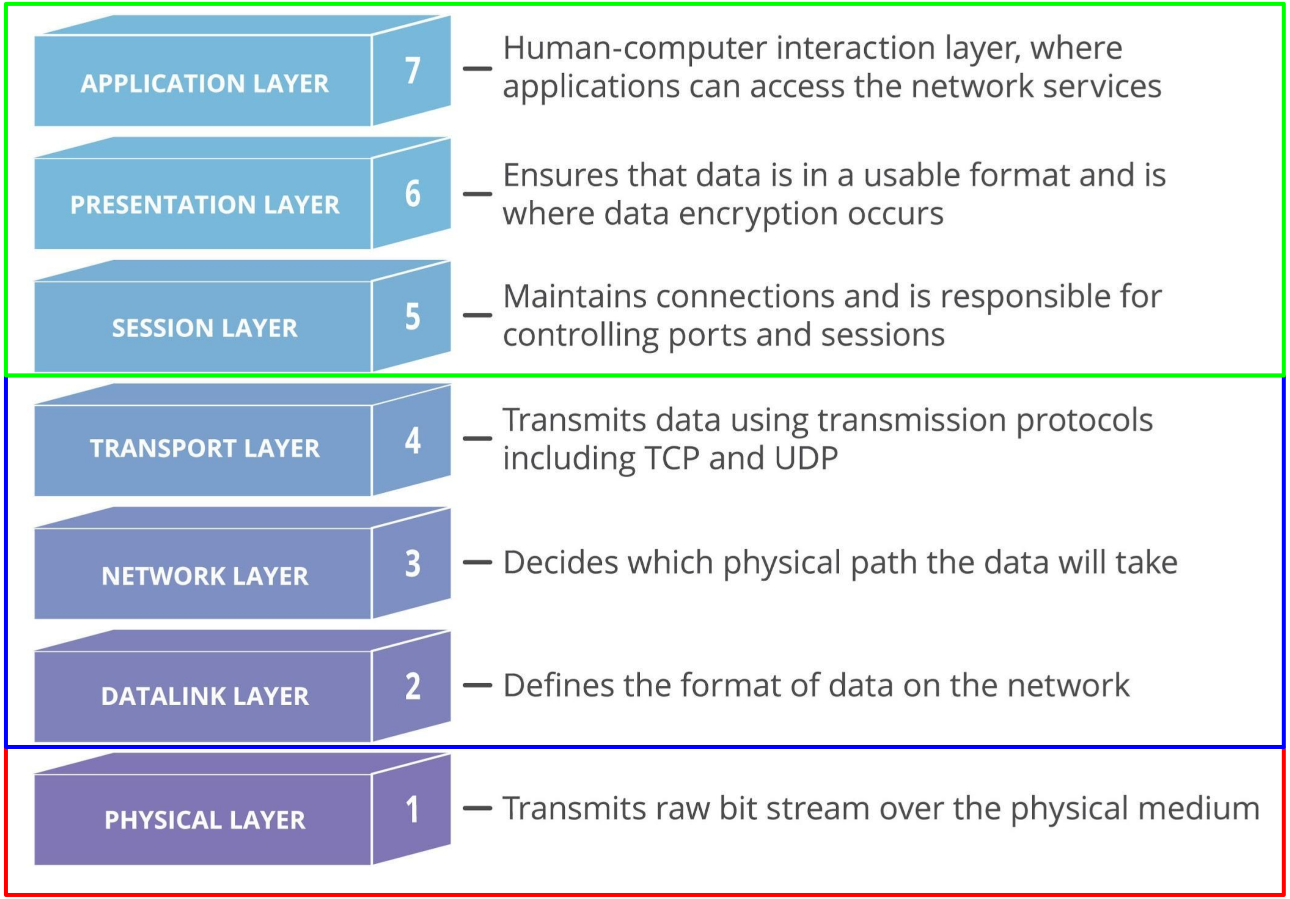
Table of Contents

Overview

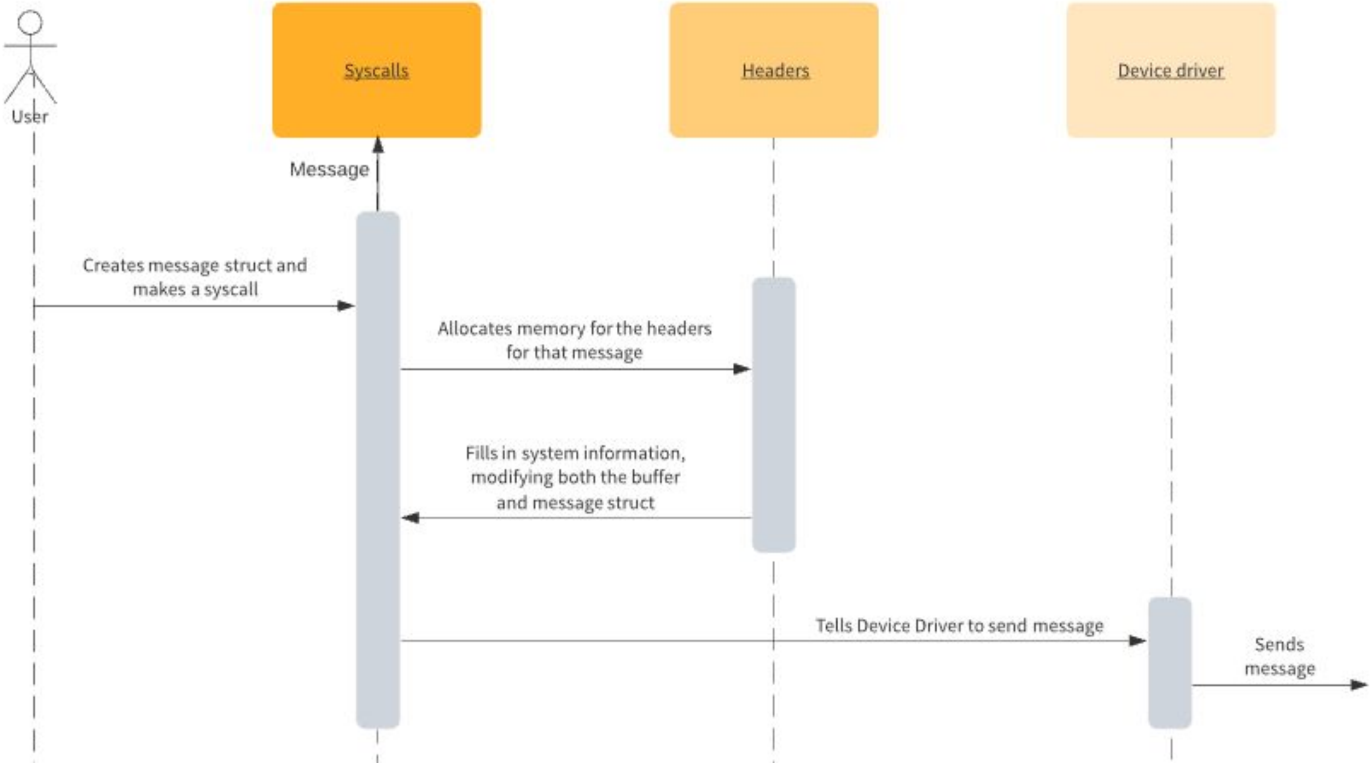
- Project focus was Networking
  - Ethernet driver for the Intel 82557
  - Ethernet, IPv4, and UDP headers for packet construction
  - Syscalls and message abstraction to be used by user programs
- Projects were closely related
  - Led to close communication, as the projects relied on each other
  - Unique development timeline
    - One part had to work before the others could

Design - OSI

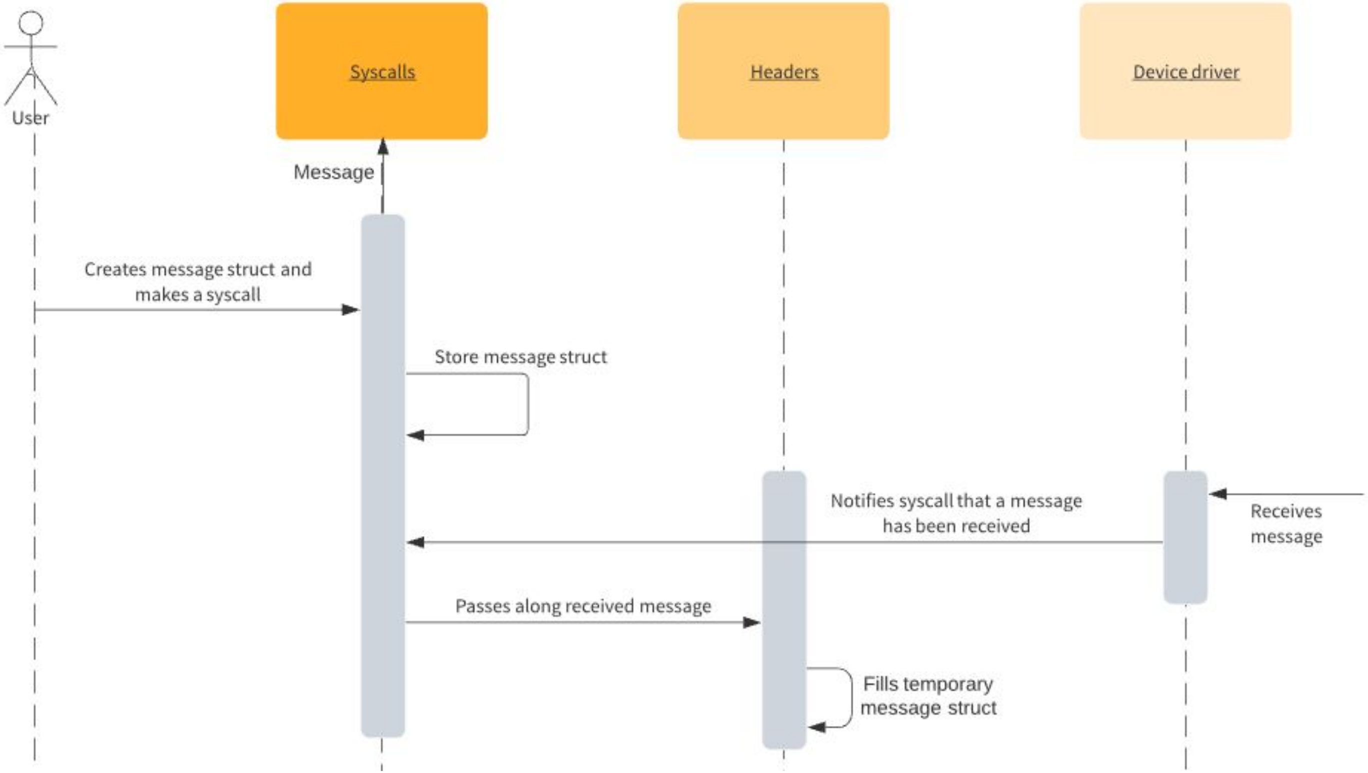
- OSI model
- Device Driver / Hardware
- Header Functions
    - Ethernet II / IPv4 / UDP
- User processes / Syscalls



Design - Sending a message



Design - Receiving a message



Project 1: Messages and Syscalls

4 syscalls

- netsend
- netrecv
- setip
- setMAC

Send/Receive messages in a standardized message format

Project 1: Message Format

msg_t - *net.h*

<table>
<tr><td>src_port</td><td>dst_port</td><td colspan="2">src_addr</td></tr>
<tr><td colspan="2">dst_addr</td><td colspan="2">dst_MAC low</td></tr>
<tr><td colspan="2">dst_MAC high</td><td colspan="2">src_MAC low</td></tr>
<tr><td colspan="2">src_MAC high</td><td>len</td><td>data* high</td></tr>
<tr><td>data* low</td></tr>
</table>

total size: 34 bytes

Creating a message

Syscalls require some info from users, and fill in the rest themselves

- For sending:
  - Source port
  - Destination port
  - Destination address
  - Destination MAC
  - Data / Data length
- For receiving:
  - Receiving port
  - Data / Data length

Setting MAC / IPv4 addresses

User programs can change MAC and IP addresses at runtime

- Setting MAC:
  - Take array of bytes to represent MAC
  - Extract relevant bytes
  - Set system MAC
- Setting IP:
  - User must convert address to 4 byte IP in network order (available with htons() call)
  - IP will be set as origin for sends and destination for receives
  - Can only set one system wide IP

OS will fill in this data during sends and receives

Project 2: Packet Headers

- Two parts to this project
  - Adding headers when sending a packet
  - Handling headers when a packet is received

- Headers (both parts of the project deal with these)
  - Ethernet II Frame Header - Link Layer
  - IPv4 Header - Network Layer
  - UDP Header - Transport Layer

Project 2: Packet Headers - Structs

LINKhdr_t - *link.h*

<table>
<tr><td colspan="2"><code>dst_mac</code></td><td><code>src_mac lo</code></td></tr>
<tr><td><code>src_mac hi</code></td><td><code>ethertype</code></td><td></td></tr>
</table>

total size: 14 bytes

Project 2: Packet Headers - Structs

NETipv4hdr_t - *ip.h*

<table>
<tr><td><code>ver_ihl</code></td><td><code>dscp_ecn</code></td><td colspan="2"><code>tot_len[2]</code></td><td colspan="2"><code>id</code></td><td colspan="2"><code>flags_offset</code></td></tr>
<tr><td><code>ttl</code></td><td><code>protocol</code></td><td colspan="2"><code>checksum[2]</code></td><td colspan="4"><code>src_addr</code></td></tr>
<tr><td colspan="4"><code>dst_addr</code></td></tr>
</table>

Total size: 20 bytes

Project 2: Packet Headers - Structs

UDPhdr_t - *transport.h*

`src_port`	`dst_port`	`len[2]`	`checksum`

total size: 8 bytes

Sending a message

- Each layer is responsible for adding its own information
    - All functions modify the same buffer
- Only the uppermost layer (udp) copies data into buffer
    - Lower time complexity

```
__link_add_header(uint8_t* buff, uint16_t len, msg_t* msg)
    fills in information relating to the link layer
    size =
    __ipv4_add_header(uint8_t* buff, uint16_t len, msg_t* msg)
        fills in information relating to the link layer
        size =
        __udp_add_header(uint8_t* buff, uint16_t len, msg_t* msg)
            fills in information relating to the link layer
            memcpy data so it comes after the headers
            return size of udp header + size of data (or 0 if error)

        compute and fill in remaining fields (length and checksum)
        return size of ipv4 header + size (or 0 if error)

    pad payload if it's too small
    return size of link header + size (or 0 if error)
```

Receiving a message - ipv4

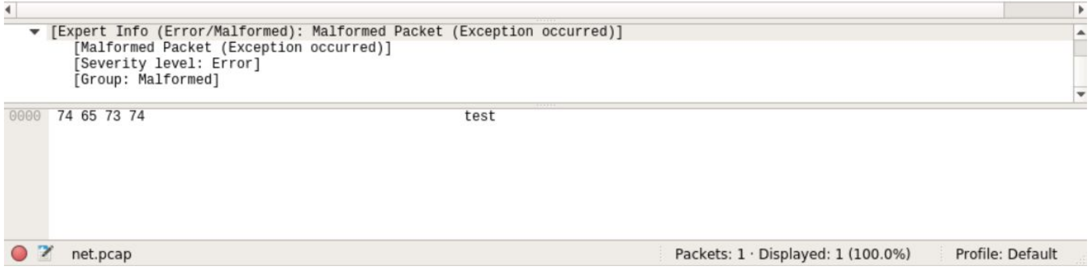
```
__link_parse_frame(msg_t* msg, uint16_t len, const uint8_t* data)
    get mac address info from data (src, dst) and fill in msg with info
    return
    __ipv4_parse_frame(msg_t* msg, uint16_t len, const uint8_t* data)
        get ip address info (src, dst) from data and fill in msg with info
        return
        __udp_parse_frame(msg_t* msg, uint16_t len, const uint8_t* data)
            get port information from data and fill in msg with info
            point msg->data to the data in data
            return 1 if packet if big enough (should be passed to user), 0 on error

        or 0 on error (i.e. protocol is not UDP)

    or 0 on error (i.e. ethertype is not ipv4 or arp)
```

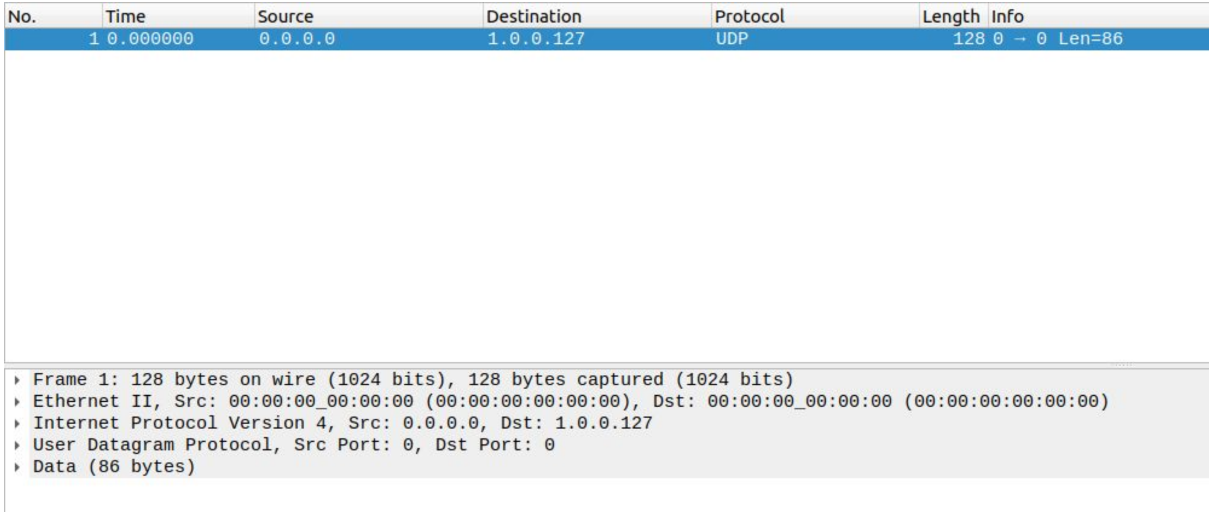
Note: We also have the ability to handle/respond to arp packets - this is additional functionality and will be talked about later on in the presentation

Wireshark



<-- Before

After -->



Project 3: Ethernet Driver

- First thing we did
- The DSL lab computers have Intel 82557's, but our driver technically supports anything in the 8255x family
- Lots of technical reading

yikes

PCI

- The NIC is a PCI device
- Accessed PCI configuration space to obtain:
    - Device ID
    - Vendor ID
    - Interrupt line
    - Base address register
- Brute force scanned the entire PCI address space
    - Probably a better way to do this
- Had to make device a PCI master
- *pci.c / pci.h*

Byte Offset (hexadecimal)	Byte 3	Byte 2	Byte 1	Byte 0
0	Device ID		Vendor ID	
4	Status Register		Command Register	
8	Class Code (200000h)			Revision ID
C	BIST	Header Type	Latency Timer	Cache Line Size
10	CSR Memory Mapped Base Address Register			
14	CSR I/O Mapped Base Address Register			
18	Flash Memory Mapped Base Address Register			
1C	Reserved			
20				
24				
28				
2C	Subsystem ID		Subsystem Vendor ID	
30	Expansion ROM Base Address Register			
34	Reserved			Cap_Ptr
38	Reserved			
3C	Max_Latency (FFh)	Min_Grant (FFh)	Interrupt Pin (01h)	Interrupt Line
DC	Power Management Capabilities		Next Item Pointer	Capability ID
E0	Reserved	Data	Power Management CSR	

The NIC

- Broken up into Command Unit (CU) and Receive Unit (RU)
- Commands are executed using the System Control Block (SCB)
  - Load CU base - 0x00
  - Load RU base - 0x00
  - CU start
  - RU start
- CSR is accessed through the I/O address space
  - Offset is obtained from PCI BAR
- PORT commands control resets
- *eth.c / eth.h*

Control / Status Register

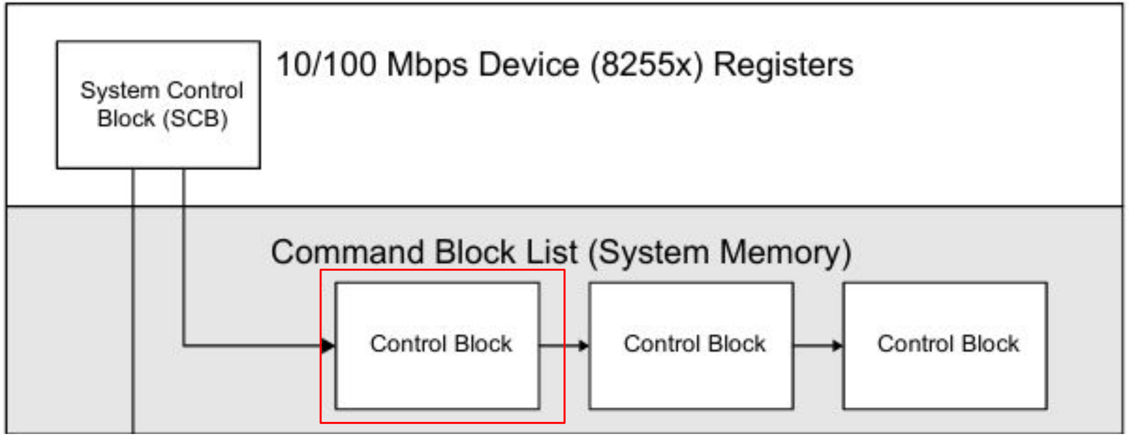
Upper Word (31–16)		Lower Word (15–0)		Offset
SCB Command Word		SCB Status Word		0h
SCB General Pointer				4h
PORT				8h
EEPROM Control Register		Reserved		Ch
MDI Control Register				10h
RX DMA Byte Count				14h
PMDR	Flow Control Register		Reserved	18h
Reserved		General Status	General Control	1Ch
Reserved				20h-2Ch
Function Event Register				30h
Function Event Mask Register				34h
Function Present State Register				38h
Force Event Register				3Ch

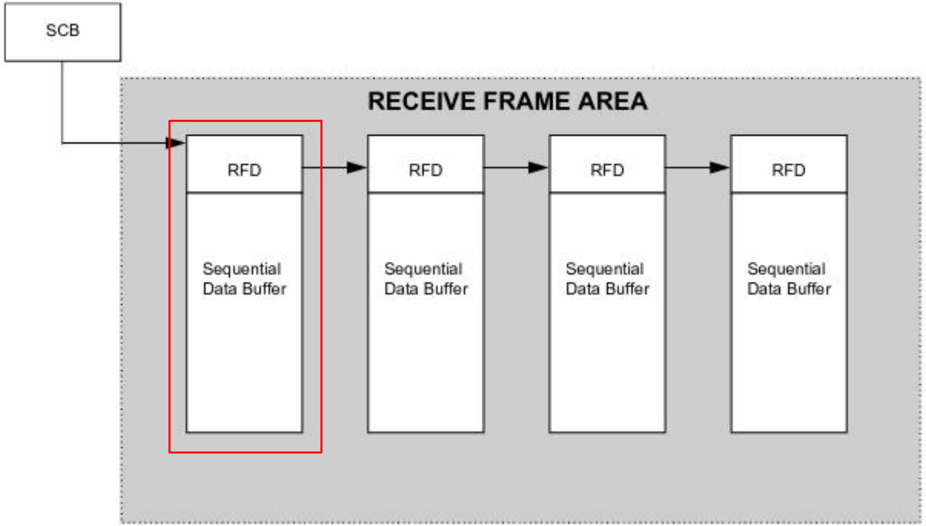
DMA - Command Block List

- A linked list of commands to execute on the CU
  - Load internal (MAC) address - makes sure packets addressed to us are not discarded
  - Transmit - send a frame
- Only ever have one command on the CBL
  - Kept our own queue of commands to be executed
  - When a command completes, change the CBL pointer and restart the CU
- Allocate command space from static memory
  - Frame data is right after the transmit command in “simple mode”, variable sized buffers
  - 2 byte align everything
- Limitations
  - Not the fastest
  - Support 50 commands concurrently, up to 8192 bytes of command data

DMA - Receive Frame Area

- RFA is made up of Receive Frame Descriptors
  - We only ever have one RFD in the RFA, makes the ISR simpler
  - 2 byte aligned
- In simple mode, receive data placed directly after RFD header
  - Flexible mode could support copying frame directly to user's memory space
- Restart the RU every "frame ready" interrupt


System Control Block (SCB)
10/100 Mbps Device (8255x) Registers
Command Block List (System Memory)
Control Block
Control Block
Control Block


SCB
RECEIVE FRAME AREA
RFD
Sequential Data Buffer
RFD
Sequential Data Buffer
RFD
Sequential Data Buffer
RFD
Sequential Data Buffer

Interrupts

- Device (should) interrupt when:
  - CU command completed (load address, transmit)
  - Received a frame
- Interrupt type determined by SCB status word
- Utilizes callback functions to get data from the driver
  - Commands are executed with an id
  - Pass a function pointer to ethernet driver to be called when commands are complete
  - Separate function pointer for received frame data to be passed for the OS to deal with
- Can also get “receive no resources” interrupt
  - Limitation of how the RFA is setup

Additional Implementation

- Added ability to respond to ARP requests
  - “50% ARP”
- Handled by the receive callback function
- *arp.h / arp.c*

2	19.291386	52:55:0a:00:02:02	Broadcast	ARP	42	Who has 10.0.2.15? Tell 10.0.2.2
3	19.295766	0f:0f:0f:0f:0f:0f	52:55:0a:00:02:02	ARP	46	10.0.2.15 is at 0f:0f:0f:0f:0f:0f
4	19.296140	10.0.2.2	10.0.2.15	UDP	49	51740 → 8081 Len=7

Difficulties

- Time and Energy
  - Hard to test
  - Some functionality of the hardware seemed to be assumed in the manual
- Never implemented TCP or ICMP
  - Always a stretch goal, UDP is much easier
- Sarah was remote
  - could not go to DSL labs - had to work on qemu
- Lots of protocols to read
  - PCI, Ethernet, IPv4, UDP
- Memory management
  - Headers and data would overwrite each other
  - Aligning data structures on allocation caused many bugs

Closing Notes

- Happy with what we accomplished
  - Design feels clean and easy to add onto if we wished
- “Front loading” work to the beginning of the semester paid off
  - Gave us a good foundation to build off of
- Valuable experience
  - Wish we could have all been there in person, but what can you do 🤷‍♀️

Questions?