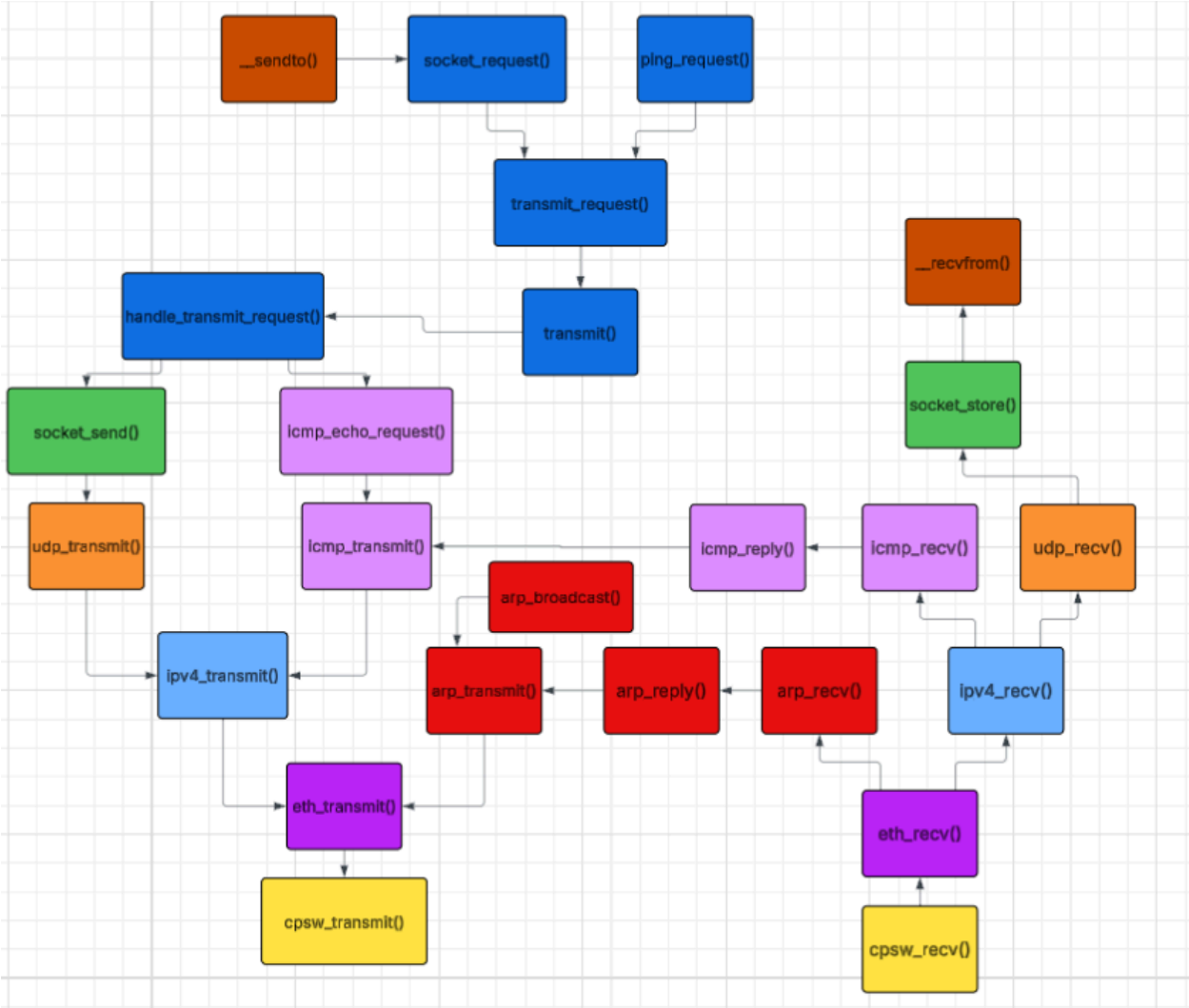
Network Stack

High Level Overview

Provides a Network stack for processes to use, processes can use the socket system call api to create, bind, and use sockets for transmitting and receiving UDP packets. The network stack supports various protocols such as Ethernet II, ARP, IPV4, ICMP, and UDP. The stack uses the Beaglebone Black's Common Platform Switch (cpsw) to send and receive frames over the ethernet cable.



- *Common Platform Switch - Yellow*
- *Ethernet II - Purple*
- *Address Resolution Protocol - Red*
- *Internet Protocol version 4 - Light Blue*
- *Internet Control Message Protocol - Light Purple*
- *User Datagram Protocol - Orange*
- *Upper Half Network Driver - Dark Blue*
- *Sockets - Green | User Socket Syscalls - Brown*

Network Stack Layers

The Network Stack works in a layered fashion which can be seen in the above figure, On frame reception the frame is passed up the stack starting in the cpsw_recv() function. The received frame is passed up to eth_recv() where the frames ethernet header is extracted, from here the frames protocol is checked if it is ARP it is passed to arp_recv(), if it is IPV4 it is passed on to ipv4_recv(). In arp_recv() the ARP header is extracted if the frame is an ARP request for our IP we send a reply. In ipv4_recv() the IPV4 header is extracted and the protocol is checked if it is an ICMP frame it is passed to icmp_recv(), if it is UDP it is passed to udp_recv(). In icmp_recv() the ICMP header is extracted, if the frame is an ICMP echo request we reply to it. In udp_recv() the UDP header is extracted and the destination port is used to check to see if any socket is bound on the port, if there is a socket bound to the port the frame is sent to socket_store() alongside the socket number. socket_store() stores the payload of the UDP frame in the sockets packet queue. User programs can retrieve the payload by calling the __recvfrom() system call with the binded sockets number, the system call returns the first pending packet in the queue.

On packet transmission User programs call the __sendto() with the socket number and frame to be transmitted. The__sendto() system communicates with the upper half network driver and creates a socket transmit request, socket_request() calls transmit_request() and places a socket transmit request on the transmit queue. When transmit() is called it checks the queue for any requests, if there is a request the request is handed over to handle_transmit_request() which checks the request type and calls the appropriate function. On socket requests the request is handed to socket_send() which calls udp_transmit(), the function appends the UDP header to the packet and passes it to ipv4_transmit(). The IPV4 header is appended to the packet and passed down to eth_transmit() which appends the ethernet frame header. The fully crafted frame is passed to cpsw_transmit() which takes the frame from memory and transmits it.

Common Platform Switch

The Beaglebone Black contains a TI Common Platform Ethernet Switch (CPSW) which is a layer 2 three port switch. Its main functionality is to place received frames in memory and transmit frames from memory using CPDMA queues. In order to access the functionality of the ethernet switch BuddyOS provides a driver for the cpsw.

CPSW Driver API

The Kernel has the following functions available to it to use the CPSW:

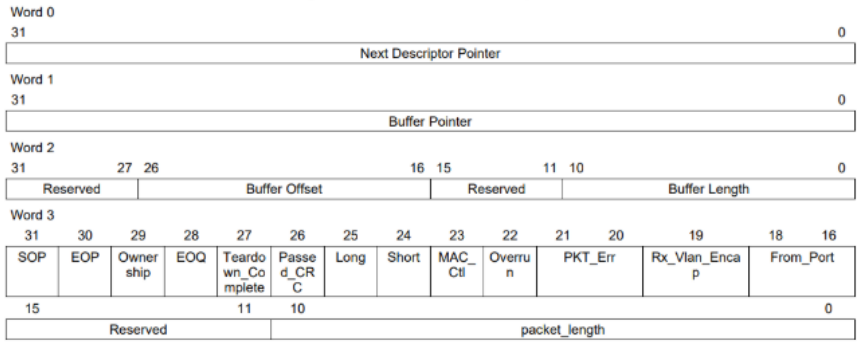
1) *void cpsw_init()*
   Description: *Initializes the CPSW for packet reception and transmission*
2) *int cpsw_recv()*
   Description: *Processes all packets in the cpdma receive queue*
3) *int cpsw_transmit(uint32_t* packet, uint32_t size)*
   Description: *Adds packet to cpdma transmit queue and transmits it*

CPSW Initialization

- Select interface
    - Choose between Reduced Media Independent Interface (RMII) and Media Independent Interface (MII).
    - The Media Independent Interface facilitates the transfer of Ethernet frames between the MAC and PHY layers
    - RMII has less pins then MII
    - Select MII interface by setting bits 0-3 to 0 in the config gmii_sel register
- Pin Muxing
    - Setting the functionality of the multiplexed pins of the cpsw
    - Each pin has a conf_ register where its functionality can be set by setting the specific bits
    - Each pin is set to a fast slew rate by setting bit 6 to 0
    - Each pin pullup/pulldown is enabled by setting bit 3 to 0
    - Each pin is set to pulldown by setting bit 4 to 0 except mdio pins
    - Pins that require input set bit 5 to 1 to enable receiving
    - Mdio pins are set to pull up by setting bit 4 to 1
- Enabling clocks
    - Enabling the 2 clocks the cpsw relies on
    - Set bit 0-1 to 0x2 in CM_PER_CPGMAC0_CLKCTRL register
    - wait until module is functional by reading bit 16-17 and waiting for it to equal 0
    - Set bit 0-1 to 0x2 in CM_PER_CPSW_CLKSTCTRL register
    - wait for clocks to be active be reading bit 4 until it equals 1
- Software reset
    - Software reset each module, this is achieved by setting bit 0 to 1 in the modules reset register. Reset is done when bit 0 is read back as 0 from the same register
    - CPDMA_SOFT_RESET register
    - CSPW_SL SOFT_RESET registers (2 of them)
    - CPSW_SS SOFT_RESET register
    - CPSW_WR SOFT_RESET register
- Initialize cpdma descriptors
    - Set cpdma descriptor registers to 0, this must be done after cpdma reset
    - Set all TX (0-7) and RX (0-7) channels Header Descriptor Pointer (HDP) registers and Completion Pointers (CP) registers to 0
    - Write 0 to TXn_HDP and TXn_CP registers (n = 0-7)
    - Write 0 to RXn_HDP and RXn_CP registers (n = 0-7)
- Configuring control registers
    - Address Lookup Engine (ALE) control register
        - Set bit 31 and 30 to 1 to enable ALE and clear ALE table
    - Management Data Input/Output (MDIO) control register
        - MDIO is how we communicate with the PHY
        - Set bit 30 to 1 to enable MDIO
        - Disable preamble by setting bit 20 to 1
        - Enable fault detection by setting bit 18 to 1
        - Set MDIO clock divisor to 124 by writing it to bits 0-15
    - Statistics control register
        - Disable statistic module by setting all bits to 0

- Read cpsw MAC address
    - MAC_ID0_LO and MAC_ID0_HI contain the cpsw MAC address
- Setting Port States and Mac Addresses
    - Set port states in PORTCTLn registers
    - Set ports 0 and 1 to forward by setting bits 0-1 to 3
    - Setting port 2 to forward causes multicast packets to not work so we do not set port 2s state to forward
    - We also set the ports MAC address to the MAC address of the cpsw by writing to Pn_SA_LO and Pn_SA_HI
- Setting up Address Lookup Engine
    - Create unicast and multicast entries in the address lookup engine table
    - One entry for our MAC address and one for broadcast
    - If an incoming packets destination MAC is not found in the ALE table then the packet is dropped
    - Entries are created by adding each word of the entry (3 words) in the corresponding TBLWn registers
    - Then writing the index in the table to bits 0-9 and 1 to bit 31 in the TBLCTL register will cause the entry to be written to the table
    - The main things to add in the entry are the entry type (unicast or multicast) and the MAC address
- Setup cpdma queues
    - Setup the RX CPDMA queue in CPPI ram
    - The queue consists of a specific linked list like data structure shown below
    - We setup a queue of these descriptors in CPPI ram with each descriptor pointing to the next
    - We also kmalloc a buffer for the descriptor, set buffer length and set the ownership flag, the cpdma control clears this flag once it has written a packet to this descriptor buffer
    - The buffer size is 1520 bytes the max size of an ethernet frame
    - We do the same for TX, however the queue only has one descriptor and we allocate the buffer at transmission time

Figure 14-8. Rx Buffer Descriptor Format



- Enable cpdma controller
    - We enable the cpdma controller by writing 1 to bit 0 of TX_CONTROL and RX_CONTROL registers

- Start reception
    - To start reception we write the address of the start of the descriptor queue we setup to the RX0_HDP register
    - Cpdma controller will use this queue for writing packets to memory

CPSW Reception

The way packet reception is intended to happen is when the cpdma controller is done writing a packet it signals an interrupt then packet is handled via interrupt service routine. Due to a silicon errata interrupts are masked and do not get signaled to the CPU however if you read the INTSTATUS_RAW register for RX you can see when an interrupt is supposed to be signaled but is masked. With this discovery we were able to bypass the silicon errata and when cpsw_recv() is called it polls the INTSTATUS_RAW register to see if a packet has been received, we call cpsw_recv() on every timer interrupt so that packet reception is responsive.

cpsw_recv()

To receive packets we call this function on every timer interrupt, the function reads the RX_INT_STATUS_RAW register to see if packets have been written to memory by the cpdma controller. If there are packets it will start proccesing all descriptors in the RX queue starting from the channel's free descriptor that has the ownership flag unset (meaning the cpdma controller is done using the descriptor). A descriptor’s buffer that contains the packet data is passed into the process_packet() function alongside the packet size. After process_packet() returns we reset the descriptor flags, write the address of the descriptor to RX0_CP and move on to the next. If we reach the end of the RX queue the descriptor's End Of Queue flag will be set, after processing the final descriptor in the queue we rewrite the start of the descriptor chain's address to the RX0_HDP register and set the channels free descriptor to the start of queue to refresh packet processing. After descriptor processing ends we write to CPDMA_EOI_VECTOR register to clear the RX interrupt.

CPSW Transmission

To transmit a packet we write the TX descriptor queue address to the TX0_HDP register, before doing so we write the packet's address to the buffer pointer, set Start of Packet flag, End of Packet flag, and set the packet's length. After transmission an interrupt is supposed to occur however as previously mentioned the interrupt is masked so we poll INTSTATUS_RAW register for TX to see when transmission is done.

cpsw_transmit()

This function is called when we want to transmit our constructed packet, it takes a pointer to the buffer containing the packet and the size of the packet. The packet's pointer is written to our single TX descriptor in the TX queue and the descriptor's flags are set. We then write the descriptor address to the TX0_HDP register to start transmission. After we start transmission we poll the TX_INT_STATUS_RAW register as an interrupt is supposed to be signaled when TX DMA operation is completed. Once transmission has been completed we write the descriptor's address to TX0_CP and write to CPDMA_EOI_VECTOR register to clear the TX interrupt.

PHY

The PHY is responsible for communicating with the physical transmission medium and auto negotiating the link parameters between 2 machines connected by a medium. Packets are Transferred from the CPSW to the MII then to the PHY for transmission over the medium.

PHY Driver

The driver for the PHY resets the PHY, auto negotiates link parameters then sets the link parameters. Due to a hardware bug the PHY doesn't power up on board startup and needs to be reset, the PHY can be reset using GPIO pin 8. This information is not listed anywhere and was found looking at the linux device trees for the beaglebone black.

```
ethphy0: ethernet-phy@0 {
        reg = <0>;
        /* Support GPIO reset on revision C3 boards */
        reset-gpios = <&gpio1 8 GPIO_ACTIVE_LOW>;
        reset-assert-us = <300>;
        reset-deassert-us = <50000>;
};
```

After resetting the PHY we need to autonegotiate link parameters, all communication with the PHY is done using the MDIO. We start Auto Negotiating by writing to the enable bit in the PHYs BCR register, after auto negotiation has been completed we read the PHYs Partner Capabilities register to check what link parameters we can use. We then set the link parameters based on the partners capabilities by writing to PORT1_MACCONTROL register.

Note on Packet Processing

For transmission of packets, the memory for the entire packet is allocated at the start of the stack. The packet is passed down the stack and each protocol's transmit function sets the bytes for its respective header byte by byte. For reception of a packet, the packet is passed up the stack and each protocol's recv function extracts the protocol header byte by byte. This design choice was made due to the Beaglebone Black not allowing unaligned memory access, which went against my initial plan of casting parts of the packet to a struct of the protocol header. Currently need to include destination mac in all transmit functions, would need an ARP table to be able to abstract this.

Ethernet II

eth_transmit(uint8_t* frame, int size, uint8_t* dest, uint16_t type)

- frame
  - Pointer to allocated memory for frame with upper layers headers set
- size
  - Size of frame
- dest
  - Destination MAC address
- type
  - Packet/Frame type (ARP or IPV4)
- Description
  - Sets the Ethernet II header for the frame and sends it to cpsw_transmit() for transmission
  - Sets the Destination MAC to dest and Source MAC to the Beaglebone Blacks MAC address
  - Sets the Frame type to the passed in value of ARP or IPV4

eth_recv(uint32_t* frame, int size)

- frame
  - Pointer to received frame from cpsw_recv()
- size
  - Size of entire frame
- Description
  - Extracts Ethernet II header from frame and stores it in ethernet frame header struct
  - Checks Packet/Frame type (ARP or IPV4)
  - Subtracts the size of ethernet header from frame size
  - Passes the rest of the frame (not including ethernet header) and ethernet header struct up the stack to the respective protocol recv function

Address Resolution Protocol

arp_transmit(uint8_t* frame, int size, uint8_t* dest_mac, uint8_t* dether_mac, uint32_t dest_ip, uint16_t opcode)

- frame
  - Pointer to allocated memory for frame with upper layers headers set
- size
  - Size of frame
- dest_mac
  - Destination MAC address for ARP header
- dether_mac
  - Destination MAC for Ethernet II layer
- dest_ip
  - Destination IP for ARP header
- opcode
  - ARP header OP code (Request or Reply)
- Description
  - Sets the ARP header for the frame and sends it to eth_transmit()

arp_recv(ethernet_header frame_header, uint32_t* frame, int size)

- frame_header
  - Ethernet II header struct
- frame
  - Pointer to frame from ARP header and onwards
- size
  - Size of frame from ARP header and onwards
- Description
  - Extracts ARP header from frame and checks ARP opcode
  - If the opcode is an ARP request call arp_reply() and pass the extracted ARP header
  - ARP replies aren't supported (would store the replies in an ARP table)

arp_reply(arp_header arp_request)

- arp_request
  - Struct of ARP header
- Description
  - Checks if the arp request is for our IP
  - If it is for our IP, uses arp_transmit() to transmit an arp reply

arp_garp()

- Description
  - Abstraction function that uses arp_transmit() to transmit an ARP gratuitous packet

arp_anounce()

- Description
  - Abstraction function that uses arp_transmit() to transmit and ARP broadcast packet

Internet Protocol Version 4

ipv4_transmit(uint8_t* frame, uint16_t size, uint8_t protocol, uint32_t dest_ip, uint8_t* dest_mac)

- frame
  - Pointer to allocated memory for frame with upper layers headers set
- size
  - Size of frame
- protocol
  - Ipv4 protocol (ICMP, UDP, TCP) TCP not supported
- dest_ip
  - Destination ip for IPV4 header
- dest_mac
  - Destination MAC for Ethernet II header
- Description
  - Sets IPV4 header for the packet and sends it to eth_transmit() alongside destination mac

ipv4_recv(ethernet_header frame_header, uint32_t* frame, int size, uint8_t* frame_ptr)

- frame_header
  - Ethernet II header struct
- frame
  - Pointer to frame from IPV4 header and onwards
- size
  - Size of frame from IPV4 header and onwards
- frame_ptr
  - Pointer to start of full frame, used for very specific case when we re use the buffer of the received frame for transmission (ICMP echo reply)
- Description
  - Extracts IPV4 header from frame
  - Checks the protocol type (UDP or ICMP) and calls respective protocol recv function

ipv4_checksum(uint8_t* ipv4_header, int size)

- ipv4_header
  - Pointer to ipv4 header
- size
  - Size of header
- Description
  - Computes ipv4 checksum following RFC1071
- Return
  - uint16_t checksum

Internet Control Message Protocol

icmp_transmit(uint8_t* frame, int size, uint8_t type, uint8_t code, uint32_t data, uint32_t dest_ip, uint8_t* dest_mac)

- frame
  - Pointer to allocated memory for frame with upper layers headers set
- size
  - Size of frame
- type
  - Packet type (echo reply or echo request)
- code
  - Packet code (echo reply code or echo request code)
- data
  - Icmp header data field
- dest_ip
  - Packet destination ip
- dest_mac
  - Destination mac for Ethernet II header
- Description
  - Sets ICMP header for packet and sends it to ipv4_transmit()
  - Uses ipv4_checksum() with icmp header to compute icmp header checksum

icmp_recv(ethernet_header eth_header, ipv4_header ip_header, uint32_t* frame, int size, uint8_t* frame_ptr)

- frame_header
  - Ethernet II header struct
- ip_header
  - IPv4 header struct
- frame
  - Pointer to frame from ICMP header and onwards
- size
  - Size of frame from ICMP header and onwards
- frame_ptr
  - Pointer to start of full frame, used for very specific case when we re use the buffer of the received frame for transmission (ICMP echo reply)
- Description
  - Extracts ICMP header from frame
  - Checks packet type if it is a ICMP echo request it call icmp_echo_reply() reusing the requests buffer to send the reply

icmp_echo_reply(ethernet_header eth_header, ipv4_header ip_header, icmp_header icmp ,uint8_t* frame, int size)

- frame_header
  - Ethernet II header struct
- ip_header
  - IPv4 header struct
- icmp
  - Icmp header struct
- frame
  - Pointer to frame from ICMP header and onwards
- size
  - Size of frame from ICMP header and onwards
- Description
  - Sends an ICMP echo reply to the requester

icmp_echo_request(uint32_t ip, uint8_t* mac)

- ip
  - Destination ip
- mac
  - Destination mac of Ethernet II header
- Description
  - Sends an ICMP echo request to the specified ip
  - Abstraction that uses icmp_transmit()

User Datagram Protocol

udp_transmit(uint8_t* frame, uint16_t size, uint16_t src_port, uint16_t dest_port, uint32_t dest_ip, uint8_t* dest_mac)

- frame
  - Pointer to memory allocated for frame with Payload set
- size
  - Size of memory allocated for frame
- src_port
  - Source port for UDP header
- dest_port
  - Destination port for UDP header
- dest_ip
  - Destination ip for IPV4 header
- dest_mac
  - Destination MAC for Ethernet II header
- Description
  - Sets UDP header for packet and sends it to ipv4_transmit()
  - Computes UDP header checksum using pseudo header (calls udp_checksum())

udp_recv(ethernet_header eth_header, ipv4_header ip_header, uint32_t* frame, int size)

- eth_header
  - Ethernet II header struct
- ip_header
  - IPv4 header struct
- frame
  - Pointer to frame from UDP header and onwards
- size
  - Size of frame from UDP header and onwards
- Description
  - Extracts UDP header from frame
  - Checks if a socket is bound on the destination port of the packet
  - If socket is bound on the port it stores the payload in the sockets queue

udp_checksum(uint8_t* frame, uint32_t src_ip, uint32_t dest_ip, uint16_t size, uint16_t src_port, uint16_t dest_port)

- frame
  - Pointer to memory allocated for frame with Payload set
- size
  - Size of memory allocated for frame
- src_ip
  - Source IP for UDP pseudo header
- src_port
  - Source port for UDP pseudo header
- dest_port
  - Destination port for UDP pseudo header
- dest_ip
  - Destination ip for UDP pseudo header
- Description
  - Computes UDP checksum by creating a pseudo udp header and calling ipv4_checksum()
- Return
  - uint16_t checksum

Sockets

socket(uint32_t pid, uint8_t* dest_mac, uint8_t protocol)

- pid
  - Pid of process
- dest_mac
  - Destination MAC for Ethernet II layer
- protocol
  - Socket packet protocol (only UDP supported)
- Description
  - Finds a free socket in the socket table, sets up some of the basic struct attributes and returns socket number
- Return
  - int socket_num

socket_recv(int socket_num)

- socket_num
  - Socket number
- Description
  - Returns the pointer to first payload in sockets receive queue

socket_send(int socket_num, uint8_t* frame, int size)

- socket_num
  - Socket number
- frame
  - Pointer to frame with payload set
- size
  - Size of frame
- Description
  - Calls udp_transmit() using sockets stored attributes

Sockets API

__socket(int pid, uint8_t* gateway, uint8_t protocol)

- pid
  - Pid of process
- gateway
  - Destination MAC for Ethernet II layer
- protocol
  - Socket packet protocol (only UDP supported)
- Description
  - User system call to create a socket
- Return
  - int socket_num

__bind(int soc, socket_info *soc_info)

- soc
    - Socket number
- soc_info
    - Pointer to socket info struct
    - Contains bind port (what port you want to bind on)
- Description
    - User system call to bind socket to a port
    - Binds socket to port which allows packets to be stored in sockets receive queue

__closesocket(int soc)

- soc
    - Socket number
- Description
    - User system call to free socket

__recvfrom(int soc, uint8_t* buff)

- soc
    - Socket number
- buff
    - Buffer to copy payload into
- Description
    - User system call to copy first payload in sockets receive queue into the provided buffer
    - Returns number of bytes copied
- Return
    - int size

__sendto(int soc, uint8_t* payload, int size, socket_info *soc_info)

- soc
    - Socket_number
- payload
    - UDP payload
- size
    - Size of payload
- soc_info
    - Pointer to socket info struct that contains dest ip and dest port
- Description
    - User system call to create socket transmit request
    - Allocates memory for frame and copies payload into it
    - Creates a socket transmit request using upper half driver

Upper Half Driver

To multiplex the Network stack BuddyOS provides an upper half driver. The upper half driver uses the transmit() function to check the transmit request queue for any requests and handles them by calling the respective protocol transmit function. The types of requests that can be made to the upper half driver are socket requests and icmp echo requests. The transmit() function is called on every timer interrupt alongside cpsw_recv() to keep the network stack responsive.

Network Testing

To test the network stack a couple of tools and test programs were made/used. First is wireshark, this was used to check that the transmitted packets fit the protocol standards and that the checksum were correct. To implement the network stack I would hex dump the received packets and compare them to wireshark, essentially reverse engineering the protocol headers using wireshark. Second was Packet Sender, a tool that allows you to send custom packets. Third is ping which allows you to send icmp echo requests and arp -a which allows you to see cached arp tables. Lastly schat, this was developed to be cross compatible with windows and BuddyOS using socket abstraction functions and our own stdlib system calls. The same c file is able to be compiled for windows and BuddyOS. Schat tested our socket implementation.