Bryan Werth

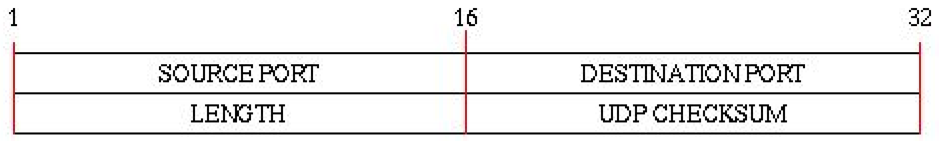
UDP/IP Final Writeup

UDP/IP Overview

To understand Ethernet communications protocols such as UDP, it is important to understand how network communications design is abstracted. The open system interconnection (OSI) model, the primary recognized model, divides network communications into seven different layers. Layers one through four, physical, data link, network, and transport, are lower level, meaning they are mostly concerned with moving data around. The physical layer conveys the actual bit stream. The data link layer, which includes media access control (MAC) and logical link control (LLC), handles low level packet flow. The MAC controls how the network gains access to data and the LLC handles frame synchronization as well as low-level flow control. Also, the network layer provides the actual routing and switching involved in transmitting data from node to node. In addition, the transport layer provides transparent transfer of data between end systems. Layers five through seven, session, presentation, and application, are higher level, meaning they contain application-level data. The session layer establishes, manages, and terminates connections between applications. The presentation layer translates network data into a format that can be accepted and understood by the application. Finally, the application layer includes everything specific to the actual application.

User Datagram Protocol (UDP) is a very thin Ethernet protocol built on top of the internet protocol (IP), that I will be implementing over the course of this project. It is an end-to-end protocol in that it contains enough information to transfer a user datagram from one process on the transmitting host to another process on the receiving host. UDP is used when reliable delivery is not required, because unlike TCP, which is the other common Ethernet protocol, there is no required acknowledgement of packet reception. As a result, UDP makes no guarantee that the transmitted data will be received properly by the end system. Applications that would benefit from a protocol designed this way value communication speed over reliability of data. For example, streaming media commonly uses UDP because speed of communication is far more important than the accuracy of the data itself.

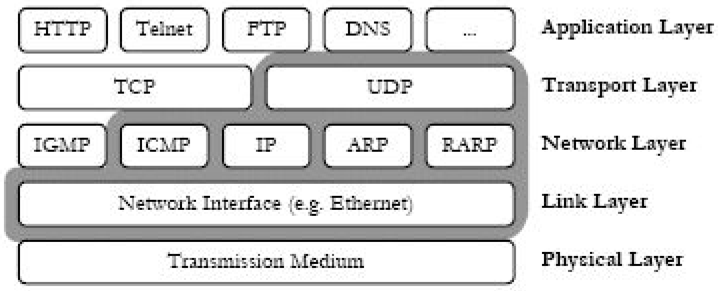
As a result of the reduced connection and error checking, a UDP packet (shown below) is much smaller than a TCP packet. Computers use ports (16 bit field) to identify the specific application that a network connection is accessing. Within a UDP packet, the source port is the port number used by the source host while the destination port is the port number used by the end application. Checksum is optional in UDP, with the field set to zero if intended to be unused. It is notable that the UDP protocol appends a 12-byte pseudo header with IP addresses of source and destination as well as a few other additional fields. The receiving host appends its own IP address with the source IP address in the same way before checking to see if the checksum matches. If it does, the receiving application knows the message has reached its destination.



1	16	32
SOURCE PORT	DESTINATION PORT	
LENGTH	UDP CHECKSUM	

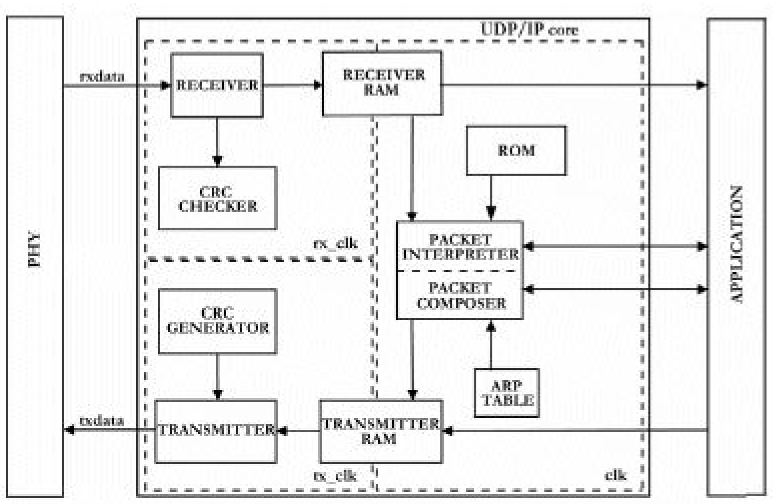
Network communications projects can span all seven of the OSI layers, but Ethernet protocols are usually implemented in transport, network, and data link layers. For the purposes of this project, we primarily care about implementing UDP (transport layer) and IP (network layer), where UDP handles end to end communication and IP handles the actual routing and switching of the message.

An early design consideration in implementing UDP/IP is whether to build the communication protocol in a microprocessor or an FPGA. Although FPGAs are faster and have a higher bandwidth, it is sometimes impractical either for size, design, or cost requirements to fully implement in hardware. In this case, UDP/IP is commonly implemented completely within hardware because of how simple it is. However, UDP/IP modules interact with other modules that are implemented in software such as any application layer systems. In addition, TCP/IP is usually partially implemented in software because of how complicated the error and connection is. In this case, although the underlying internet protocol is probably still built on an FPGA, the actual transport layer TCP module is design on a microprocessor.



UDP/IP Implementation

There are a few critical sub-modules that make up a typical UDP/IP core (seen below):



Receiver: detects new packets and checks CRC as well as destination MAC-address, discarding the packet if either check fails

CRC checker: calculates the cyclic redundancy check, an error-detecting code commonly used to check for accidental changes in raw data

Receiver RAM: temporarily stores the entire packet, bad CRC's and MAC-addresses are filtered out in the receiver

ROM: non-changing control values are stored here to save space

Packet Composer/Interpreter: the main UDP control block, which manages incoming and outgoing packets, validating UDP packets as well as managing ARP and ICMP requests and responses

Address Resolution Protocol: Stores IP and MAC addresses for reference

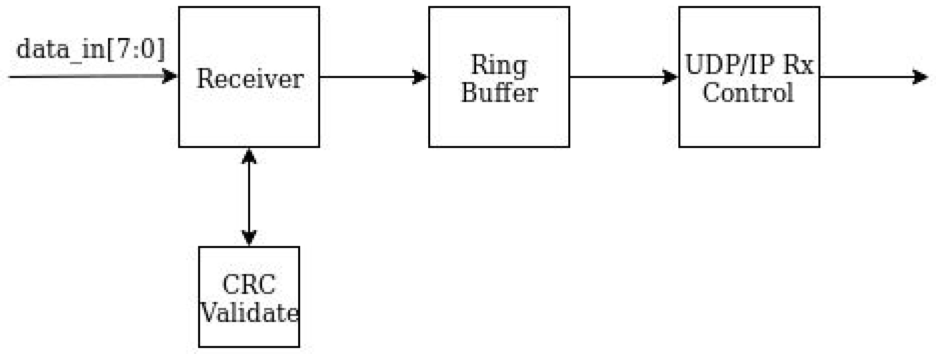
Transmitter RAM: temporarily stores entire packet before transmitting

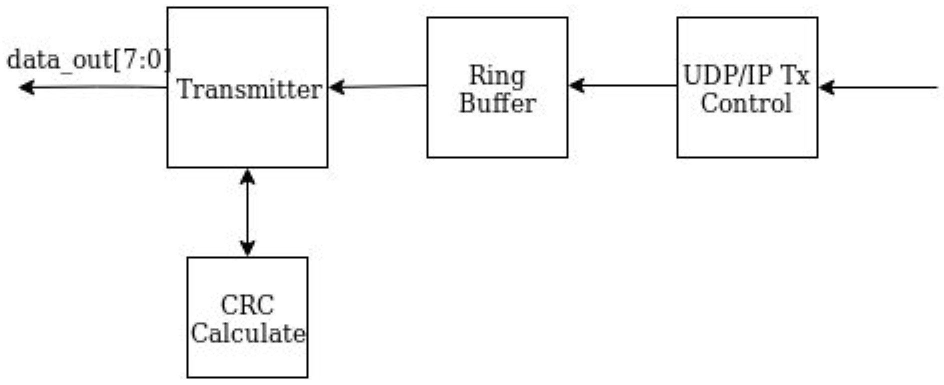
Transmitter: sends the generated packet along with the calculated CRC that is appended at the end

In my version of UDP/IP, some novel design decisions were made to make success more achievable. Most significantly, I decided not to integrate an ARP Table into the project. I considered the interesting part of this project to be the development and iteration of a packet validation protocol. By removing some of the complexity of fully realizing the UDP/IP protocol, I gave myself more time to iterate on and optimize my design. In addition, some sacrifices were made in the durability of the design. For example, I did not include a ring buffer full detector to indicate when the ring buffer is full. For this reason, there are strict requirements on the size and frequency of communication for the udp/ip receiver and transmitter to remain in operation. Despite these design decisions that reduce the usability of the final product, it would be fairly straight forward to produce something better now with a working full-pipeline design.

Test Strategy

I used an incremental design validation strategy to confirm the functionality of my UDP/IP receiver and transmitter. For each separate module that was designed, I developed a test bench that was specifically designed to confirm the functionality of that module. For some of these modules, I broke out internal control signals to confirm the state changes within the module. With all of the modules separately confirmed, I gradually connected them while testing each incremental version until everything was connected. First I connected and tested the crc receiver/transmitter and ring buffer. Following this, I connected the actual UDP control module to the ring buffer and tested the complete transmitter on its own. Finally, I confirmed the full receiver by connecting it to the transmitter. With a confirmed ability to input data to the transmitter and receive it correctly at the output of the receiver, I was satisfied with the functionality of the full design.





Design Review

Although the designed UDP/IP transmitter and receiver work reliably given some input restrictions discussed above, there is room for improvement in the speed and efficiency of the final product. First, each of the three major sub-modules, CRC, ring buffer, and udp control, is designed to only process one packet at a time. For example, the UDP control module handles gradual serial reception of a packet, UDP and IP checksum calculation, and gradual transmission of confirmed packet. The throughput of the UDP control module is reduced by the fact that only one packet can be processing within the module at a time. Pipelining this design would allow for a packet to be in the process of being received while another packet is having UDP and IP checksum confirmed and a fourth packet is being transmitted, drastically increasing throughput. Implementing similar improvements to the CRC module would have a similar effect.

In addition, flexibility of input packet size as well as module latency is worsened by the way packets are aggregated. Within the CRC module, each packet is received from serial into a two-dimensional byte array to be transmitted to the CRC validation sub-module in parallel. Packet size is confirmed by waiting for the last byte control line to be driven high. Also, the design does not rely on the packet length fields at all. As a result of both these facts, packet size is restricted by the size of the two-dimensional byte array. Additionally, latency is increased by the fact that the entire packet has to be received before processing can begin. I could solve this problem by avoiding aggregation of data into a two-dimensional byte array entirely. Instead, I could handle processing gradually as the packet is received while relying on the packet length fields to determine how many bytes to wait for. On the output of each module, I could add a FIFO (first in first out) with an enable to allow for packets to pass if data reliability is confirmed. This way, data aggregation would happen independent of actual processing, allowing for lower latency and better packet length flexibility.

Conclusion

This project was largely a success, but there is room for improvement to the design in the future. I produced a working UDP/IP receiver and transmitter, but there were some sacrifices to functionality, reliability, and performance in the process of achieving what I did. Moving forward, I am going to integrate an ARP table module to allow for full Ethernet compatibility. I will also pipeline my design for better throughput and remove up-front data aggregation as previously discussed.