Data Transmission Protocols

CS 3410 Systems Programming

The OSI Model

One common way of thinking about communication between computers is the *OSI Model*. Traditionally, there are 7 layers:

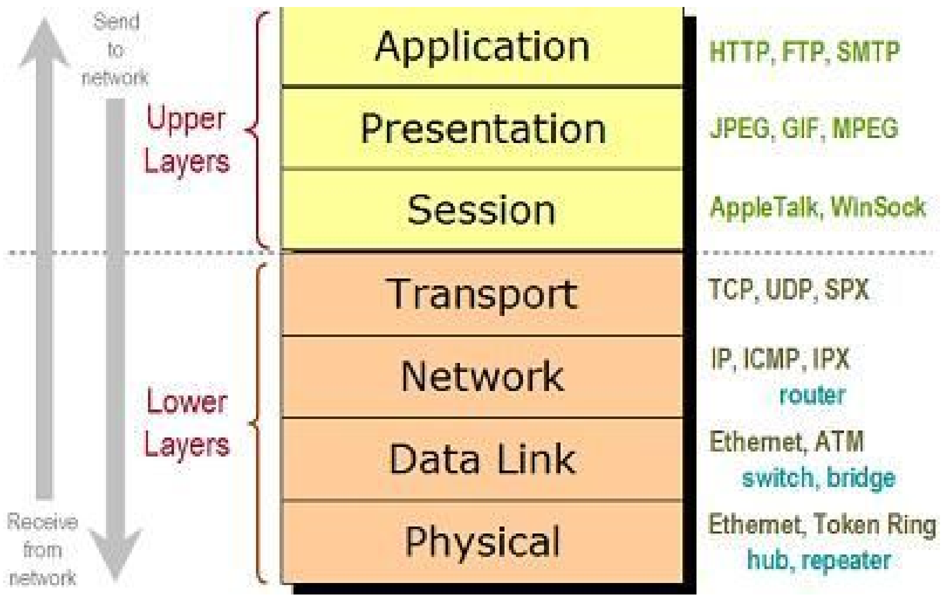

Figure 1: 7 Layers of the OSI Model

As you may be able to deduce, our Serial connection is at the Data Link level - it ensures that the bytes we send between our host program and the Arduino actually arrive. We can assume that the data link layer is capable of transmitting entire *frames* of data without error. In our case, a frame is a single byte of data. In a traditional network, the Ethernet protocol is at the Data Link layer. For this project, we are implementing a protocol at Layer 3/4: the network/transport layers. Our protocol must impose structure on the data we're sending between devices. This structure can include reliability, segmentation, and acknowledgement of transmitted packets. In traditional networks, this is done by the IP and TCP/UDP protocols. More information about the OSI model can be found on Wikipedia.

Framing

As outlined above, a *data frame* is one transmission unit of data (at the Link layer level). The link layer makes no guarantees about the ordering of frames. As we have done in the past, sending one frame of data between devices is easy. However, complications arise when you want to send a *packet*, composed of multiple frames. When listening on a serial port, you only see a stream of bytes go by. How do you know when a packet starts or ends? How about the length of a packet?

One simple solution is denoting byte flags for the start and end of a packet. After your program reads the (predetermined and constant) start byte, it knows that all following bytes are a part of the message until the ending flag is read. However, what happens when your application actually wants to send one of the flag bytes without confusing the protocol? One solution is to prepend all flag bytes with a different `ESC` byte that the protocol knows to ignore and interpret the following frame literally. For example, in many programming languages, if you want to have a string literal that contains a quote character, you would escape the quotes with a backslash. This technique is called *byte stuffing*. If you want to build more features into your protocol, you can increase the size of the packet *header* and *footer* (the constant size start and end sections of a packet).

Error Handling

It is often a good idea to implement some sort of data integrity check in your protocol. You want to ensure that you are properly accounting for all bytes (that none are missing or wrongly ordered). This is an important sanity check, since it prevents your system from taking potentially devastating action based on incoming garbage data. Two possibilities for implementing this idea are checksums and cyclic redundancy checks (CRC).

Another approach to handling errors (often in conjunction with the data integrity checks above) is for the receiver to send an `ACK` or `NACK` after verifying incoming packets. The receiver would validate incoming packets by calculating the CRC independently and ensuring it matches the one received. If they match, send an `ACK` message back. If there is an error, send a `NACK` which prompts the sender to attempting transmitting the message again.

Packet Ordering

Once we allow messages to be retransmitted upon error, there is the possibility than one message is received and processed twice due. We don't want to wrongly double process a message, so we'll need a way of deduplicating packets. One common approach (used by TCP) is sending a *sequence number* with each packet. That way, when the receiver sees an incoming packet with a duplicate sequence number, it knows that packet can be safely ignored.

Example Protocol

One commonly used protocol in embedded systems is called Point-to-Point Protocol (PPP). PPP is very similar to the idea outlined above. It is a simple and naive protocol lacking error checking features, but it may serve as a good start for your own protocol. It has the following characteristics:

- Both the start and end of a message is denoted with the byte `0x7E`
- The escape byte is `0x7D`
- Whenever a flag or escape byte appears in the message data, it is predeeded with `0x7D` and XOR-ed with `0x20`

When implementing a protocol for your project, you should *not* simply take and implement an existing protocol. PPP as defined above will likely be too simple (and lacking in reliability features) for you to use. You must design your own protocol (although you can take inspiration from existing protocols and add other features "on top"). Your Part 2 write-up must explain from where you took inspiration and why you think the idea has value.

Further Reading

- Serial Communication framing
- Point-to-Point Protocol
- Consistent Overhead Byte Stuffing
- Forming Serial Packets
- Serial Error Correction
- Error Detection Methods