Accelerated Learning Series

(www.ALearnOnline.com – A site dedicated to education)

Modules in Computer Science & Engineering

Concepts in Digital Communications

A set of notes detailing fundamental concepts in Computer Communications.

Author: SKG.

Dec 2010.

Revision History

Version 1.0 (Dec 2010)

- First version created

Table of Contents.

Prerequisites

- General Purpose Computer Architecture (ALS notes in Hardware Engineering)

Preface

Scientific development almost always builds upon previous knowledge. This process is often referred to as “innovation”. Most of the time innovation happens in small distinct steps and on rare occasions innovation can happen in a sudden leap. I like to refer to those sudden leaps as “fundamental innovation” or an “invention”. Fundamental innovations can be characterized as an almost magical alteration in the way we perceive possibilities.

Looking back at the last couple of centuries we observe a series of fundamental innovations that starts with the widespread use of electrical and fossil fuel energy. The following is a list of some of the inventions that changed human life over the last 200 years:

1800: Electric Battery (Alessandro Volta - Italian)
1822: Mechanical Calculator (Charles Babbage – British)
1827: Camera (Joseph Niepce – French)
1868: Typewriter (Christopher Sholes – American)
1876: Internal Combustion Engine (NicolausAugust Otto - German)
1877: Light Bulb (Thomas Edison - American)
1876: Telephone (Alexander Graham Bell – Scottish/Canadian/American)
1895: Radio (Guglielmo Marconi – Italian)
1945: Stored Program Computer (John von Neumann – Hungarian/American)
1947: Semiconductor Transistor (John Bardeen, Walter Brattain, William Shockely – Americans)
1969: ARPANET – predecessor to the Internet (DoD – America)

The dates and inventors noted above are debatable, but what is not debatable is that these were some of the inventions that changed the way we live and the way we conceive of possibilities. It is fascinating to stop and think of what it would have been like to live at a time when any one of those fundamental innovations came to pass. Fortunately for us, we are living at such a moment in human history.

History will undoubtedly record data processing and data communications as the fundamental innovation of the late 20th century. Today we take for granted the ability to communicate via emails. But even as late as the eighties and early nineties, it was a form of communication reserved for the laboratories. I still recall the first time I sent an email to a classmate of mine in engineering school in the late eighties and got a response back almost instantly. My classmate was sitting on a chair next to mine in the laboratory. I heard the beep when he received my email. I glanced over his Cathode Ray Tube (CRT) Monitor and confirmed that what he was seeing was exactly what I had typed in the email I sent and then I watched as he typed his response and was struck in awe as I heard the beep on my terminal almost as quickly as he could press his “Return” key. It was a truly magical experience.

Over the last twenty years I was blessed with the opportunity to work at different layers and with different technologies in data communications; it has left me with an even greater appreciation for this magical ability

to transmit and receive information at lightning speed. To this day I feel that same magic every time I send that first command to a remote device and get a response back. To me, nothing has done more to expand the horizon of human possibilities than data processing and communications. It is the culmination of generations of investment in mathematics, physics, chemistry and engineering and it opens the doors to new frontiers in human discovery. While scientific innovation is often appreciated for its immediate utility to mankind, what is perhaps more profound is that it brings us that one step closer to uncovering the ultimate mystery of nature. There is no denying that data processing and communication will be recognized as key milestones and invaluable tools in that ultimate pursuit.

In this set of notes on Digital Communications we will cover fundamental concepts that allow the transmission and reception of data and then discuss how modern technologies exploit these concepts. The focus will be to relay a conceptual understanding of how things work and what communication terminologies mean, rather than emphasizing the mathematical rigors of background knowledge. The ultimate goal is to impart an interest in the subject.

Acknowledgements

I am sincerely grateful to A. Walker and M. Benson for taking the time to review this document and for providing very valuable and constructive feedback.

1.0 Communication Concepts

Digital communications refers to the field related to sending discrete information from a source to a destination. There are several aspects that factor into the successful communication of information. To start with, we need to acquire discrete data. Then we need to convert the data into a signal that can traverse the medium between the source and the destination. This signal has to be transmitted over the medium and finally the destination has to receive this data and authenticate its integrity.

In this section, we will explore the different engineering processes involved in realizing the goal of sending information from a source to a destination.

1.1 Terminology

At its most rudimentary level, communication involves the moving of information (or data) from a source to a destination.

The source acquires the data, encodes a signal representing the data and finally transmits the encoded signal over some media.

The destination receives the encoded signal, extracts the data from the signal, confirms the integrity of the transmitted data and finally avails the data.

As you can see there are five distinct terms that come up in this definition of communication:

1. Data
2. Media
3. Signal
4. Transmitter
5. Receiver

1.1.1 Data

Data refers to the information that is being communicated. In general, most of the information in nature is constantly changing, though the pace of change may vary depending on a number of factors including the subject of the information.

We can represent constantly changing Data in a **Continuous** or **Discrete** form. A Continuous representation involves tracking the data at every instant in time. A Discrete representation involves taking a sample at certain intervals in time. **Digital communications** deals with the Discrete representation of data, while **Analog communications** deals with Continuous representation of data.

1.1.2 Media

Media refers to the bridge between the source and the destination of Data. Sometimes the source and destination are tethered and other times they are simply separated in space.

1.1.3 Signal

In the field of data communications, a **Signal** almost always refers to an electromagnetic wave form. This is because an electromagnetic wave can traverse most media.

1.1.4 Transmitter

Transmitter refers to the module responsible for sending a signal over a media.

1.1.5 Receiver

Receiver refers to the module responsible for receiving a signal over a media.

1.1.6 Simplex

If Data is only transmitted from a source to a destination, but not from the destination to the source, we refer to such a communication system as a **Simplex** communication system.

1.1.7 Half-Duplex

If Data can be transmitted in both directions between two communicating devices, but only in one direction at a time, we refer to such a communication system as a **Half-Duplex** communication system.

1.1.8 Full-Duplex

If Data can be simultaneously transmitted in both directions between two communication devices, we refer to such a system as a **Full-Duplex** communication system.

1.1.9 Connection-oriented

If a logical or physical connection is first established before data is exchanged between two communication devices, we refer to this as a connection-oriented communication protocol.

1.1.10 Connectionless

If a message can be sent between two communication devices without the prior establishment of a logical or physical connection, we refer to this as a connectionless communication protocol.

1.2 Data Concepts

1.2.1 Data Sampling

Representation of naturally continuous data in a discrete form involves **sampling** the continuous data at certain intervals in time. This results in a number of data samples defined by the sampling interval. The longer the sampling interval, the less the number of data samples.

Take a 100Hz sine wave as our data. This sine wave has a period of 0.01s. (Note: Period is the reciprocal of the frequency - 1/100=0.01)

If we took samples of this wave every 0.01s, we will get exactly the same value for every sample.

If we took samples at intervals greater than 0.01s, we will never get two samples within any given period.

If we took samples at intervals less than 0.01s, we will get more than 1 sample per period.

Intuitively we can guess that the more number of samples we have within a given period, the more accurate will be the discrete representation of our continuous data.

If two different continuous signals are indistinguishable after they are sampled into discrete signals, we know that our sampling rate was inadequate and this phenomenon is referred to as **aliasing**.

The Swedish Engineer, Harry Nyquist, studied this problem and concluded mathematically that you can avoid aliasing by sampling at twice the highest frequency content in the data. This is often referred to as the **Nyquist sampling rate**.

In our example, the data is a pure sine wave of 100Hz. Hence a sampling rate of 200Hz or 0.005s would be adequate to avoid aliasing.

1.2.2 Time Domain vs. Frequency Domain

In the real world however, data is never a pure sine wave whose frequency is known to us. How do we determine a sampling rate in those cases?

Fortunately for us, the French mathematician, Joseph Fourier proved in 1822 that any continuous function can be represented as the sum of an infinite series of sine functions. In other words any data observed in the time domain can be represented in the frequency domain as consisting of an infinite number of

frequencies. In the case of a pure sine wave the strength of the all other frequencies, other than the frequency of the pure sine wave, will be zero. **Fourier transform** refers to the technique of taking a time domain signal and identifying its constituent frequency components.

As it turns out, the analysis of data in the frequency domain is often more useful in communications than the analysis in the time domain. One reason for this is that the medium of transmission can have inherent frequency limitations. In other words, certain media will only allow certain frequencies bands. Yet another reason is that by limiting certain transmissions to certain frequencies we can multiplex the use of the frequency bands available in a given transmission medium.

1.2.3 Bandwidth vs. Bits-per-second

When referring to data transmission throughput we use the unit of **bandwidth** in the frequency domain and the number of **bits-per-second** in the time domain. Let us use an example to elaborate on these units.

Assume we have a digital bit stream of "101010…". Further assume that we generate a rectangular voltage pulse where the "1" is represented by 1v and a "0" is represented by a -1v. We choose this pattern for mathematical ease, since it makes this a periodic signal. Periodic signals have certain properties that help in frequency domain analysis and knowledge thus gathered can be extrapolated to non-periodic signals as well.

We then construct a sine wave with the same period as our rectangular periodic pulse and another sine wave with 1/3 the period of the rectangular periodic pulse and half its amplitude (assuming both the waves share the same phase). We then add these two sine waves. The result closely approximates the rectangular periodic pulse (See figure below).

We could continue to add sine waves with 1/5 the period and then 1/7 the period and so on, each with lesser amplitude (representing lesser energy). Our approximation to the original rectangular pulse will get better with each addition. **As it turns out, what we have done is extract the individual frequencies in our periodic rectangular pulse.** The fundamental frequency is the same as the frequency of the periodic rectangular pulse. Subsequent odd multiples (also known as harmonics) of the fundamental frequency represent the additional frequencies.

Using frequency filters, we can limit the number of harmonics that we want to maintain. Note that higher harmonics have lesser energy and thus eliminating them will have lesser impact on the quality of the signal.

Assume the original bit stream had a period of 10^{-3}s (frequency of 1KHz) and in 1 period we are sending 2 bits (a "1" and a "0"). This would mean that in the time domain we are transmitting $2x10^{3}$bps or 2Kbps.

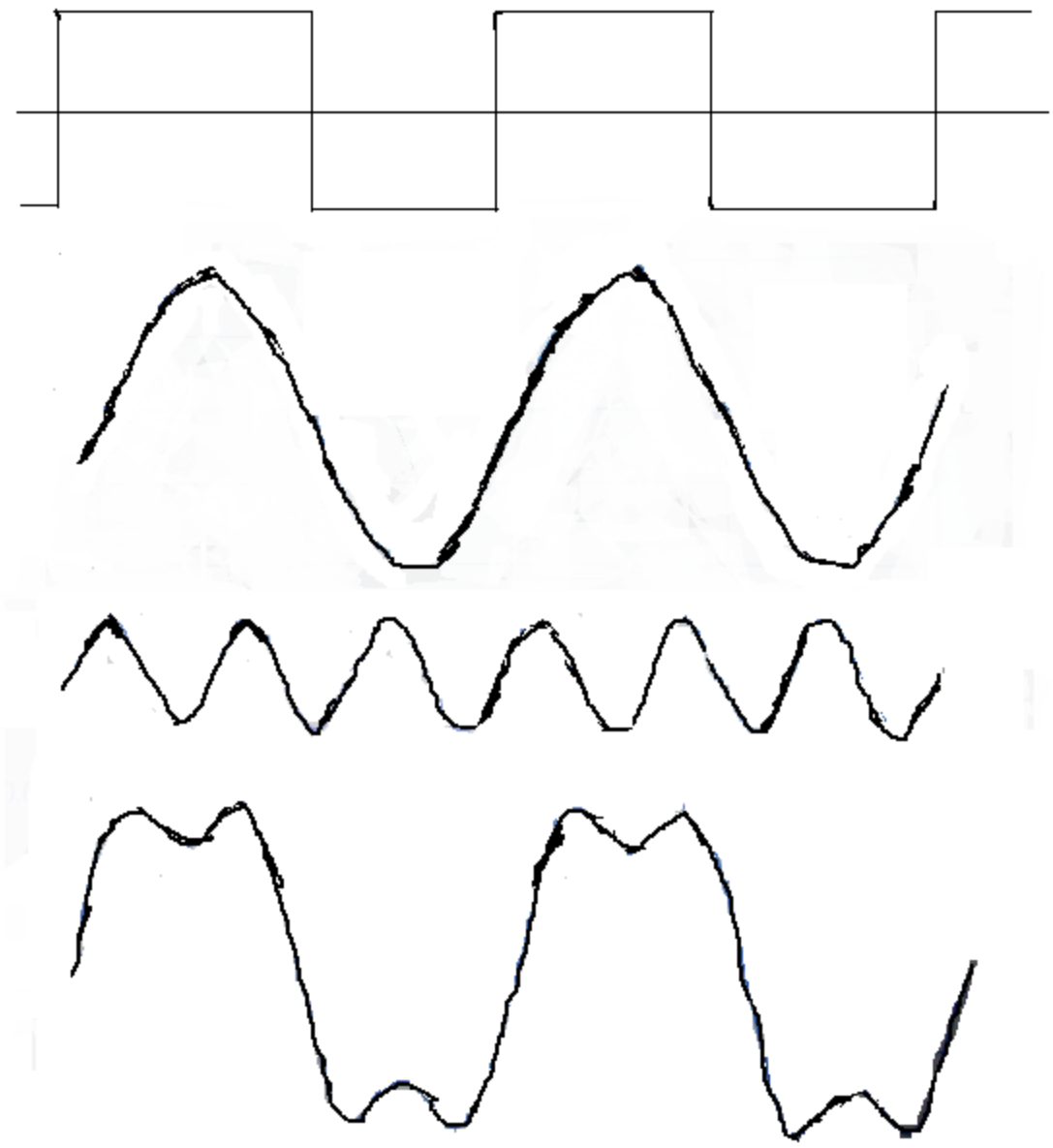

Figure 1.2.3.1

If we used a frequency filter that only allowed 4 harmonics and the fundamental frequency to be transmitted, we would have the following frequencies:

1 KHz
3 KHz
5 KHz
7 KHz
9 KHz

This would work out to a 8 KHz (9 -1) bandwidth.

In other words a bandwidth of 8 KHz in this case works out to a transmission speed of 2 Kbps in the time domain.

The number of harmonics we choose to preserve depends on the original signal. We need as many harmonics as is needed to recoverably approximate the original time domain signal.

1.3 Signal Concepts

Once data is sampled, we need to convert this data into a signal. This is because we can't send number representations directly over a medium. What can be sent over a medium is an electromagnetic wave. Hence we need to generate an electromagnetic wave that embeds in it the sampled data. Such an **electromagnetic wave** is referred to as a **signal**.

Sampled data is also referred to as **discrete** or **digital data**. The signal that we generate to represent digital data can be either discrete or continuous (also referred to as analog).

In the example in our previous section where we had a bit stream of "101010…", we converted this to a discrete signal using a voltage of 1V to represent a "1" and a voltage of -1V to represent a "0". We refer to this as a **discrete signal** because the signal does not take on random voltage values. The signal will always be at either a 1V or a -1V (or some predefined set of discrete values). The process of converting discrete data to a discrete signal is referred to as "**encoding**". At the receiver, the encoded signal is "**decoded**" to retrieve the original discrete data. The encoding scheme used is often designed to optimize the bandwidth consumption over a particular medium. The throughput of an encoded signal is measured in **bits-per-second**.

Discrete data can also be used to generate a continuous signal. A **continuous signal** refers to a signal that is not limited to a set of voltage values. They are a continually changing function of time and usually can take on an infinite number of values, often within a range of electrical parameters. The conversion of discrete data to a continuous signal is referred to as "**modulation**". Modulation involves embedding the discrete data over a carrier signal with a specific frequency. A **carrier signal** is a continuous signal defined by a **frequency**, **amplitude** and **phase**. The process of embedding discrete data over a carrier signal invariably involves an encoding operation on one of these three frequency domain parameters. The throughput of a modulated signal is measured in "**bauds**", which represents the signal element change rate per second. The signal element that is used will depend on which of the three frequency parameters is used in modulation.

Though modulation equipment used to generate continuous signals are more expensive and more complex than encoding equipment used to generate a discrete signal, sometimes the transmission media will dictate the use of a continuous signal instead of a discrete signal. Eg. Transmission between devices that are not tethered by conductive cables is always a continuous signal.

1.3.1 Discrete Signals

Discrete signal encoding schemes are guided by the ease in decoding these signals. We will now discuss some of the common encoding schemes. There are two primary challenges that a receiver faces: Noise and

clock drift. Noise impacts the quality of the signal while clock drift causes the receiver to sample the data at an incorrect location in time. Each of the encoding schemes below address these two issues with varying levels of success.

Non-return to Zero (NRZ) is perhaps the easiest way to encode discrete data. This is similar to what we discussed in our earlier bit stream example. Here we define two voltage levels to define a “0” and a “1”. The voltage level remains unchanged for the duration of the bit transmission interval and then the voltage level switches to the level corresponding to the next bit. Note that there is no reference transition voltage level between the two defined voltage levels (hence the term non-return to Zero). To decode such a signal, the receiver must be synchronized in time with the transmitter so that it knows where each bit starts and ends. The receiver will sample the bit as close to the mid-point of each bit transmission. The receiver can re-synchronize its clock with the transmitter at each transition in voltage level.

Non-return to Zero – Invert on zeros (NRZI) is a slight variation on NRZ in that the transition to a different voltage level is not based on the value of the data but on the change in the data pattern from a “0” or “1” to a “0”. In other words, every transition indicates a “0” for that particular bit slot. No transition indicates a “1” for that particular bit slot. This is an example of a **differential encoding** scheme. Differential encoding relies on comparing two signals to determine a bit value rather than measuring an absolute value. This has the advantage of being more reliable in **noisy media** where the absolute value of a signal is subject to distortion.

Multilevel Binary can use more than one voltage level to represent a “1” or a “0”. One example may be to use a zero voltage level for a “0” and oscillate between a 1v and -1v for each “1” in the digital data. This has the advantage of offering the receiver more **clock synchronization** opportunities in the case of a long stream of “1”s in the data stream. It however doesn’t help in the case of a long stream of “0”s.

Biphase is a technique that overcomes the clock synchronization issue in the previous coding schemes. The most common example of the Biphase encoding scheme is the **Manchester** encoded signal. Here a transition occurs in the middle of each bit slot. A Low-to-High transition represents a “1” and a High-to-Low transition represents a “0”. This allows for receiver clock synchronization at every bit slot. There is still the need to approximate the mid-point of a bit slot, since transitions at the beginning of the slot are just setting up for the correct data transition. **Differential Manchester** is variation on the Manchester encoding by adding differential encoding to Manchester encoding. Here the transition at the mid-point of a bit slot is purely for clock synchronization. The data value of “0” is represented by the presence of a transition at the beginning of a bit slot and a “1” by the lack of a transition at the beginning of a slot.

1.3.2 Continuous Signals

Continuous signal modulation involves altering the frequency, amplitude or the phase of the carrier signal to represent the discrete data. The following are some of the common modulation schemes.

Amplitude-Shift Keying (ASK) alters the amplitude of the carrier signal based on the discrete data. For example the amplitude may be “0” for a “0” and 5v for a “1”.

Frequency-Shift Keying (FSK) alters the frequency to a lower than carrier frequency for a “0” and a higher than carrier frequency for a “1” or vice-versa. Note that the alteration is always centered on the carrier frequency.

Phase-Shift Keying (PSK) alters the phase of the carrier signal to represent the “0” to “1” or “1” to “0” transitions.

1.3.3 Signal Strength

The strength of a signal is of particular importance in communications because as the signal traverses a transmission medium, it will lose some of its energy. This is also referred to as the **attenuation** of the signal. Signal strength often drops logarithmically and hence it is represented as a relative logarithm unit. The decibel (dB) is the unit of measure of signal strength and is defined as follows:

$$\text{Relative Signal Strength in dB} = 10 \log_{10} ((V1^2/R1) / (V2^2/R2))$$

Where $(V1^2/R1)$ represents the power in the first signal and $(V2^2/R2)$ represents the power in the second signal.

If a signal has a power of 500Watts when it is transmitted and has a power for 250Watts when it is received, the loss in power is:

$$\text{Loss} = 10 \times \log_{10} (250/500) = 10 \times (-0.3) = -3\text{dB}.$$

Since the $\log_{10}$ of 0.5 is -0.3, a “-3dB” loss refers to losing half the signal strength.

Similarly a doubling of signal strength is always a “3dB” gain ($\log_{10}$ of 2 is 0.3).

In some cases it is convenient to use 1-Watt as the power of the second, or relative, signal and in this case a signal with a power of 1-Watt would end up defined as an absolute 0 dBW (decibel Watt), as $\log_{10}(1/1) = 0$

The **Signal to Noise ratio** (**SNR**) is yet another common measure of relative signal strength. It refers to the ratio of the signal strength to the noise strength and is commonly expressed in decibels. An SNR of 0dB implies that the noise is as strong as the signal. A negative SNR implies Noise is stronger than the Signal and a positive SNR implies the Signal is stronger than Noise.

1.4 Transmission Concepts

The most critical issue when transmitting a signal from a transmitter to a receiver is the synchronization of the clocks between them. Even a small drift in the clock of the receiver relative to the transmitter will lead to errors very quickly. As we discussed in the section on signals, certain forms of encoding allows for more synchronization opportunities than others.

Generally speaking there are two transmission techniques to reduce the negative impact of clock drifts. These are referred to as Asynchronous Transmission and Synchronous Transmission.

1.4.1 Asynchronous Transmission

Here data is transmitted in very small number of bits (usually 7 bits) where the clock is synchronized at the start of the pattern of bits. The assumption here is that even if there was a clock drift at the receiver, the cumulative effect over a small number of bits will not be catastrophic and that the receiver will get a chance to resynchronize at the start of the next pattern. The disadvantage of this scheme however is that it usually involves additional “Start” and “Stop” bits that envelop a very small number of bits and that adds substantial overhead in throughput. The term “asynchronous” is not particularly accurate in that there is still some level of clock synchronization at the start of each set of transmission.

1.4.2 Synchronous Transmission

An alternative to asynchronous transmission is to either provide a separate clock line that is shared by both the transmitter and receiver or embed the clock information in the signal, as in the case of Manchester coded signals. This will allow the possibility of sending a large number of bits in sequence without the fear of clock drifts at the receiver. Each block of bits would still need to be enveloped in some START and END synchronization bits to indicate the beginning and end of a block of bits. Such an envelope of a data block is often referred to as a frame.

1.4.3 Serial Communication

Serial Communication refers to the case where bits are sent in sequence over a single transmission line. The fact that there is only a single line to send data, forces the serialization of the transmission. Note that serial communication can exploit either Asynchronous or Synchronous transmission techniques discussed previously. Often times in common parlance you will hear the terms “Serial” and “Asynchronous” being used interchangeably. This is however inaccurate. Perhaps the reason for this is that the ubiquitous Personal Computer (PC) has a serial port that uses the Universal Asynchronous Receiver/Transmitter (UART). In that particular case, the serial port exploits the asynchronous transmission scheme and sends a single ASCII character at a time.

1.4.4 Parallel Communication

Parallel Communication refers to the case where there are multiple physical lines between a source and a destination. This allows for sending multiple bits simultaneously. This is an effective technique that increases throughput efficiency, but comes at the additional cost of maintaining multiple physical lines. For

communication between modules that are close in vicinity, it can however be a viable option. Note that the transmission over any particular physical line is still governed by the signaling and synchronicity concepts discussed previously.

1.4.5 Error detection

Once data is received at the destination, it is important to confirm that what is assumed as received is actually what was sent by the source. There are numerous techniques to do this and all of them involve the transmitter embedding some correlation information in the frame that is sent to receiver. The receiver uses the same algorithm used by the transmitter to regenerate the correlation information on the data received and then compares its value with the value that the transmitter embedded in the frame.

One such technique involves the use of a parity-bit in asynchronous transmissions. The idea is to count the number of “1”s in the pattern being sent and determine if this is even or odd. If **even parity** is used, then the parity bit is set if the number of “1”s is even, else the parity bit is set to “0”. Similarly if **odd parity** is used, then the parity bit is set if the number of “1”s is odd, else the parity bit is set to “0”. The parity bit is then appended to the end of the transmission. The receiver will recalculate the parity and confirm that it matches what was sent by the transmitter. As you might have guessed, such a scheme does not account for the case where multiple bits are flipped at the receiver. However it is a reasonable technique in asynchronous communications where only a small number of bits are sent at a time.

Another common technique used with larger blocks of data is called the **Cyclic Redundancy Check (CRC)**. In this scheme, the transmitter and receiver agree on a predetermined number. For every message block, the transmitter pads the message with a number of bits such that if the padded pattern is divided by the predetermined number there will be no remainder. At the receiver, the received pattern is once again divided by the predetermined number to confirm that there is no reminder. CRC is particularly popular because it can be efficiently implemented in both hardware and software.

If errors are detected, the receiver generally will have a means of requesting a retransmission. The actual means of doing this will depend on the protocol between the transmitter and the receiver.

2.0 Networking Concepts

In the previous section on Communication concepts we discussed the techniques surrounding the sending of information from a source to a destination. In communication parlance, this is also referred to as point-to-point communication. While these concepts are fundamental, a natural extension of this is to be able to connect multiple devices such that any one device could communicate with any other device in the network. In essence, anyone in a network can be a source or a destination to one or more devices in the same network of devices.

One way to achieve this would be to have a point-to-point physical connection between every possible permutation of devices on the network. As you can imagine this would be prohibitive in terms of cost and management as the number of devices increase.

A more reasonable approach is to make each device a node in a commonly shared piece of physical connection. This approach however raises new and additional issues to what was discussed in the previous section. To start with, we need an arbitration mechanism to manage access to the common physical connection. Further, we need some addressing scheme to be able to direct transmissions from one node to another. These concepts will be discussed in this section.

2.1 The Layered Architecture

As scientists and engineers attempt to resolve problems, they discover common themes that they categorize based on the common concepts that apply to these situations. This natural evolution leads to distinct areas of interest and standards that define the best practices to address particular situations.

These standards are particularly important in Network Communications because to communicate with other devices in a network, one has to agree on a protocol for communication. The type of signaling, transmission, error detection and other such communication parameters have to be standardized for successful interaction between various nodes in a network. This requirement becomes even more critical when you consider that there are a multitude of hardware and software modules distributed by a multitude of product vendors that have to work in concert to achieve the successful transmission and reception of data.

In 1977 the **International Standardization Organization (ISO)** created a committee to study and develop an architecture based on the various tasks involved in computer communications. This committee was tasked with describing the roles of the various modules involved and how they interfaced with each other. This resulted in the "**Open System Interconnection** (**OSI**)" reference model that was ratified in 1983. While this model does not bear a strict resemblance to existing networking architectures, it does do a good job of identifying and isolating the variety of roles in computer communication.

The OSI model consists of 7 layers:

Layer Number	Layer Name	Description
1	Physical Layer	This is the lowest layer in the 7 layer OSI model and is responsible for the physical medium over which signal transmission occurs.
2	Data Link Layer	This is the layer responsible for arbitration, synchronization, error and flow control.
3	Network Layer	This layer is responsible address management, packet routing and fragmentation (if required over a particular Data Link Layer).
4	Transport Layer	This layer is responsible for providing a reliable communication channel for all upper layers.
5	Session Layer	This layer is responsible for abstracting upper layer applications from the intricacies of creating, maintaining, arbitrating and terminating connection sessions.
6	Presentation Layer	This layer is responsible for abstracting differences in data representation from the applications.
7	Application Layer	Provides access to the underlying communication layers to end users.

Most of the material we covered in the previous section relates to layer 1 of the OSI model. In this and the following section we will expand on layers 2, 3 and 4 in some detail use prevailing networks as examples. Layers 5, 6 and 7 are less distinct in most prevailing network architectures.

The general idea in this layered architecture is that each layer in a node can communicate with the same layer in a different node. The application layer generates the data that it gives to the presentation layer. This data is referred to as the "**payload**" since it is the essence of the communication.

The presentation layer does some processing and adds a header to create an envelope around its processing and then sends this enveloped data to the session layer. The session layer in turn does some processing, adds its header to create a new envelope and sends that down to the Transport layer and so on. The fully encapsulated envelope that goes over the physical layer is also referred to as a **datagram**.

At the receiver, each layer will undo the processing done by the same layer at the sender and removes its respective header and passes the remaining envelope to the immediately higher layer. Thus by the time the envelope reaches the application layer at the receiver, the original payload should be seen.

2.2 Network Layer Management & Routing

Perhaps the most intriguing layer in the OSI layered model is the Networking layer. The networking layer is responsible for the maintenance of network addresses that are unique across the entire network as well as identifying the next hop in the routing paths to get to a destination from a source. The networking layer is perhaps best described with an example and the best example of a networking layer is the **Internet Protocol (IP)** that is used by the World Wide Web.

2.2.1 IP Header

The most expedient way to understand a communication protocol is to study the protocol's header. The figure below shows the fields in the IP header.

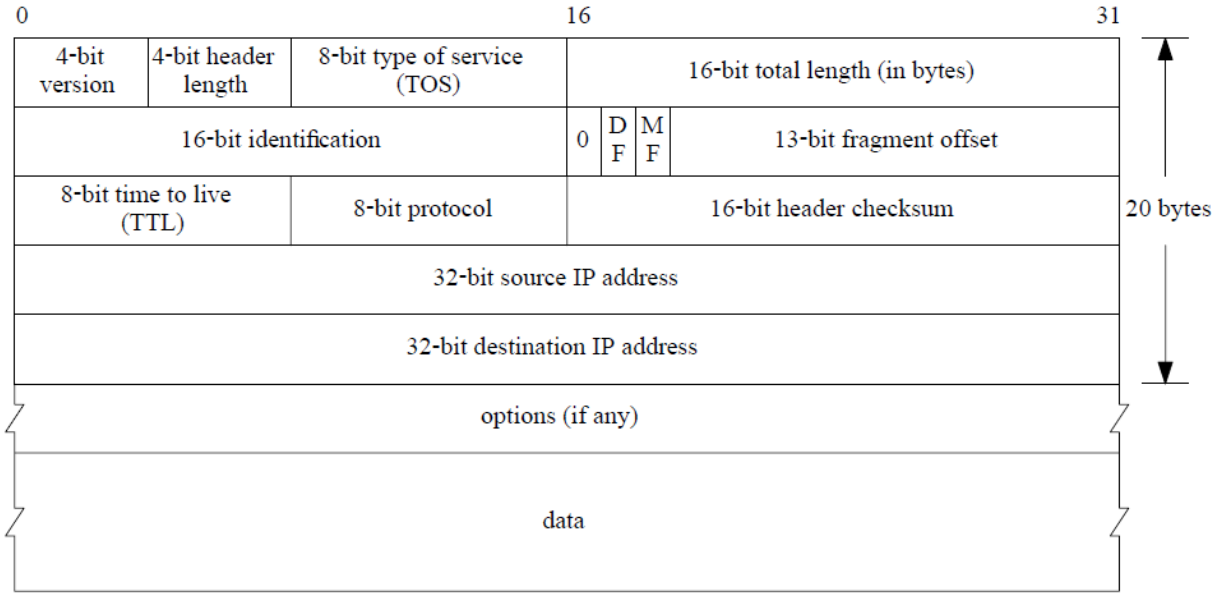

Figure 2.2.1.1

Version

This is a 4-bit value that identifies the version of IP that is used. In our discussion we will assume that this is always set to "4".

Header Length

This is a 4-bit value that reflects the number of 32-bit words that form the IP header. Usually this is set to "5".

Type of Service

This is an 8-bit field used to identify packet prioritization. Routers that honor this field can use it to prioritize traffic.

Total Length

This a 16-bit value that refers to the total length of the packet measured in octets.

Identification

This is a 16-bit identifier that a sender can use to aid in the reassembly of the packets at the receiver.

Flags

Bit 0: Reserved and always set to “0”

DF: If set, Don’t fragment

MF: If set, More fragments are associated with this datagram

Fragment Offset

This is a 13-bit value indicating the relative position of this fragment (measured in octets) within the larger datagram. The first fragment has an offset of zero. Note that the upper layer gives a datagram to the IP layer and the IP layer can fragment this datagram based on network layer restrictions.

Time to Live

This is an 8-bit value indicating the maximum time this packet can remain in the internet. If this value is zero, the packet is discarded. The time is measured in seconds, but since each processing nodes decrements this value by at least one, it can also been seen as the number of hops that a packet can have in an attempt to reach its destination.

Protocol

This is an 8-bit field that identifies the upper layer protocol that is using the IP layer. At its final destination, the IP layer will route this packet to the protocol defined in this field.

Header Checksum

This is 16-bit header-only checksum.

Source Address

A 32-bit source address field.

Destination Address

A 32-bit destination address field.

The Internet Protocol uses an address of the format “xxx.xxx.xxx.xxx”. These are four bytes, each delimited by a period, for a total of 32 bits. Each byte can take a value of 0 to 255. These addresses are also referred to as **IPV4** addresses. Over the last few years, as the number of nodes in the World Wide Web have increased, the limitations of 32-bit addresses have become more pronounced. This has led to a new

addressing scheme that uses 128-bit address. This new address is also referred to as an **IPV6** address. The version field in the IP header can be used to find the expected size of the source and destination addresses.

2.2.2 Dynamic Host Configuration Protocol (DHCP)

The World Wide Web is the interconnection of several IP networks. Each node in any given IP network is issued a unique IP address. This issuance can be maintained manually by the system administrator of that network. In such cases, each node is hard-coded to have that IP address and is also referred to as a **static IP address**. A more efficient way of issuing IP addresses is for a node to request an address when it joins the network and have one dynamically allocated when such a request is made. The **Dynamic Host Configuration Protocol (DHCP)** defines the specification for the request and issuance of a **Dynamic IP** address within an IP network. The DHCP protocol requires a DHCP server to issue an IP address from a pool of addresses that it has available and to maintain the lifetime of that address. If a node fails to renew its IP address after its lease term expires, the DHCP server can reclaim that IP address and add it to its free pool for reissuance to another node at a later time. This is a far more efficient mechanism in maintaining IP addresses than the static allocation of IP addresses.

The DHCP protocol is commonly broken into four phases and referred to as the **DORA**. This stands for **DISCOVER**, **OFFER**, **REQUEST** and **ACK**. A node broadcasts a DISCOVER request. A DHCP server responds with an OFFER. The node formally REQUESTs the use of the IP address in the OFFER. The DHCP server acknowledges (ACKs) this request. The figure below shows a capture of this traffic at the physical layer along with the fields in the DHCP OFFER packet. Note how DHCP relies on UDP and IP protocols. We will discuss UDP later in the next section.

Frame Number	Time Offset	Source	Destination	Description
274	2.4348325	0.0.0.0	255.255.255.255	DHCP:Request, MsgType = DISCOVER, TransactionID = 0x0BF4C1FE
326	2.9479926	123.124.125.1	255.255.255.255	DHCP:Reply, MsgType = OFFER, TransactionID = 0x0BF4C1FE
328	2.9762165	0.0.0.0	255.255.255.255	DHCP:Request, MsgType = REQUEST, TransactionID = 0x0BF4C1FE
330	2.9828386	123.124.125.1	255.255.255.255	DHCP:Reply, MsgType = ACK, TransactionID = 0x0BF4C1FE

Figure 2.2.2.1

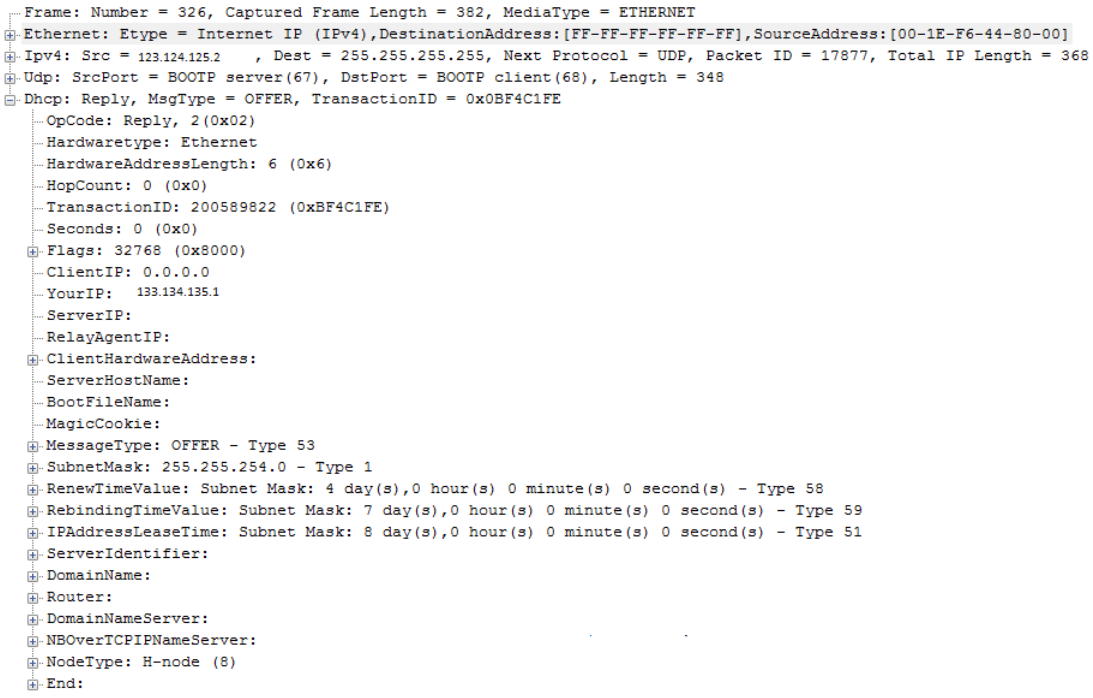

Figure 2.2.2.2

2.2.3 Domain Name System (DNS)

When an application in a node in a given IP network wishes to send a datagram to another application on another node in the same or different IP network, it will likely identify the destination node by its user friendly name. The IP layer in the source node will translate the user-friendly name to the destination's IP address and use this to populate the IP header in that datagram. The way the IP layer translates the user-friendly name to an IP address is defined by the **Domain Name System (DNS)** protocol. The DNS is much like a phonebook, in that it translates the user-friendly names to the IP addresses. This level of redirection allows for the changing of underlying IP addresses without changing the user-friendly names. The way DNS works is best described by an example. Assume you wanted to find the IP address of www.msnbc.com. The IP network on which you are currently located will have a "**root name server**" that will field your query to identify www.msnbc.com. It will then suggest that you contact the "**com name server**" because the address string contains the ".com" suffix and it will provide you with the IP address of the "com name server". The "com name server" in turn will ask you to contact the "msnbc name server" and it will provide you with the IP address of the msnbc name server. Finally the "msnbc name server" will give you the current IP address of www.msnbc.com.

Once a source and destination IP address are available to the IP layer, a datagram can be sent from the source to the destination. The sender does not need to know the exact path that the datagram is going to traverse to get to its destination; it just needs to know the next hop in the path. If the destination IP address is a member of the source's IP network, then the next hop is likely the destination node since the sender could directly address the destination. If the destination IP address is not part of the sender's IP address, a special node on the source IP network, referred to as a "**router**" will take ownership of this datagram.

2.2.4 Routers & Bridges

A **Router** is a special node in that it connects the IP network to the next higher backbone on the World Wide Web. If there are no data link layer differences as the datagram is moved from one backbone to the next, such a router is also referred to as a **Bridge**. You can visualize the World Wide Web as a series of hierarchical backbones that eventually end at the **Network Service Provider (NSP)** backbone. Network Service Providers are very large IP networks that have been established over the years. Some examples of NSPs included **UUNet**, **CerfNet** and **SprintNet**. Each NSP is required to connect to 3 **Network Access Points (NAP)**. NAPs allow for datagrams to jump from the backbone of one NSP to another. As the datagram moves higher up these World Wide Web backbones, they eventually hit a router or an NAP that determines that the datagram should move side-ways and then back down another set of routers until the datagram reaches the IP network that hosts the destination node. Note that the exact path of the datagram is not predetermined at the sender's node. Instead it is dynamically determined as the datagram flows across the internet.

The routers play a crucial role in determining if the datagram needs to be forwarded up, sideways or down the internet hierarchy. To enable this function, routers maintain tables of IP address ranges that fall below them. As routers come across datagrams on the upper interfaces addressed to IP ranges below them, they forward them to the IP networks beneath them. Similarly as they come across datagrams on the lower interfaces not addressed to IP ranges beneath them, they forward them to the IP networks above them. Routers higher up in the internet hierarchy have much larger tables to maintain since they have larger number of IP networks below them.

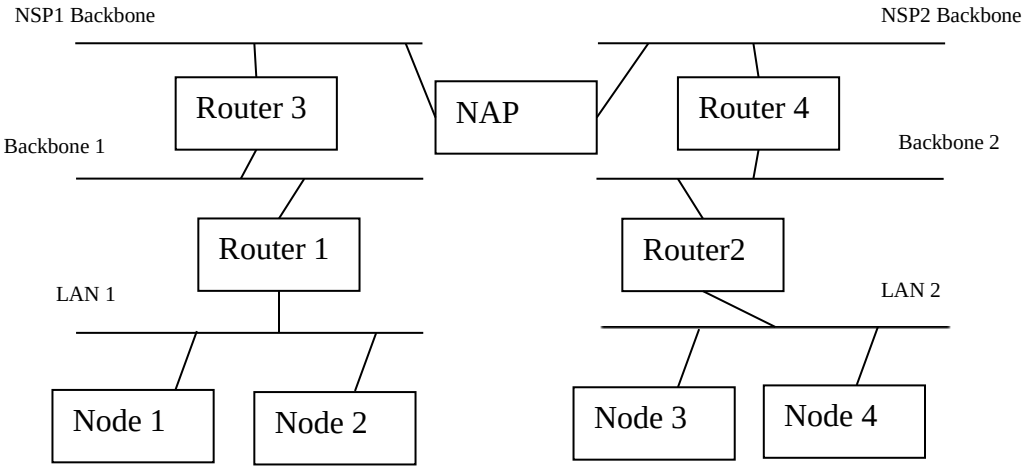

Figure 2.2.4.1

2.2.5 Address Resolution Protocol (ARP)

When a source node wants to send an IP datagram to a destination node within its own IP network, it will need to go through a Data Link Layer to access the shared physical layer. Most Data Link Layer protocols will have a unique hardware address associated with a node. The IP layer will need to know the unique hardware address for the destination node and wrap the datagram with a Data Link Layer header that includes this hardware address. This is because the Data Link Layer of each node on the network will only accept broadcast datagrams or datagrams specifically addressed to it. This requires an ability to map an IP address to a Data Link Layer hardware address. The **Address Resolution Protocol (ARP)** is responsible for this task. It allows a node to **broadcast** a probe ARP packet to all nodes in the network and get the node that owns a particular IP address to respond with its Data Link Layer address. Similarly, when a router forwards a packet to an IP network, it is responsible for identifying the Data Link Layer address of the destination node on that network. The figures below show an ARP request and an ARP response. Note that the ARP packet goes directly over Ethernet and doesn't rely on IP. The Ethernet header identifies this packet with an Etype of "ARP".

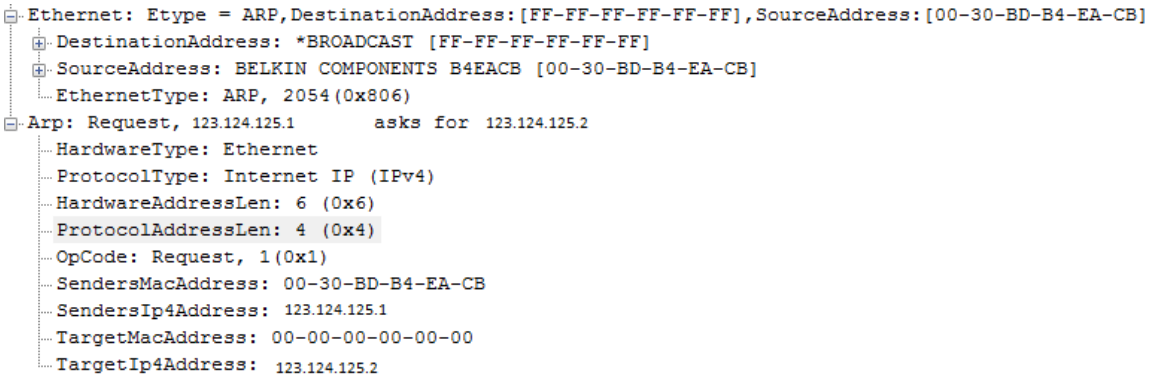

Figure 2.2.5.1

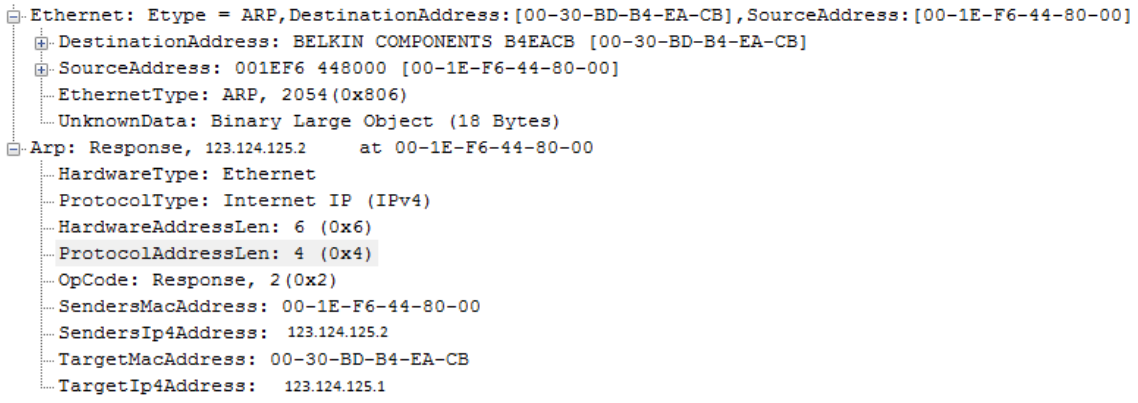

Figure 2.2.5.2

The Address Resolution Protocol also allows for the advertising of a node's IP address before accepting it the first time. This is a safety precaution to guard against the possibility that another node is already using that IP address. Such an ARP packet is often referred to as a **Gratuitous ARP**.

2.2.6 Internet Control Message Protocol (ICMP)

The **Internet Control Message Protocol (ICMP)** is a component of the Internet Protocol (IP) and is responsible for conveying control and status information about the IP. ICMP is not involved in the communication of payload packets.

Two of the most common applications that use the ICMP are **Ping** and **Traceroute**. Ping is used to probe a remote host for responsiveness, while Traceroute is used find the intermediary hops in packet routing over the IP.

The figures below show ICMP Request and Reply packet examples. Note that ICMP relies on IP.

Frame Number	Time Offset	Source	Destination	Description
288	5.9255178	100.101.102.1	100.101.102.2	ICMP:Echo Request Message, From 100.101.102.1 To 100.101.102.2
289	5.9279559	100.101.102.2	100.101.102.1	ICMP:Echo Reply Message, From 100.101.102.2 100.101.102.1

Figure 2.2.6.1

```
Frame: Number = 288, Captured Frame Length = 74, MediaType = ETHERNET
Ethernet: Etype = Internet IP (IPv4),DestinationAddress:[00-1E-F6-44-80-00],SourceAddress:[00-30-BD-B4-EA-CB]
Ipv4: Src = 100.101.102.1      , Dest = 100.101.102.2   , Next Protocol = ICMP, Packet ID = 6115, Total IP Length = 60
  Versions: IPv4, Internet Protocol; Header Length = 20
  DifferentiatedServicesField: DSCP: 0, ECN: 0
  TotalLength: 60 (0x3C)
  Identification: 6115 (0x17E3)
  FragmentFlags: 0 (0x0)
  TimeToLive: 128 (0x80)
  NextProtocol: ICMP, 1(0x1)
  Checksum: 57644 (0xE12C)
  SourceAddress:  100.101.102.1
  DestinationAddress:  100.101.102.2
Icmp: Echo Request Message, From 100.101.102.1    To 100.101.102.2
  Type: Echo Request Message, 8(0x8)
  Code: 0 (0x0)
  Checksum: 19783 (0x4D47)
  ID: 1 (0x1)
  SequenceNumber: 20 (0x14)
  ImplementationSpecificData: Binary Large Object (32 Bytes)
```

Figure 2.2.6.2

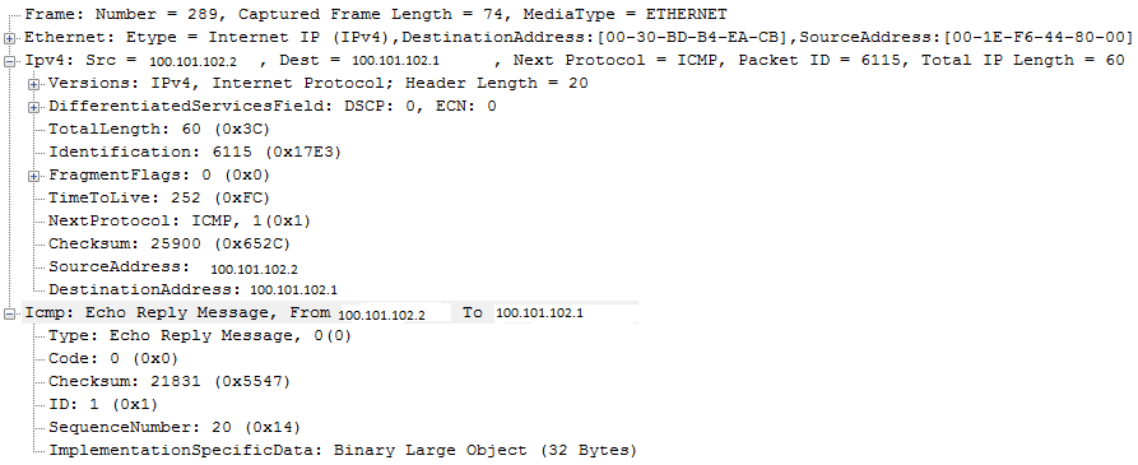
Frame: Number = 289, Captured Frame Length = 74, MediaType = ETHERNET
Ethernet: Etype = Internet IP (IPv4),DestinationAddress:[00-30-BD-B4-EA-CB],SourceAddress:[00-1E-F6-44-80-00]
Ipv4: Src = 100.101.102.2 , Dest = 100.101.102.1 , Next Protocol = ICMP, Packet ID = 6115, Total IP Length = 60
Versions: IPv4, Internet Protocol; Header Length = 20
DifferentiatedServicesField: DSCP: 0, ECN: 0
TotalLength: 60 (0x3C)
Identification: 6115 (0x17E3)
FragmentFlags: 0 (0x0)
TimeToLive: 252 (0xFC)
NextProtocol: ICMP, 1(0x1)
Checksum: 25900 (0x652C)
SourceAddress: 100.101.102.2
DestinationAddress: 100.101.102.1
Icmp: Echo Reply Message, From 100.101.102.2 To 100.101.102.1
Type: Echo Reply Message, 0(0)
Code: 0 (0x0)
Checksum: 21831 (0x5547)
ID: 1 (0x1)
SequenceNumber: 20 (0x14)
ImplementationSpecificData: Binary Large Object (32 Bytes)

Figure 2.2.6.3

2.3 Data Link Layer & Local Area Networks

A **Local Area Network (LAN)** refers to a computer network that connects computers in relatively close proximity of each other. The computers in a school or a business are usually member nodes in a Local Area Network. Two or more LANs could be potentially interconnected by routers placed between them.

The datagrams generated at the Network Layer end up in the physical layer of a LAN by going through its **Data Link Layer**. The Data Link Layer defines a protocol to access the physical layer of any particular LAN. The single most important function of the Data Link Layer is to arbitrate the access to the shared physical bus. This function is also referred to as **Medium Access Control (MAC).**

There are several MAC techniques used in LANs. The choice of a particular MAC technique will depend on the type of Network traffic that is generated in that particular network.

2.3.1 Round Robin

The Round Robin MAC technique refers to a scheme where each node in the network is in turn given a time slot when it can transmit. A node can accept or decline its turn depending on whether it has data to transmit. If a node accepts its turn to transmit, it has a maximum amount of time during which it has access to the physical medium for transmission. The node can relinquish its access to the physical medium prior to the expiration of the maximum time, if it has no further data to transmit. If a node has more data to send than can be accommodated within the maximum time slot, it can transmit all that it can with the maximum time slot and then yield the physical medium for the next node in turn; the remainder of data can then be transmitted next time around.

The Round Robin technique is rather efficient when all the nodes in network have a lot to transmit all the time. If that is not the case, the overhead of maintaining time slots for each node can be substantial. The advantage of the Round Robin scheme is that once you know the number of nodes in a network, you can predict the worst case latency in transmission of data. For this reason such networks are sometimes referred to as **Deterministic Networks**. **Token Bus (IEEE 802.4)** and **Token Ring (IEEE 802.5)** are examples of a Round Robin MAC scheme.

2.3.2 Timeslot Reservation

Yet another MAC technique is to allow a node to reserve time slots in advance. This can be a useful technique for nodes with traffic that is fairly large and continuous in nature such as voice or file transfers. Such traffic is also referred to as **Stream traffic**.

This technique can be more efficient than Round Robin if a node does not always require time slots, but when it does require time slots, it needs it for an extended period. Once the reservation is successful, the transmission latency is deterministic and hence such networks are also referred to as **Deterministic networks**. The **Distributed Queue Dual Bus (IEEE 802.6)** is an example of a Timeslot Reservation MAC scheme.

2.3.3 Random Contention

For traffic is that sporadic and short, neither of the techniques described above are efficient. Such traffic is referred to as **Bursty traffic**. The management overhead in maintaining tokens or reserving timeslots will prove cumbersome and inefficient in these cases. As it turns out, the bulk of computer communications are of a bursty nature. In such cases it is far more efficient to let anarchy prevail. Any node can transmit at any time and if a node's transmission collides with another node's transmission, each of the transmitting nodes retry after a random interval.

Such networks are **Non-Deterministic** by nature since it is impossible to predict how long it would take to transmit a datagram. The most common example of a Random Contention MAC scheme is the **Carrier Sense Multiple Access with Collision Detection (CSMA/CD - IEEE 802.3)**. Twisted-Pair Ethernet is an example of CSMA/CD.

The figure below depicts an Ethernet header.

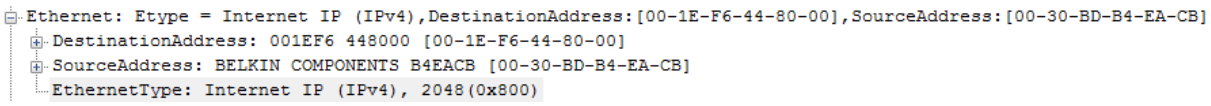

Figure 2.3.3.1

3.0 Prevailing Communication Technologies

The previous sections dealt with the first three layers (Physical, Data Link and Network) of the OSI model. In this section we will discuss some of the prevailing technologies that exploit these layers. In the process, we will visit a common implementation of the Transport layer (TCP) that is used in the World Wide Web.

We will also discuss the prevailing technologies to address communication between computers and peripheral devices. While the concepts of routing and network management are less rigorous in these types of communication compared to the Internet Protocol, the demarcation of the various communication layers of the OSI model are still largely relevant.

3.1 Transport Control Protocol/Internet Protocol (TCP/IP)

The **TCP/IP protocol suite** refers to two types of **Transport layer protocols**. The first is the **Transmission Control Protocol (TCP)** and the other is the **User Datagram Protocol (UDP)**. The TCP protocol is a full-duplex Connection-Oriented protocol while UDP is connectionless. Both protocols rely on the Internet Protocol (IP) for the Network layer.

As discussed in the section on the layered OSI model, the Transport layer is responsible for providing **reliability** in communication. Some of the common factors that contribute to Transport layer reliability are the following:

- Connection Management (Applicable to Connection-oriented protocols)
- Guaranteed Delivery of Data (Acknowledgment and Error recovery)
- Traffic sequencing
- Traffic prioritization

The TCP Transport layer addresses all of the above reliability factors. UDP on the other hand, does not address any of the above reliability factors. UDP is a datagram protocol, much like the IP Network layer, but it adds some additional addressing features (ports) to the IP Network layer to help with peer-to-peer application level communications. Hence UDP does not qualify as a Transport layer protocol in the definition of the OSI model. Nevertheless for historical reasons UDP is still treated as a Transport layer protocol.

3.1.1 TCP Header

As before with the IP header, the fastest way to grasp a communication protocol is to study the fields in the protocol's header. The figure below shows the TCP header structure.

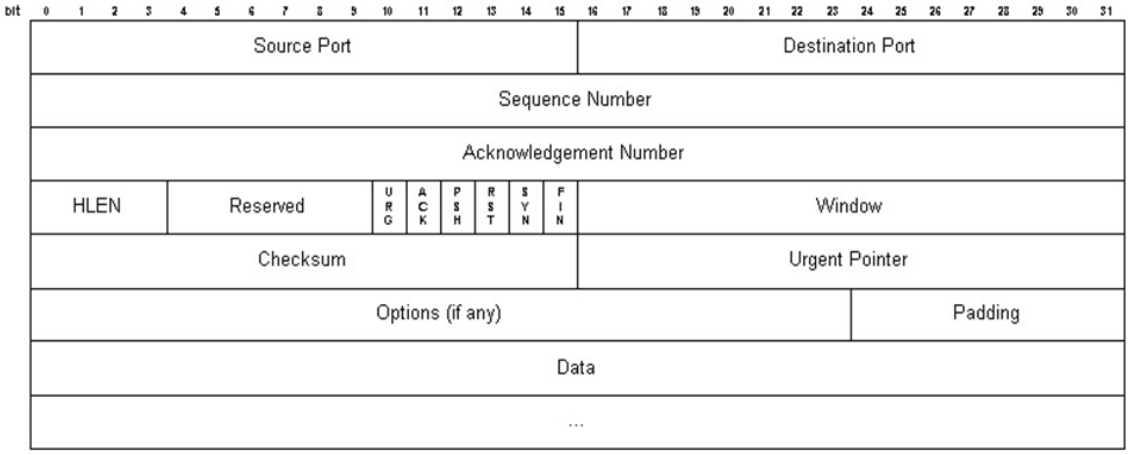

Figure 3.1.1.1

Port Number

The 16-bit Source Port and Destination Port is an additional level of addressing provided by the TCP transport layer to help with peer-to-peer application level communication. Applications can reserve specific port numbers and so when traffic arrives at the TCP layer, it will allow TCP to route the traffic to the appropriate application based on the Destination port number. The port numbers are governed by global standards and ports 0 through 1023 are reserved for commonly used applications.

Sequence Number

This is a 32-bit number identifying the position of the first data byte in the current segment of transmission within the entire stream of traffic over the TCP connection. The number will wrap back to zero after reaching 2^{32} -1. This field is used by TCP to ensure that the receiving application gets the data in the same sequence in which the sender sent the byte stream.

Acknowledgment Number

This is a 32-bit number that identifies the next data byte the sender expects from the receiver. This field is used when the ACK control bit is set.

Header Length

This 4-bit field specifies the offset to the data in this data packet. It is the size of the TCP header.

Control Bits

Urgent Pointer (URG): If set, the receiver must interpret the “Urgent Pointer” field.
Acknowledgement (ACK): If set, the Acknowledgement field is in use.
Push Function (PSH): If set, the data segment should be forwarded to application as soon as possible.
Reset Connection (RST): If set, sender is aborting the connection. All queued data can be discarded.
Synchronize (SYN): Used during connection establishment to synchronize sequence numbers.
No More Data form Sender (FIN): If set, sender has completed transmission.

Window

This is a 16-bit field used for data throttling. It indicates to the sender how much data the receiver is willing to accept.

Checksum

This is a 16-bit checksum value used to confirm data integrity. If an error is detected, TCP will issue a NACK, thus forcing a retransmission by the sender.

Urgent Pointer

This field is relevant only when the “URG” control bit is set. It indicates where the last byte of urgent data in this segment ends.

3.1.2 TCP Connection Establishment

A TCP connection establishment requires a 3-way handshake as described in the diagram below.

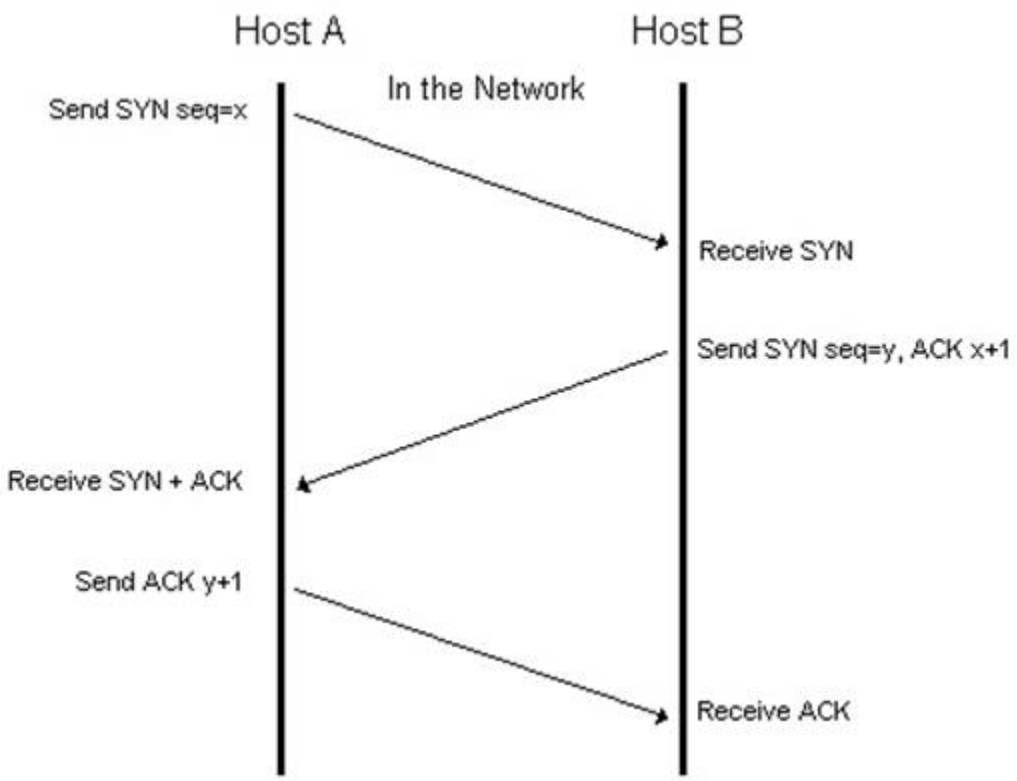

Figure 3.1.2.1

First, Host A sends a SYC packet to Host B with a Sequence number of “x”.
Second, Host B sends a SYN packet to Host A with a Sequence number of ‘y” and an ACK of “x+1”.
Third, Host A sends a SYN packet to Host B with an ACK of “y+1”.

After these three packets have been successfully transmitted and received, the connection is established.

3.1.3 TCP Connection Termination

A TCP connection termination requires four packet transfers as shown below.

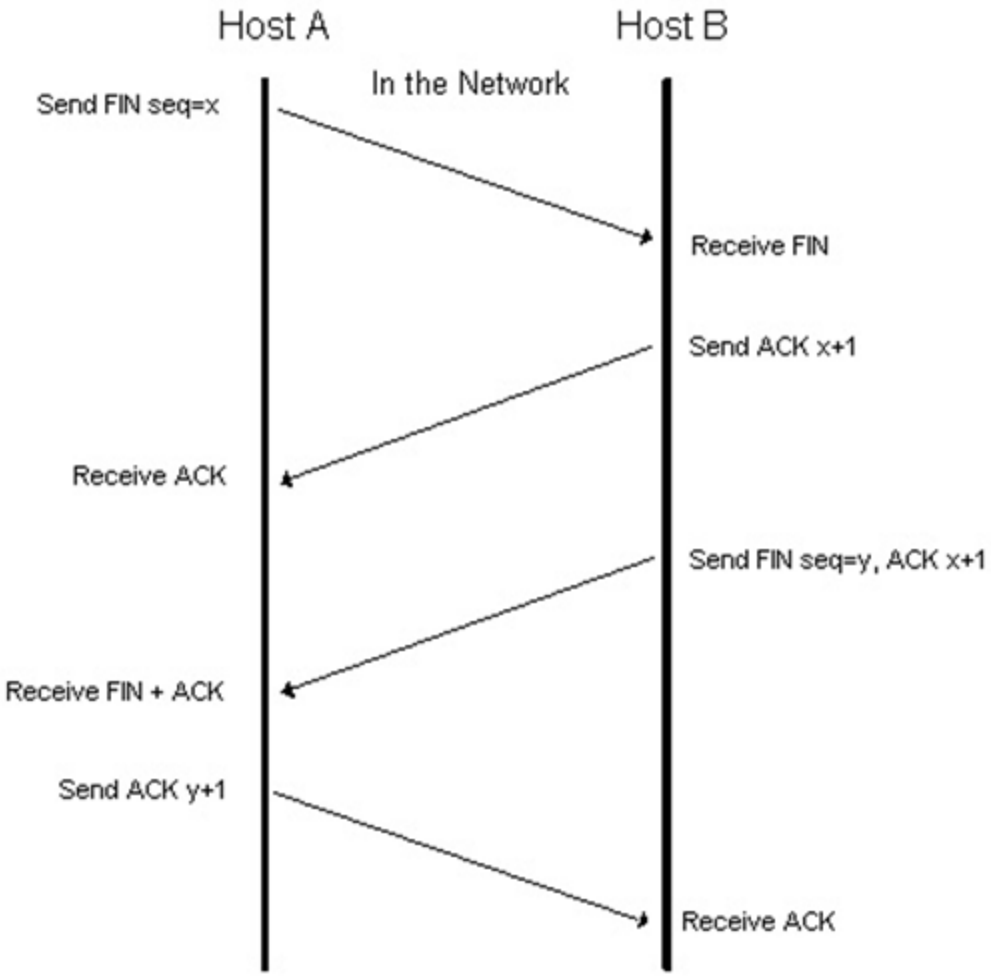

Figure 3.1.3.1

First, Host A sends a packet with Sequence number “x” with the FIN bit set.
Second, Host B acknowledges the previous packet by using an ACK of “x+1”.
Third, Host B sends another packet with the FIN bit set with a Sequence of “y”.
Fourth, Host A acknowledges the previous packet by using an ACK of “y+1”.

The reason for issuing a FIN packet from both sides is because TCP is a full-duplex communication channel.

3.1.4 TCP Data Transfer & Nagle Algorithm

Once a connection is established, both sides can transmit and receive packets. Each host TCP layer is responsible for sequencing and checksum validation prior to forwarding the data buffer in each packet to the application.

As long as the application is requesting to send data, TCP will send data to a receiver provided it does not exceed the window advertised by the receiver. Note that each acknowledgement packet also contains a valid window field.

TCP uses several timers. Some of these are designed to take corrective actions in the event of unresponsive connections. Others are designed to delay actual transmission requests to optimize bandwidth consumption of the underlying network. Note that sending very small chunks of payload at a time can cause substantial network overhead. A common TCP algorithm to avoid this is the **Nagle algorithm** which is described as follows:

```
if there is new data to send
  if the window size >= MSS and available data is >= MSS
    send complete MSS segment now
  else
    if there is unconfirmed data still in the pipe
      enqueue data in the buffer until an acknowledge is received
    else
      send data immediately
    end if
  end if
end if
```

Where MSS is the Maximum Segment Size.

3.1.5 TCP Client Sample Source Code

The following is a simple TCP client sample that shows how to use the Windows Socket interface to access the TCP Transport Layer in the Microsoft Windows Operating system.

```
// TCPClient.cpp :

#include "stdafx.h"
#include <winsock2.h>

//Constants
#define    MAX_LOADSTRING                  100
#define    MAX_BUFLEN                      512
#define    GET_HOST_NAME_FAILURE           1
#define    SOCKET_FAILURE                  2
#define    CONNECT_FAILURE                 3
#define    SEND_FAILURE                    4
#define    TEST_PORT                       1910

// Main Entry
int _tmain(int argc, _TCHAR* argv[])
{
   char    szHostId[MAX_LOADSTRING];
   char    buffer[MAX_BUFLEN  ];
   char    buf_num[MAX_BUFLEN  ];
```

```
   WSADATA info;
   SOCKET  s = INVALID_SOCKET;
   struct  sockaddr_in server;
   struct  hostent *h;
   BOOL    retVal = FALSE;
   int     errorCode=0, lineNum=0;

   WSAStartup(MAKEWORD(1,1), &info);

   /* Look up the server in the host file to get its IP address*/
   strcpy(szHostId, "skg10");
   if( (h=gethostbyname(szHostId)) == NULL)
      {errorCode= GET_HOST_NAME_FAILURE; lineNum=__LINE__;goto exit;}

   /* Create a socket */
   if( (s=socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
      {errorCode= SOCKET_FAILURE; lineNum=__LINE__; goto exit;}

   /* Fill in the server's address  and port*/
   server.sin_family = AF_INET;
   memcpy(&server.sin_addr, h->h_addr, h->h_length);
   server.sin_port   = htons(TEST_PORT);

   /* Connect to the Server */
   if (connect(s, (sockaddr *)&server, sizeof(server)) == SOCKET_ERROR )
      {errorCode= CONNECT_FAILURE; lineNum=__LINE__; goto exit;}

   /* Copy a greeting message into the buffer */
   strcpy(buffer, "hello from SKG");

   /* Send it to the Server */
   if( (send(s, buffer, strlen(buffer)+1, 0)) == SOCKET_ERROR )
      {errorCode= SEND_FAILURE; lineNum=__LINE__; goto exit;}

   /* Copy more stuff to send & send it */
   for (int i=0; i < 100; i++)
   {
      strcpy(buffer, "This is a longer message to send");
      if( (send(s, buffer, strlen(buffer)+1, 0)) == SOCKET_ERROR )
         {errorCode= SEND_FAILURE; lineNum=__LINE__; goto exit;}
      sprintf(buf_num, "Buffer Number %d", i);
      if( (send(s, buf_num, strlen(buf_num)+1, 0)) == SOCKET_ERROR )
         {errorCode= SEND_FAILURE; lineNum=__LINE__; goto exit;}
   }

   retVal = TRUE;
exit:
   if(retVal == FALSE)
   {
      switch(errorCode)
      {
         case( GET_HOST_NAME_FAILURE ):
            printf("TCPClient: Failure Detected in call to \"gethostbyname\" on line %d LastError=%d!!", lineNum, GetLastError());
            break;
         case( SOCKET_FAILURE ):
            printf("TCPClient: Failure Detected in call to \"socket\" on line %d LastError=%d!!", lineNum, GetLastError());
            break;
         case( CONNECT_FAILURE ):
            printf("TCPClient: Failure Detected in call to \"connect\" on line %d LastError=%d!!", lineNum, GetLastError());
            break;
         case( SEND_FAILURE ):
            printf("TCPClient: Failure Detected in \"send\" on line %d LastError=%d!!", lineNum, GetLastError());
            break;
      }
   }

   /* Disconnect from the Server */
   if( s != INVALID_SOCKET ) closesocket(s);
   WSACleanup();
```

```
    return(0);
}
```

3.1.6 TCP Sever Sample Source Code

The following is a simple TCP server sample that shows how to use the Windows Socket interface to access the TCP Transport Layer in the Microsoft Windows Operating system.

```
// TCPServer.cpp :

#include "stdafx.h"
#include <winsock2.h>

#define    SOCKET_FAILURE                1
#define    SEND_FAILURE                  2
#define    LISTEN_FAILURE                3
#define    BIND_FAILURE                  4
#define    ACCEPT_FAILURE                5
#define    RECV_FAILURE                  6
#define    TEST_PORT                     1910

int _tmain(int argc, _TCHAR* argv[])
{
   SOCKET         msgsock;
   int            client_len;
   int            bytes_read = 0;
   char           buf[1600];
   SOCKET         sock = INVALID_SOCKET;
   struct         sockaddr_in server, client;
   WSADATA        wsadata;
   BOOL           retVal = FALSE;
   int            errorCode=0, lineNum=0;

   /* Initialize the socket library */
   WSAStartup(MAKEWORD(1,1), &wsadata);

   /* Create a socket to listen on */
   if( (sock=socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
      {errorCode= SOCKET_FAILURE; lineNum=__LINE__; goto exit;}

   server.sin_family      = AF_INET;
   server.sin_addr.s_addr = htonl(INADDR_ANY);
   server.sin_port        = htons(TEST_PORT);

   /* Bind Socket to Port */
   if (bind(sock, (struct sockaddr *) &server, sizeof (server)) == SOCKET_ERROR)
      {errorCode= BIND_FAILURE; lineNum=__LINE__; goto exit;}

  /* Set to Listen mode */
   if (listen(sock, 5) != 0 )
      {errorCode= LISTEN_FAILURE; lineNum=__LINE__; goto exit;}

   fprintf(stderr, "TCP/IP Test server: Version 1.00\r\n\n");

   /* Wait for the client to "connect" */
   while (1)
   {
         fprintf(stderr, "Waiting for Client to Connect...\r\n");
         client_len = sizeof(client);
         /* Block waiting to accept connection */
         if( (msgsock = accept(sock, (struct sockaddr *) &client, &client_len)) < 0 )
            {errorCode= ACCEPT_FAILURE; lineNum=__LINE__; goto exit;}
```

```
        fprintf(stderr,"Accepted connection -- \r\n");

        /* Show how many bytes arrive in each packet, until the client closes the connection */
        do
        {
           if( (bytes_read = recv(msgsock, buf, sizeof(buf), 0)) == SOCKET_ERROR )
              {errorCode= RECV_FAILURE; lineNum=__LINE__; goto exit;}
           fprintf(stderr, "RECV'd %d bytes\r\n", bytes_read);
        }while(bytes_read);
   }

   retVal = TRUE;
exit:
   if(retVal == FALSE)
   {
      switch(errorCode)
      {
         case( SOCKET_FAILURE ):
            printf("TCPServer: Failure Detected in call to \"socket\" on line %d LastError=%d !!", lineNum, GetLastError());
            break;
         case( BIND_FAILURE ):
            printf("TCPServer: Failure Detected in call to \"bind\" on line %d LastError=%d !!", lineNum, GetLastError());
            break;
         case( LISTEN_FAILURE ):
            printf("TCPServer: Failure Detected in \"listen\" on line %d LastError=%d !!", lineNum, GetLastError());
            break;
         case( ACCEPT_FAILURE ):
            printf("TCPServer: Failure Detected in \"accept\" on line %d LastError=%d !!", lineNum, GetLastError());
            break;
         case( RECV_FAILURE ):
            printf("TCPServer: Failure Detected in \"recv\" on line %d LastError=%d !!", lineNum, GetLastError());
            break;
      }
   }


   /* Disconnect from the Server */
   if( sock != INVALID_SOCKET ) closesocket(sock);
   WSACleanup();

   return 0;
}
```

3.1.7 TCP Sample Network Trace

The following is a network trace that captured the traffic at the physical layer when the above sample client and server were communicating. A network trace is captured with a network monitor tool that puts the Data Link Layer in a **promiscuous mode** so that it forwards all frames at the physical layer (not just the ones addressed to its MAC address) up the networking stack and the network capture tool then disassembles the frames to individual protocol headers and generates a report like the one below. Such a tool is invaluable in debugging communication issues.

Frames 509, 510 and 511 represent the TCP connection establishment.
Frames 519, 520, 521 and 522 represent the TCP connection terminating.
The frames in between the connection establishment and termination represent the payload transactions. Pay attention to the Sequence, ACK and Window sizes. Also notice how the 100 sends translated to just a few frames at the physical layer.

Frame Number	Time Offset	Source	Destination	Description
509	13.2000912	10.81.153.115	10.125.148.226	TCP:Flags=.....S., SrcPort=65504, DstPort=1910, PayloadLen=0, Seq=1750010279, Ack=0, Win=8192 (Negotiating scale factor 0x2) = 8192
510	13.2008556	10.125.148.226	10.81.153.115	TCP:Flags=...A..S., SrcPort=1910, DstPort=65504, PayloadLen=0, Seq=19797847, Ack=1750010280, Win=8192 (Negotiated scale factor 0x8) = 2097152
511	13.2045016	10.81.153.115	10.125.148.226	TCP:Flags=...A...., SrcPort=65504, DstPort=1910, PayloadLen=0, Seq=1750010280, Ack=19797848, Win=4182 (scale factor 0x2) = 16728
512	13.2047565	10.81.153.115	10.125.148.226	TCP:Flags=...AP..., SrcPort=65504, DstPort=1910, PayloadLen=16, Seq=1750010280 - 1750010296, Ack=19797848, Win=4182 (scale factor 0x2) = 16728
513	13.2058729	10.81.153.115	10.125.148.226	TCP:Flags=...AP..., SrcPort=65504, DstPort=1910, PayloadLen=1394, Seq=1750010296 - 1750011690, Ack=19797848, Win=4182 (scale factor 0x2) = 16728
514	13.2059315	10.125.148.226	10.81.153.115	TCP:Flags=...A...., SrcPort=1910, DstPort=65504, PayloadLen=0, Seq=19797848, Ack=1750011690, Win=261 (scale factor 0x8) = 66816
515	13.2059417	10.81.153.115	10.125.148.226	TCP:Flags=...AP..., SrcPort=65504, DstPort=1910, PayloadLen=29, Seq=1750011690 - 1750011719, Ack=19797848, Win=4182 (scale factor 0x2) = 16728
516	13.2065696	10.81.153.115	10.125.148.226	TCP:Flags=...AP..., SrcPort=65504, DstPort=1910, PayloadLen=1349, Seq=1750011719 - 1750013068, Ack=19797848, Win=4182 (scale factor 0x2) = 16728
517	13.2066193	10.125.148.226	10.81.153.115	TCP:Flags=...A...., SrcPort=1910, DstPort=65504, PayloadLen=0, Seq=19797848, Ack=1750013068, Win=255 (scale factor 0x8) = 65280
518	13.2097036	10.81.153.115	10.125.148.226	TCP:Flags=...AP..., SrcPort=65504, DstPort=1910, PayloadLen=1394, Seq=1750013068 - 1750014462, Ack=19797848, Win=4182 (scale factor 0x2) = 16728
519	13.2099286	10.81.153.115	10.125.148.226	TCP:Flags=...AP..F, SrcPort=65504, DstPort=1910, PayloadLen=824, Seq=1750014462 - 1750015287, Ack=19797848, Win=4182 (scale factor 0x2) = 16728
520	13.2099902	10.125.148.226	10.81.153.115	TCP:Flags=...A...., SrcPort=1910, DstPort=65504, PayloadLen=0, Seq=19797848, Ack=1750015287, Win=261 (scale factor 0x8) = 66816
521	13.2112105	10.125.148.226	10.81.153.115	TCP:Flags=...A...F, SrcPort=1910, DstPort=65504, PayloadLen=0, Seq=19797848, Ack=1750015287, Win=261 (scale factor 0x8) = 66816
522	13.2141345	10.81.153.115	10.125.148.226	TCP:Flags=...A...., SrcPort=65504, DstPort=1910, PayloadLen=0, Seq=1750015287, Ack=19797849, Win=4182 (scale factor 0x2) = 16728

3.1.8 UDP Header

As mentioned previously the **User Datagram Protocol (UDP)** is connectionless and unreliable Transport layer protocol and does not meet the strict definition of a Transport layer in the OSI model. Nevertheless it is classified as a Transport layer protocol for historical reasons.

The figure below shows the UDP header. Notice that UDP allows for an additional addressing parameter (to that in the IP layer) in the form of port numbers. This allows for application level addressing much like TCP. However unlike TCP, there are no Acknowledgements, Sequence numbers or Data buffer Windows. This means that UDP packets can get lost, received out of sequence or tossed for lack of buffer. One can think of UDP as essentially IP traffic with the additional ability to use port number to associate itself with applications within a node in an IP network.

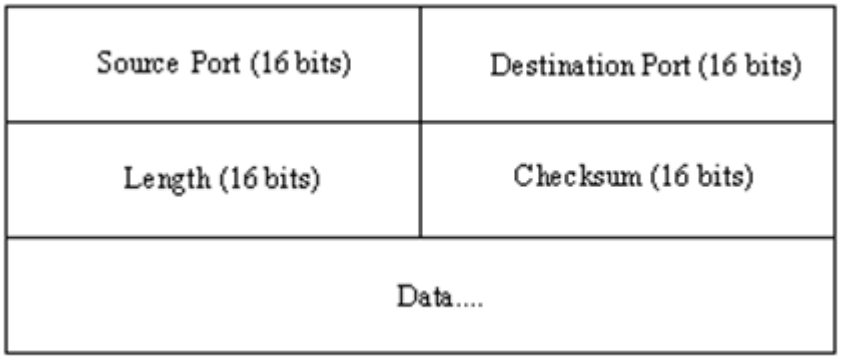

Figure 3.1.8.1

3.1.9 UDP Client Sample Source Code

The following is a simple UDP client sample that shows how to use the Windows Socket interface to access the UDP Transport Layer in the Microsoft Windows Operating system.

```
// UDPClient.cpp :

#include "stdafx.h"
#include <winsock2.h>

//Constants
#define    MAX_LOADSTRING                  100
#define    MAX_BUFLEN                      512
#define    GET_HOST_NAME_FAILURE           1
#define    SOCKET_FAILURE                  2
#define    SENDTO_FAILURE                  3
#define    TEST_PORT                       1910


int _tmain(int argc, _TCHAR* argv[])
{
   char          szHostId[MAX_LOADSTRING];
   WSADATA       info;
   SOCKET        s;
   struct        sockaddr_in server;
   char          buffer[MAX_BUFLEN];
   char          buf_num[MAX_BUFLEN];
   struct        hostent *h;
   BOOL          retVal = FALSE;
   int           errorCode=0, lineNum=0;

   WSAStartup(MAKEWORD(1,1), &info);

   /* Look up the server in the host file */
   strcpy(szHostId, "skg10");
   if( (h = gethostbyname(szHostId)) == NULL )
      {errorCode= GET_HOST_NAME_FAILURE; lineNum=__LINE__; goto exit;}

   /* Create a datagram socket */
   if( (s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == SOCKET_ERROR )
      {errorCode= SOCKET_FAILURE; lineNum=__LINE__; goto exit;}

   /* Fill in the server's address */
   server.sin_family = AF_INET;
   memcpy(&server.sin_addr, h->h_addr, h->h_length);
   server.sin_port   = htons(TEST_PORT);

   /* Copy a greeting message into the buffer */
   strcpy(buffer, "Hello from SKG");

   /* Send it to the Server */
   if( sendto(s, buffer, strlen(buffer)+1, 0, (LPSOCKADDR)&server, sizeof(struct sockaddr)) == SOCKET_ERROR)
      {errorCode= SENDTO_FAILURE; lineNum=__LINE__; goto exit;}

   for (int i=0; i < 100; i++)
   {
      strcpy(buffer, "This is a longer message to send");
      if( sendto(s, buffer, strlen(buffer)+1, 0, (LPSOCKADDR)&server, sizeof(struct sockaddr)) == SOCKET_ERROR)
         {errorCode= SENDTO_FAILURE; lineNum=__LINE__; goto exit;}
      sprintf(buf_num, "Buffer Number %d", i);
      if( sendto(s, buf_num, strlen(buf_num)+1, 0, (LPSOCKADDR)&server, sizeof(struct sockaddr)) == SOCKET_ERROR)
         {errorCode= SENDTO_FAILURE; lineNum=__LINE__; goto exit;}
   }

   retVal = TRUE;
exit:
   if(retVal == FALSE)
   {
      switch(errorCode)
      {
         case( GET_HOST_NAME_FAILURE ):
            printf("UDPClient: Failure Detected in \"gethostbyname\" on line %d LastError=%d!!", lineNum, GetLastError());
            break;
```

```
        case( SOCKET_FAILURE ):
           printf("UDPClient: Failure Detected in call to \"socket\" on line %d LastError=%d!!", lineNum, GetLastError());
           break;
        case( SENDTO_FAILURE ):
           printf("UDPClient: Failure Detected in call to \"sendto\" on line %d LastError=%d!!", lineNum, GetLastError());
           break;
     }
  }

  /* Disconnect from the Server */
  if( s != INVALID_SOCKET ) closesocket(s);
  WSACleanup();

  return(0);
}
```

3.1.10 UDP Server Sample Source Code

The following is a simple UDP server sample that shows how to use the Windows Socket interface to access the UDP Transport Layer in the Microsoft Windows Operating system.

```
// UDPServer.cpp :

#include "stdafx.h"
#include "stdafx.h"
#include <winsock2.h>

#define    MAX_BUFLEN                1600
#define    SOCKET_FAILURE            1
#define    BIND_FAILURE              4
#define    RECVFROM_FAILURE          6
#define    TEST_PORT                 1910


int _tmain(int argc, _TCHAR* argv[])
{
  SOCKET        sock = INVALID_SOCKET;
  SOCKET        msgsock=INVALID_SOCKET;
  int           client_len=sizeof(SOCKADDR);
  int           bytes_read = 0;
  char          buf[MAX_BUFLEN];
  struct        sockaddr_in server, client;
  WSADATA       wsadata;
  BOOL          retVal = FALSE;
  int           errorCode=0, lineNum=0;


  /* Initialize the socket library */
  WSAStartup(MAKEWORD(1,1), &wsadata);

  /* Create a datagram socket to listen on IPPROTO_UDP */
  if( (sock=socket(AF_INET, SOCK_DGRAM,IPPROTO_UDP)) == SOCKET_ERROR )
     {errorCode= SOCKET_FAILURE; lineNum=__LINE__; goto exit;}

  server.sin_family      = AF_INET;
  server.sin_addr.s_addr = htonl(INADDR_ANY);
  server.sin_port        = htons(TEST_PORT);

  if (bind(sock, (struct sockaddr *) &server, sizeof (server)) == SOCKET_ERROR)
     {errorCode= BIND_FAILURE; lineNum=__LINE__; goto exit;}

  fprintf(stderr, "UDP Test server: Version 1.00\r\n\n");

  /* Wait for the client to "connect" */
```

```
   while (1)
   {
      fprintf(stderr, "Waiting for UDP Client to Send datagram...\r\n");

      /* Show how many bytes arrive in each packet*/
      do
      {
         if( (bytes_read = recvfrom(sock, buf, sizeof(buf), 0, (LPSOCKADDR)&client, &client_len)) == SOCKET_ERROR)
               {errorCode= RECVFROM_FAILURE; lineNum=__LINE__; goto exit;}
         fprintf(stderr, "RECV'd %d bytes\r\n", bytes_read);
      }while(bytes_read);
   }

   retVal = TRUE;

exit:
   if(retVal == FALSE)
   {
      switch(errorCode)
      {
         case( SOCKET_FAILURE ):
            printf("UDPServer: Failure Detected in call to \"socket\" on line %d LastError=%d!!", lineNum, GetLastError());
            break;
         case( BIND_FAILURE ):
            printf("UDPServer: Failure Detected in call to \"bind\" on line %d LastError=%d!!", lineNum, GetLastError());
            break;
         case( RECVFROM_FAILURE ):
            printf("UDPServer: Failure Detected in \"recvfrom\" on line %d LastError=%d!!", lineNum, GetLastError());
            break;
      }
   }


   /* Disconnect from the Server */
   if( sock != INVALID_SOCKET ) closesocket(sock);
   WSACleanup();

   return 0;
}
```

3.1.11 UDP Sample Network Trace

The following is a capture of some of the physical layer traffic associated with the samples above. Note that there is one frame corresponding every UDP send in the sample. Also note that there no ACKs in response to the sends.

Frame Number	Time Offset	Source	Destination	Protocol Name	Description
66	1.9860585	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 23
67	1.9890646	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 41
68	1.9891643	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 24
69	1.9892494	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 41
70	1.9893345	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 24
71	1.9894184	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 41
72	1.9894975	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 24
73	1.9895813	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 41
74	1.9896647	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 24
75	1.9897486	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 41
76	1.9898311	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 24
78	1.9899389	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 41
79	1.9900227	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 24
80	1.9901053	10.81.153.132	10.125.148.226	UDP	UDP:SrcPort = 49938, DstPort = 1910, Length = 41

3.2 Universal Serial Bus (USB)

When I first had the opportunity to study USB, I felt overwhelmed with the amount of disjoint information on the subject. First there is the USB 2.0 specification (assuming you are using USB 2.0) which is a voluminous 622 page document. Then there is the Host Controller Interface Specifications; there are 3 different varieties of those (EHCI, OHCI and UHCI). Finally each USB class is defined by its own standard. It takes a while for the student of USB to put all this information in some sort of context.

Once you gain a basic perspective how the USB protocols accomplish the fundamental task of communication, you will suddenly find that every source of information on the topic that you had previously thought didn't make any sense, suddenly appears very comprehensible. Further, once you reach that point, it is often sufficient to use this wide array of documentation purely as reference material.

The background information on Digital Communications that we have discussed thus far puts us in a better position to fully appreciate some of the motivations for the USB protocol. Nevertheless there are some circular dependencies that make presenting this subject a little challenging. While I will do my best to present the material is some hierarchical fashion, avoiding reference to abstract specifications as far as possible, know that it may take a few readings to digest the subject matter.

3.2.1 Motivation for USB

As the Personal Computer (PC) revolution was underway in the nineties, it became obvious that some standardization was required in interfacing with computer peripherals. Up until then, each peripheral had its own physical interface. Most computers provided for a serial port, a parallel port, a keyboard port and a mouse port. The only extension capability to add a different physical port was to use an ISA card that attached to the ISA bus on the motherboard and exposed its own port. Graphics cards were a common example of this and they provided another port for the display monitor. Any data acquisition peripheral would similarly attach to the ISA bus. However, ISA slots were also limited and not easily accessible to end users.

The brilliance of USB was not just that it addressed the standardization of physical ports, it also leveraged the knowledge of the bandwidth demands of varying peripherals and did a reasonable job of accommodating most peripherals in common use. In the sections below we will see how this is done. The following lists some of the milestones in the evolution of USB (Note that some of the terms will be clearer later):

- USB 1.0 – Spec published in 1996 - Low speed 1.5Mbps, Full speed 12Mbps.
- USB 1.1 – Spec published in 1998 – Addition of Interrupt OUT transfer type.
- USB 2.0 – Spec published in 2000 – Addition of High Speed (40 times full speed – 480Mb/s).
- USB OTG – Spec published in 2001 – Attempt to overcome peer-to-peer communication limitation

3.2.2 USB Terminology

As with any subject, nomenclature can be a stumbling block until it is understood. Once understood however, it adds substantially to the ease of conveying ideas more precisely. The following are some of the common terminologies used in USB discussions:

- **Host** – Master on USB bus. There can only be one Host in a USB bus. The Host is responsible for enumeration (detecting a function and loading its driver), bandwidth management, error checking, power supply (500mA for bus powered devices and 100mA for self-powered devices) and data exchange. The host is always the initiator of activity on the bus.
- **Function** – Slave on USB bus. There can be a maximum of 127 function devices on a USB bus. A function is responsible for filtering data based on address, respond to directed requests, error checking, limiting power consumption (500mA/100mA during Active state and 2.5mA during suspend) and data exchange (respond to requests from the host).
- **Hub** – Allows for additional ports on the USB bus. The Host exposes a "**root**" hub with a limited number of ports. An external hub can be plugged into any port on the root hub or another external hub to expose additional ports. The responsibilities of a hub are similar to a function with the additional task of playing host to the functions attached to its ports.
- **Device** – The term "USB device" is used to refer to either a function or a hub.
- **Compound Device** – A single physical device that has multiple functions attached to an internal hub.
- **Composite Device** – A single physical device that has multiple functions, any one or more of which may be active. These functions are not attached to an internal hub, instead enumerate separately over a single port.

3.2.3 USB Physical Layer

A USB device is hooked up to Computer using a USB cable. The cable has 4 wires (+5V power, Ground, twisted pair data). The end that attaches to the computer is often called the "A" end and the end that attaches to the device is called the "B" end. The computer is referred to as a "Host" and the device is referred to as a "Function".

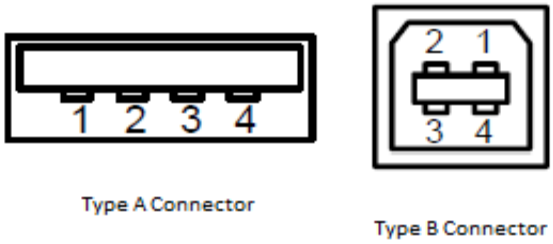

Figure 3.2.3.1

Pin 1: V_{Bus} is 5V.
Pin 2: Data (D-)
Pin 3: Data (D+)
Pin 4: Gnd

In addition to the form factors displayed above, there are "Mini" and "Micro" form factors which are also in common use.

The maximum allowed cable length is about 5 meters. This is primarily because the USB protocol only allows for a maximum round-trip delay of about 1500ns.

The encoding used for data is a differential NRZI with bit stuffing. Bit stuffing is a technique that helps with clock synchronization in the event that there are 6 consecutive 1s in the bit stream. In this case, the transmitter forces a “1” to “0” and back to “1” transition and the receiver is expected to use the stuffed bit for clock synchronization but then discard it from the data. Note that USB does not use a separate clock line.

Figure 3.2.3.1

Actual signaling methods are dependent on the speed (Low, Full or High).

3.2.4 USB Enumeration

The process of enumeration begins when a USB device is physically connected to a host and ends when the appropriate USB class driver takes ownership for the device. The following are some of the tasks involved in the process:

- Speed Negotiation between Host and Function
- Host Assigns an Address
- Host queries for Descriptors and identifies appropriate Driver
- Driver selects appropriate configuration

We will discuss each of these phases below.

Speed Negotiation between Host and Function

All full-speed and high-speed USB devices pull up the D+ line with a 1.5KOhm resistor which causes the D+ line to stabilize at high level in approximately 200ms. This tells the host that the device connected is either a full-speed or a high-speed device.

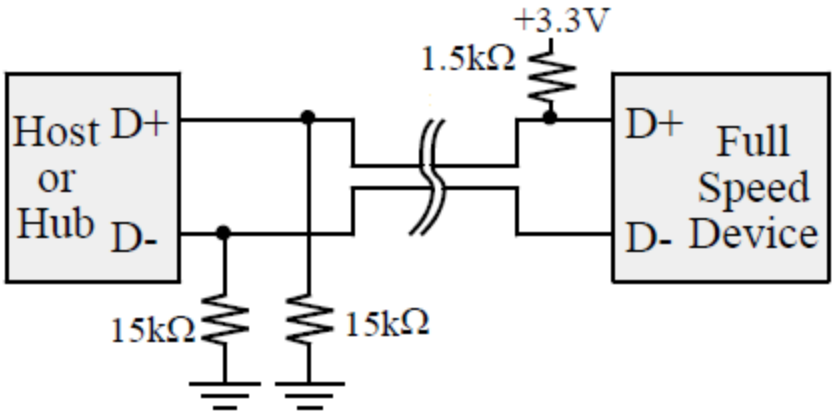

Figure 3.2.4.1

All low-speed devices pull up the D- line with 1.5KOhm resistor which causes the D- line to stabilize at high voltage level in approximately 200ms. This tells the host that the device connected is a low-speed device.

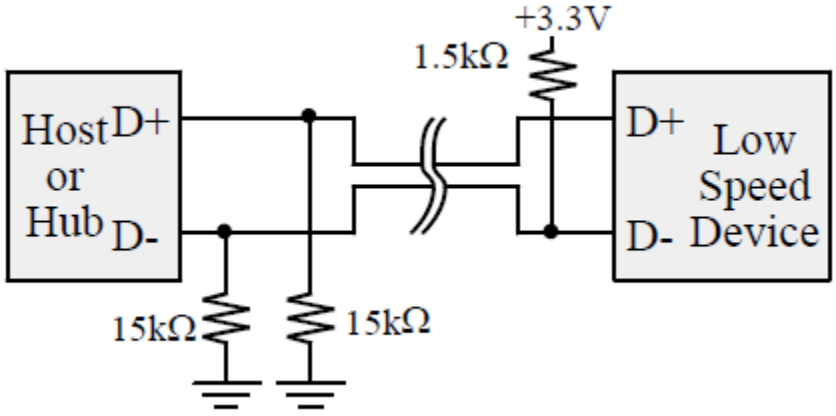

Figure 3.2.4.2

The host will then issue a RESET on the connected port to force both the D+ and D- lines to the low level state (Single Ended Zero – SE0) for at least 10ms.

If the device is capable of High Speed transmission, it will source current into the D- line to issue a slight toggle on that line for approximately 1ms to 7ms immediately after the host RESET. This is also referred to as the “Chirp K” (A slight toggle on the D+ line is referred to a “Chirp J”).

Within 100us after detecting a “Chirp K” from the device, the host will issue an alternating sequence of Chirp Ks and Chirp Js. Once the device detects 3 sets of KJ chirps, it has 500us to transition to high speed operation.

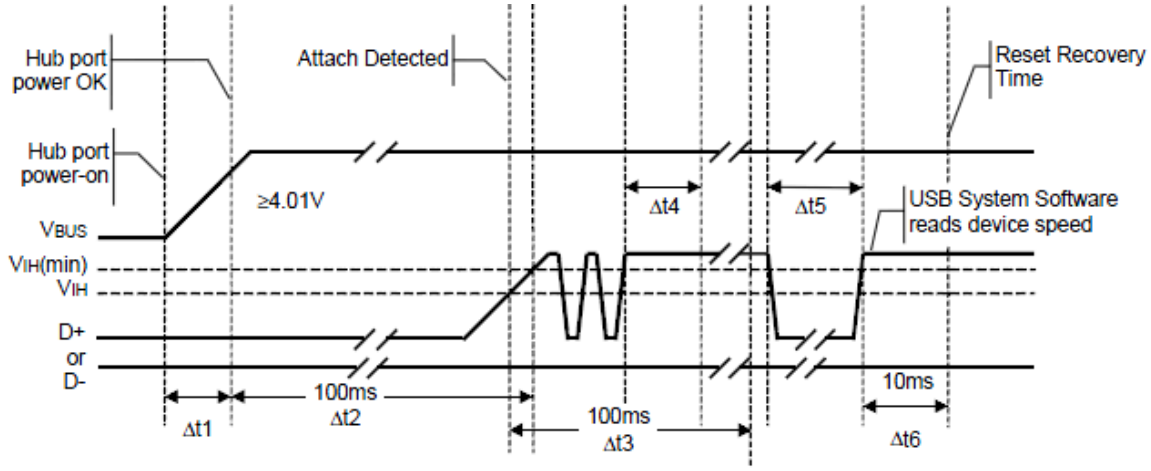

Figure 3.2.4.3 (Figure 7-29 in USB 2.0 spec – See spec section 7.1.7.3 for timings)

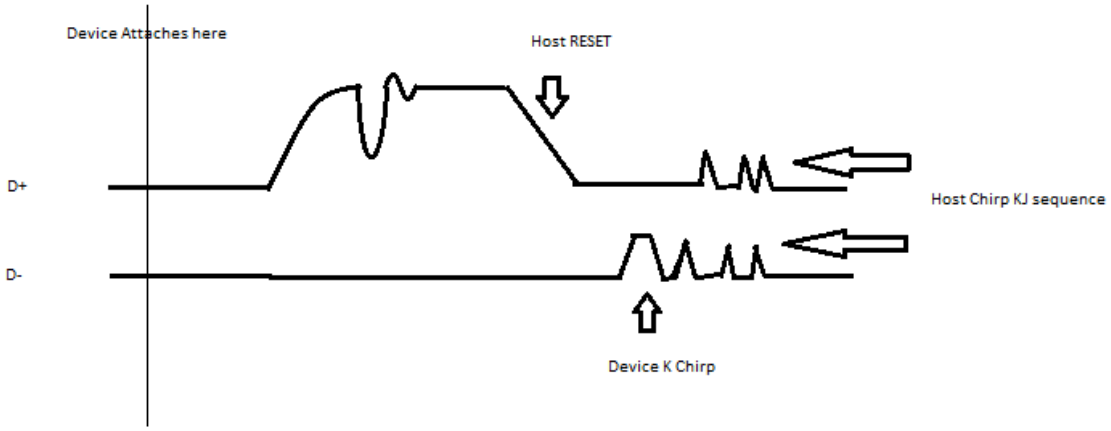

Figure 3.2.4.4 (Showing chirps during High-speed detection)

Once speed negotiation is complete the host takes the device off the reset state.

Host Assigns an Address

USB communications occur over “Endpoints” and they can be viewed as sources or sinks of data. All devices must support a special Endpoint referred to as Endpoint Zero. This is the Endpoint that will receive control and status communication during enumeration. The early communications to a newly enumerated device will be addressed to device “0” and Endpoint “0”. Since the host will only enumerate one device at a time, Device “0” is the currently enumerating device. One of the first transactions to EndPoint zero will be a “**Set_Address**” request. This will give the device a unique address on the USB bus.

Host queries for Descriptors and identifies appropriate Driver

Once an address is assigned, the host will then issue a “Get_Descriptor” request to the new address to read the “Device Descriptor”. The USB protocol defines several kinds of “Descriptors”, each of which gives information on the type of device, configuration and interfaces supported by the device. We will discuss this in more detail in the next section. Suffice to say that the host will use this information to identify the appropriate driver that can communicate with this device and load that driver. This driver is also referred to as the “class driver” since it represents a particular class of USB devices.

Driver selects appropriate configuration

Once the driver loads, it will first select the configuration it is willing to support and then open additional channels of communication (also called pipes) to the device.

3.2.5 USB Descriptors

USB devices indicate their capabilities to the host with the help of Descriptors. You can think of Descriptors as data structures with fields that expose the capabilities of a device. There are five common types of Descriptors defined by the USB Spec.

- Device Descriptors
- Configuration Descriptors
- Interface Descriptors
- Endpoint Descriptors
- String Descriptors

Descriptor Hierarchy

The diagram below shows the relationship between descriptors for a device with a single configuration, 2 interfaces and each interface with 2 endpoints.

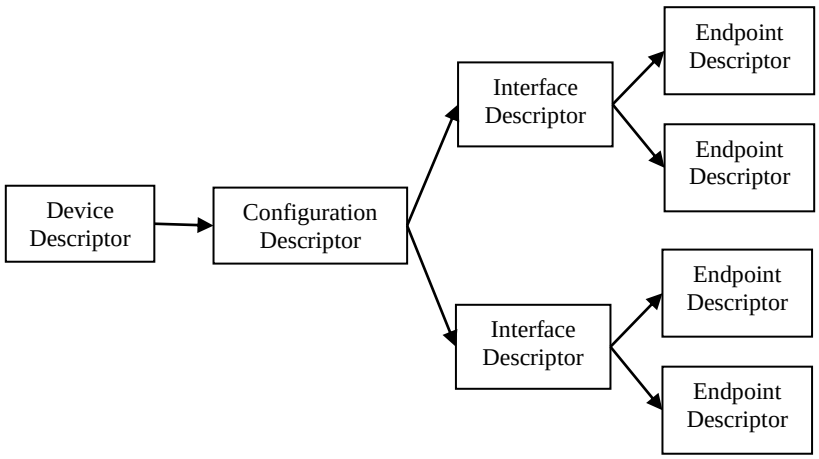

Figure 3.2.5.1

Device Descriptors

The following is how the USB spec defines the Device Descriptor:

A device descriptor describes general information about a USB device. It includes information that applies globally to the device and all of the device's configurations. A USB device has only one device descriptor.

A high-speed capable device that has different device information for full-speed and high-speed must also have a device_qualifier descriptor (see Section 9.6.2).

The device descriptor of a high-speed capable device has a version number of 2.0 (0200H). If the device is full-speed only or low-speed only, this version number indicates that it will respond correctly to a request for the device_qualifier desciptor (i.e., it will respond with a request error).

The *bcdUSB* field contains a BCD version number. The value of the *bcdUSB* field is 0xJJMN for version JJ.M.N (JJ – major version number, M – minor version number, N – sub-minor version number), e.g., version 2.1.3 is represented with value 0x0213 and version 2.0 is represented with a value of 0x0200.

The *bNumConfigurations* field indicates the number of configurations at the current operating speed. Configurations for the other operating speed are not included in the count. If there are specific configurations of the device for specific speeds, the *bNumConfigurations* field only reflects the number of configurations for a single speed, not the total number of configurations for both speeds.

If the device is operating at high-speed, the *bMaxPacketSize0* field must be 64, indicating a 64 byte maximum packet. High-speed operation does not allow other maximum packet sizes for the control endpoint (endpoint 0).

All USB devices have a Default Control Pipe. The maximum packet size of a device's Default Control Pipe is described in the device descriptor. Endpoints specific to a configuration and its interface(s) are described in the configuration descriptor. A configuration and its interface(s) do not include an endpoint descriptor for the Default Control Pipe. Other than the maximum packet size, the characteristics of the Default Control Pipe are defined by this specification and are the same for all USB devices.

The *bNumConfigurations* field identifies the number of configurations the device supports. Table 9-8 shows the standard device descriptor.

Table 9-8 in the USB 2.0 spec details the fields in the Device Descriptor as follows:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (18 bytes)
1	bDescriptorType	1	Constant	DEVICE Descriptor Type (0x01)
2	bcdUSB	2	BCD	USB spec version in Binary Coded Decimal
4	bDeviceClass	1	Class	Class code that reflects this type of device

5	bDeviceSubClass	1	SubClass	Subclass that reflects this type of device
6	bDeviceProtocol	1	Protocol	Protocol code – qualifies the class code
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero.
8	idVendor	2	ID	Vendor ID
10	idProduct	2	ID	Product ID
12	bcdDevice	2	BCD	Device Release version in Binary Coded Decimal
14	iManufacturer	1	Index	Index of the string descriptor describing manufacturer
15	iProduct	1	Index	Index of the string descriptor describing product
16	iSerialNumber	1	Index	Index of the string descriptor describing product
17	bNumConfigurations	1	Number	Number of possible configurations

Configuration Descriptors

The following is how the USB spec defines the Configuration Descriptor:

The configuration descriptor describes information about a specific device configuration. The descriptor contains a *bConfigurationValue* field with a value that, when used as a parameter to the SetConfiguration() request, causes the device to assume the described configuration.

The descriptor describes the number of interfaces provided by the configuration. Each interface may operate independently. For example, an ISDN device might be configured with two interfaces, each providing 64 Kb/s bi-directional channels that have separate data sources or sinks on the host. Another configuration might present the ISDN device as a single interface, bonding the two channels into one 128 Kb/s bi-directional channel.

When the host requests the configuration descriptor, all related interface and endpoint descriptors are returned (refer to Section 9.4.3).

A USB device has one or more configuration descriptors. Each configuration has one or more interfaces and each interface has zero or more endpoints. An endpoint is not shared among interfaces within a single configuration unless the endpoint is used by alternate settings of the same interface. Endpoints may be shared among interfaces that are part of different configurations without this restriction. Once configured, devices may support limited adjustments to the configuration. If a particular interface has alternate settings, an alternate may be selected after configuration. Table 9-10 shows the standard configuration descriptor.

Table 9-10 in the USB 2.0 spec details the fields in the Configuration Descriptor as follows:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	CONFIGURATION Descriptor Type (0x02)
2	wTotalLength	2	Number	Total Length of data returned (includes all interface and Endpoint Descriptors associated with this configuration)

4	bNumInterfaces	1	Number	Number of Interfaces
5	bConfigurationValue	1	Number	Use this value to select this configuration
6	iConfiguration	1	Index	String Descriptor index describing this configuration
7	bmAttributes	1	Bitmap	D7-Bus Powered, D6 Self Powered, D5 Remote Wakeup, D4..0- Reserved
8	bMaxPower	1	mA	Maximum Power Consumption

Interface Descriptors

The following is how the USB spec defines the Interface Descriptor:

The interface descriptor describes a specific interface within a configuration. A configuration provides one or more interfaces, each with zero or more endpoint descriptors describing a unique set of endpoints within the configuration. When a configuration supports more than one interface, the endpoint descriptors for a particular interface follow the interface descriptor in the data returned by the GetConfiguration() request. An interface descriptor is always returned as part of a configuration descriptor. Interface descriptors cannot be directly accessed with a GetDescriptor() or SetDescriptor() request.

An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always alternate setting zero. The SetInterface() request is used to select an alternate setting or to return to the default setting. The GetInterface() request returns the selected alternate setting.

Alternate settings allow a portion of the device configuration to be varied while other interfaces remain in operation. If a configuration has alternate settings for one or more of its interfaces, a separate interface descriptor and its associated endpoints are included for each setting.

If a device configuration supported a single interface with two alternate settings, the configuration descriptor would be followed by an interface descriptor with the *bInterfaceNumber* and *bAlternateSetting* fields set to zero and then the endpoint descriptors for that setting, followed by another interface descriptor and its associated endpoint descriptors. The second interface descriptor's *bInterfaceNumber* field would also be set to zero, but the *bAlternateSetting* field of the second interface descriptor would be set to one. If an interface uses only endpoint zero, no endpoint descriptors follow the interface descriptor. In this case, the *bNumEndpoints* field must be set to zero.

An interface descriptor never includes endpoint zero in the number of endpoints.

Table 9-12 in the USB 2.0 spec details the fields in the Interface Descriptor as follows:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes

1	bDescriptorType	1	Constant	Interface Descriptor Type (0x04)
2	bInterfaceNumber	1	Number	Number of Interface
3	bAlternateSetting	1	Number	Used to select an alternative setting
4	bNumEndpoints	1	Number	Number of Endpoints
5	bInterfaceClass	1	Class	Class Code
6	bInterfaceSubClass	1	SubClass	Subclass Code
7	bIntefaceProtocol	1	Protocol	Protocol Code
8	iInterface	1	Index	String Descriptor Index describing this Interface

Endpoint Descriptors

The following is how the USB spec defines the Endpoint Descriptor:

Each endpoint used for an interface has its own descriptor. This descriptor contains the information required by the host to determine the bandwidth requirements of each endpoint. An endpoint descriptor is always returned as part of the configuration information returned by a GetDescriptor(Configuration) request. An endpoint descriptor cannot be directly accessed with a GetDescriptor() or SetDescriptor() request. There is never an endpoint descriptor for endpoint zero. Table 9-13 shows the standard endpoint descriptor.

Table 9-13 in the USB 2.0 spec details the fields in the Endpoint Descriptor as follows:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	DEVICE Descriptor Type
2	bEndpointAddress	1	Endpoint	EP Address(bits3-0), Direction(bit7: 0-Out 1-In)
3	bmAttributes	1	Bitmap	Transfer Type(bits 1-0), Sync Type(bits 3-2)
4	wMaxPacketSize	2	Number	Maximum packet size this EP can buffer
6	bInterval	1	Number	Interval for polling EP defined in frames or microframes

String Descriptors

Table 9-15 in the USB 2.0 spec details the fields in the String Descriptor Zero as follows:

Offset	Field	Size	Value	Description
0	bLength	1	N+2	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	STRING Descriptor Type
2	wLANGID[0]	2	Number	LANGID code zero
…	…	…	…	…
N	wLANGID[x]	2	Number	LANGID code x

This is a special string index detailing the languages supported. When requesting a string descriptor, the requestor has to specify the desired language suing a 16-bit language ID that is defined in this table.

Note that String Descriptors are optional and not required to be supported by a Function device.

3.2.6 USB Packets

Packet Types

USB uses four types of packets in communication between the host and a function device as discussed in the table below. Note that the host is always responsible for initiating a transaction in the USB bus. The function only responds to requests from a host.

Packet Types	Description
Token	Token packets are used to describe the transaction that is to follow. There are four types of token packets: IN, OUT, SETUP, SOF. IN – Informs USB function that host is requesting a READ transaction OUT – Informs USB function that host is requesting a WRITE transaction SETUP – Used for configuration. SOF – Indicates a start of frame that is issued by the host every 1ms Note that “IN” and “OUT” are always with respect to the host. “IN” implies data into the host and “OUT” implies data out of the host.
Data	Data packets convey payload either from the host to the function (OUT) or from the function to the host (IN).
Handshake	There are three type of Handshake packets: ACK – Acknowledge packet has been successfully received NAK – Data is currently unavailable. STALL – Device in troubled state, require host intervention.
Special	These cover a range of special cases like when there are speed mismatches and packet splitting is require across high-speed and full-speed functions.

Packet fields & formats

Every packet starts with a **SYNC** field. This is a clock synchronization mechanism between the host and the function devices.

The SYNC field is usually followed by a Packet Identifier (**PID**) field. This is 4-bit value (transmitted as 8-bits) that identifies the type of packet that is about to be sent.

Most packets will have a **CRC** field at the end and this will be followed by an End-of-Packet (**EOP**) signal.

Token Packets will usually follow the PID with an Address (**ADDR**) field, identifying the function that the host is addressing. This will be followed by the Endpoint (**ENDP**) to further qualify the destination of the packet.

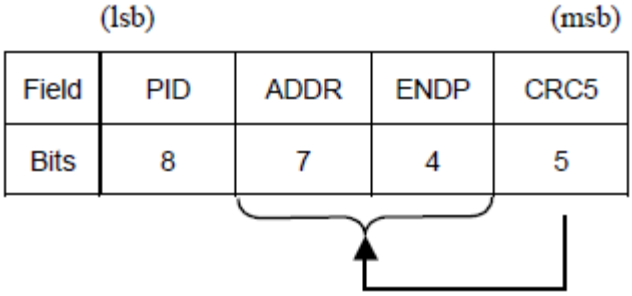

Figure 3.2.6.1

Data packets will contain the “SYNC”, “PID”, Data, “CRC” and “EOP” fields.

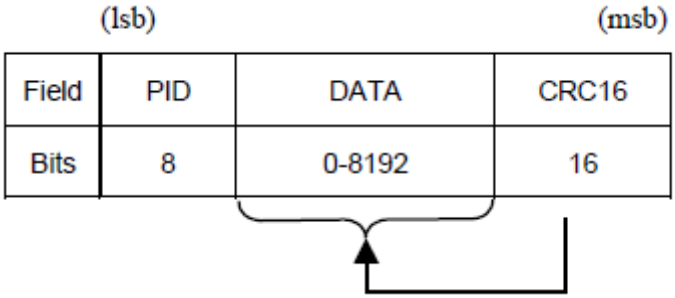

Figure 3.2.6.2

Handshake packets will contain the “SYNC”, “PID”, and “EOP” fields.

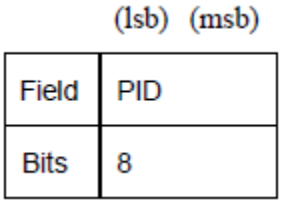

Figure 3.2.6.3

Start of Frame packets will contain the “SYNC”, “PID”, 11-bit Frame number, “CRC” and “EOP” fields.

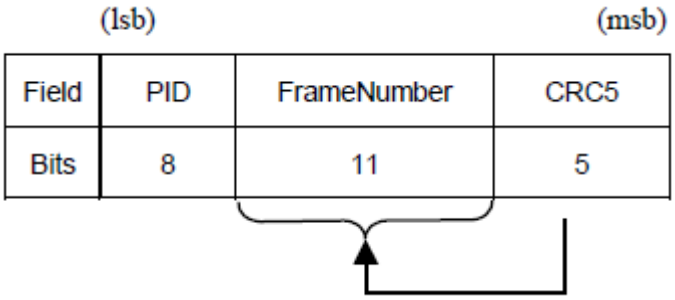

Figure 3.2.6.4

The USB 2.0 spec has the following description on Start of Frame packets:

USB defines a full-speed 1 ms frame time indicated by a Start Of Frame (SOF) packet each and every 1ms period with defined jitter tolerances. USB also defines a high-speed microframe with a 125 µs frame time. SOF packets are generated (by the host controller or hub transaction translator) every 1ms for full-speed links. SOF packets are also generated after the next seven 125 µs periods for high-speed links. High-speed devices see an SOF packet with the same frame number eight times (every 125 µs) during each 1 ms period. If desired, a high-speed device can locally determine a particular microframe "number" by detecting the SOF that had a different frame number than the previous SOF and treating that as the zeroth microframe. The next seven SOFs with the same frame number can be treated as microframes 1 through 7.

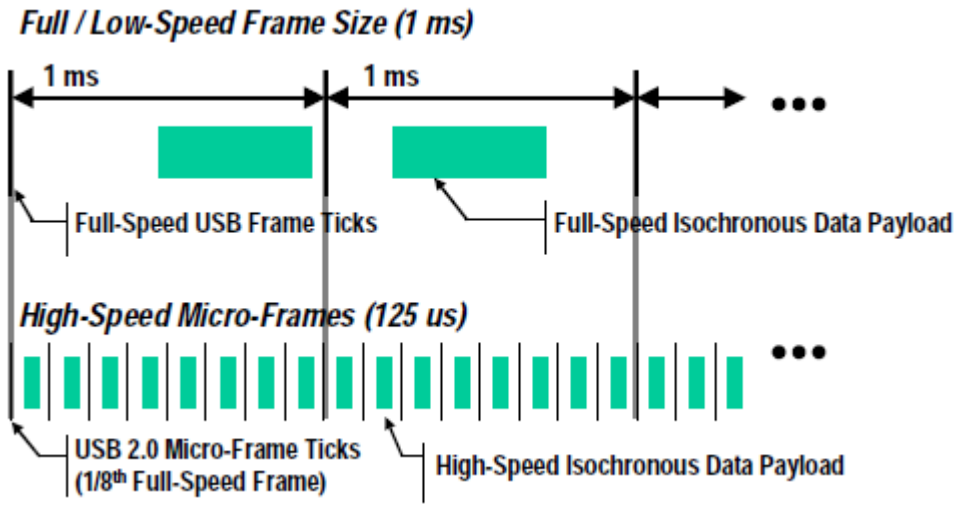

Figure 3.2.6.5

3.2.7 USB Pipes & EndPoints

A logical connection between a host and an Endpoint on the device is referred to as a **Pipe**. USB defines two kinds of pipes: **Stream pipes** and **Message pipes**. Message pipes have predefined formats while

stream pipes are free format. Message pipes are generally used only for control traffic. Stream pipes are used for general payload traffic.

USB classifies Endpoint as four types. The following table describes the distinctions between these Endpoints.

Endpoint Type	Description
Control	A control endpoint is primarily used for configuration and setup of a device. Most of the traffic over a control endpoint would be command and status operations. These are usually of random and bursty nature.
Interrupt	An interrupt endpoint is generally used when the payload is small but requires prompt attention. Both data integrity and guaranteed latency are the characteristic requirement for such traffic. A mouse or a keyboard would be ideal candidates for exposing interrupt endpoints. In general Interrupt endpoints are periodically polled by the host.
Isochronous	An isochronous endpoint is ideal for cases where a real-time stream of data is involved. In such cases data recovery in the event of data loss is less important than ensuring the real-time stream continues to be sent or received. Audio streaming is a good candidate for an Isochronous endpoint. In general Isochronous endpoints are periodically polled by the host.
Bulk	A bulk endpoint is used when large bursty data is involved. File transfer is a good candidate for a bulk endpoint. Here data integrity is the primary concern.

The type of USB transfer is usually named after the type of Endpoint involved.

The host is responsible for managing the total bandwidth available on the USB bus. The USB specification dictates that no more than 90% of any frame should be allocated to periodic transfers and on high-speed buses the requirement is 80% instead of 90%.

This implies that if there is enough Interrupt and Isochronous endpoint activity to consume the full periodic capacity within a given frame, then we are left with just 10% or 20% for Control and Bulk activity. Note also that Control transfers have priority over Bulk transfers.

The host maintains two queues to accomplish the goals of bandwidth allocation: A Periodic queue and an Asynchronous queue. Interrupt and Isochronous transfers are populated in the periodic queue while Control and Bulk transfers are populated in the asynchronous queue. Within each frame, the periodic queue is given the required priority and at a minimum the asynchronous queue will get 10-20% of the time slot.

In the following sections we will outline what the USB 2.0 spec demands of each transfer type in more detail.

3.2.8 USB Bulk Transfer (Section 8.5.2 of USB 2.0 spec)

Bulk transaction types are characterized by the ability to guarantee error-free delivery of data between the host and a function by means of error detection and retry. Bulk transactions use a three-phase transaction consisting of token, data, and handshake packets as shown in figure below. Under certain flow control and halt conditions, the data phase may be replaced with a handshake resulting in a two-phase transaction in which no data is transmitted. The PING and NYET packets must only be used with devices operating at high-speed.

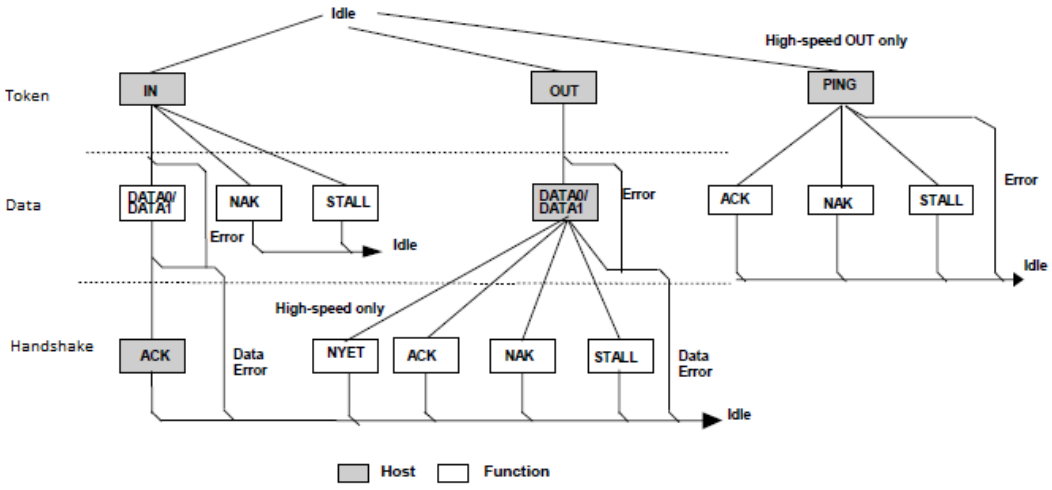

Figure 3.2.8.1

When the host is ready to receive bulk data, it issues an IN token. The function endpoint responds by returning either a data packet or, should it be unable to return data, a NAK or STALL handshake. NAK indicates that the function is temporarily unable to return data, while STALL indicates that the endpoint is permanently halted and requires USB System Software intervention. If the host receives a valid data packet, it responds with an ACK handshake. If the host detects an error while receiving data, it returns no handshake packet to the function.

When the host is ready to transmit bulk data, it first issues an OUT token packet followed by a data packet (or PING special token packet, see Section 8.5.1). If the data is received without error by the function, it will return one of three (or four including NYET, for a device operating at high-speed) handshakes:

- ACK indicates that the data packet was received without errors and informs the host that it may send the next packet in the sequence.
- NAK indicates that the data was received without error but that the host should resend the data because the function was in a temporary condition preventing it from accepting the data (e.g., buffer full).
- If the endpoint was halted, STALL is returned to indicate that the host should not retry the transmission because there is an error condition on the function.

If the data packet was received with a CRC or bit stuff error, no handshake is returned.

3.2.9 USB Control Transfer (Section 8.5.3 of USB 2.0 spec)

Control transfers minimally have two transaction stages: Setup and Status. A control transfer may optionally contain a Data stage between the Setup and Status stages. During the Setup stage, a SETUP transaction is used to transmit information to the control endpoint of a function. SETUP transactions are similar in format to an OUT but use a SETUP rather than an OUT PID. Figure below shows the SETUP transaction format. A SETUP always uses a DATA0 PID for the data field of the SETUP transaction. The function receiving a SETUP must accept the SETUP data and respond with ACK; if the data is corrupted, discard the data and return no handshake.

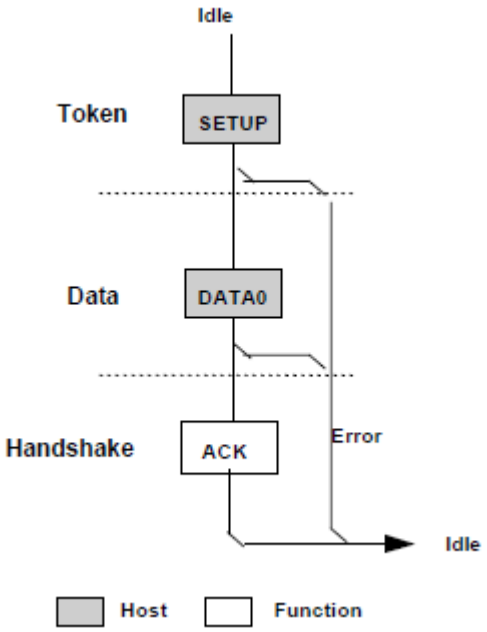

Figure 3.2.9.1

The Data stage, if present, of a control transfer consists of one or more IN or OUT transactions and follows the same protocol rules as bulk transfers. All the transactions in the Data stage must be in the same direction (i.e., all INs or all OUTs). The amount of data to be sent during the data stage and its direction are specified during the Setup stage. If the amount of data exceeds the pre-negotiated data packet size, the data is sent in multiple transactions (INs or OUTs) that carry the maximum packet size. Any remaining data is sent as a residual in the last transaction.

The Status stage of a control transfer is the last transaction in the sequence. The status stage transactions follow the same protocol sequence as bulk transactions. Status stage for devices operating at high-speed also includes the PING protocol. A Status stage is delineated by a change in direction of data flow from the previous stage and always uses a DATA1 PID. If, for example, the Data stage consists of OUTs, the status is a single IN transaction. If the control sequence has no Data stage, then it consists of a Setup stage followed by a Status stage consisting of an IN transaction.

3.2.10 USB Interrupt Transfer (Section 8.5.4 of USB 2.0 spec)

Interrupt transactions may consist of IN or OUT transfers. Upon receipt of an IN token, a function may return data, NAK, or STALL. If the endpoint has no new interrupt information to return (i.e., no interrupt is pending), the function returns a NAK handshake during the data phase. If the *Halt* feature is set for the interrupt endpoint, the function will return a STALL handshake. If an interrupt is pending, the function returns the interrupt information as a data packet. The host, in response to receipt of the data packet, issues either an ACK handshake if data was received error-free or returns no handshake if the data packet was received corrupted. Figure 8-38 shows the interrupt transaction format.

Section 5.9.1 (of USB 2.0 spec) contains additional information about high-speed, high-bandwidth interrupt endpoints. Such endpoints use multiple transactions in a micro-frame as defined in that section. Each transaction for a high-bandwidth endpoint follows the transaction format shown in the figure below.

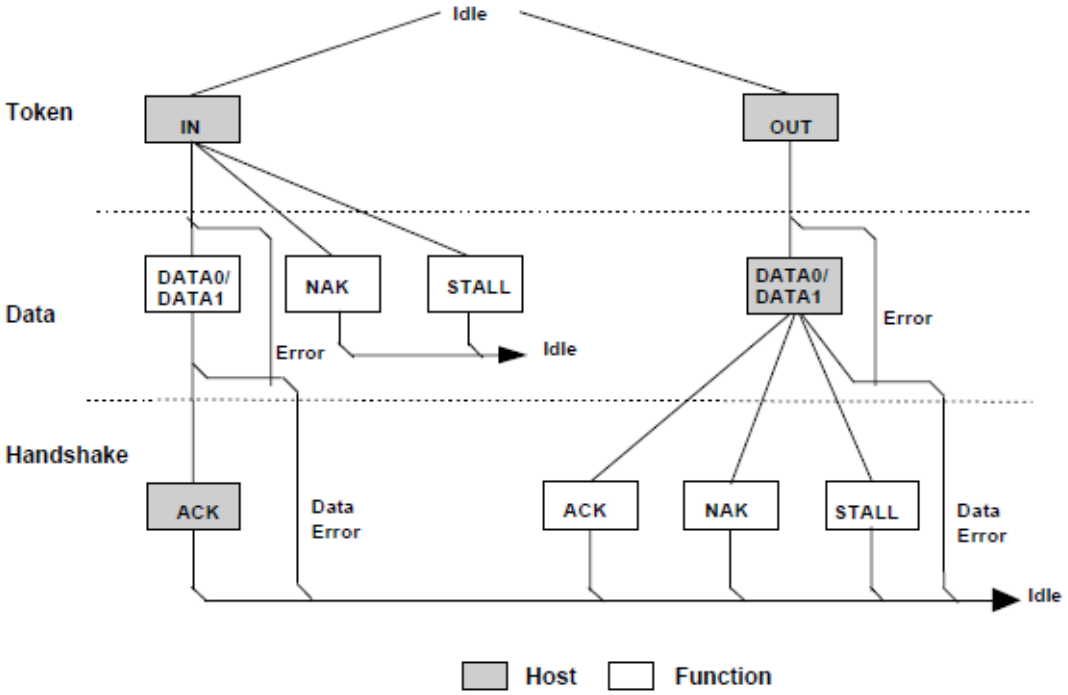

Figure 3.2.10.1

When an endpoint is using the interrupt transfer mechanism for actual interrupt data, the data toggle protocol must be followed. This allows the function to know that the data has been received by the host and the event condition may be cleared. This “guaranteed” delivery of events allows the function to only send the interrupt information until it has been received by the host rather than having to send the interrupt data every time the function is polled and until the USB System Software clears the interrupt condition. When used in the toggle mode, an interrupt endpoint is initialized to the DATA0 PID by any configuration event on the endpoint and behaves the same as the bulk transactions.

3.2.11 USB Isochronous Transfer (Section 8.5.5 of USB 2.0 spec)

Isochronous transactions have a token and data phase, but no handshake phase, as shown in Figure 3.2.11.1. The host issues either an IN or an OUT token followed by the data phase in which the endpoint (for INs) or the host (for OUTs) transmits data. Isochronous transactions do not support a handshake phase or retry capability.

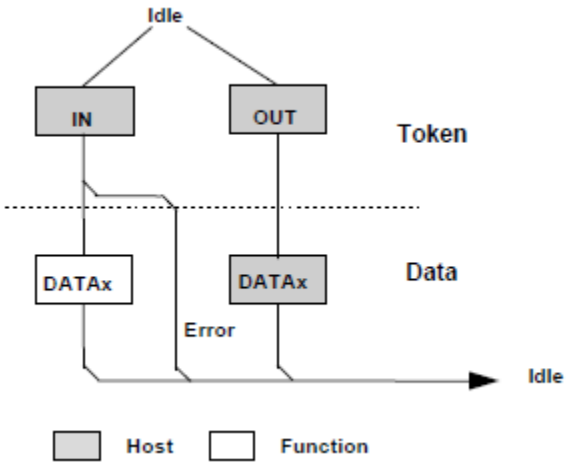

Figure 3.2.11.1

4.0 – Conclusion

This set of notes takes the student from the very basics in physical layer concepts all the way to how advanced communication protocols rely of software and hardware to accomplish the complex goals of network and peer-to-peer communication. The goal was to convey the concepts in digital communications rather than the detailed mathematical and engineering background. It is hoped that the student armed with this background conceptual knowledge will be better positioned to understand and contribute to other communication protocols that they are faced with.