# TCP/IP for Embedded Engineers, Session 404,424

**By Thomas F. Herbert**
**TFHerbert Consulting**

**herbert@tfherbert.net**

**http://www.tfherbert.net**

## Introduction

Most modern embedded systems are intended for a connected world. TCP/IP is the enabling technology to move data, video, and voice for diverse purposes. Embedded engineers are faced with requirements to put Internet connectivity in virtually everything they design and build. The TCP/IP protocol suite is the core technology for connectivity in the modern world. TCP/IP is the technology of choice for data, including video and voice. Web browsing and remote device configuration now also require TCP/IP.

TCP/IP stands for Transmission Control Protocol and Internet Protocol, two of the protocols in the protocol suite. Many embedded engineers are familiar with TCP/IP as a mechanism to access the Internet from their PCs or laptops. Or they know it as a source of problems when their PCs can no longer surf the internet. Now, these same engineers are responsible for designing and delivering an embedded system that implements the same TCP/IP protocols. Many engineers are not familiar with the memory requirements or the CPU performance necessary or the real-time constraints imposed on their system. After reading this article, any engineer familiar with embedded software, system, or hardware design can grasp the essentials necessary to obtain a TCP/IP stack and integrate it into their design. I will cover some of the choices in obtaining TCP/IP. For example, what are the places to get the protocols? What is available for my target hardware? Is it necessary to have a Commercial Off the Shelf (COTS) Real-Time Operating System (RTOS) to run the protocols?

This article introduces TCP/IP and its components and layers, along with issues that are relevant to embedded systems designers. There are many written materials to be used resources for network administrators. There are also excellent materials about the TCP/IP protocol and its implementation. In this article though, I will address issues relevant to TCP/IP selection, porting and implementation in embedded systems. In this article, I tend to take a low level view of the protocols therefore I emphasize what happens when a packet is received and transmitted at the lower layers of the protocol. After developing an understanding of the foundation below the networking applications, the you may continue and explore the upper layer protocols and services that give the internet the enormous value and interest that it has in today's world.

The TCP/IP protocols or "stack" are laid out in terms of the OSI layered model. The physical, logical, network and transport layers are introduced. Ethernet is used as an example of a specific physical layer. The network layer, IP is introduced including IP addressing. IP routing is not covered in this article. I cover transport methods such as TCP for connection oriented communication and UDP for connectionless message exchange. Also, I discuss APIs for networking applications such as TLI and sockets.

## History of TCP/IP

The TCP/IP protocols were first adopted by the DARPA Defense Advanced Research Projects Agency (DARPA ) as part of a project to connect primary government funded research institutions in a computer network. Around the same time at the University of California at Berkeley, the BSD (Berkeley Systems Distribution) version of Unix was widely distributed to university computer science departments. DARPA contracted to have the TCP/IP protocols merged into the Berkeley BSD operating system. Then, DARPA required anyone hooking up to the Internet to run the TCP/IP protocols.

Because of DARPA, the TCP/IP protocol source code was widely distributed as part of BSD. All TCP/IP source code leveraged from the BSD sources had to contain the University of California at Berkeley copyright. From this leveraged source code, vendors ported the TCP/IP protocols to most mainframe and mini-computers. Then as the embedded industry grew, RTOS vendors and separate protocol stack vendors ported the TCP/IP sources from BSD to many embedded

environments. Most of the products using TCP/IP today contain much of the unaltered code from Berkeley and much of it still carries the BSD Copyright.

## The OSI Layered Model

The Open System Interconnect (OSI) model was introduced in the seventies by the International Standards Organization (ISO) to promote inter-operability between diverse vendor's systems. The final OSI standard was released in 1984.

TCP/IP was already established while the ISO networking standards were evolving. Nowadays, it is common to explain all networking in terms of the OSI model because it introduces a common terminology to discuss issues related to computer networking.

The principle behind layering is each layer hides its implementation details from the layer below and the layer above. Each layer on the transmitting machine has a logical peer-to-peer connection with the corresponding layer in the receiving machine. This is accomplished through the use of encapsulation. Figure 1 shows the TCP/IP stack in terms of the OSI layers. In this illustration, the stack is shown in a typical LAN application. For WAN (Wide Area Network) or Point-to-point applications the lower layers can be somewhat more complicated.

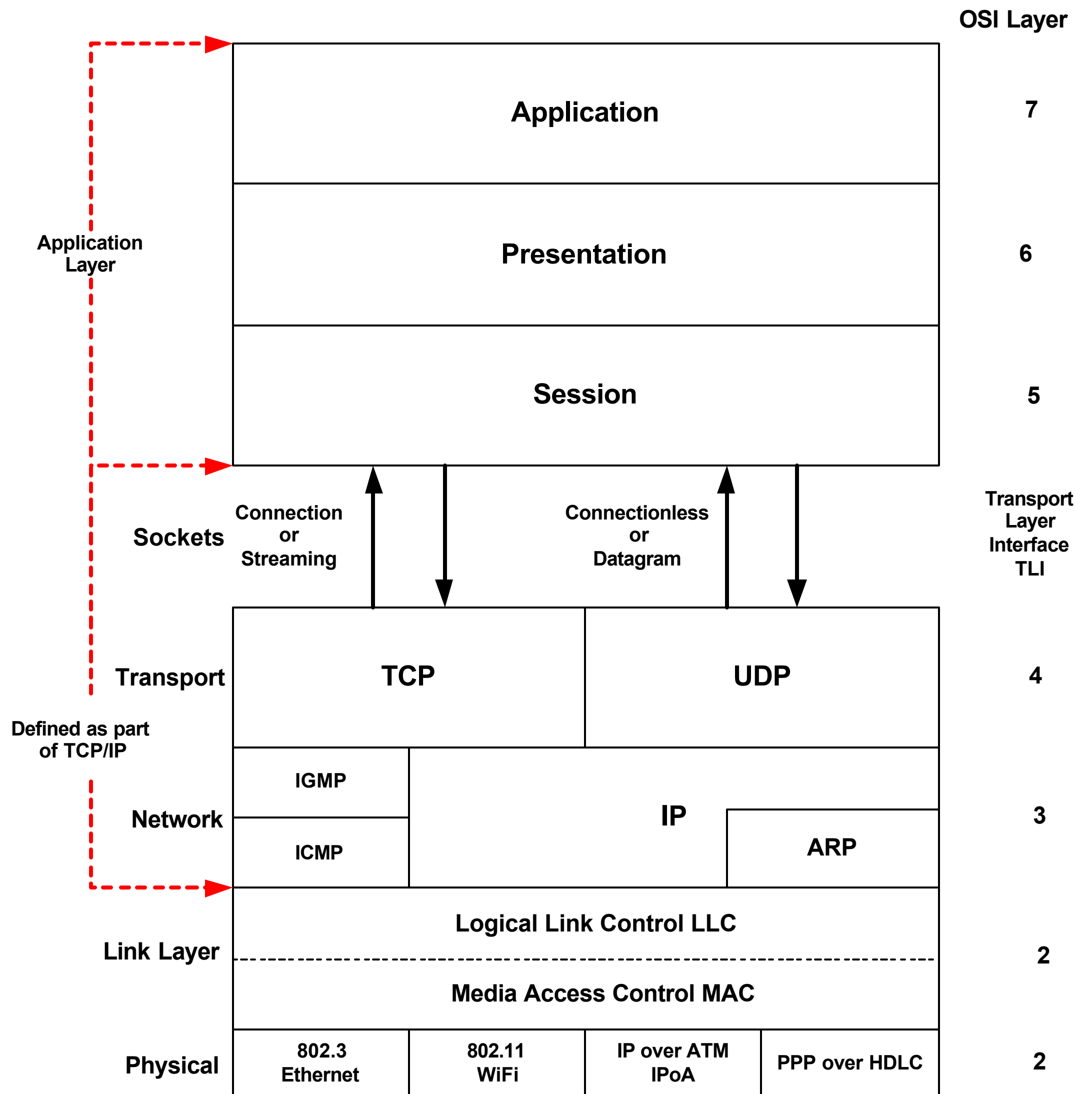

**Figure 1 TCP/IP Stack**

Each layer in a receiving machine gets received frames from the layer below and transmits sending frames to the layer above. Similarly each layer in a sending machine gets received frames from the layer above and transmits its frames to the layer below.

## 1. Physical

The physical layer, or PHY, encompasses the actual physical data transmission. Examples are Ethernet (CSMA/CD) or Token Ring for LAN applications or various high speed serial interfaces for WANs. Physical layer implementation is specific to the type of transmission. In this paper, I generally use Ethernet as an example of a PHY. Each machine on the physical network has an address called the Media Access Control (MAC) address. IN Wide Area Networks (WANs), this address is often called a station ID. Most MACs provide several addressing modes. They have an individual address that uniquely identifies each actual interface card. They also have a concept of multicast and broadcast addresses, where a transmitted frame can be shutgunned out to the wire with the target specified as all machines in the case of broadcast or some subset of machines in the case of multicast. In Ethernet, the multicast and broadcast addresses are implemented by setting magic bits in the MAC address.

## 2. Data Link

This layer isolates the network layer from the physical layer and the electrical or optical transmission details. Technically, the Data Link layer is responsible for presenting an interface for an error free transmission otherwise known as a connection oriented service to the layer above. Of course as discussed elsewhere, the legacy of TCP/IP is to not assume that the lower layers have any sort of error recovery. Therefore, most implementations of the link layer for TCP/IP don't implement any mechanism for reliability.

The data link layer can be thought of as having two internal sub-layers. The upper part handles the framing details and the interface with the upper layers. The lower part consists of the device driver including device management, interrupt handling, and DMA control.

As we shall see, there are two types of framing used for local area networks. In this paper, I talk about both types of framing. An example of one type of framing is 802.2 LLC. 802.2 calls for 3 types of service, connectionless, connection mode and acknowledged connection mode. In this paper, Two of these , the connection mode and the acknowledged connection mode are for providing a reliable or guaranteed channel where packets are acknowledged and sequenced. Since TCP/IP does not require an underlying reliable connection at the data link layer, this paper will only consider address connectionless service at this layer.

Also, it is up to the link layer to do some filtering of packets or frames. The most elemental filtering actually occurs at the physical layer (PHY). The PHY can be set to receive only packets with its own MAC in the destination field. It can also receive all packets with a hardware multicast or broadcast address. With Ethernet for example, any machine has the capability to sniff all packets on the network and this is called promiscuous mode.

## 3. Network

The network layer encompasses the Internet domain knowledge. This the layer that contains the routing protocols, and understands the Internet addressing scheme. Before a packet is transmitted or received, the IP addressing mechanism must be converted to or from Physical (MAC) addresses that can be understood by the link and physical layers.

## IP (Internet Protocol)

IP (Internet Protocol) [RFC 791] is the part of the stack known as the network layer. The network layer handles routing of packets across network interfaces. Domain naming and address management are considered to be part of this layer as well. PHYs are limited to a maximum packet size, Maximum Transmission Unit (MTU). Therefore, IP also includes a mechanism for fragmentation and re-assembly of packets that exceed the link's maximum size.

## IP Addressing

IP addresses are used to insure packets are routed to the correct destination. IPv4 includes three fundamental types of addresses, unicast, multicast and broadcast addresses. It is the convention to use the dotted decimal notation to describe IP addresses. It is generally more convenient to express netmasks as hex numbers. In addition to broadcast and multicast addresses, unicast addresses are divided into classes. The first three bits of the IP address determine the class [RFC 796]. The class of the address divides the 32 bit address between a network portion and a host portion. This determines the number of machines on that can be on a network. When TCP/IP was young, there weren't many users, the address space was sufficient and routers were expensive. In those days, class A and B addresses were common. Now they are not used very much any more because they allow too many hosts on a subnet. Generally, Class C addressing is used within the enterprise.

Also, since IP address space is very limited, a concept called Classless Interdomain Routing (CIDR) was developed. [RFC 1519] With this scheme is not necessary that the subnet be divided on a byte boundary. Instead the netmask determines the size of the network portion and the size of the host portion of the address. For example, we could have a subnet address allowing 512 network ID's and 128 host ID's. This address would be `0xffffff80`.

| Class | Lower Bound of network | Upper Bound of network | Lower bound of host | Upper Bound of Host |
|---|---|---|---|---|
| Default Route<br>Is all zeros | `0.0.0.0` | `0.0.0.0` | `0.0.0.0` | `0.0.0.0` |
| A - 8 bits net<br>24 bits host | `0.0.0.0` | `126.0.0.0` | `126.0.0.0` | `126.255.255.255` |
| Loopback | `127.0.0.0` | `127.0.0.0` | `127.0.0.0` | `127.x.x.x` |
| B - 16 bits net<br>16 bits host | `128.0.0.0` | `191.255.0.0` | `128.0.0.0` | `128.0.255.255` |
| C – 24 bits net<br>8 bits host | `192.0.0.0` | `223.255.255.0` | `192.0.0.0` | `192.0.0.255` |
| Multicast | `224.0.0.0` | | | `239.255.255.255` |

**Table 1 IP Addressing**

**ARP**

One important function of TCP/IP or any network stack is to convert the network layer address to the physical framing details necessary for transmission. For Ethernet interfaces, IP uses a protocol called Address Resolution Protocol (ARP). Although ARP part of the network layer, it can be considered either to be in the network or the link layer. In this paper, I will consider it to be a part of the network layer. The purpose of ARP is to map IP addresses used at the network layer into MAC addresses used at the link layer. We show how ARP is called from the link layer when we follow the packets as they are being transmitted later in this paper.

ARP has a dynamic table called the ARP cache which is used to do the mapping. If ARP does not have an entry in the cache, it determines the physical address from the IP address by sending out a packet and waiting for the response. ARP must receive packets from the link layer and incoming packets that have the ARP type in the type field are sent to ARP by the link layer.

## 4. Transport

The transport layer implements reliable sequenced packet delivery known as connection-oriented transfer and also provides the data as either a stream of bytes of discrete packets for the layers above. This layer contains the retrying and sequencing necessary to correct for lost information at the lower layers. The transport for TCP/IP provides two protocols, TCP for reliable or connection-oriented transmission and UDP for unreliable, or connectionless transmission.

Why do we need a reliable delivery mechanism at the transport layer? IP makes no assumptions about the reliability in the underlying data link and physical layers. When there is a requirement for a mechanism for reliable transfer, TCP is the protocol of choice. There is a long-standing debate between the Telephony world and the IP world about how much error checking and reliability needs to be designed into the lower layers. The WAN protocols which are usually deployed underneath TCP/IP provide a reliable connection oriented service which is used with telephony, therefore they have a great deal of error checking in the lower layers. TCP/IP is intended to work with LAN protocols, such as CSMA/CD, that drops packets at the physical layer when collisions occur or checksums don't match. However, with sufficient raw bandwidth, TCP can fix up transmission errors at layer 4 before noticeable data loss occurs.

More recently, TCP/IP is used for telephony and other time critical data distribution. IP contains a field called the Type of Service (TOS) field or Differentiated Services. Routers are supposed can be configured to make routing priority decisions based on the contents of this field. The TOS field values can be specified by the application through the transport layer interface, sockets.

When discussing TCP/IP, the transport layer refers to the application level API for the transport layer. There are actually two common API's, Transport Layer Interface (TLI) and sockets. Therefore, we usually refer to the application API as the transport layer. The transport layer provides an interface for both connectionless transport or connection-oriented transport. If the socket API is used, the connectionless sockets are called datagram sockets and the connection-oriented sockets are called stream sockets.

UDP – Unreliable or Connectionless Transport

UDP allows the application to send and receive only individual packets of data. Each packet is sent as a separate transmission. UDP only provides a simple interface to the IP layer. It has no mechanism to retry or retransmit lost packets. If the application has a large amount of

data to send, it has to worry about breaking up the data into individual small enough pieces. The network layer, IP does allow for fragmentation and reassemble, but generally UDP applications will pay attention to the Maximum Transmission Unit (MTU) and avoid sending packets that require excessive fragmentation. The most important aspect of UDP is that the routers in between the sending and receiving machine can change paths at any time, therefore the application can make no assumption that the packets were received in the order that they were sent.

TCP Connection Oriented or Streaming Sockets

This streaming type transport is implemented by the TCP protocol. The application can write a stream of continuous data to the socket and TCP breaks up the data into packets. It sequences the packets so they can arrive out of order but be put back in the order that they were sent at the receiving end. As part of this mechanism, TCP retransmits packets that were lost by the lower layers.

The TCP or stream type sockets are buffered full duplex. The application in the transmitting machine can write an undifferentiated stream of data in the socket as if it were a pipe and the application on the receiving machine will receive the same data in the same order.

**5. Session**

The session layer was originally conceived to support virtual terminal applications between remote login terminals and a central terminal server. In the early days of networking, when most users used computers through a terminal interface, session layer was intended to keep track of a logged in user talking to a central time-sharing system. Telnet is an example of a protocol that contains session management. Since TCP/IP only incorporates protocols through the transport layer, all the software above the transport layer is generally lumped together as networking applications so the session layer is not differentiated from the application layer

**6. Presentation**

The presentation layer maps between the user's view of the data as a structured entity and the lower layer protocol's view of the data as a stream of bytes. This is accomplished through Abstract Syntax Notation (ASN.1) which is a high level language that describes a user level representation of the data. The transport layer has no specific role in TCP/IP because TCP/IP is only implemented from the physical through the network layers. Therefore, any presentation level functionality is viewed as part of the application layer. Also, object exchange methods such as Distributed Common Object Model (DCOM) and Common Object Request Broker Architecture (CORBA) are really at the presentation layer, but are considered as part of the application layer.

**7. Application**

Application layer encompasses virtual all applications of TCP/IP networking including network file systems, web server or browser, or client server transaction protocols. Network management is really in the application layer although there are hooks down in the protocols to gather statistics and events and pass them up to the management application.

# Implementation

In this section, I discuss how the protocols are actually implemented in an embedded system. The information is general for many implementations although we will point out some critical differences between BSD and Linux implementations.

## Encapsulation

Encapsulation is used to isolate each of the layers in the protocol stack. Each layer frames the data in its own way and has its own header format. In the sending machine, the layer places its own header information in front of the data it gets from the layer above before passing it to the layer below. In the receiving machine, each layer first interprets and then strips the header information from frames received from the layer below before passing them up to the layer above. In reality, it is not quite so simple. In many cases, the headers are saved for use by the layers above.

## Buffer Management

TCP/IP breaks data down into a series of packets or frames. Buffers are created to hold the frames and the data contained in them. Because of the inherent asynchronous nature of a network, the buffers typically are allocated and de-allocated dynamically at a very high rate.

Essentially, there are three main requirements for any network buffer implementation.

- There must be no fragmentation.
- The network buffer implementation must allow buffers to be easily placed on queues without copying. Also, the it should provide the ability to build packets from lists of buffers also without copying.
- The buffer implementation should provide the ability to pre-pend and remove headers without copying the data in the packet.

There are several methods of buffer allocation. They can be deployed from a pre-allocated series of buffer pools or they can be allocated from the global memory pool when needed. They have variable lifetimes depending on any number of factors such as network throughput, whether the included frame is part of a stream transmission, or whether the packet is a datagram. An understanding of buffer management is fundamental to an understanding of the protocol implementation. Linux uses a method called the slab cache which does not require pre-allocated buffers but avoids fragmentation associated with the use of heaps.

### Mblock

Mblocks are the buffering mechanism used in STREAMS. It is conceptually similar to the Berkeley mbuffs which are used in most non-Linux TCP/IP implementations. Although not used widely, we mention it here because it is a relatively simple variable size fixed buffer scheme. The basic memory units in STREAMS are called message blocks or mblk_t and data blocks or dblk_t. The data blocks contain a pointer to an associated buffer. They also contain pointers used to keep track of the data's position in the buffer. The mblk_t is used to link messages on a message queue. Multiple message blocks may point to the same data to decrease the amount of copying of packets when multiplexing packets into multiple streams.

### Mbuffs

The basic buffering mechanism used in Berkeley-based TCP/IP is called mbufs, or memory buffers. Mbufs can be expanded to allow for variable length frames used often in IP based communications. The mbuf data structure is shown in table 2. Small amounts of data can be placed directly in the data area of the mbuf. Larger amounts of data require the mbuf to be extended with another data structure called a cluster. The data area of the mbuf generally holds header information but the cluster holds heterogeneous payload data for extending the data carrying capability of an mbuf. Mbufs can be chained to allow for easy implementation of packet queues as shown in Figure 2. The basic mbuf contains some header information with a data area

which can be extended with a cluster if the frame can't fit into the data area. Mbufs are also sometimes used to put control information or other temporary data used within the protocols.

| | |
|---|---|
| m_next | Next mbuf in chain of mbufs. A chain of 1 or more mbufs is used to contain a single packet. |
| m_nextpkt | Next mbuf in a queue of packets. |
| m_len | Contains total length of data in the mbuf including cluster if one is attached. |
| m_data | Address of the beginning of the data. This may point to a location in the data area in the bottom of the mbuf or it may point to a location within a cluster. M_data does not necessarily point to beginning of data area but can be incremented to where useable data starts. |
| m_type | |
| m_flags | Described in Table 3 below. |
| m_pkthdr.len | This location will be the beginning of data if flags field is 0. |
| m_pkthdr.rcvif | |

**Table 2 Fields in mbuf structure**

There are 4 types of mbufs. The mbuf type is differentiated by the m_flags field in the mbuf structure.

| | |
|---|---|
| 0 flags | Mbuf contains only data which can be up to 108 bytes. |
| M_PKTHDR | Mbuf contains a packet header and up to 100 bytes of data. |
| M_EXT | Mbuf is extended with a cluster allowing up to 2048 bytes of data. |
| M_PKTHDR \| M_EXT | Mbuf contains the packet header and a pointer to a cluster. This allows it to contain up to 2048 bytes of data. |

**Table 3 Types of mbufs.**

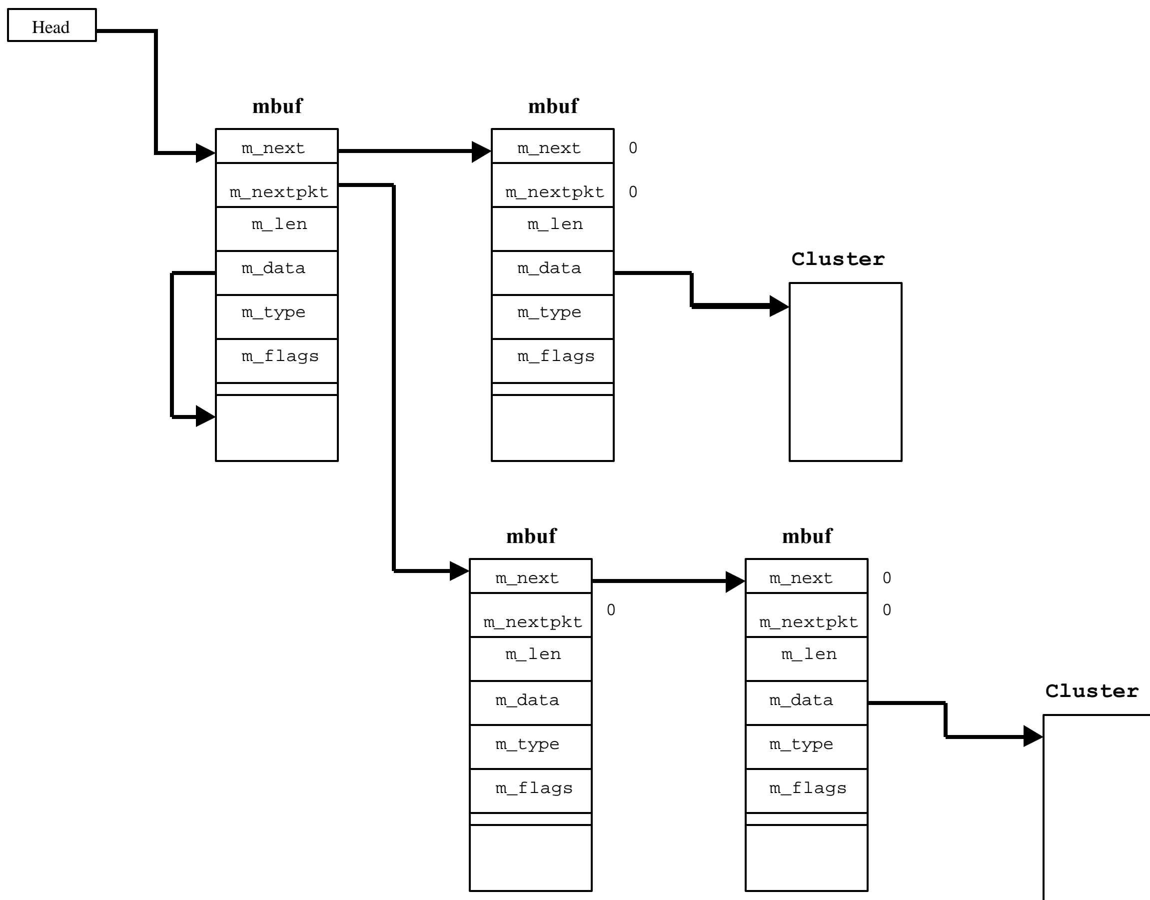


**Figure 2 Linking of mbufs**

## Linux Slab Cache

Linux takes a different approach from the BSD derived systems. While BSD systems use a separate allocator from the rest of the OS kernel to allocate network buffers, Linux uses the same slab allocator for network buffers that is used for all other kernel memory allocation.

The slab allocator also avoids fragmentation. Linux maintains a cache of network buffer structures and when each buffer is de-allocated it is returned to the cache. As with mbufs, each network buffer structure or sk_buff has an attached data buffer. In Linux the data buffers are allocated directly from the kernel's page allocation cache. These attached buffers are also returned to cache when they are de-allocated. The caches are specific for each type of object. Network buffers in the cache are re-initialized when they are de-allocated back to the cache. Therefore, each subsequent user does not have to spend time initializing common fields.

## Network Security

As there is more and more emphasis on network security, much of this concern has become important to us as embedded systems engineers and some of the aspects of network security can be addressed by TCP/IP implementations in embedded systems. There are three fundamental types of attacks on networks.

- Denial of Service (Dos)
- Modification or deleting of information.

- Eavesdropping or unauthorized sniffing of packets.

Also there are a few essential parts of network security.

- Confidentiality – information should be protected by reading by unauthorized parties.
- Integrity – ability to detect changes to either stored or transmitted information.
- Obligation – no party involved in a transaction can deny the sending, receiving or storing of information.

The TCP/IP stack itself does not address all aspects of network security by incorporating changes within the stack itself. However there are a few components fundamental to TCP/IP security.

The first part is IP security (IPSec) for confidentiality. To assure confidentially, we use encryption and this is generally done with IPSec in combination with Internet Key Exchange (IKE). In most implementations, IPSec is implemented outside TCP/IP itself. However, recent versions of Linux (version 2.6 and later) include a kernel implementation of IPsec called XFRM.

The second part is protection of DoS attacks. Some modern TCP and IP implementations have the ability to prevent DoS attacks by incorporating fixes in TCP, ICMP and IP. For example, the recent versions of Linux (mostly after 2.2) include changes to reduce DoS attacks.

A third part of security in TCP/IP is the Secure Socket Layer (SSL). SSL is implemented by a library above TCP and does not involve any changes to the TCP/IP stack itself.

## A Note on IPv6

The internet protocol is now at least 24 years old. As noted in the introduction Ipv4 is quite old and has earned a few grey hairs. It was originally specified in 1980 [RFC760]. In the early days of IP, it was never envisioned how popular the internet would become. The most fundamental limitation of IPv4 is its limited address size. IPv4 addresses are 32 bits long limiting the total number of addresses. Back in the early 1990s, when there began to be dramatic increases in the usage of the internet, it was becoming clear that the address space was too restricted. There were simply not enough addresses available because of limitation of the four byte address in IPv4. In the early 1990s, people started to think about ways to expand the address space. Most organizations on the internet were assigned network IDs from the class C address space and this only made three bytes available. An innovation called Classless Inter-domain Routing (CIDR) was developed in 1993 [RFC 1519] to free up network IDs by removing restrictions of the class based addressing scheme. Earlier in this paper, we discuss the IPv4 addressing scheme in detail. This has kept us going until now, but it is anticipated that we will run out of address space "soon".

In the mid 1990s, IPv6 was specified. [RFC1883] IPv6 increases the address length to 128 bits which is enough for practically every person place or thing to have a unique Ipv6 address. As of 2004, the world is still largely running IPv4 but IPv6 is starting to enjoy wider use. Right now, in the 2.6 kernel, Linux IPv6 is fairly stable and is becoming the focus of attention for those working on IPv6 protocols and utilities.

There is much more to IPv6 than a longer network address. IPv6 has a completely different network layer implementation.

- It has the capability of auto-configuring addresses. Basic addresses can be formed by incorporating the 6 byte MAC or link layer (Ethernet) addresses. This auto-configuring mechanism eliminates the need for Dynamic Host Configuration Protocol (DHCP) which is commonly used to configure addresses in IPv4.

- Also, with IPv6, hosts can dynamically discover each other or discover the default router using the built in protocol called Neighbor Discover Protocol (NDP) [RFC2461] . ND is similar to ARP but has quite a bit more capability.
- Another difference is that IPv6 has Quality of Service (QoS) built into the protocol specification. In contrast, IPv4 does QOS based routing by adding to the basic protocol generally by using a combination of the IP TOS field, the use of 802.1P extended headers, and RSVP or similar protocols. Using these mechanisms, modern routers do a pretty good job of QoS routing in IPv4, there really is no built in QoS capability in the IPv4 protocol.
- The basic IPv6 header is simpler than IPv4. There are also extended headers to provide IP options or IP fragmentation, or IP routing information.

The basic IPv6 header is shown in Figure 3. This is to facilitate quick packet processing because quick IP processing may not need to check the extended headers. The next header field in the basic header and each extended header points to the start of the next header. A fragment header is shown in Figure 4 which is a good example of a typical extended header. The IPv6 header types include the fragment header, routing header or destination header. These headers have a recommended order.

1. Hop-by-hop Options Header
2. Destination Options Header
3. Routing Header
4. Fragment Header
5. Upper layer Header

The upper layer headers include UDP and TCP. These headers follow the IPv6 option headers. The format for the upper layer headers is identical to the format for IPv4. The values for the next-header fields are shown in Table 4.

| Name | Value | Header Description |
|---|---|---|
| NEXTHDR_HOP | 0 | The Hop-by-hop option header. |
| NEXTHDR_TCP | 6 | The next header is a TCP header. |
| NEXTHDR_UDP | 17 | The next header is a UDP header. |
| NEXTHDR_IPV6 | 41 | The next header is for IPv6 in IPv6 tunnel. |
| NEXTHDR_ROUTING | 43 | The next header is the routing header. |
| NEXTHDR_FRAGMENT | 44 | Header for fragmentation or reassembly. |
| NEXTHDR_ESP | 50 | Header for Encapsulating Security Payload (ESP). |
| NEXTHDR_AUTH | 51 | This is for the authentication header. |
| NEXTHDR_ICMP | 58 | IPv6 ICMP |
| NEXTHDR_NONE | 59 | This value means that there is no next header. |

| NEXTHDR_DEST | 60 | This is the destination options header. |
|---|---|---|
| NEXTHDR_MAX | 255 | This is the maximum value for the next-header field. |

**Table 4 Next-header Field Values**

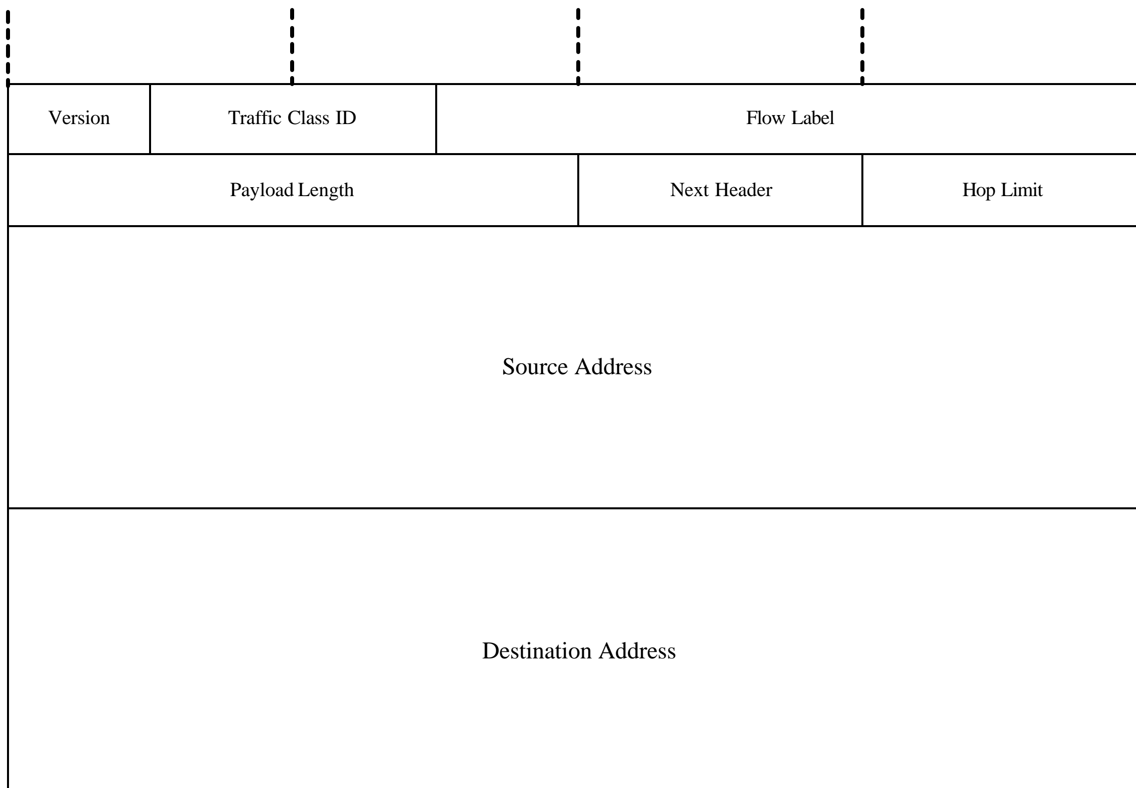


**Figure 3 IPv6 Basic Header Format**

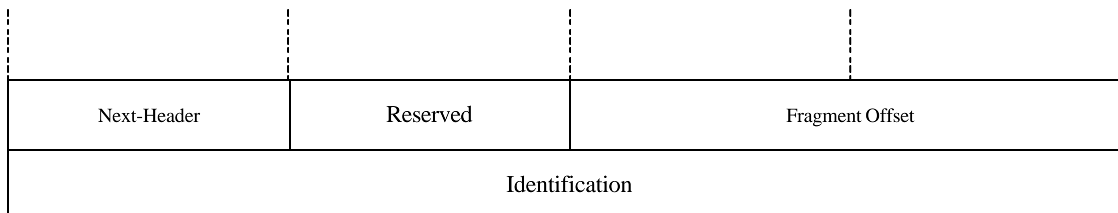

**Figure 4 IPv6 Fragment Header Format**

## The Data's Journey

The best way to understand any protocol is to follow the data as it travels from the application layer, down the stack of the sending machine, out the wire and up the stack of the receiving machine.

Typically, an application scribbles the data in a socket. A socket can be thought of as a mated pair of logical endpoints. One endpoint is on the sending machine and one is on the receiving machine.

### Down the Stack we go

Here is how a piece of information actually travels down the TCP/IP stack. The first thing that the application or user layer does is create a socket with the socket() system call.

**Sending data via UDP or connectionless transmission**

For example, lets assume that the user wishes to send a single message to another system via the UDP or datagram protocol. To use UDP, the application specifies the SOCK_DGRAM protocol in the socket() system call. For example, the user invokes the sendmsg() system call passing a message pointer as an argument. Figure 5 shows the UDP encapsulation, the header information that is prepended to the user data. The user call blocks until there is sufficient space for the message in the send buffer of the UDP protocol. When there is sufficient space available, the message is placed into a chain of mbufs until the complete message is queued for transmission. If the implementation supports zero copy buffers, to avoid copying the data, an mbuf structure is allocated with the cluster pointers pointing the user data area. The UDP protocol then prepends its header to the data. It calculates the checksum and calls ip_output() to send the frame.

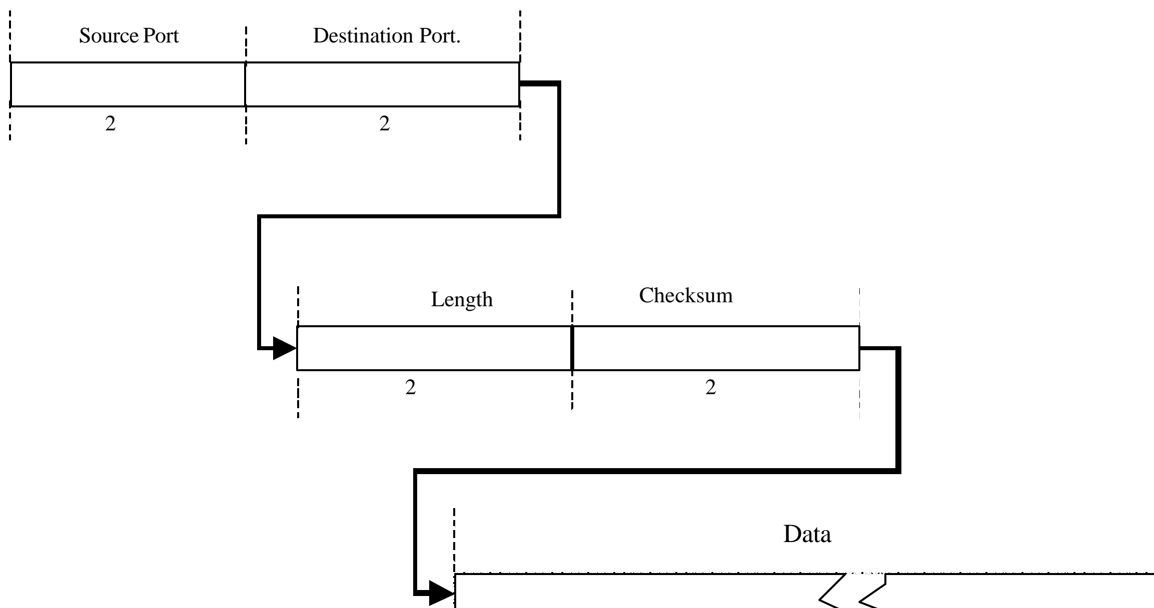


**Figure 5 UDP Encapsulation**

**Sending streaming data via TCP or SOCK_STREAM protocol**

If the user wishes to transmit a reliable stream of data, they use TCP by specifying the SOCK_STREAM protocol in the socket() system call. On the machine that wishes to use a remote service or the client end, the connect() call is used to establish a connection between the local socket and the remote endpoint. The connect() call waits until the connection is established. The TCP protocol maintains a state machine to keep track of the connection. Once the connect() call returns, the socket is ready to accept data.

The application can then write data in the socket with sendmsg() or another write call. The user data is accumulated in the send buffer of TCP. If your implementation supports zero-copy sockets, the data is not copied. It is tied up in mbuf and cluster structures by using pointer manipulation only.

When a streaming socket receives the data, it breaks the data into frames and allocates buffers to hold the data. The TCP headers are then placed in front of the data and the frame is passed down to the IP layer. The data is split up into individual frames to insure that the data length is less than the MTU.

Figure 6 shows the TCP encapsulation. Some of the fields shown in figure 6 are used to keep track of the state of the connection. For example, TCP uses the sequence number field and acknowledgement number field shown below to maintain the order of the individual datagrams. The state bits (Finish, Synch, Reset, Push, Acknowledge and Urgent) are used by the protocols on both end to keep track of the state of the connection and manage the establishment and break down of the connection.

It is important to note that the actual data transmission is buffered and asynchronous. The embedded system must have some kind of timer management to implement the retransmits necessary to compensate for lost packets. It is also must have a dynamic buffer mechanism to hold packets until they can be retransmitted or discarded.

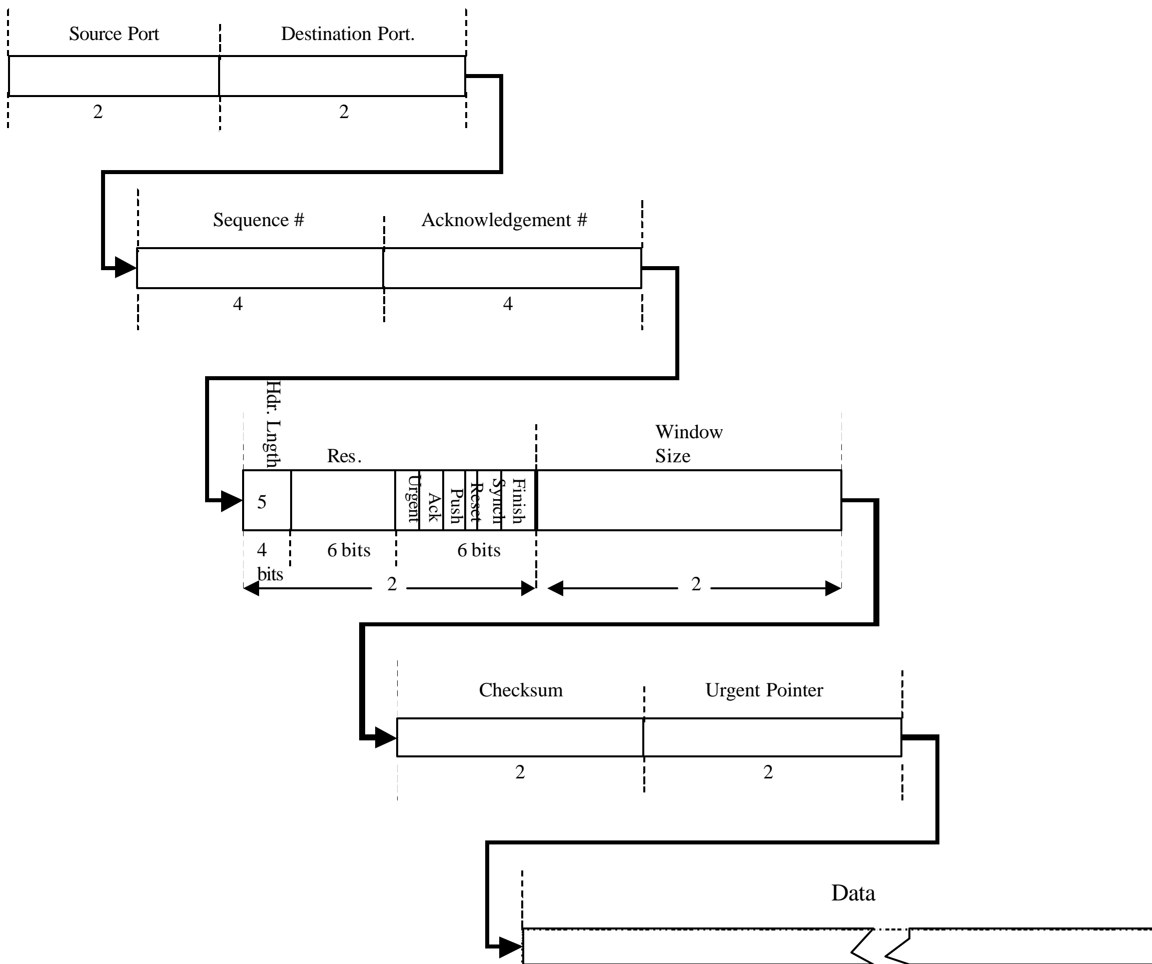


**Figure 6 TCP Encapsulation**

**IP layer Encapsulation and fragmentation**

IP layer encapsulation is shown in figure 7. The IP layer puts its headers on the front of the data it receives from the transport protocols. It also establishes the route for the packet according to its routing tables and inserts the appropriate address in the IP header. It calculates the checksum and sets the TTL (Time-to-live) field to 0 for packets originated above.

The transport protocols don't necessarily know the Maximum Transmission Unit (MTU) for a particular link layer. As discussed earlier, the MTU is the maximum packet size that can be transmitted across a PHY. In particular, streaming sockets used for connection-oriented transmission are completely independent of the implementation of the lower layers. IP's job to fragment the data into small enough packets to fit within the MTU. Routers or networking equipment between the sending and receiving machine may break the data into even smaller frames if the data must pass through link layers that have a smaller MTU than the one at the transmitters interface. IP keeps track of the packet fragmentation and assures that the packet is reassembled in the receiving machine in the same way it was assembled.

IP implements packet routing. Therefore, it has the IP forwarding capability if there are more than two interfaces multiplexed below it. Routing tables are maintained that IP uses to determine what

to gateway to send a packet in order to reach a particular final destination machine. In this paper I won't cover routing for that is a substantial topic on its own.

The examples I use in this paper are for IEEE 802.2 type link layer framing. It is important to note that many of the legacy BSD 4.3 implementations of TCP/IP have the IP layer written with hard coded assumptions about the link layer. They assume that the interface uses the old Ethernet type II framing. They could get away with it because packets with this type of framing can co-exist with IEEE 802.2 framed packets on the same Ethernet. The protocol wouldn't work with MACs other than Ethernet without modifying the TCP/IP sources.

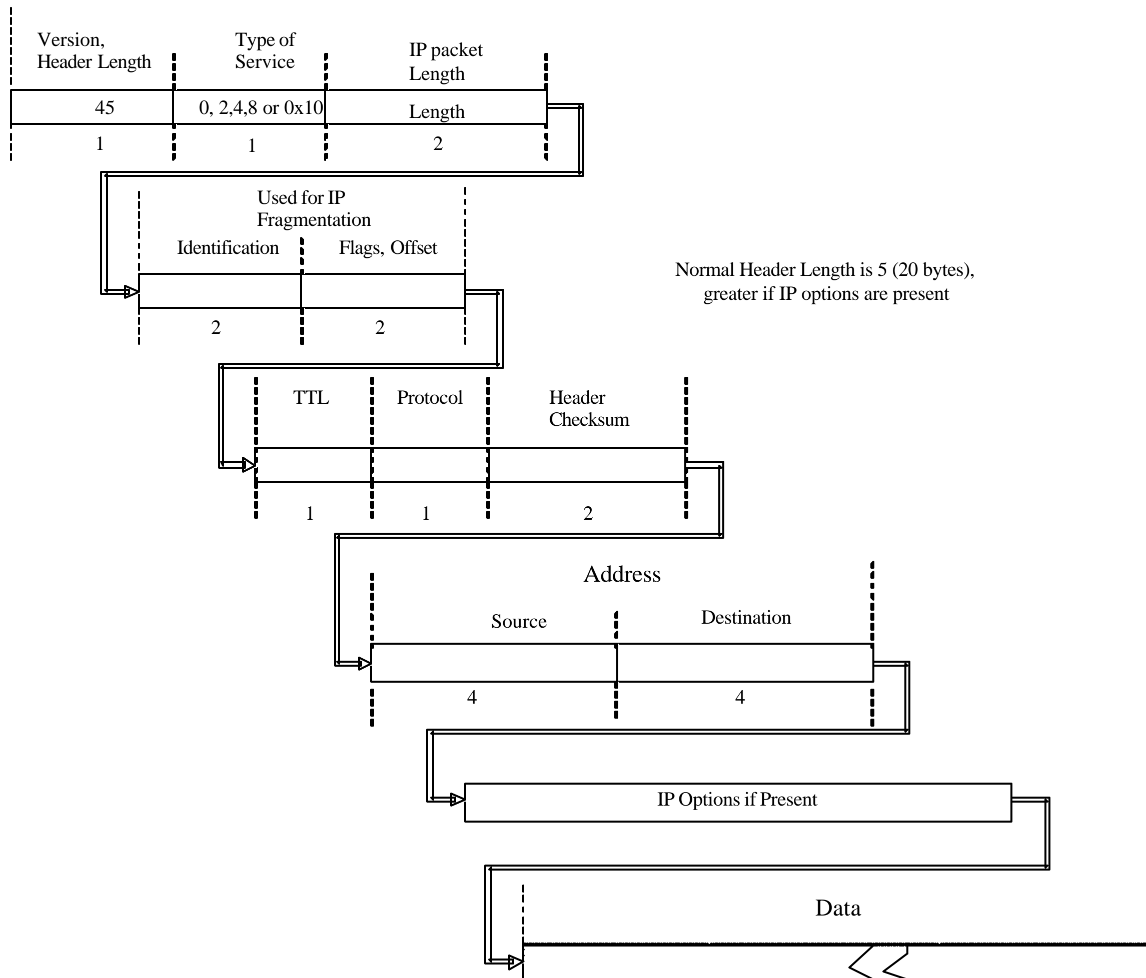


**Figure 7 IP Encapsulation**

## Link Layer Encapsulation

In this paper, I concentrate on 802.2 LLC encapsulation which is shown in figure 9. Since legacy TCP/IP ports such as BSD 4.3 were hard coded to handle Ethernet Type II encapsulation I show it in Figure 8 below. 802.3 type encapsulation which works with other link layers then Ethernet. Problems can occur with confusion between the two types of encapsulation if the TCP/IP stack was originally written to assume Ethernet type II type encapsulation and the stack is later used over other link layers..

If link layer multiplexing is used, one of more separate instances of data link layers can receive an individual packet from . For each interface that is attached to IP, it calls the link-layers-output function to transmit an IP frame. The packet is passed along with a pointer to the destination address in a sockaddr{} structure. For ethernet drivers, the output function calls etherOutput(). EtherOutput calls addrResolve() to convert the IP address in the sockaddr{} structure to a MAC address.

This conversion from IP to MAC addresses is done by the Address Resolution Protocol (ARP). When ARP is called by the link layer it is passed a pointer to a frame and a pointer to the sockaddr{} structure. It then looks in its tables to see if the address has already been resolved. If it finds the address, it queues the packet to the interface for transmission with the correct MAC address. If the address is not already resolved, ARP sends a broadcast packet containing the IP address to see if an ARP server somewhere on the network knows the MAC address that corresponds with the enclosed IP address.

At this point, any intelligence from the IP layer about fragmentation, routing, or IP addressing is perceived as mere data by the link layer to be blindly buried in the LLC frame. The link layer puts the destination address in the frame which is passed to it from IP. Of course as discussed above, IP determines which IP address to use from its routing tables. Then, the source address is put in the frame and the length is calculated. For TCP/IP most of the stuff in the LLC and SNAP headers is hard coded as is shown in figure 9. The type field is set to the protocol type depending on whether it is an IP, ARP or RARP packet.

The link layer is generally thought of as having an upper and lower part. The upper part handles the encapsulation and the multiplexing with the layers above. The lower layer handles the device interface including DMA and interrupt handling.After prepending the LLC headers on front of the data, the lower half of the link layer picks up the data. The lower half sets up the DMA and hardware for frame transmission.

Each PHY has a maximum transmission length. The MTU (Maximum Transmission Unit) size is maintained by the link layer. For IP, it can be up to 32767 bytes or as short as a few hundred bytes. For Ethernet, the MTU is 1500 bytes. It is up to IP to see if the packet can fit into the MTU. When an interface or link layer is bound to the network layer or IP, the MTU size is made available along with other hardware characteristics. This is used by IP to determine whether to fragment the packet or not.

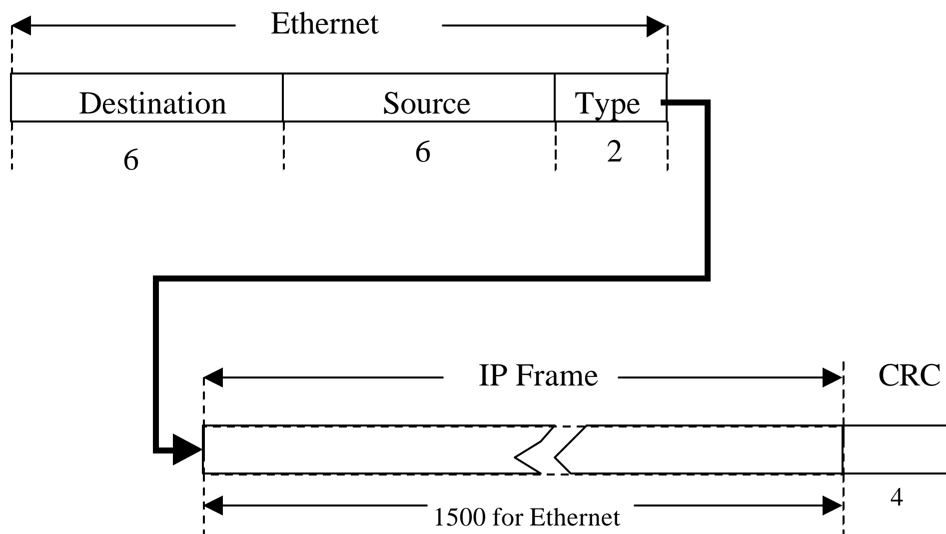

**Figure 8 Ethernet Type II Encapsulation**

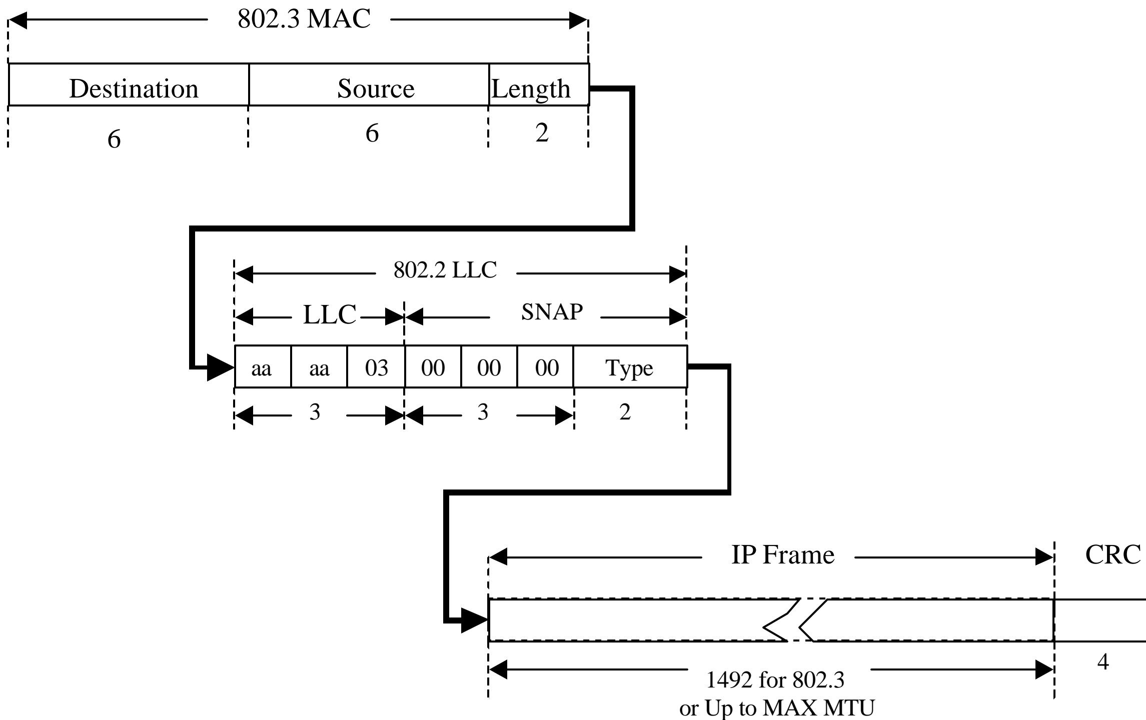


**Figure 9 Data Link LLC Encapsulation**

**Physical layer**

The management of the physical layer is really part of the lower half of the link layer networking interface driver. The frame is now ready for transmission and it typically is placed in a circular buffer ready for transmission by the DMA hardware

## The Data's Journey up the stack

In this section the path of data will be traced as it is received by the PHY and passed up the stack through the protocols to the socket layer.

**Received by the Link Layer.**

The data is received at the link layer via DMA into a circular buffer by the interface hardware. A driver in a zero copy implementation will place the circular buffer in memory in such a way that packets can be massaged into clusters which can later have mbuf wrappers placed around them without unnecessary copying by the CPU.

The link layer determines the type of framing. If the incoming packet is coming from an Ethernet interface, it may have either Ether type II framing or 802.3 type framing. The length field in the 802.3 type framing is at the same displacement as the type field in the Ether type II frame. The allowable types are in the table 5. All the protocol numbers are maintained by the Internet Assigned Numbers Authority (IANA), [IAPROT03]

| 0x0800 | IPv4 |
|---|---|

| 0x0806 | ARP |
| --- | --- |
| 0x8035 | RARP |
| 0x86DD | IPv6 |

**Table 5 Packet Types for IP**

If any of these types were to be interpreted as a length, they would represent a packet length greater than the .MTU of Ethernet of 1500 bytes. Therefore the two types of framing can be distinguished.

The interface's input function compares the link layer or MAC address of the incoming packet with the broadcast address. If the incoming packet is a broadcast packet, it is examined for a match on each protocol that has is attached to this interface. If the multicast bits are set for this interface, the packet is passed to any protocol that has requested multicast packets.

Then, if the incoming packet is a unicast packet directed to the local system or if it is a broadcast packet, the type field is examined. If the system has a protocol of a matching type, that protocol receives the input packet. If there is a protocol that has requested to receive data promiscuously, it would receive all the frames that are received at the link layer.

At this point, if the incoming frame is typed as an ARP packet, it is queued to the ARP protocol's input function. If it typed for IPv4, it is queued to IPv4's input function. Similarly for IPv6, it is queued to IPv6's input function.

**Received by IP.**

Most of the complexity in the IP layer has to do with routing or IP forwarding. For purposes of this section on IP, I assume that your embedded application is not expected to do routing. Layer 3 routing is a much larger topic and should be treated in a separate venue. (Below I discuss criteria for selecting a stack vendor to insure that the implementation can be extended for routing.) The IP layer compares the IP destination address of the incoming packet to its own IP address and subnet mask to see if the packet is destined for itself. If IP forwarding is not turned on and the packet is not destined for itself, it is discarded.

IP must now determine whether the packet is fragmented and requires re-assembly, or whether it is a complete IP datagram. The packet is reassembled, if required and once it is determined to be a complete IP datagram, it is time for transport de-multiplexing. IP has to be able to tell which transport is to receive the packet. It could be a UDP datagram or it could be destined for TCP. This is determined by IP when it checks the protocol ID in the IP header The data is then sent to the appropriate transport by way of the protocol switch table.

**Received by UDP**

If the data is sent to UDP by a lower layer protocol, it is received by the UDP_input() function. First the IP header is stripped and saved for later use. The UDP length is verified and the checksum is checked which requires some of the fields from the previously saved IP header.

The incoming packet is then checked to see if it was sent to a multicast or a broadcast address. If it was sent to a broadcast or multicast address, it is forwarded to all the matching sockets' input queues. If the data was sent to the correct unicast address but the port number is wrong, an ICMP packet is sent out indicating the port is unreachable. Otherwise if everything is OK, it is placed on the socket's input queue.

**Received by TDP**

TCP input is very complicated and I won't be able to do it complete justice in this article. The first thing it does is to verify the TCP checksum. Then, the timestamp in the packet is examined to verify the state of the connection. TCP must now decide if more data needs to be sent to the other end or if a timeout has occurred. If the incoming packet is data and not an acknowledgement, the IP and TCP headers are dropped and the data is placed into the listening socket's input buffer.

# Implementation in Embedded Systems

Embedded systems have inherited the programming practices used in larger systems. As in the case of other aspects of embedded systems, network protocols and TCP/IP in particular incorporate the lore of programming practices used in larger systems. This means that most TCP/IP stacks used in embedded systems started out as ports of TCP/IP implementations used in Unix systems. The Berkeley stack is the basis for most of these ports and is the basis of most of the TCP/IP stacks available commercially for embedded systems. Of course, there are many issues particular to real-time and embedded systems. A straight port of the Berkeley stack is not the best implementation for the particular needs of an embedded and real-time system.

Most vendors have modified the Berkeley code over the years to improve the performance of the stack in embedded systems. The engineers doing the port should address the following issues when modifying a Berkeley stack for embedded systems. If you are purchasing a TCP/IP stack you should verify that your vendor has taken these things into account.

- **Buffer Management**

  The TCP/IP mbuf buffer management should avoid fragmentation, allow fast prepending of headers without copying and be able to be formed into queues and lists without copying.

- **Timers**

  The timers used in the protocols for connection management, timeouts, and retries should be managed by the RTOS, They should not be a separate implementation that will secretly steal bandwidth from the CPU or cause concurrency problems.

- **Latency**

  If there is an RTOS, it should not add any additional latency. Interrupt handling interfaces should be fast and deterministic. The RTOS should not add any latency to the interrupt processing required with the physical transmission and reception of a frame. The large amount of context switches and CPU processing required in dealing with a packet increases the importance of using an OS with minimum thread context switches.

- **Concurrency**

  All buffering mechanisms should have semaphore protection to allow higher performance potential in real-time systems. Legacy TCP/IP protocol implementation was on Unix systems and depended on manipulating hardware interrupt levels to eliminate resource contention problems. Semaphore protection should be available to the timers to reduce concurrency problems.

- **Minimized data copying**

  The TCP/IP implementation should minimize the amount of data copying. The frame data generally does not need to be copied by the CPU. The networking chip's DMA places the packets directly in the managed buffer pool where the packet is passed up through the stack by manipulating pointers and not by copying data. Also, some vendors have extended the mbuf mechanism to allow the data to be shared between mbufs and mblocks where there are STREAMS protocols also present in the system.

- **Link layer multiplexing**

  Protocol implementation requires a framework with mechanisms for queuing and buffer management. Also, modern protocols require more flexible device driver interfaces and more flexible multiplexing. This is particularly true where serial point-to-point protocols such as PPP, are now extended to support IP tunneling and Virtual Private Networks (VPN). The original Berkeley implementation is not sufficiently flexible to meet all of these needs. The better protocol stack implementations use a framework that allows the stack to be extended as new protocols and interfaces are developed. This can be accomplished by extended the basic Berkeley driver interface scheme or the protocols can be rewritten to use a different framework.

- **CPU Bandwidth**

  An embedded system may have widely varying requirements for its TCP/IP stack. For example, a TCP/IP stack in most Internet appliances probably would not be considered real-time. Also, if the network is used for control and management functions, the hard bandwidth requirements will be fairly low. If the device is receiving streaming video or is used for IP telephony, then a faster rate of frame processing would move the stack into the realm of real-time application.

STREAMS is a generic framework and API for implementation of layered networking protocols developed at AT&T Bell Labs. There are several commercially available implementations of TCP/IP using STREAMS and there are also several separate implementations of STREAMS framework available for embedded markets. STREAMS is very well suited for systems that have to multiplex packets to and from multiple protocol stacks. Since the whole world is standardizing on the Internet, there is less interest in supporting protocols other than TCP/IP. Therefore, most embedded TCP/IP implementations are based on Berkeley. I this article, I concentrate on the Berkeley based TCP/IP implementation, but in the last section, I have included some references about STREAMS.

## Link Layer Interfaces and Device Drivers

As described above, the OSI model shows an interface between the physical and data link layers. In actual implementation, this interface is implemented in several ways.

### 1. BSD 4.3

Most of the examples in this paper show the BSD 4.3 type of structures and interfaces. The traditional Berkeley stack could multiplex between multiple interfaces if they were using a common IP stack. Originally, it could only interface cleanly with link layers that are compatible with Ethernet and only among IP and its related protocols such as ARP and RARP. Subsequently some implementers have hacked the BSD code to allow it to be used with serial interfaces such as SL/IP (Serial Line Interface Protocol) or PPP. This generally was done is a somewhat clugy way to make them look like Ethernet and as such was not particularly efficient. Also, each vendor of COTS TCP/IP for embedded systems have put their own nuances in the interface mechanism as well.

The BSD 4.3 compatible stacks uses the ifattach() function and the ifnet{} structure. This structure and its associated attachment mechanism were inherited by most ports of the Berkeley stack used in embedded systems. Typically, the network device driver initialization function is specific to a particular OS. When this function is called, it first allocates space for its own internal data structures usually called a "softc" structure. This data structure generally includes space for the ifnet{} structure. It determines its own MAC address by reading it from the hardware. Then it fills in the fields in the ifnet{} structure. It sets the device specific information such as the MTU, the MAC address, and the if_name and if_unit. It then fills in the function pointer fields with pointers to the driver's interface functions. Once this structure is appropriately initialized, the initialization code calls the if_attach() function with a pointer to the ifnet{} structure as an argument.

The ifnet structure is illustrated in table 6. The ifnet structure contains, among other things, a pointer to a list of address structures for each interface. This address structure called ifaddr{} contains the interface's MAC address and the broadcast address.

| char | *If_name | Two character interface name. | Defines interface. |
|---|---|---|---|
| struct ifnet | *if_next | Pointer to next ifnet | All of these are in a linked list. |
| struct ifaddr | *if_addrlist | Pointer to address for interface. | These are also in a linked list |
| int | if_pcount | Number of promiscuous listeners. | Used for packet filtering. |
| caddr_t | if_bpf | | Used for packet filtering. |
| u_short | if_index | | Defines interface. |
| short | if_unit | | Defines particular interface instance |
| short | if_timer | | Time until if_watchdog is called. |
| short | if_flags | | Flags keep track of state of interface such as up or down. |
| struct if_data | if_data | Volatile Statistics. | Used to keep statistics about interface. |
| Int | (*if_init) | Initialization function for this interface. | |
| Int | (*if_output) | Output function. | This function is to queue packets for sending. |
| int | (*if_start) | Start function. | This function is to initiate sending of packets. |

| int | (*if_ioctl) | IO Command function. | This function implements IO commands specific to this interface. |
|---|---|---|---|
| int | (*if_reset) | Device reset function. | |
| void | (*if_watchdog) | Timer function. | Checks to see that interface has not timed out. |

**Table 6 ifnet structure used in BSD 4.3**

## 2. BSD 4.4

BSD 4.4 extends the traditional interface in a few significant ways. Inside the stack, BSD 4.3 had hard coded address lengths in its sockaddr{} structure. In 4.4 BSD, this was extended to allow variable length addresses to allow for protocols other than TCP/IP that have longer network address formats. Also the device driver interface was generalized to make it less specific to Ethernet. Also, a pointer to the interface, netif{} is passed along by the link layer when incoming data is queued to IP or another other protocol in the network layer.

## 3. Data Link Provider Interface

The Data Link Provider Interface (DLPI) is found in most implementations of STREAMS. DLPI interface is not specific to TCP/IP and can be generalized for almost any protocol. It does, however, require that the protocols be implemented as STREAMS modules and the link layer be implemented as a STREAMS driver. It also requires 802.2 type framing to properly multiplex between the link layer and the network layer of the protocols bound to the link layer.

The legacy BSD interface between the link layer and the network layer doesn't incorporate any mechanism for interfacing specifically to connection-oriented or connectionless link layers. DLPI deals with that requirement by providing different types of binding depending on whether the Service Access Provider (SAP) is capable of providing connectionless or connection oriented transmissions. Also, DLPI allows the Data Link SAP (DLSAP) to identify itself as a promiscuous SAP, that is one that can grab all the packets on the net, not just those directed for itself.

Fundamental to DLPI is the concept of the SAP. In figure 9 you can see how the SAP identifiers are incorporated in the LLC framing. The network layer above the interface is a Data Link Service User (DLSU) and the Link Layer driver is a DLSAP (Data Link Service Access Provider) or just SAP for short. DLPI uses a set of request primitives passed as messages from the DLSU to the DLSAP. In response, the DLSAP passes a set of acknowledgment primitives back to the DLSU.

Figure 10 shows the relationship between the Service Access Provider and the DLS User. DLPI manages the state of the relationship between the DLS user and the DLSAP with a state machine. Table 7 lists some of the common primitives likely to be used for TCP/IP connectionless link layer service and their responses.

| Message | Response | Purpose |
|---|---|---|
| DL_ATTACH_REQ | DL_BIND_ACK | Used to tell driver to initialize hardware and associate physical layer with a STREAM. |

| | | Required for Style 2 providers to initialize themselves. |
|---|---|---|
| DL_INFO_REQ | DL_INFO_REQ | Asks the provider to send information about the interface including MAC type, service mode, QOS (Quality of Service), |
| DL_BIND_REQ | DL_BIND_ACK | Requests DLP to bind a SAP to a STREAM. Used to indicate the type of service, connection oriented or connectionless SAP. |
| DL_ENABLMULTI_REQ | DL_OK_ACK | Requests the provider to enable a multicast address fir the specific DLSAP. |
| DL_PROMISCON_REQ | DL_OK_ACK | Asks provider to enable promiscuous mode for this SAP. |
| DL_PROMISCOFF_REQ | DL_OK_ACK | Asks provider to disable promiscuous mode for this SAP. |
| DL_UNITDATA_REQ | DL_UNITDATA_IND | Requests connectionless data grams be passed to the DLS user from the SAP. The datagrams are passed in a message block preceded by a DL_UNITDATA_IND primitive. |

**Table 7 DLPI primitives**

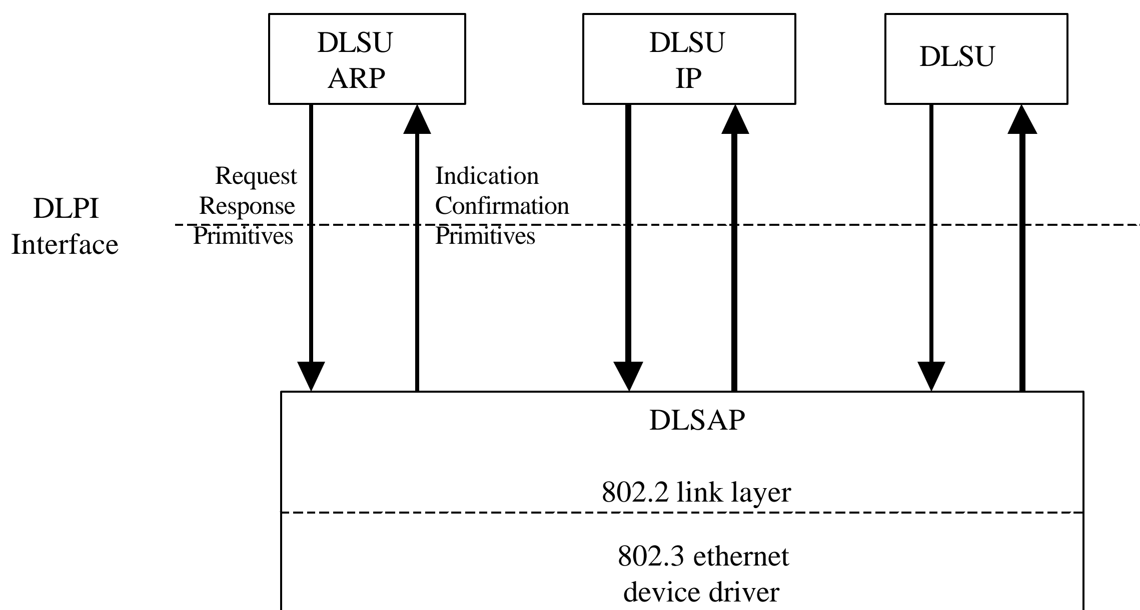

**Figure 10 Data Link Provider Interface**

## 4. Extended Multiplexing Interface

An extended multiplexing interface, found in many RTOSs, extends the BSD 4.4 interface to include the multiplexing and multi-protocol capability which was previously only found in DLPI. A good implementation will do this while still maintaining compatibility with the traditional Berkeley type TCP/IP stacks and drivers.

Traditionally, there have been various attempts to specify drivers to support simultaneous interfaces to both Berkeley and STREAMS type stacks. This was done with a layer of glue code that would copy the messages from STREAMS mblocks to Berkeley stype mbufs.

An good extended multiplexing interface mechanism gives you a DLPI type binding mechanism to bind a stack to a layer interface but still allows compatibility with Berkeley style code. As in DLPI, this advanced mechanism should allow multiple stacks to be bound to the same link level interface. It provides a generic mechanism for address resolution between MAC addresses and protocol addresses. This address resolution capability should still support the BSD ARP protocol discussed under BSD 4.3 above. As with DLPI, the advanced interface has a state machine to keep track of the relationship between the stack and the link layer. It also provides a means of enabling multicasting address on an interface. This sort of mechanism was absent from BSD 4.3 and only partially present in BSD 4.4.

## Promiscuity – Sometimes a good thing.

Most network interfaces can be set to promiscuous mode so that all packets are received. For example, Ethernet interfaces can be set to receive all packets on the cable, including those that are not broadcast or multicast packets and those that have somebody's MAC address. It is interesting to note that most protocols would work even if the interfaces were accidentally set for promiscuous mode. Packet filtering then takes place at higher levels of the stack which is very inefficient but probably functional. Some software above the link layer may want to use promiscuous mode by telling the interface to receive everything and send it all up. An example of this is a software protocol analyzer such as the tcpdump or snoop utility found in Unix systems.

At the transport layer, the API provides a promiscuous socket. Generally, you can't stop processing and sending other information merely because the user wants to observe the network traffic. In order to provide the capability of receiving promiscuously while preserving other network functionality, the data link interface must support some sort of promiscuous binding. It has to provide a mechanism for the layer above to request to see all packets.

## Selecting and hosting the TCP/IP protocols in your embedded system

Since, the BSD stack has been available in source form for many years, most people implementing TCP/IP for embedded systems have used it as a base. Hardly anyone implements his/her own protocol suite from scratch. There are a number of fundamental choices you will have if you want to put TCP/IP in their product for the first time whether or not you want to use a COTS implementation of TCP/IP.

Of course we can't consider all the choices for embedded engineers with considering Linux which is even more ubiquitous than BSD and comes with the TCP/IP stack preconfigured in the kernel. Another advantage of Linux, is that as of 2.6, it supports a stable implementation of Ipv6. However, you may want to apply the latest USAGI kernel patches. As of this writing, the latest IPv6 was not rolled into the core kernel release.

You may want to look at a number of factors before you decide which path to take for incorporating TCP/IP. You may want to ask yourself a few basic questions about the connectivity requirements in your design. For example, you may want to ask how sensitive your project is to

UMC (Unit Manufacturing Cost)? Also ask about the future and reuse potential of your design. Is the project a platform from which you may expect leverage other products? Below are a few broad categories of products used in connected embedded systems. Each of them implies a different direction in choosing TCP/IP in your embedded project.

1. The embedded product is based on a legacy system where there is only a limited need for remote access to the product's serial port. UMC is not the strongest determining factor.
2. A net connection is needed for remote management of the embedded product. Generally, this presumes the use of SNMP (Simple Network Management Protocol) and the ability to support the standard MIBs (Management Information Base) and perhaps the ability to create and link in new MIBs.
3. The networking connection is required for data capture and analysis.
4. The product is a router, gateway, switch, broadband modem or a similar product where networking is a fundamental part.
5. The product is a consumer device, such as a Personal Digital Assistant (PDA) that must have embedded web browsing capability.
6. The product primarily must support an embedded web server.

Below I list some of the choices you have today for incorporating TCP/IP in your product.

- **Total hardware implementation.**

There are a number of companies that are developing completely self-contained chips that have implemented a TCP/IP stack in firmware. The chip can be designed into your board and generally is designed to be interfaced easily. Usually it has a serial interface so the system is made to think it is talking to out a serial port.

**Advantages:**

- Vendor owns all networking and inter-operability problems.
- It is easy to specify a working solution.
- There is very little software and firmware redesign required for legacy products.
- You can concentrate on your product's added value.

**Disadvantages:**

- It is not a flexible solution.
- This system can not be easily configured to give remote users a web based interface.
- The fact that networking is not integrated into your product can make it seem clunky.

- **Rolling In Your Own with no RTOS**

This involves snarfing a free, source licensable, or public domain TCP/IP generally derived from BSD and porting it straight to the hardware without the benefit of an RTOS. Even if you don't have an RTOS, you will need to implement a basic scheduler, a timer mechanism, and

a buffer allocation mechanism. This solution may be appropriate for the first two categories above if the requirement domain is fairly well defined and there is not likely to be a need for reuse.

**Advantages:**

- This solution has low Unit Manufacturing Cost (UMC) because of reduced royalties.
- You own all the sources.
- There are no royalties.
- It could be fun to do if you have a lot of time.

**Disadvantages:**

- You own all the problems.
- There is no RTOS to help allocate CPU bandwidth between the networking stack and other parts of your application.
- It is inflexible and doesn't allow for future product growth.
- It can't be easily brought up to date with new networking standards.
- Compliance and inter-operability testing is an enormous burden.
- It is hard to upgrade because of incompatibility with other implementations.
- It has a higher engineering cost do to the necessity of creating basic infrastructure and porting the TCP/IP stack.
- It is bad for time-to-market constrained projects.

## Third Party Integration

With this method, you would purchase a commercial RTOS from one vendor and a commercial or open source TCP/IP stack from somewhere else.

**Advantages:**

- You can use a commercial RTOS which are generally fairly low cost and have a small memory footprint.

**Disadvantages:**

- It is up to the you to select and integrate your own networking stack.
- You may have to design implement your own transport and link layer interfaces.
- The smallest RTOS often don't' support standard interfaces such as sockets which make it hard to port application code.
- This is not a time-to-market friendly solution.
- You own all the interface problems.
- There may be little or no support.

- **All in One**

  With this option, you would purchase a bundled product from a vendor that that includes an RTOS, development tools and a TCP/IP stack. This solution can be used for all the product categories above. Of course with category 1, you would have to occur some redesign to incorporate a new RTOS into your new design.

  **Advantages:**

  - This solution works well for design cycles with critical time to market constraints.
  - It is scalable to meet evolving needs.
  - Should give you access to added value upper layer protocols, such as embedded web servers and clients, network management and other applications.

  **Disadvantages:**

  - May add some royalty cost to your product.
  - Some redesign of legacy products may be required to add the networking feature.

- **Complete Open Source Solution**

  With this final option, you could use one of the standard Linux distributions for your embedded system. Linux comes with the TCP/IP stack in the kernel. It is trusted, well known and widely used. It includes all the latest security updates and performance enhancements. It also includes a stable IPv6 solution, however you may need to apply the latest USAGI patches as of the time of this writing. This solution will work with any of the categories above.

  It is important to note that the Linux implementation of TCP/IP can't easily be separated from the kernel. It is very tightly integrated.

  **Advantages:**

  - Because of the popularity of Linux as an embedded OS, this solution also works well for design cycles with critical time to market constraints.
  - The TCP/IP stack comes built in. It already supports all the standards. It can do routing, support web browsing or a web server. It supports management too.
  - Supports standard interfaces such as sockets.

  **Disadvantages:**

  - May be some additional engineering time, if there isn't a cut and paste port of Linux to your hardware.
  - Linux has a larger memory footprint than the BSD derived stacks.
  - You will need to run on one of the 32 bit CPUs which supports hardware memory management.

### Selection Criteria for commercial or other RTOSs and TCP/IP stacks.

The following is a list of items, you may want to consider when selecting an RTOS and networking stack vendor.

- Choose an RTOS that has a variety of interfaces and available device drivers. It should be able to smoothly support interfaces other than common LAN PHYs such as Ethernet.
- The OS should have stable and tested support for your target CPU and networking interfaces.
- If you are building a platform to be used in a variety of products, make sure that the OS can support your needs into the future.
- The TCP/IP stack should be BSD 4.4 compatible. It should support standard sockets and the various performance and security enhancements found in recent implementations
- The stack should have the hooks to allow it to be managed via remote management. It should support SNMP and the standard MIBs.
- The stack should be implemented to minimize copying of data when moving buffers in the system. This is accomplished in conjunction with some kind of enhanced buffer management.
- The OS should support multiplexing at the data link layer and above the socket layer to allow the TCP/IP stack to co-exist with other protocols and allow protocols such as IP tunneling, IPSec, and complex WAN interfaces to be used.

## Conclusion

TCP/IP is close to 20 years old. It was developed as part of one of the oldest attempts in establishing a computer inter-network, now known as the Internet. The protocol has out lasted many more recent and modern protocols and by now many of these more recent networking protocols have been forgotten. TCP/IP and the concept of the Internet is elegant in its simplicity. Largely because of its ubiquity and its simplicity, this old TCP/IP based Internet has become the global fabric uniting much of modern life and is now the basis for much of the hot new "High Technology" of today.

Obviously, I shouldn't try to dictate the choices you make as an embedded engineer when you move your legacy design to the Internet enabled world or as you develop your brand new internet appliance. As an embedded engineer, you will follow your own dictates given the business and technical constraints particular to your project. As discussed above, there are many choices for selecting and integrating TCP/IP in your design. Many new ASICS today have extra gates to burn and extra bandwidth in the CPU core as well as more available memory. After making the right choice for your design, you may find that in today's world most designs have enough headroom to incorporate Internet ability. Even on small embedded systems, adding TCP/IP won't be a large draw on your system for CPU and memory.

## Further Reading

There are many excellent books written on TCP/IP internals, application programming using TCP/IP, and maintenance and configuration of IP networks. I would like to suggest a few

references below which I have found valuable for explaining the internals of TCP/IP that may be useful to the embedded engineers that want to bone up on networking.

The first and most important references for TCP/IP are the following two volumes. If you really need both to see how the protocols work and how they are implemented these books are must haves.

W. Richard Stevens, *TCP/IP Illustrated, vol. 1*,Addison-Wesley, ISBN 0-201-63346-9

Gary R. Wright and W. Richard Stevens, *TCP/IP Illustrated, vol. 2*,Addison-Wesley, ISBN 0-201-63354-X

For an authoritative work on BSD 4.4 I suggest the following book:

Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quaterman; The Design and Implementation of the 4.4BSD Operating System, Addison-Wesley, ISBN 0-201-54974-4

For general information on STREAMS, you may read my article on STREAMS:

Thomas F. Herbert, *Implementing Network Protocols and Drivers with STREAMS*, Embedded Systems Programming, VOL. 10 NO. 4 April 1997, http://www.embedded.com/97/feat9704.htm

Or you may want to read the definitive reference on SVR4 STREAMS. Even if you are not specifically interested in STREAMS, chapter 11 on DLPI (Data Link Provider Interface) is a good look at what functionality needs to be in a good multiplexing data link interface. Unfortunately, at the time of the writing of this article, this book is out of print.

Unix Press, *STREAMS Modules and Drivers*, Prentice-Hall, ISBN 0-13-066879-6

Internet Assigned Numbers Authority, *Protocol Numbers*, http://www.iana.org/assignments/protocol-numbers

## Bio

Tom is an independent consulting specializing in embedded networking. Earlier, he was at Wind River where he worked for Wind River Services as a network specialist. Before Wind River, he was a Systems Scientist at Divx, where he worked on embedded systems in DVD players. Earlier, he has held various software engineering positions at Xerox Corporation and Eastman Kodak Company. He holds a patent in pattern recognition in embedded applications. He received his BS in computer science from Binghamton University in New York State.