1. Introduction

This manual is intended for embedded systems engineers and support professionals who are not familiar with basic networking concepts. An overview of an Ethernet network and the TCP/IP suite of protocols used to communicate across the network will be given.

The implementation details that are discussed in this manual pertain to versions of Dynamic C prior to 7.05. Improvements and additions to the TCP/IP suite of protocols are fully documented in the *Dynamic C TCP/IP User's Manual*.

2. Ethernet Basics

TCP/IP (Transmission Control Protocol/Internet Protocol) is a set of protocols independent of the physical medium used to transmit data, but most data transmission for Internet communication begins and ends with Ethernet frames.

The Ethernet can use either a bus or star topology. A bus topology attaches all devices in sequence on a single cable. In a star topology all devices are wired directly to a central hub. 10Base-T uses a combination called a star-shaped bus topology because while the attached devices can share all data coming in on the cable, the actual wiring is in a star shape.

The access method used by the Ethernet is called Carrier Sense Multiple Access with Collision Detect (CSMA/CD). This is a contention protocol, meaning it is a set of rules to follow when there is competition for shared resources.

2.1 Ethernet Address

All Ethernet interfaces have a unique 48-bit address that is supplied by the manufacturer. It is called the Ethernet address (also known as the MAC address, for Media Access Control). Ethernet-enabled Z-World boards store this value in Flash Memory (EEPROM) that is programmed at the factory. If you need unique Ethernet addresses for some product you are making, you can obtain them from the IEEE Registration Authority.

To read the MAC address of a TCP/IP Development Board, run the utility program **display_MAC.c**. It is located on the Technical Support Sample Program Web page: http://www.rabbitsemiconductor.com/support_center/rab20_support.html. It is also included with Dynamic C, version 7.04 and above.

2.2 Physical Connections

A Realtek RTL8019 10Base-T interface chip provides a 10 Mbps Ethernet connection. This chip is used on many Ethernet-enabled Z-World boards. The corresponding port can be connected directly to an Ethernet network.

By using *hubs* and *routers*, a network can include a large number of computers. A network might include all the computers in a particular building. A local network can be connected to the Internet by means of a *gateway*. The gateway is a computer that is connected both to the local network and to the Internet. Data that must be sent out over the Internet are sent to the local network interface of the gateway, and then the gateway sends them on to the Internet for routing to some other computer in the world. Data coming in from the Internet are directed to the gateway, which then sends them to the correct recipient on the local network.

2.2.1 Cables

Ethernet cables are similar to U.S. telephone plug cables, except they have eight connectors. For our purposes, there are two types of cables—crossover and straight-through. In most instances, the straight-through cables are used. It is necessary to use a crossover cable when two computers are connected directly without a hub (for example, if you want to connect your PC's Ethernet directly to the Rabbit Semiconductor TCP/IP Development Board.) Some hubs have one input that can accept either a straight-through or crossover cable depending on the position of a switch. In this case make sure that the switch position and cable type agree.

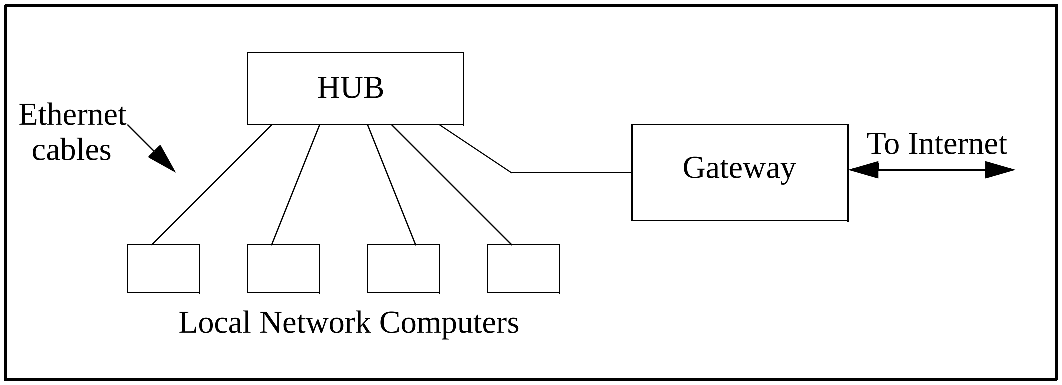

Figure 1. Ethernet Network

2.3 Frames

Bits flowing across the Ethernet are grouped into structures called frames. A frame must be between 46 and 1500 bytes in size. An Ethernet frame has four parts:

1. A *Preamble* of 8 bytes that helps synchronize the circuitry, thus allowing small bit rate differences between sender and receiver.
2. A *Header* of 14 bytes that contains a 6 byte destination address, 6 byte source address and a 2 byte type field.
3. A *Data* area of variable length that, along with the header, is passed to the IP layer (aka. the Network layer).
4. A *Trailer* of 4 bytes that contains a CRC to guard against corrupted frames.

If the destination address is all 1 bits, it defines a *broadcast* frame and all systems on the local network process the frame. There are also *multicast* frames. A subset of systems can form a “multicast” group that has an address that does not match any other system on the network. All systems in a particular subset process a packet with a destination address that matches their subset. A system can belong to any number of subsets.

A system may put its interface(s) into *promiscuous* mode and process all frames sent across its Ethernet. This is known as "sniffing the ether." It is used for network debugging and spying.

2.3.1 Collisions

In a star-shaped bus topology, all systems have access to the network at any time. Before sending data, a system must determine if the network is free or if it is already sending a frame. If a frame is already being sent, a system will wait. Two systems can “listen” on the network and “hear” silence and then proceed to send data at the same time. This is called a collision. Ethernet hardware has collision detection sensors to take care of this problem. This is the Collision Detect (CD) part of CSMA/CD. The colliding data is ignored, and the systems involved will wait a random amount of time before resending their data.

3. Networks

A network is a system of hardware and software, put together for the purpose of communication and resource sharing. A network includes transmission hardware, devices to interconnect transmission media and to control transmissions, and software to decode and format data, as well as to detect and correct problems.

There are several types of networks in use today. This chapter will focus on three of them:

- LAN - Local Area Network
- WAN - Wide Area Network
- VPN - Virtual Private Network

3.1 LAN

The most widely deployed type of network, LANs were designed as an alternative to the more expensive point-to-point connection. A LAN has high throughput for relatively low cost. LANs often rely on shared media, usually a cable, for connecting many computers. This reduces cost. The computers take turns using the cable to send data.

3.1.1 Repeaters and Bridges

LANs typically connect computers located in close physical proximity, i.e., all the computers in a building. *Repeaters* are used to join network segments when the distance spanned causes electrical signals to weaken. Repeaters are basically amplifiers that work at the bit level; they do not actively modify data that is amplified and sent to the next segment.

Like repeaters, *bridges* are used to connect two LANs together. Unlike repeaters, bridges work at the frame level. This is useful, allowing bridges to detect and discard corrupted frames. They can also perform frame filtering, only forwarding a frame when necessary. Both of these capabilities decrease network congestion.

Bridged LANs can span arbitrary distances when using a satellite channel for the bridge. The resulting network is still considered a LAN and not a WAN.

3.2 WAN

To be considered a WAN, a network must be able to connect an arbitrary number of sites across an arbitrary distance, with an arbitrary number of computers at each site. In addition, it must have reasonable performance (no long delays) and allow all of the computers connected to it to communicate simultaneously. This is accomplished with *packet switches*.

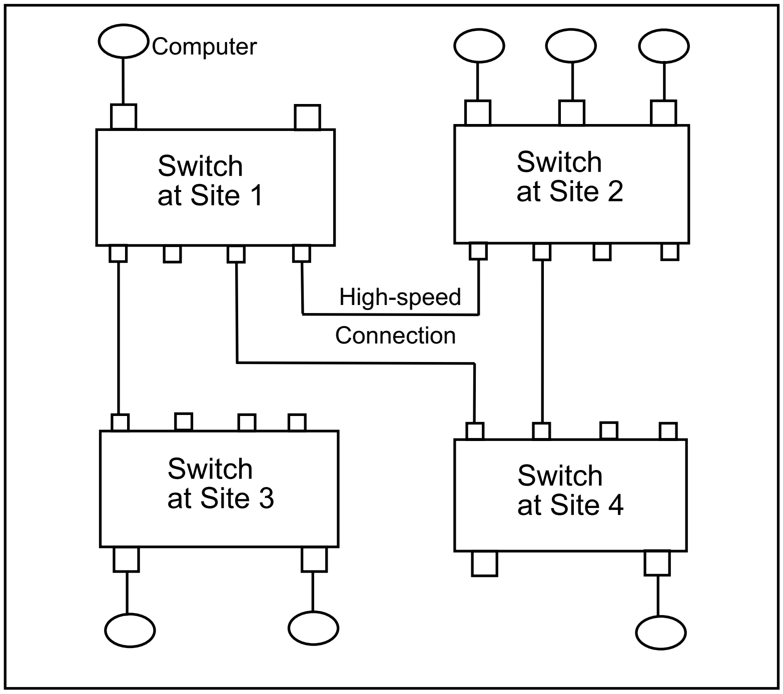

Figure 2. WAN with 4 switches.

3.2.1 Packet Switches

Packet switches are small computers with CPUs, memory and I/O devices. They move complete packets, using a technique called Store and Forward. An incoming packet is stored in a memory buffer and the CPU is interrupted. The processor examines the packet and forwards it to the appropriate place. This strategy allows the switch to accept multiple packets simultaneously.

As the figure above illustrates, WANs currently do not need to be symmetric.

3.2.2 Forwarding a Packet

A data structure contains the information that tells the switch where to send the packet next. This is called a routing table. The destination address in the packet header determines the routing table entry that is used to forward the packet. It could be forwarded to a computer attached to the switch that is examining the packet or it could be to another switch in the WAN.

3.3 VPN

VPNs are built on top of a publicly-accessible infrastructure, such as the Internet or the public telephone network. They use some form of encryption and have strong user authentication. Essentially a VPN is a form of WAN; the difference is their ability to use public networks rather than private leased lines. A VPN supports the same intranet services as a traditional WAN, but also supports remote access service. This is good for telecommuting, as leased lines don’t usually extend to private homes and travel destinations.

A remote VPN user can connect via an Internet Service Provider (ISP) in the usual way. This eliminates long-distance charges. The user can then initiate a tunnel request to the destination server. The server authenticates the user and creates the other end of the tunnel. VPN software encrypts the data, packages it in an IP packet (for compatibility with the Internet) and sends it through the tunnel, where it is decrypted at the other end.

There are several tunneling protocols available: IP security (IPsec), Point-to-Point Tunneling Protocol (PPTP) and Layer 2 Tunneling Protocol (L2TP).

3.4 Network Devices

Some network devices (repeaters, bridges and switches) were discussed in the previous sections. These are all dedicated hardware devices. Network devices can also be non-dedicated systems running network software.

3.4.1 Routers

A *router* is a hardware device that connects two or more networks. Routers are the primary backbone device of the Internet, connecting different network technologies into a seamless whole. Each router is assigned two or more IP addresses because each IP address contains a prefix that specifies a physical network.

Before a packet is passed to the routing software, it is examined. If it is corrupted, it is discarded. If it is not corrupted, a routing table is consulted to determine where to send it next. By default, routers do not propagate *broadcast* packets (see “Directed Broadcast Address” on page 17). A router can be configured to pass certain types of broadcasts.

3.4.2 Firewalls

A *firewall* is a computer, router, or some other communications device that controls data flow between networks. Generally, a firewall is a first-line defense against attacks from the outside world. A firewall can be hardware-based or software-based. A hardware-based firewall is a special router with additional filter and management capabilities. A software-based firewall runs on top of the operating system and turns a PC into a firewall.

Conceptually, firewalls can be categorized as Network layer (aka Data Link layer) or Application layer. Network layer firewalls tend to be very fast. They control traffic based on the source and destination addresses and port numbers, using this information to decide whether to pass the data on or discard it.

Application layer firewalls do not allow traffic to flow directly between networks. They are typically hosts running proxy servers. Proxy servers can implement protocol specific security because they understand the application protocol being used. For instance, an application layer firewall can be configured to allow only e-mail into and out of the local network it protects.

3.4.3 Gateways

A *gateway* performs routing functions. The term *default gateway* is used to identify the router that connects a LAN to an internet. A gateway can do more than a router; it also performs protocol conversions from one network to another.

3.5 Network Architecture

There are two network architectures widely used today: peer-to-peer and client/server. In peer-to-peer networks each workstation has the same capabilities and responsibilities. These networks are usually less expensive and simpler to design than client/server networks, but they do not offer the same performance with heavy traffic.

3.5.1 Client/Server Networks

The client/server paradigm requires some computers to be dedicated to serving other computers. A server application waits for a client application to initiate contact.

Table 1. Summary of Differences between Client and Server Software

Client Software	Server Software
An arbitrary application program that becomes a client when a remote service is desired. It also performs other local computations.	A special-purpose, privileged program dedicated to providing one service. It can handle multiple remote clients at the same time.
Actively initiates contact.	Passively waits for contact.
Invoked by a user and executes for one session.	Invoked when the system boots and executes through many sessions.
Capable of accessing multiple services as needed, but actively contacts only one remote server at a time.	Accepts contact from an arbitrary number of clients, but offers a single service or a fixed set of services.
Does not require special hardware or a sophisticated operating system.	Can require powerful hardware and a sophisticated operating system, depending on how many clients are being served.

3.5.1.1 Port Numbers

Port numbers are the mechanism for identifying particular client and server applications. Servers select a port to wait for a connection. Most services have well-known port numbers. For example, HTTP uses port 80. When a web browser (the client) requests a web page it specifies port 80 when contacting the server. Clients usually have ephemeral port numbers since they exist only as long as the session lasts.

Some of the common well-known TCP port numbers are listed in the table below.

Port Number	Listening Application
7	Echo request
20/21	File Transfer Protocol (FTP)
23	Telnet
25	Simple Mail Transfer Protocol (SMTP)
53	Domain Name Server
80	HTTP Server

4. Network Protocol Layers

Computers on a network communicate in agreed upon ways called protocols. The complexity of networking protocol software calls for the problem to be divided into smaller pieces. A layering model aids this division and provides the conceptual basis for understanding how software protocols together with hardware devices provide a powerful communication system.

4.1 Layering Models

In the early days of networking, before the rise of the ubiquitous Internet, the International Organization for Standardization (ISO) developed a layering model whose terminology persists today.

Table 2. ISO 7-Layer Reference Model

	Name of Layer	**Purpose of Layer**
Layer 7	Application	Specifies how a particular application uses a network.
Layer 6	Presentation	Specifies how to represent data.
Layer 5	Session	Specifies how to establish communication with a remote system.
Layer 4	Transport	Specifies how to reliably handle data transfer.
Layer 3	Network	Specifies addressing assignments and how packets are forwarded.
Layer 2	Data Link	Specifies the organization of data into frames and how to send frames over a network.
Layer 1	Physical	Specifies the basic network hardware.

The 7-layer model has been revised to the 5-layer TCP/IP reference model to meet the current needs of protocol designers.

Table 3. TCP/IP 5-Layer Reference Model

	Name of Layer	**Purpose of Layer**
Layer 5	Application	Specifies how a particular application uses a network.
Layer 4	Transport	Specifies how to ensure reliable transport of data.
Layer 3	Internet	Specifies packet format and routing.
Layer 2	Network	Specifies frame organization and transmittal.
Layer 1	Physical	Specifies the basic network hardware.

4.2 TCP/IP Protocol Stack

TCP/IP is the protocol suite upon which all Internet communication is based. Different vendors have developed other networking protocols, but even most network operating systems with their own protocols, such as Netware, support TCP/IP. It has become the de facto standard.

Protocols are sometimes referred to as protocol stacks or protocol suites. A protocol stack is an appropriate term because it indicates the layered approach used to design the networking software

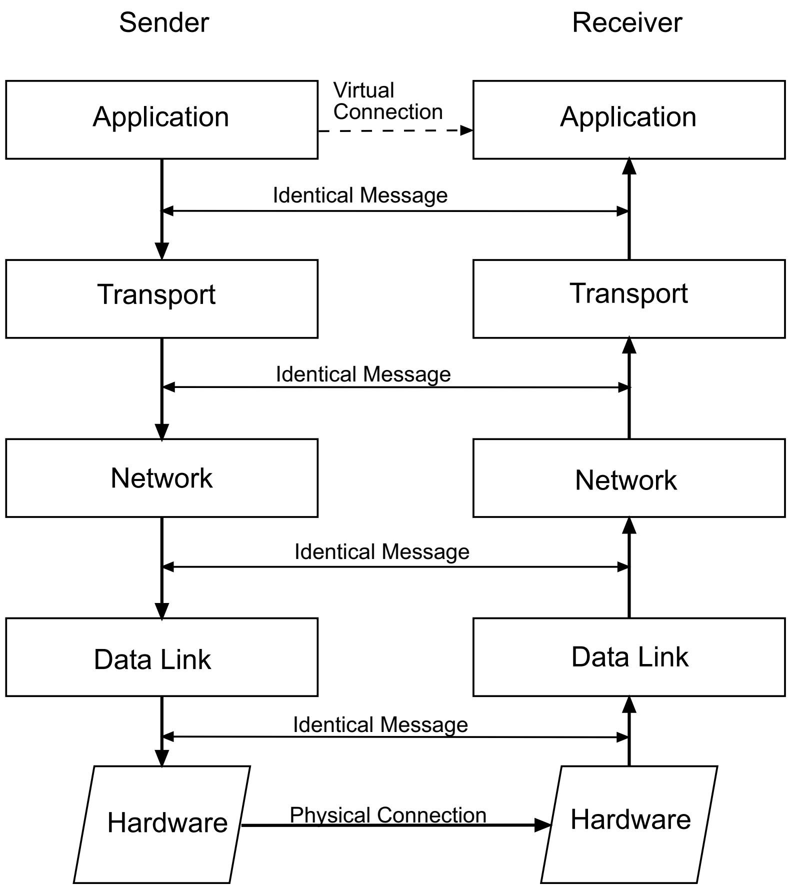

Figure 3. Flow of data between two computers using TCP/IP stacks.

Each host or router in the internet must run a protocol stack. The details of the underlying physical connections are hidden by the software. The sending software at each layer communicates with the corresponding layer at the receiving side through information stored in headers. Each layer adds its header to the front of the message from the next higher layer. The header is removed by the corresponding layer on the receiving side.

5. TCP/IP Protocols

This chapter discusses the protocols available in the TCP/IP protocol suite. The following figure shows how they correspond to the 5-layer TCP/IP Reference Model. This is not a perfect one-to-one correspondence; for instance, Internet Protocol (IP) uses the Address Resolution Protocol (ARP), but is shown here at the same layer in the stack.

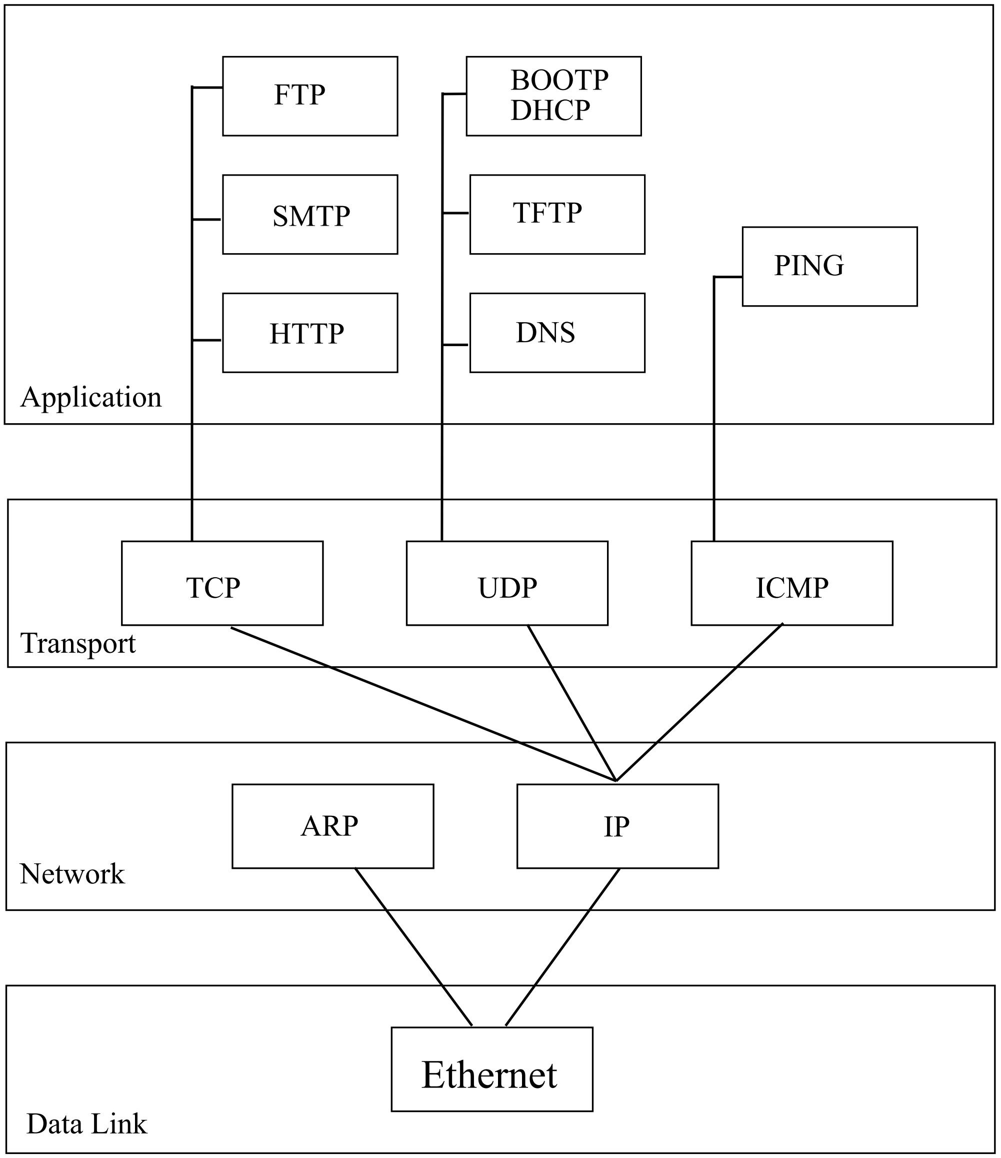

Figure 4. TCP/IP Protocol Flow

5.1 IP

IP provides communication between hosts on different kinds of networks (i.e., different data-link implementations such as Ethenet and Token Ring). It is a connectionless, unreliable packet delivery service. Connectionless means that there is no handshaking, each packet is independent of any other packet. It is unreliable because there is no guarantee that a packet gets delivered; higher-level protocols must deal with that.

5.1.1 IP Address

IP defines an addressing scheme that is independent of the underlying physical address (e.g, 48-bit MAC address). IP specifies a unique 32-bit number for each host on a network. This number is known as the Internet Protocol Address, the IP Address or the Internet Address. These terms are interchangeable. Each packet sent across the internet contains the IP address of the source of the packet and the IP address of its destination.

For routing efficiency, the IP address is considered in two parts: the prefix which identifies the physical network, and the suffix which identifies a computer on the network. A unique prefix is needed for each network in an internet. For the global Internet, network numbers are obtained from Internet Service Providers (ISPs). ISPs coordinate with a central organization called the Internet Assigned Number Authority (IANA).

5.1.2 IP Address Classes

The first four bits of an IP address determine the class of the network. The class specifies how many of the remaining bits belong to the prefix (aka Network ID) and to the suffix (aka Host ID). The first three classes, A, B and C, are the primary network classes.

Class	First 4 Bits	Number Of Prefix Bits	Max # Of Networks	Number Of Suffix Bits	Max # Of Hosts Per Network
A	0xxx	7	128	24	16,777,216
B	10xx	14	16,384	16	65,536
C	110x	21	2,097,152	8	256
D	1110	Multicast			
E	1111	Reserved for future use.			

When interacting with mere humans, software uses dotted decimal notation; each 8 bits is treated as an unsigned binary integer separated by periods. IP reserves host address 0 to denote a network. 140.211.0.0 denotes the network that was assigned the class B prefix 140.211.

5.1.3 Netmasks

Netmasks are used to identify which part of the address is the Network ID and which part is the Host ID. This is done by a logical bitwise-AND of the IP address and the netmask. For class A networks the netmask is always 255.0.0.0; for class B networks it is 255.255.0.0 and for class C networks the netmask is 255.255.255.0.

5.1.4 Subnet Address

All hosts are required to support subnet addressing. While the IP address classes are the convention, IP addresses are typically subnetted to smaller address sets that do not match the class system. The suffix bits are divided into a subnet ID and a host ID. This makes sense for class A and B networks, since no one attaches as many hosts to these networks as is allowed. Whether to subnet and how many bits to use for the subnet ID is determined by the local network administrator of each network.

If subnetting is used, then the netmask will have to reflect this fact. On a class B network with subnetting, the netmask would not be 255.255.0.0. The bits of the Host ID that were used for the subnet would need to be set in the netmask.

5.1.5 Directed Broadcast Address

IP defines a directed broadcast address for each physical network as all ones in the host ID part of the address. The network ID and the subnet ID must be valid network and subnet values. When a packet is sent to a network's broadcast address, a single copy travels to the network, and then the packet is sent to every host on that network or subnetwork.

5.1.6 Limited Broadcast Address

If the IP address is all ones (255.255.255.255), this is a limited broadcast address; the packet is addressed to all hosts on the current (sub)network. A router will not forward this type of broadcast to other (sub)networks.

5.2 IP Routing

Each IP datagram travels from its source to its destination by means of routers. All hosts and routers on an internet contain IP protocol software and use a routing table to determine where to send a packet next. The destination IP address in the IP header contains the ultimate destination of the IP datagram, but it might go through several other IP addresses (routers) before reaching that destination.

Routing table entries are created when TCP/IP initializes. The entries can be updated manually by a network administrator or automatically by employing a routing protocol such as Routing Information Protocol (RIP). Routing table entries provide needed information to each local host regarding how to communicate with remote networks and hosts.

When IP receives a packet from a higher-level protocol, like TCP or UDP, the routing table is searched for the route that is the closest match to the destination IP address. The most specific to the least specific route is in the following order:

- A route that matches the destination IP address (host route).
- A route that matches the network ID of the destination IP address (network route).
- The default route.

If a matching route is not found, IP discards the datagram.

IP provides several other services:

- **Fragmentation**: IP packets may be divided into smaller packets. This permits a large packet to travel across a network which only accepts smaller packets. IP fragments and reassembles packets transparent to the higher layers.
- **Timeouts**: Each IP packet has a Time To Live (TTL) field, that is decremented every time a packet moves through a router. If TTL reaches zero, the packet is discarded.
- **Options**: IP allows a packet's sender to set requirements on the path the packet takes through the network (source route); the route taken by a packet may be traced (record route) and packets may be labeled with security features.

5.3 ARP

The Address Resolution Protocol is used to translate virtual addresses to physical ones. The network hardware does not understand the software-maintained IP addresses. IP uses ARP to translate the 32-bit IP address to a physical address that matches the addressing scheme of the underlying hardware (for Ethernet, the 48-bit MAC address).

There are three general addressing strategies:

1. Table lookup
2. Translation performed by a mathematical function
3. Message exchange

TCP/IP can use any of the three. ARP employs the third strategy, message exchange. ARP defines a request and a response. A request message is placed in a hardware frame (e.g., an Ethernet frame), and broadcast to all computers on the network. Only the computer whose IP address matches the request sends a response.

5.4 The Transport Layer

There are two primary transport layer protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). They provide end-to-end communication services for applications.

5.4.1 UDP

This is a minimal service over IP, adding only optional checksumming of data and multiplexing by port number. UDP is often used by applications that need multicast or broadcast delivery, services not offered by TCP. Like IP, UDP is connectionless and works with datagrams.

5.4.2 TCP

TCP is a connection-oriented transport service; it provides end-to-end reliability, resequencing, and flow control. TCP enables two hosts to establish a connection and exchange streams of data, which are treated in bytes. The delivery of data in the proper order is guaranteed.

TCP can detect errors or lost data and can trigger retransmission until the data is received, complete and without errors.

5.4.2.1 TCP Connection/Socket

A TCP connection is done with a 3-way handshake between a client and a server. The following is a simplified explanation of this process.

- The client asks for a connection by sending a TCP segment with the SYN control bit set.
- The server responds with its own SYN segment that includes identifying information that was sent by the client in the initial SYN segment.
- The client acknowledges the server's SYN segment.

The connection is then established and is uniquely identified by a 4-tuple called a *socket* or *socket pair*:

> (destination IP address, destination port number)
> (source IP address, source port number)

During the connection setup phase, these values are entered in a table and saved for the duration of the connection.

5.4.2.2 TCP Header

Every TCP segment has a header. The header comprises all necessary information for reliable, complete delivery of data. Among other things, such as IP addresses, the header contains the following fields:

> **Sequence Number -** This 32-bit number contains either the sequence number of the first byte of data in this particular segment or the Initial Sequence Number (ISN) that identifies the first byte of data that will be sent for this particular connection.
>
> The ISN is sent during the connection setup phase by setting the SYN control bit. An ISN is chosen by both client and server. The first byte of data sent by either side will be identified by the sequence number ISN + 1 because the SYN control bit consumes a sequence number. The following figure illustrates the three-way handshake.

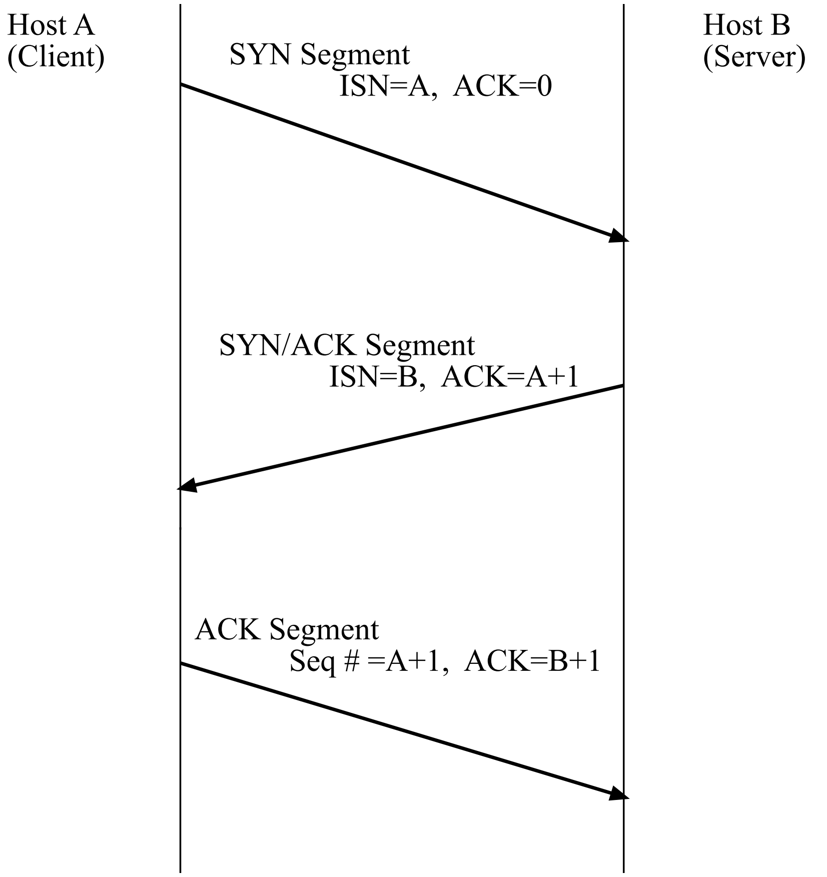

Figure 5. Synchronizing Sequence Numbers for TCP Connection

The sequence number is used to ensure the data is reassembled in the proper order before being passed to an application protocol.

Acknowledgement Number - This 32-bit number is the other host's sequence number + 1 of the last successfully received byte of data. It is the sequence number of the next expected byte of data. This field is only valid when the ACK control bit is set. Since sending an ACK costs nothing, (because it and the Acknowledgement Number field are part of the header) the ACK control bit is always set after a connection has been established.

The Acknowledgement Number ensures that the TCP segment arrived at its destination.

Control Bits - This 6-bit field comprises the following 1-bit flags (left to right):

- URG - Makes the Urgent Pointer field significant.
- ACK - Makes the Acknowledgement Number field significant.
- PSH - The Push Function causes TCP to promptly deliver data.
- RST - Reset the connection.
- SYN - Synchronize sequence numbers.
- FIN - No more data from sender, but can still receive data.

Window Size - This 16-bit number states how much data the receiving end of the TCP connection will allow. The sending end of the TCP connection must stop and wait for an acknowledgement after it has sent the amount of data allowed.

Checksum - This 16-bit number is the one's complement of the one's complement sum of all bytes in the TCP header, any data that is in the segment and part of the IP packet. A checksum can only detect some errors, not all, and cannot correct any.

5.4.3 ICMP

Internet Control Message Protocol is a set of messages that communicate errors and other conditions that require attention. ICMP messages, delivered in IP datagrams, are usually acted on by either IP, TCP or UDP. Some ICMP messages are returned to application protocols.

A common use of ICMP is "pinging" a host. The Ping command (Packet INternet Groper) is a utility that determines whether a specific IP address is accessible. It sends an ICMP echo request and waits for a reply. Ping can be used to transmit a series of packets to measure average round-trip times and packet loss percentages.

5.5 The Application Layer

There are many applications available in the TCP/IP suite of protocols. Some of the most useful ones are for sending mail (SMTP), transferring files (FTP), and displaying web pages (HTTP). These applications are discussed in detail in the *TCP/IP User's Manual*.

Another important application layer protocol is the Domain Name System (DNS). Domain names are significant because they guide users to where they want to go on the Internet.

5.5.1 DNS

The Domain Name System is a distributed database of domain name and IP address bindings. A domain name is simply an alphanumeric character string separated into segments by periods. It represents a specific and unique place in the "domain name space." DNS makes it possible for us to use identifiers such as zworld.com to refer to an IP address on the Internet. Name servers contain information on some segment of the DNS and make that information available to clients who are called resolvers.

5.5.1.1 DCRTCP.LIB Implementation of DNS

The **resolve()** function in **DCRTCP.LIB** immediately converts a dotted decimal IP address to its corresponding binary IP address and returns this value.

If **resolve()** is passed a domain name, a series of queries take place between the computer that called **resolve()** and computers running name server software. For example, to resolve the domain name www.rabbitsemiconductor.com, the following code (available in **SAMPLES\TCP\DNS.C**) can be used.

```
#define MY_IP_ADDRESS "10.10.6.101"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.19"
#define MY_NAMESERVER "10.10.6.19"

#memmap xmem
#use dcrtcp.lib

main() {
   longword ip;
   char buffer[20];

   sock_init();

   ip=resolve("www.rabbitsemiconductor.com");
   if(ip==0)
     printf("couldn’t find www.rabbitsemiconductor.com\n");
   else

     printf("%s is www.rabbitsemiconductors address.\n”,
      inet_ntoa(buffer,ip));
}
```

Your local name server is specified by the configuration macro **MY_NAMESERVER**. Chances are that your local name server does not have the requested information, so it queries the root server. The root server will not know the IP address either, but it will know where to find the name server that contains authoritative information for the .com zone. This information is returned to your local name server, which then sends a query to the name server for the .com zone. Again, this name server does not know the requested IP address, but does know the local name server that handles rabbitsemiconductor.com. This information is sent back to your local name server, who sends a final query to the local name server of rabbitsemiconductor.com. This local name server returns the requested IP address of www.rabbitsemiconductor.com to your local name server, who then passes it to your computer.

6. Dynamic C TCP/IP Implementation

The Dynamic C TCP/IP protocol suite is contained in a number of Dynamic C libraries. The main library file is **DCRTCP.LIB**. IP version 4 is supported, not version 6. This chapter will describe the configuration macros and the functions used to initialize and drive TCP/IP.

The implementation details that are discussed here pertain to versions of Dynamic C prior to 7.05. Improvements and additions to the TCP/IP suite of protocols are fully documented in the *Dynamic C TCP/IP User's Manual*.

6.1 TCP/IP Configuration Macros

TCP/IP can be configured by defining configuration macros at compile time, by using the **tcp_config()** function (and other functions) at runtime or by using the Dynamic Host Configuration Protocol (DHCP). Some ISPs require that the user provide them with a MAC address from the controller. Run the utility program discussed in Section 2.1 to display the MAC address.

6.1.1 IP Addresses Set Manually

Four pieces of information are needed by any host on a network:

1. The IP address of the host (e.g., the TCP/IP Development Board).
2. The part of the IP address that distinguishes machines on the host's network from machines on other networks (the netmask).
3. The IP address of the router that connects the host's network to the rest of the world (the default gateway).
4. The IP address of the local DNS server for the host's network. This is only necessary if DNS backups are needed.

MY_IP_ADDRESS, **MY_NETMASK**, **MY_GATEWAY** and **MY_NAMESERVER** respectively correspond to these four critical addresses.

6.1.2 IP Addresses Set Dynamically

The macro **USE_DHCP** enables the Dynamic Host Configuration Protocol (DHCP). If this option is enabled, a DHCP client contacts a DHCP server for the values of **MY_IP_ADDRESS**, **MY_NETMASK**, **MY_GATEWAY** and **MY_NAMESERVER**.

DHCP servers are usually centrally located on a local network and operated by the network administrator.

6.1.3 Default Buffer Size

There are two macros used to define the size of the buffer that is used for UDP datagram reads and TCP packet reads and writes: **tcp_MaxBufSize** and **SOCK_BUF_SIZE**.

tcp_MaxBufSize is deprecated in Dynamic C version 6.57 and higher and is being kept for backwards compatibility. It has been replaced by **SOCK_BUF_SIZE**.

If **SOCK_BUF_SIZE** is 4096 bytes, the UDP buffer is 4096 bytes, the TCP read buffer is 2048 bytes and the TCP write buffer is 2048 bytes.

In Dynamic C versions 6.56 and earlier, **tcp_MaxBufSize** determines the size of the input and output buffers for TCP/IP sockets. The **sizeof(tcp_Socket)** will be about 200 bytes more than double **tcp_MaxBufSize**. The optimum value for local Ethernet connections is greater than the Maximum Segment Size (MSS). The MSS is 1460 bytes. You may want to lower **tcp_MaxBufSize**, which defaults to 2048 bytes, to reduce RAM usage. It can be reduced to as little as 600 bytes.

tcp_MaxBufSize will work slightly differently in Dynamic C versions 6.57 and higher. In these later versions the buffer for the UDP socket will be **tcp_MaxBufSize * 2**, which is twice as large as before.

6.1.4 Delay a Connection

Sometimes it is appropriate to accept a connection request when the resources to do so are not available. This happens with web servers when web pages have several graphic images, each requiring a separate socket.

The macro **USE_RESERVEPORTS** is defined by default. It allows the use of the function **tcp_reserveport()**. When a connection to the port specified in **tcp_reserveport()** is attempted, the 3-way handshaking is done even if there is not yet a socket available. This is done by setting the window parameter in the TCP header to zero, meaning, “I can take 0 bytes of data at this time.” The other side of the connection will wait until the value in the window parameter indicates that data can be sent.

When using **tcp_reserveport()**, the 2MSL (for Maximum Segment Lifetime) waiting period for closing a socket is avoided.

Using the companion function, **tcp_clearreserve()**, causes the connection to the port to be done in the conventional way.

6.1.5 Runtime Configuration

Functions are provided to change configuration values at runtime. The most general one is **tcp_config()**. It takes two strings. The first string is the setting to be changed and the second string is the value to change it to. The configuration macros **MY_IP_ADDRESS**, **MY_NETMASK**, **MY_GATEWAY**, and **MY_NAMESERVER** can all be overridden by this function.

```
tcp_config("MY_IP_ADDRESS","10.10.6.101");
```

Some of the **tcp_config()** functionality is duplicated by other Dynamic C TCP/IP functions. **tcp_config()** can override the macro **MY_IP_ADDRESS**, and so can the **sethostid** function.

6.2 Skeleton Program

The following program is a general outline for a Dynamic C TCP/IP program. The first couple of defines set up the default IP configuration information. The “memmap” line causes the program to compile as much code as it can in the extended code window. The “use” line causes the compiler to compile in the Dynamic C TCP/IP code using the configuration data provided above it.

```
#define MY_IP_ADDRESS "10.10.6.101"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.19"
#memmap xmem
#use dcrtcp.lib
main() {
   sock_init();
   for (;;) {
      tcp_tick(NULL);
   }
}
```

To run this program, start Dynamic C and open the **SAMPLES\TCPIP\PINGME.C** file. Edit the **MY_IP_ADDRESS**, **MY_NETMASK**, and **MY_GATEWAY** macros to reflect the appropriate values for your device. Run the program and try to run **ping 10.10.6.101** from a command line on a computer on the same physical network, replacing **10.10.6.101** with your value for **MY_IP_ADDRESS**.

The **main()** function first initializes the **DCRTCP.LIB** TCP/IP stack with a call to **sock_init()**. This call initializes internal data structures and enables the Ethernet chip, which will take a couple of seconds with the RealTek chip. At this point, **DCRTCP.LIB** is ready to handle incoming packets.

DCRTCP.LIB processes incoming packets only when **tcp_tick()** is called. Internally, the functions **tcp_open()**, **udp_open()**, **sock_read()**, **sock_write()**, **sock_close()**, and **sock_abort()** all call **tcp_tick()**. It is a good practice to make sure that **tcp_tick()** is called periodically in your program to insure that the system has had a chance to process packets.

When you ping your device, the Ethernet chip temporarily stores the packet, waiting for **DCRTCP.LIB** to process it. **DCRTCP.LIB** removes the packet the next time **tcp_tick()** gets called, and responds appropriately.

A rule of thumb is to call **tcp_tick()** around 10 times per second, although slower or faster call rates should also work. The Ethernet interface chip has a large buffer memory, and TCP/IP is adaptive to the data rates that both end of the connection can handle; thus the system will generally keep working over a wide variety of tick rates.

A more difficult question is how much computing time is consumed by each call to **tcp_tick()**. Rough numbers are less than a millisecond if there is nothing to do, 10s of milliseconds for typical packet processing, and 100s of milliseconds under exceptional circumstances.

6.3 TCP Socket

For Dynamic C version 6.57 and above, each socket must have an associated **tcp_Socket** of 145 bytes or a **udp_Socket** of 62 bytes. The I/O buffers are in extended memory.

For earlier versions of Dynamic C, each socket must have a **tcp_Socket** data structure that holds the socket state and I/O buffers. These structures are, by default, around 4200 bytes each. The majority of this space is used by the input and output buffers.

There are two ways to open a TCP socket: passive or active.

6.3.1 Passive Open

To wait for someone to contact your device, open a socket with **tcp_listen()**. This type of open is commonly used for Internet servers that listen on a well-known port, like 80 for HTTP. You supply **tcp_listen()** with a pointer to a **tcp_Socket** data structure, the local port number others will be contacting on your device, and the IP address and port number that are valid for the device. If you want to be able to accept connections from any IP address or any port number, set one or both to zero.

To handle multiple simultaneous connections, each new connection will require its own **tcp_Socket** structure and a separate call to **tcp_listen()**, but using the same local port number (**lport** value.)

The **tcp_listen()** call will immediately return, and you must poll for the incoming connection. You can use the **sock_wait_established** macro, which will call **tcp_tick()** and block until the connection is established or you can manually poll the socket using **sock_established**.

6.3.1.1 Example of Passive Open

The following example waits for a connection on port 7, and echoes back each line as you enter it. To test this program, change the configuration information and start it running. From a connected PC, telnet to the device using port 7.

```
#define MY_IP_ADDRESS "10.10.6.101"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.19"

#memmap xmem
#use "dcrtcp.lib"

#define PORT 7

tcp_Socket echosock;

main() {
   char buffer[2048];
   int status;

   sock_init();

   while(1) {
      tcp_listen(&echosock,PORT,0,0,NULL,0);
      sock_wait_established(&echosock,0,NULL,&status);

      printf("Receiving incoming connection\n");
      sock_mode(&echosock,TCP_MODE_ASCII);

      while(tcp_tick(&echosock)) {
         sock_wait_input(&echosock,0,NULL,&status);
         if(sock_gets(&echosock,buffer,2048))
            sock_puts(&echosock,buffer);
      }

   sock_err:
      switch(status) {
         case 1: /* foreign host closed */
            printf("User closed session\n");
            break;

         case -1: /* time-out */
            printf("\nConnection timed out\n");
            break;
      }
   }
}
```

6.3.2 Active Open

When your Web browser retrieves a page, it is actively opening one or more connections to the Web server's passively opened sockets. To actively open a connection, call **`tcp_open()`**, which uses parameters that are similar to the **`tcp_listen()`** call. It is necessary to supply exact parameters for **`ina`** and **`port`**, but the **`lport`** parameter can be zero, which tells **`DCRTCP.LIB`** to select an unused port between 1024 and 65536.

When you call **`tcp_open()`**, Dynamic C tries to contact the other device to establish the connection. **`tcp_open()`** will fail and return a zero if the connection could not be opened due to routing difficulties, such as an inability to resolve the remote host's hardware address with ARP.

```
#define MY_IP_ADDRESS "10.10.6.101"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.19"
#define MY_NAMESERVER "10.10.6.19"

#define WEBSITE "www.zweng.com"
#define FILE "/"
#define PORT 80

#memmap xmem
#use "dcrtcp.lib"

main() {
   int status;
   tcp_Socket s;
   char buffer[2048];
   longword ip;

   sock_init();

   ip=resolve(WEBSITE);
   tcp_open(&s,0,ip,PORT,NULL);
   sock_wait_established(&s,0,NULL,&status);

   sock_mode(&s,TCP_MODE_ASCII);

   sprintf(buffer,"GET %s\r\n",FILE);
   sock_puts(&s,buffer);

   while(tcp_tick(&s)) {
      sock_wait_input(&s,0,NULL,&status);
      if(sock_gets(&s,buffer,2048))
         printf("%s\n",buffer);
   }
   return 0;

   sock_err:
   switch(status) {
      case 1: /* foreign host closed */
         printf("User closed session\n");
         break;

      case -1: /* time-out */
         printf("\nConnection timed out\n");
         break;
   }
}
```

6.3.3 TCP Socket Functions

There are many functions that can be applied to an open TCP socket. They fall into three main categories: control, status, and I/O. Each function is explained in the *Dynamic C TCP/IP User's Manual*.

6.3.3.1 Control Functions

- **`sock_abort()`**
- **`sock_close()`**
- **`sock_flush()`**
- **`sock_flushnext()`**
- **`tcp_listen()`**
- **`tcp_open()`**

`tcp_open()` and **`tcp_listen()`** have already been explained in the active and passive sections.

`sock_close()` should be called when you want to end a connection. A call to **`sock_close()`** may not immediately close the connection because it may take some time to send the request to end the connection and receive the acknowledgements. If you want to be sure that the connection is completely closed before continuing your program, you can call **`tcp_tick()`** with the address of the socket. When **`tcp_tick()`** returns zero, then the socket is completely closed. Please note that if there is data left to be read on the socket, the socket will not completely close.

There may be some reason that you want to cancel an open connection. In this case, you can call **`sock_abort()`**. This function will cause a TCP reset to be sent to the other end, and other future packets sent on this connection will be ignored.

For performance reasons, data may not be immediately sent from a socket to its destination. If your application requires the data to be sent immediately, you can call **`sock_flush()`**. This function will cause **`DCRTCP.LIB`** to try sending any pending data immediately. If you know ahead of time that data will need to be sent immediately, call **`sock_flushnext()`**on the socket. This function will cause the next set of data written to the socket to be sent immediately, and is more efficient than **`sock_flush()`**.

6.3.3.2 Status Functions

- **sock_bytesready()**
- **sock_established()**
- **sock_rbleft()**
- **sock_rbsize()**
- **sock_rbused()**
- **sock_tbleft()**
- **sock_tbsize()**
- **sock_tbused()**
- **tcp_tick()**

When you supply **tcp_tick()** with a pointer to a TCP socket, it will first process the packets and then check to see if the socket has an established connection. It returns a zero if the socket is no longer open because of an error condition or if the socket has been closed. You can use this functionality after calling **sock_close()** on the socket to determine whether the socket is completely closed.

```
sock_close(&my_socket);
while(tcp_tick(&my_socket)) {
   // check time-out, do idle work...
}
```

The status functions can be used to avoid blocking when using **sock_write()** and some of the other I/O functions. The following blocks of code illustrate a way of using the buffer management and socket management functions to avoid blocking. The first block of code checks to make sure that there is enough room in the buffer before adding data with a blocking function. The second makes sure that there is a string terminated with a new line in the buffer, or that the buffer is full before calling **sock_gets()**.

```
if(sock_tbleft(&my_socket,size)) {
   sock_write(&my_socket,buffer,size);
}
```

or:

```
sock_mode(&my_socket,TCP_MODE_ASCII);
if(sock_bytesready(&my_socket) != -1) {
   sock_gets(buffer,MAX_BUFFER);
}
```

6.3.3.3 I/O Functions

- **`sock_fastread()`**
- **`sock_fastwrite()`**
- **`sock_getc()`**
- **`sock_gets`**
- **`sock_preread()`**
- **`sock_putc()`**
- **`sock_puts()`**
- **`sock_read()`**
- **`sock_write()`**

There are two modes of reading and writing to TCP sockets: ASCII and binary. By default, a socket is opened in binary mode, but you can change that with a call to **`sock_mode()`**.

When a socket is in ASCII mode, **`DCRTCP.LIB`** assumes that the data is an ASCII stream with record boundaries on the newline characters for some of the functions. This behavior means **`sock_bytesready()`** will return >=0 only when a complete newline-terminated string is in the buffer or the buffer is full. The **`sock_puts()`** function will automatically place a newline character at the end of a string, and **`sock_gets()`** will strip the newline character.

When in binary mode, do not use **`sock_gets()`**.

6.4 UDP Interface

- **`sock_fastread()`**
- **`sock_fastwrite()`**
- **`sock_getc()`**
- **`sock_gets`**
- **`sock_putc()`**
- **`sock_puts()`**
- **`sock_read()`**
- **`sock_recv()`**
- **`sock_recv_from()`**
- **`sock_recv_init()`**
- **`sock_write()`**
- **`udp_open()`**

The UDP protocol is useful when sending messages where either a lost message does not cause a system failure or is handled by the application. Since UDP is a simple protocol and you have control over the retransmissions, you can decide if you can trade low latency for high reliability. Another advantage of UDP is the ability to broadcast packets to a number of computers on the same network. When done properly, broadcasts can reduce overall network traffic because information does not have to be duplicated when there are multiple destinations.

6.4.1 Opening and Closing

The **udp_open()** function takes a remote IP address and port number. If they are set to a specific value, all incoming and outgoing packets are filtered on that value (i.e., you talk only to the one socket).

If the remote IP address is set to -1, it receives any packet, and outgoing packets are broadcast. If the remote IP address is set to 0, no outgoing packets may be sent until a packet has been received. This first packet completes the socket, filling in the remote IP address and port number with the return address of the incoming packet. Multiple sockets can be opened on the same local port, with the remote address set to 0, to accept multiple incoming connections from separate remote hosts. When you are done communicating on a socket that was started with a 0 IP address, you can close it with **sock_close()** and reopen to make it ready for another source.

6.4.2 Writing

The normal socket functions you used for writing to a TCP socket will work for a UDP socket, but since UDP is a significantly different service, the result could be different. Each atomic write—**sock_putc()**, **sock_puts()**, **sock_write()**, or **sock_fastwrite()—**places its data into a single UDP packet. Since UDP does not guarantee delivery or ordering of packets, the data received may be different either in order or content than the data sent.

6.4.3 Reading

There are two ways to read packets using **DCRTCP.LIB**. The first method uses the normal **sock_getc()**, **sock_gets()**, **sock_read()**, and **sock_fastread()** functions. These functions will read the data as it came into the socket, which is not necessarily the data that was written to the socket.

The second mode of operation for reading uses **sock_recv_init()**, **sock_recv()**, and **sock_recv_from()**. The **sock_recv_init()** function installs a large buffer area that gets divided into smaller buffers. Whenever a datagram arrives, **DCRTCP.LIB** stuffs that datagram into one of these new buffers. **sock_recv()** scans the buffers for any datagrams received by that socket. If there is a datagram, the length is returned and the user buffer is filled, otherwise it returns zero.

The **sock_recv_from()** function works like **sock_recv()**, but it allows you to record the IP address where the datagram originated. If you want to reply, you can open a new UDP socket with the IP address modified by **sock_recv_from()**. There is no way to send UDP packets without a socket.

After calling **sock_recv_init()** on the socket, you should not use **sock_getc()**, **sock_read()**, or **sock_fastread()**.

6.4.4 Checksums

There is an optional **checksum** field inside the UDP header. This field verifies only the header portion of the packet and doesn't cover any part of the data. This feature can be disabled on a reliable network where the application has the ability to detect transmission errors. Disabling the UDP **checksum** can increase the performance of UDP packets moving through **DCRTCP.LIB**. This feature can be modified by:

```
sock_mode(s, UDP_MODE_CHK);      // enable checksums
sock_mode(s, UDP_MODE_NOCHK);    // disable checksums
```

6.5 Program Design

When designing your program, you must place some thought into how it will be structured. If you plan on using the state-based approach, you need to select the appropriate functions.

6.5.1 State-Based Program Design

One strategy for designing your program with Dynamic C is to create a state machine within a function where you pass it the socket. This method allows you to handle multiple sockets without the services of a multitasking kernel. This is the way the **HTTP.LIB** functions are organized (see HTTP in the *Dynamic C TCP/IP User's Manual)*. The general states are waiting to be initialized, waiting for a connection, a bunch of connected states, and waiting for the socket to be closed. Many of the common Internet protocols fit well into this state machine model. An example of state-based programming is **SAMPLES\TCPIP\STATE.C**. This program is a basic Web server that should work with most browsers. It allows a single connection at a time, but could easily be extended to allow multiple connections.

6.5.2 Blocking vs. Non-Blocking

The **sock_fastread()** and **sock_preread()** functions read as much data as is available in the buffers, and return immediately. Similarly, **sock_fastwrite()** fills the buffers and returns the number of characters that were written. When using these functions, it is your responsibility to ensure that all of the data were written completely.

```
offset=0;

while(offset<length) {
   bytes_written=sock_fastwrite(&socket,buffer+offset,length-offset);
   if(bytes_written<0) {
      // error handling
   }
   offset+=bytes_written;
}
```

The other functions do not return until they have completed or there is an error. If it is important to avoid blocking, you can check the conditions of an operation to insure that it will not block.

```
sock_mode(socket,TCP_MODE_ASCII);
...
if (sock_bytesready(&my_socket) != -1){
   sock_gets(buffer,MAX_BUFFER);
}
```

In this case **sock_gets()** will not block because it will be called only when there is a complete new line terminated record to read.

6.5.3 Blocking Macros

To block at a certain point and wait for a condition, **DCRTCP.LIB** provides some macros to make this task easier. In this program fragment, **sock_wait_established** is used to block the program until a connection is established. Notice the time-out (second parameter) value of zero. This tells Dynamic C to never time-out. Associated with these macros is a **sock_err** label to jump to when there is an error. If you supply a pointer to a status integer, it will set the status to an error code. Valid error codes are -1 for time-out and 1 for a reset connection.

```
tcp_open(&s,0,ip,PORT,NULL);
sock_wait_established(&s,0,NULL,&status);

//...

sock_err:
switch(status) {
   case 1:      /* foreign host closed */
      printf("User closed session\n");
      break;
   case -1:     /* time-out */
      printf("\nConnection timed out\n");
      break;
}
```

6.6 Multitasking and TCP/IP

The TCP/IP engine may be used with the µC/OS real-time kernel. The line

```
#use ucos2.lib
```

must appear before the line

```
#use dcrtcp.lib
```

7. References

1. A two-part article, *Introduction to TCP/IP*, in Embedded Systems Programming discusses issues related to programming embedded systems.

   http://www.embedded.com/internet/9912/9912ia1.htm

2. Ethereal is a good, free program for viewing network traffic. It works under various Unix operating systems and under Windows.

   http://www.ethereal.com

3. *Computer Networks and Internets*, Douglas E. Comer. Published by Prentice Hall. ISBN 0-13-239070-1. This book gives an excellent high-level description of networks and their interfaces.

4. *TCP/IP Illustrated, Volume 1 The Protocols*, W. Richard Stevens. Published by Addison-Wesley. ISBN 0-20-163346-9. This book gives many useful low-level details about TCP/IP, UDP and ICMP.

Notice to Users

Z-WORLD PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE-SUPPORT DEVICES OR SYSTEMS UNLESS A SPECIFIC WRITTEN AGREEMENT REGARDING SUCH INTENDED USE IS ENTERED INTO BETWEEN THE CUSTOMER AND Z-WORLD PRIOR TO USE. Life-support devices or systems are devices or systems intended for surgical implantation into the body or to sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling and user's manual, can be reasonably expected to result in significant injury.

No complex software or hardware system is perfect. Bugs are always present in a system of any size. In order to prevent danger to life or property, it is the responsibility of the system designer to incorporate redundant protective mechanisms appropriate to the risk involved.

The TCP/IP software is designed for use only with Rabbit Semiconductor chips.