Treck Inc.

Treck TCP/IP User Manual

Version 4.0

Complete Internet Solutions

Technical Support:
5041 Lamart Drive #240 Riverside, California 92507
Phone: (909) 787-7056 Fax: (909) 787-8803

Corporate Headquarters:
Treck, Inc. 431 Ohio Pike Suite 210 North Cincinnati, Ohio 45255
Phone: (513) 528-5732 Fax: (513) 528-5740

Contents

Introduction to TCP/IP

Short Background of the Internet

The TCP/IP protocol suite is also known as the "Internet" Protocol Suite since it is used to move data across the Internet. The Internet is a worldwide network of computers located at many different companies, institutions, and government offices. The Internet was started over three decades ago. Realizing the need for a common method to share data and send messages in electronic form, the US Department of Defense (DOD) started the Internet project within the Defense Advanced Research Projects Agency. The first Internet was called ARPANET and started operation in 1968. As more and more researchers joined ARPANET, people started to realize the need for a global network. In 1979, a group called the Internet Control and Configuration Board (ICCB) was founded by ARPA and met to coordinate and structure the architecture of the now growing ARPANET. In 1980, the ARPANET was extended to a global network when the computers that needed network connectivity were outfitted with the TCP/IP protocol suite. In 1983, the MILNET was created to allow military sites to have a separate network from the researchers. Both of these networks (ARPANET for researchers and MILNET for the Military) used TCP/IP for network communication. Now that the military was on its own network, ARPA decided to encourage universities to use the new TCP/IP protocols and utilize the new ARPANET. At most universities at that time, the computer science departments were using the UNIX operating system. ARPA then funded a project to adapt the TCP/IP protocol suite to the BSD (Berkeley Software Distribution) UNIX operating system and provided a low cost means to allow the universities to connect to the ARPANET. The BSD variant of UNIX also had many new tools for the TCP/IP Protocol Suite that was integrated with it. These tools allowed the user to use simple UNIX commands to move files across the network with ease. It was so simple to use that it became popular very quickly. The BSD UNIX also included a new application program interface (API) to allow programmers to access the TCP/IP protocols. This new API was called *sockets* and allowed the user a uniform way to program network applications without regard to the type of computer that it was running on. This allowed researchers to write new programs and protocols to run on top of the TCP/IP protocol. Since BSD UNIX was so popular within computer science departments, TCP/IP was also used for local area networking. The ARPANET became so popular, that in 1986 the National Science Foundation funded the first wide area backbone network called NSFNET.

This new backbone network allowed even more educational institutions to connect to what was now being called the Internet (although the term ARPANET was still being used). Because the National Science Foundation funded NSFNET, universities used it almost exclusively. As commercial sites needed local area connectivity to share files within their company, most of them turned to the IPX/SPX protocols developed by Novell. In fact at one time, Novell was being used by more than 80% of commercial sites using local area networking. As the need for Electronic Mail (e-mail) and wide area networking grew in the commercial sector, companies turned to the Internet and the TCP/IP protocols that it used. New companies were formed to create new commercial backbone networks. One of the first of these was UUNET. Over the past few years, the Internet has exploded onto the commercial scene because of a new way of advertising called web pages. Today, it is difficult to watch television without being bombarded with web page addresses.

What is a Protocol?

The definition of a protocol is "A set of formal rules describing how to transmit data, especially across a network." What are these rules? Well these rules consist of how the packet is formatted, how much data it carries, if and how it can be sure that the remote computer has actually received the data, as well as any additional information that the protocol needs to send to the remote side. In our software, we utilize the TCP/IP (or Internet) protocol suite set of rules. These rules are outlined in loose specifications called Request for Comments (RFC's). An RFC is submitted by anyone who thinks that some change or addition to the Internet protocol suite needs to happen. This proposed RFC is sent to the appropriate working group at the Internet Engineering Task Force (IETF) for comments by others who are involved with that working group. If others in the working group agree with the proposed idea, then they may ask for some changes and adopt the RFC as a "Standards Track" document. If they reject the idea, then the RFC can still be published as an "Informational" RFC. Standards Track documents describe what the protocol should do in all implementations. Informational RFC's describe what a vendor would like others to implement. There are other types of RFC's, but most are beyond the scope of this document. You can visit the IETF web site at
http://www.ietf.org. If you are looking for a particular RFC, it is always easier to locate the repository closest to you via a web search engine (such as Altavista, Infoseek, and Yahoo).

The TCP/IP Protocol Stack

A "stack" architecture divides out the functionality of a data communications capability into several separate layers. Each layer then communicates with its peer layer on remote machine or on the same machine. Rather than tightly coupling the hardware interface with addressing, the stack model identifies a separate interface through which these functions shall cooperate. Figure 1-1 shows a sample of an OSI architecture model. The lower levels of the stack consist of independent protocols that support specific hardware interfaces, transport mechanisms, and so forth while the higher layer protocols are interfaces into the user application code.

7	Application
6	Presentation
5	Session
4	Transmission
3	Network
2	Data Link
1	Physical

Fig. 1-1
The OSI Architecture Model

Figure 1-2 shows the TCP/IP model. We have mapped some of the protocols in use with TCP/IP into the OSI reference model. Note that BSD Sockets is at the session layer. BSD Sockets is *not* a protocol. It is a user API to allow access to the protocol stack beneath.

<table>
<tr><td>7 Application</td><td colspan="6">Your Application and/or Treck Applications</td></tr>
<tr><td>6 Presentation</td><td colspan="6">Host / Network Byte Order Conversion</td></tr>
<tr><td>5 Session</td><td colspan="6">RPC</td></tr>
<tr><td>4 Transmission</td><td colspan="3">TCP</td><td colspan="3">UDP</td></tr>
<tr><td>3 Network</td><td>IP</td><td colspan="3">ICMP</td><td colspan="2">ARP</td></tr>
<tr><td>2 Data Link</td><td colspan="2">Ethernet</td><td colspan="3">PPP</td><td>SLIP</td></tr>
<tr><td>1 Physical</td><td colspan="6">Your Network Hardware</td></tr>
</table>

Fig. 1-2
The TCP/IP Model

The TCP/IP stack layers perform the following functions:

Media Access (Physical) Protocols
Specify the mechanisms for client and server nodes on a network to interface to the transmission media.

Data-Link Protocols
Specify the control characters and the lowest level mechanisms for transmitting packets of data in successive small segments (called *frames*) between nodes.

Network Protocols
Means by which packets of data are routed through the network from sender to receiver. This level is more concerned with the path the packets take not the content of those packets.

Transport Protocols
Assume responsibility for the delivering of a potentially large message from the sending application on one network to the receiving destination.

Session Protocols
Responsible for negotiating parameters for the link (i.e. which layer 3 protocol to use).

Presentation Protocols
Responsible for making sure that data is viewed the same on both ends of the communication. Can also be used for encryption and compression.

Application Protocols
Forms the working toolset for network users and the applications that are written to support them.

In the original TCP/IP stack, the network layer consisted of the Internet Protocol (IP), and the transport layer consisted of the Transport Control Protocol (TCP) for reliable delivery of application messages and User Datagram Protocol for (UDP) for efficient exchange of small packets-primarily for control and administrative purposes.

The Ethernet Protocol

From its inception in 1978 by the Xerox Corporation, Intel Corporation, and Digital Equipment Corporation, Ethernet has become a very popular LAN technology. It has grown to be one of the most widely used methods of packet switching between computers.

The original configuration of Ethernet technology involved setting up a connection between two computers using a coaxial cable. This cable was generally ½” diameter and could be up to 500 meters long. (See figure 1-3.)

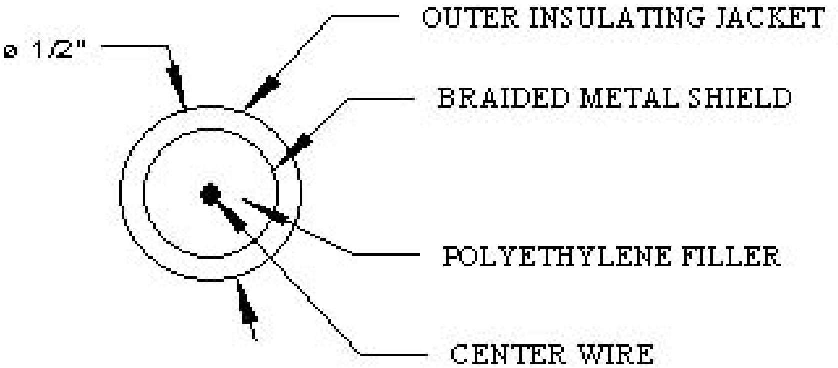


Fig. 1-3
Cross Section of a Coaxial Cable Used in the Original Ethernet

The cable itself was completely passive. It carried no electrical current. The only active electrical components that made the network function were associated with the computers attached to the network.

The connection between a computer and an Ethernet cable required a piece of hardware called a *transceiver*. Looking at the physical cable, there needed to be a hole in the outer layers for placement of the transceiver. This was commonly referred to as a *tap*. Usually small metal pins mounted to the transceiver would go through the hole in the cable to provide electrical contacts to the center wire and the braided shield. Some manufactures made “T’s” that required the cable to be cut and inserted.

Besides the *transceiver*, there also needed to be a *host interface* or *host adapter* that would plug into the computer’s bus. This enabled the connection to be made from the computer to the network.

The *transceiver* is a small piece of hardware that was placed adjacent to the ether. It contains digital circuitry that allows it communicate with a digital computer. The *transceiver* can sense if the ether is in use and can translate analog signals to and from digital form. The cable that connects the *transceiver* and the *host adapter* is called the *Attachment Unit Interface* (AUI). This cable contains many wires. These wires carry electrical current needed to operate the *transceiver*, the signals that control the *transceiver* operation, and the contents of the packet being sent or received. (See fig. 1.4.)

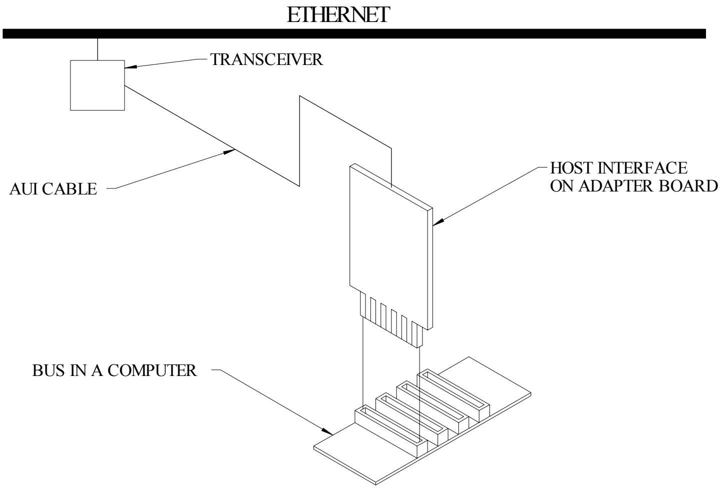


Fig. 1-4
Host to Ethernet Connection

The AUI cable connects the host interface and the transceiver. It carries power, signals, and transmitting or receiving packets.

Because of several obvious reasons, like a big, hard to bend, bulky wire, and the chance of electrical interference, engineers developed a new method of Ethernet wiring. It is called *thin wire Ethernet* or *thinnet*. This cabling system proved to be more flexible, thinner, and less expensive. However, because of using such a thin wire, the cable provided less protection against electrical interference. It could not be placed near powerful electrical equipment, like that found in a factory. It also covers shorter distances and supports fewer computer connections per network than thick Ethernet. On a thin-wire Ethernet network, the costly, bulky transceivers were replaced with digital, high-speed circuits. That enabled direct connection from a computer to the Ethernet. Now instead, the computer contains both the circuitry and the host interface. This is beneficial to manufacturers of small computers and workstations because they can integrate Ethernet hardware into single board computers and mount connectors directly to the back of the computer.

This method is also beneficial if many computers occupy a single room. The thinnet cable runs from one computer workstation directly to another. Another computer can be added by simply connecting that computer to another on the thinnet chain. The disadvantage to this is that it enables users to manipulate the ether. If it is disconnected, all the computers in the chain are unable to communicate. However, in most cases the advantages outweigh the disadvantages.

This method uses BNC connectors to connect to the Ethernet. This is much simpler that coaxial cable. A BNC connector can be installed easily without the use of any tools. Therefore, a user can connect to the Ethernet without the aid of a technician.

Twisted Pair Ethernet

In the last few years, technology has made advancements that have made it possible to construct an Ethernet that no longer needs electrical shielding of a coaxial cable. This type of Ethernet technology is called *twisted pair Ethernet*. This technology allows the user to connect to the Ethernet using a pair of conventional unshielded copper wires similar to the wires used to connect telephones. One obvious advantage to this method is reduced cost, but it also protects other computers on the network from a user who disconnects a single computer. Sometimes it is even possible to use existing telephone wiring for the Ethernet.

Known technically as *10Base-T*, a twisted pair wiring scheme connects each computer on a network using Ethernet *hubs* . An Ethernet hub is an electronic device that simulates the signals on an Ethernet cable. The physical unit is a small box or hub that usually reside in the wiring closet or telephone room.

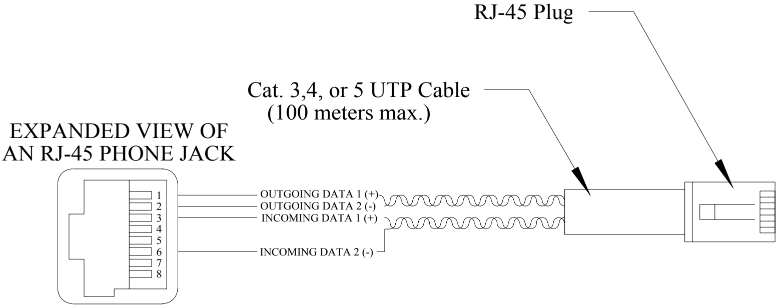


Fig. 1-5
Exploded View of Twisted Pair Cable

Using this method only allows for a connection to be made between a computer and a *hub* within 100 meters. A hub requires power, and can be set up for authorized personnel to monitor and control its operation over the network. An Ethernet hub provides the same communication capability as a thin or thick Ethernet. They merely allow for alternative wiring schemes.

As we have discussed, when using a thick Ethernet setup, the connection requires an AUI connection. When using a thin Ethernet setup, the connection requires a BNC connector, and when using a *10Base-T* connection, an *RJ-45* connector must be used. The connectors resemble a modular telephone plug. Many Ethernet manufacturers provide the option for the user to select which one of the three they want to use. Although only one connector can be used at a time, this allows the user to integrate easily between the three options.

Note that pair 1 and 2 and pairs 3 and 6 are “twisted”. This is done for noise cancellation.

Pin 1	Outgoing Data 1 (+)
Pin 2	Outgoing Data 2 (-)
Pin 3	Incoming Data 1 (+)
Pin 4	No connection
Pin 5	No connection
Pin 6	Incoming Data 2 (-)
Pin 7	No connection
Pin 8	No connection

Fig. 1-6
Pin Assignment of an RJ-45 Connector

Properties of an Ethernet

The Ethernet is a 10 Mbps broadcast bus technology with best-effort delivery semantics and distributed access control. It is a *bus* because all stations share a single communication channel. However, it is also a *broadcast* system because all transceivers receive every transmission. It is important to know that all transceivers do not distinguish among transmissions.

A transceiver passes all packets from the cable to the *host interface*. The *host interface* then chooses which packets the computer should receive, and filters out all others. It is called a *best-effort* delivery system because the hardware provides no information to the sender about whether the packet was successfully delivered. With that in mind, if a destination machine happens to be powered down, the packets sent to that machine will be lost without notifying the sender of that fact. Later, we will see how the TCP/IP protocols accommodate best effort delivery systems.

Ethernet has no central authority to grant access; therefore we say that access control is distributed. The Ethernet access scheme is called *Carrier Sense Multiple Access* with *Collision Detect.* It is CSMA because multiple machines can access the Ethernet simultaneously and each machine determines whether the ether is idle by sensing if there is a carrier wave present. Before a host interface sends a packet, it listens to the ether to see if a message is already being transmitted. We call this *carrier sensing.* If there is no transmission sensed, then it begins transmitting. Each transmission is limited in duration, because there is a maximum packet size (which we will discuss later).

Also, the hardware must observe a minimum idle time between transmissions. That means that no single pair of communicating machines can use the network without giving other machines an opportunity for access.

Collision Detect and Recovery

When a transceiver begins transmission, the signal does not reach all parts of the network at the same time. A transmission travels along the cable at approximately 80% the speed of light. It is possible that two machines can sense that the ether is idle and begin transmitting at the same time. When two electrical signals get crossed they become scrambled, such that neither is meaningful. These incidents are called *collisions*.

The Ethernet makes provisions for this happening. Each transceiver is constantly monitoring the cable while it is transmitting to see if a foreign signal interferes with its transmission. Technically, this monitoring is called *collision detect*. When a collision is detected, the host interface aborts transmission, and then waits for activity to subside, and then tries to transmit again. Care must be taken however, or the network could wind up with all the transceivers busily trying to transmit resulting in collisions. This is where the Ethernet excels, it uses a binary exponential back-off policy where a sender delays a random time after the first collision, twice as long if a second collision occurs, fours times as long if a third attempt results in a collision, and so on. The logic behind exponential back-off is that in the unlikely event that many stations try to transmit simultaneously, a severe "traffic jam" could occur. In such a jam, there is a very good chance that two stations will choose very similar back-off times, resulting in a high chance for another collision. By doubling the random delay time, the exponential back-off strategy quickly spreads the attempts to retransmit over a reasonably long period of time, therefore resulting in fewer collisions.

Because an Ethernet is a bus with the possibility of collisions, one should not over utilize the Ethernet. 80% utilization is about the maximum any Ethernet should be loaded with.

Ethernet Capacity

The standard Ethernet is rated at 10 Mbps. That means that data can be transmitted onto the cable at 10 million bits per second. Although computers can generate data at Ethernet speed, raw network speed should not be thought of as the rate at which two computers can exchange data. Instead, network speed should be thought of as a measure of the network's traffic capacity. Think of a network as a water supply line in a house, and think of packets as the water. A large diameter pipe can carry a large quantity of water, while a small diameter pipe can only carry a small amount of water. So if you have a ½" supply line to 20 sinks, chances are that it will not work efficiently if all the sinks are turned on at the same time. However, if you have a 3" supply line to those same 20 sinks, it could easily handle supplying water if those sinks were all turned on at the same time. This is the same way with an Ethernet. For example, a 10Mbps Ethernet can handle a few computers that generate heavy loads or many computers generating light loads.

Ethernet Hardware Addressing

An Ethernet uses a 48-bit addressing scheme. Each computer that attached to an Ethernet network is assigned a unique 48-bit number that is known as its *Ethernet Address*. To assign an Ethernet address, Ethernet hardware manufacturers request blocks of Ethernet addresses and assign them in sequence as they manufacture Ethernet interface hardware. Therefore, no two hardware interfaces have the same Ethernet address. The first 24 bits in an *Ethernet Frame* are the manufacturer's ID.

Fig. 1-7
Ethernet Addressing Scheme

Usually, the Ethernet address is fixed in machine-readable code on the host interface hardware. Because Ethernet addresses belong to hardware devices, they are sometimes called *hardware address* or *Physical Addresses*.

Note: Physical addresses are associated with the Ethernet interface hardware; moving the hardware interface to a new machine or replacing a hardware [redacted] ***ss.***

t clear why higher h changes.

The host interface hardware examines packets and determines which packets should be sent to the host. Remember that each interface receives a copy of every packet, even those addressed to other machines. The host interface uses the destination address field in the packet as a filter. The interface ignores those packets that are addressed to other machines, and passes to the host only those packets addressed to it. The addressing mechanism and the hardware filter are needed to prevent the computer from becoming overwhelmed with incoming data. Although the computer's CPU could perform the check, doing so in the host interface keeps the traffic on the Ethernet from slowing down processing on all computers.

A 48-bit Ethernet address can do more than specify a single destination computer. An address can be one of three types:

- The physical address of one network interface (a *unicast* address)
- The network *broadcast* address
- A *multicast* address

We typically write the Ethernet address in Hexadecimal. For example:

`0x0ce134121978`By convention, the broadcast address (all 1's) is reserved for sending to all stations simultaneously.

48 Bits
11111111111111111111111111111111111111

or in hexadecimal:

48 Bits
FFFFFFFFFFFF

Fig. 1-8
Broadcast Address

Multicast addresses provide a limited form of broadcast in which a subset of the computers on the network agree to listen to a given multicast address. This set of computers is called a *multicast group.* To join a multicast group, the computer must instruct its host interface to accept the group's multicast address. The advantage of multicasting lies in the ability to limit broadcasts. Every computer in a multicast group can be reached with a single packet transmission. Computers that choose not to participate in a particular multicast group do not receive packets sent to the group.

To accommodate broadcast and multicast addressing, Ethernet interface hardware must recognize more than its physical address. A host interface usually accepts at least two kinds of packets: those addressed to the interface's physical address, and those addressed to the network broadcast address. Some interfaces can be programmed to recognize multicast addresses or even alternate physical addresses. When the operating system starts, it initializes the Ethernet interface, giving it a set of addresses to recognize. The interface then examines the destination address field in each packet, passing on to the host only those transmissions designated for one of the specified addresses.

Special Bits of an Ethernet Address

Under universally administrated addressing, a unique address is embedded in ROM on each network interface. The first three bytes of the 48-bit address identify the manufacturer of the adapter; the remaining bits identify the adapter card number. Under locally administrated addressing, the user is responsible for configuring the source address of each workstation.

0 8 16 24 31

<table>
<tr><td colspan="2">Hardware Type</td><td>Protocol Type</td></tr>
<tr><td>Hardware Address Length</td><td>Protocol Address Length</td><td>Operation</td></tr>
<tr><td colspan="3">Sender Hardware Address (Bytes 0-3)</td></tr>
<tr><td colspan="2">Sender Hardware Address (Bytes 4-5)</td><td>Sender IP Address (Bytes 0-1)</td></tr>
<tr><td colspan="2">Sender IP Address (Bytes 2-3)</td><td>Target Hardware Address (Bytes 0-1)</td></tr>
<tr><td colspan="3">Target Hardware Address (Bytes 2-5)</td></tr>
<tr><td colspan="3">Target IP Address (Bytes 0-3)</td></tr>
</table>

Fig. 1-9
Ethernet and IEEE 802.3 Source/Destination Address Fields

Obtaining an Ethernet Address Block

To obtain an Ethernet address block for your company, you need to contact IEEE Standards or visit their website at http://standards.ieee.org/faqs/OUI.html. This will provide information on getting a Company ID/OUI.

Ethernet Frame Format

The Ethernet should be thought of as a link-level connection among machines. Thus, it makes sense to view the data transmitted as a *frame*. The term frame originated from communication over serial lines in which the sender frames the data by adding special characters before and after the transmitted data. Ethernet frames are variable lengths, with no frame smaller than 64 bytes or larger than 1518 bytes (header, data, and CRC). As in all packet switched networks, each Ethernet frame includes a field that contains the address of its destination. Figure 1-10 shows that the Ethernet frame format contains the physical source address as well as the destination address.

Preamble	Destination Address	Source Address	Frame Type	Frame Data	CRC
8 bytes	6 bytes	6 bytes	2 bytes	46-1500 bytes	4 bytes

Fig. 1-10
Ethernet Frame Format

In addition to identifying the source and destination, each frame transmitted across the Ethernet contains a *preamble*, *type field*, *data field*, and *Cyclic Redundancy Check* (CRC). The preamble consists of 64 bits of alternating *0*'s and *1*'s to help receiving nodes synchronize. The 32-bit CRC helps the interface detect the transmission errors. The sender computes the CRC as a function of the data in the frame, and the receiver recomputes the CRC to verify that the packet has been received intact.

The frame type field contains a 16-bit integer that identifies the type of data being carried in the frame. From the Internet point of view, the frame type field is essential because it means that Ethernet frames are *self-identifying*. When a frame arrives at a given machine, the operating system uses the frame type to determine which protocol software module should process the frame. The chief advantage of self-identifying frames is that they allow multiple protocols to be intermixed on the same physical network without interference. For example, one frame could have an application program using Internet protocols while another is used for local experimental protocol. The operating system uses the type field of an arriving frame to decide how to process the contents. We will see that TCP/IP uses self-identifying Ethernet frames to distinguish among several protocols.

The Address Resolution Protocol (ARP)

The Address Resolution Protocol (ARP) is a low-level protocol used to bind addresses dynamically. To better understand this concept, look at Figure 1-11. This shows that the host (H) wants to know a machines IP address (B). To resolve this, host (H) broadcasts a special packet that asks the host with a specific IP address to respond with its physical address. All hosts receive the request, but only (B) recognizes its IP address and sends a reply that contains its physical address. When host (H) receives the reply from host (B), it uses its physical address to send the Internet packet directly to (B).

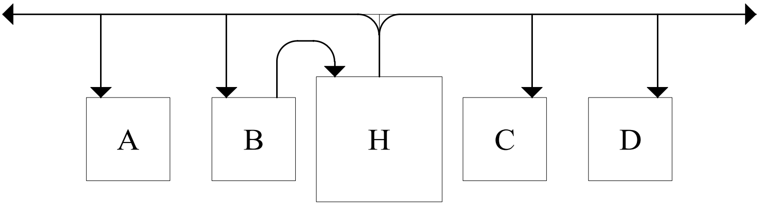


Fig. 1-11
ARP Transmission Sequence

When a host makes a broadcast to all the machines on a network, it makes every machine on the network process the broadcast data. This can be very costly and time consuming. To reduce communication costs and time, machines that ARP maintains a *cache* of recently acquired IP-to-physical address bindings so they do not have to use ARP repeatedly. Whenever a computer receives an ARP reply, it saves the senders IP address and corresponding hardware address in its cache for successive lookups. When transmitting a packet, a computer always looks in its cache for a binding before sending an ARP request. If a computer finds the desired binding in its ARP cache, it need not send a broadcast on the network. Experience shows that because most network communication involves more than one packet transfer, even a small cache is beneficial.

There are several important points to remember about ARP. First, notice that when one machine is about to use ARP (because it needs to send information to another machine), there is a high probability that the second machine will need to communicate with the first machine in the near future. To anticipate the need of the second machine, and to reduce network traffic, the first machine sends its IP-to-physical address binding when sending a request to that machine.

That machine then extracts the first machine's binding from the request and saves that information in its ARP cache. It then sends a reply to the first machine. Because the first machine broadcasts its initial request, all machines on the network receive it and can extract the IP-to-physical address and store it their cache. This saves time in future transmissions. When a computer has its host interface replaced (e.g., because the hardware has failed) its physical address changes. Other computers on the network that have stored a binding in their ARP cache must be notified of the change. A system can notify others of a new address by sending an ARP broadcast when it boots.

To summarize, remember:
The sender's IP-to-physical address binding is included in every ARP broadcast; receivers update the IP-to-physical address binding information in their cache before processing an ARP packet.

ARP is a low-level protocol that hides the underlying network physical addressing, permitting one to assign an arbitrary IP address to every machine. We think of ARP as a part of the physical network system and not as a part of the Internet protocols.

ARP Implementation

Functionally, ARP is divided into two parts. The first part maps an IP address to a physical address when sending a packet. The second part answers requests from other machines. Address resolution for outgoing packets may seem to be pretty straightforward, but some small details can complicate an implementation. When given a destination IP address, the software consults its ARP cache to see if it knows the mapping from IP address to physical address. If it does, the software extracts the physical address, places the data in a frame using that address, and transmits the frame. If it does not know the mapping, then the software must broadcast an ARP request and wait for a reply.

Many things can happen to complicate an address mapping. The target machine can be down, or just too busy to accept the request. If so, the sender may not receive a reply, or the reply may be delayed. Because an Ethernet is best-effort delivery system, the initial ARP broadcast can also be lost (if this happens, then the sender should retransmit, at least once). Meanwhile, the sender must store the original outgoing packet so that it can be sent once the address has been resolved. In fact, the host must decide whether to allow other application programs to proceed while it processes an ARP request.

The second part of the ARP code handles ARP packets that arrive from the network. When an ARP packet arrives, the software first extracts the sender's IP address and hardware address pair, and examines the local cache to see if it already has an entry for that sender. If the cache entry exists for the given IP address, the handler updates that entry by overwriting the physical address with the physical address obtained from the packet. The receiver then processes the rest of the ARP packet. It is important to understand this, for reference to the next section.

ARP Encapsulation and Identification

Before we continue, it is important to know just what *encapsulation* is. Encapsulation is treating a collection of structured information as a whole without affecting, or taking notice of its internal structure. This means that we can take any information, and essentially put it in between pertinent information. There are guidelines to follow that dictate the format of encapsulation. Keeping that in mind, when ARP messages travel from one machine to another, they must be carried in physical frames.

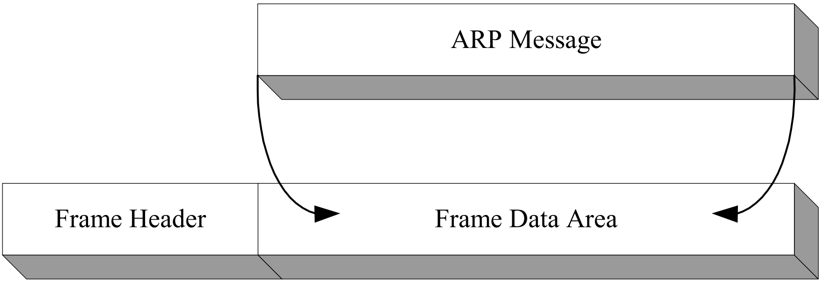


Fig. 1-12
ARP Message Encapsulated in a Physical Network Frame

To identify the frame as carrying an ARP message, the sender assigns a special value to the type field in the frame header and places the ARP message in the frames data field. When a frame arrives at a computer, the network software uses the frame type to determine its contents. In most technologies, a single type value is used for all frames that carry an ARP message. Network software in the receiver must then further examine the ARP message to distinguish between ARP requests and ARP replies.

ARP Protocol Format

Unlike most protocols, the data in ARP packets does not have a fixed format header. Instead to make ARP useful for a variety of network technologies, the length of fields that contain addresses depend on the type of network. To make it possible to interpret an arbitrary ARP message, the header includes the fixed fields near the beginning that specify the lengths of the addresses found in the succeeding fields. In fact, the ARP message format is general enough to allow it to be used with arbitrary physical addresses and arbitrary protocol addresses. Fig. 1.13 shows an ARP message with 4 bytes per line.

0 8 16 24 31

<table>
<tr><td colspan="2">Hardware Type</td><td colspan="2">Protocol Type</td></tr>
<tr><td>Hardware Address Length</td><td>Protocol Address Length</td><td colspan="2">Operation</td></tr>
<tr><td colspan="4">Sender Hardware Address (Bytes 0-3)</td></tr>
<tr><td colspan="2">Sender Hardware Address (Bytes 4-5)</td><td colspan="2">Sender IP Address (Bytes 0-1)</td></tr>
<tr><td colspan="2">Sender IP Address (Bytes 2-3)</td><td colspan="2">Target Hardware Address (Bytes 0-1)</td></tr>
<tr><td colspan="4">Target Hardware Address (Bytes 2-5)</td></tr>
<tr><td colspan="4">Target IP Address (Bytes 0-3)</td></tr>
</table>

Fig. 1-13
Example of ARP Format when Used for IP-to-Ethernet Address Resolution

The length fields depend on the hardware and protocol address lengths, which are 6 bytes for an Ethernet address and 4 bytes for an IP address.

Hardware Type
Specifies a hardware interface type for which the sender seeks an answer. Contains a value of *1* for Ethernet.

Protocol Type
Specifies the type of high-level protocol address the sender has supplied. It contains *0800* for IP addresses.

Operation
Specifies an ARP Request (*1*), ARP Response (*2*), Reverse ARP (RARP) Request (*3*), or RARP Response (*4*).

Hardware Address Length
Specifies the length of the hardware address.

Protocol Address Length
Specifies the length of the high-level protocol address.

Sender Hardware Address
Specifies the sender's hardware address if known.

Sender IP Address
Specifies the sender's Internet Protocol address if known.

Target Hardware Address
When making a request specifies the sender's hardware address.

Target IP Address
When making a request specifies the sender's IP address.

Unfortunately, the variable length fields in ARP packets do not align neatly on 32-bit boundaries, making Figure 1-13 difficult to read. For example, the sender's hardware address, labeled *Sender Hardware Address*, occupies 6 contiguous bytes, so it spans two lines in the diagram. Field *Hardware Type* specifies a hardware interface type for which the sender seeks an answer; it contains the value *1* for Ethernet. Similarly, field *Protocol Type* specifies the type of high-level protocol address the sender has supplied; it contains *0800* for IP addresses. Field *Operation* specifies an ARP request or ARP response. Fields *Hardware Address Length* and *Protocol Address Length* allow ARP to be used with arbitrary networks because they specify the length of the hardware address and the length of the high-level protocol address. The sender supplies its hardware address and IP address, if known, in the fields *Sender Hardware Address* and *Sender IP Address*.

When making a request, the sender also supplies the target IP address or target hardware address using fields *Target Hardware Address* and *Target IP Address*. Before the target machine responds, it fills in the missing addresses, swaps the target and sender pairs, and changes the operation to a reply. Therefore a reply carries the IP and hardware addresses of the original requester, as well as the IP and hardware addresses of the machine for which a binding was sought.

Big Endian/Little Endian

To create an Internet that is independent of any particular vendor's machine architecture or network hardware, the software must define a standard representation for data. To illustrate that point, consider what happens when software on one computer sends a 32-bit binary integer to another computer. The physical transport hardware moves the sequence of bits from the first machine to the second without changing the order. However, not all machines store 32 and/or 16-bit integers in the same way. On some machines, referred to as *Little Endian*, the lowest memory address contains the low-order byte of the integer. On others, referred to as *Big Endian*, the lowest memory address holds the high-order byte of the integer. Thus, direct copying of bytes from one machine to another may change the value of the number.

Standardizing byte-order for integers is especially important in an Internet because Internet packets carry binary numbers that specify information like destination addresses and packet lengths. Both the senders and receivers must understand such quantities. The TCP/IP protocols solve the byte-order problem by defining a *network standard byte order* that all machines must use for binary fields in Internet packets. Each host converts numeric items from the host specific order to network standard byte order before sending a packet, and converts from network byte order to the host-specific order when a packet arrives. Naturally, the user data field in a packet is exempt from this standard – users are free to format their own data however they choose. Of course, most application programs use the *Big Endian* format as well. This is why we transfer packets using the Big Endian method.

The Internet standard for byte order specifies that the most significant byte is sent first (i.e. Big Endian style). If one considers the successive bytes in a packet as it travels from one machine to another, a binary integer in that packet has its most significant byte nearest the beginning of the packet and its least significant byte nearest the end of the packet. Motorola processors are typically *Big Endian* types, while Intel processors are typically *Little Endian* types. We transfer packets using the *Big Endian* format.

Short Integer	16 Bits
Long Integer	32 Bits

MSB			LSB
12	34	56	78

Fig. 1-14
Difference Between Big Endian & Little Endian

MSB = Most Significant Byte
LSB = Least Significant Byte

To illustrate the difference between *Big Endian* and *Little Endian*, look at Figure 1-15. If we store the number `0x12345678` at memory location 2000 it would look like this:

Memory Location	Big Endian	Little Endian
2000	12	78
2001	34	56
2002	56	34
2003	78	12

Fig. 1-15
Example of a Number Stored in a Big and Little Endian Processor

The Point to Point Protocol (PPP)

PPP or *Point-to-Point* protocol is designed to transport datagrams from multiple protocols over point-to-point links in a dynamically changing network. As a result, the design of PPP addresses has three areas of functionality:

Encapsulation How PPP nests within the stack of protocols that make up the entire communications environment in a network.

Link Control Protocol How PPP establishes, configures, and monitors the data-link connection.

Network Control Protocols How PPP interacts with a variety of network-layer protocols, including IP.

PPP requires that both sides negotiate a link for what they both will accept and how the link will operate. Characteristics such as the maximum size of the datagram that a given peer will accept, the authentication protocol (if any) that should be applied to datagrams originating from that sender, and compression schemes are all open to negotiations between the two systems being linked via PPP. This negotiation takes the form of a series of packet exchanges until both systems have agreed to the parameters under which the link will operate.

PPP is intended for use in simple links that transport datagrams between two peers. PPP supports full-duplex lines with simultaneous bi-directional traffic. Unlike some link-level protocols, PPP assumes that datagrams arrive in the order they were sent. Within this limitation, PPP offers an easy connection protocol between hosts, bridges, routers, and client computers. Previously SLIP was the preferred protocol for dial-up links. Now however, PPP is almost exclusively used for dial-up links because of its flexibility. In particular, the link-testing features of PPP enable more detailed transfer of graphics, binary files, and World Wide Web pages to and from PC's and the public Internet or private Intranets.

Link Control Protocol

When we use PPP the first thing we need to use is LCP or *Link Control Protocol*. LCP sends a packet, or a configuration request. A configuration request establishes what specifics it needs for this request. Both sides of a connection go through the same procedures. Sometimes, one side will wait for the other side; sometimes both machines will transmit this information simultaneously. The configuration request is simply a packet that specifies certain information about how it is going to utilize the link. If we were to look at a sample frame format it would look similar to this:

Code	Identifer	Total Length	Option	Length	Value	
1 Byte	1 Byte	2 Bytes	1 Byte	1 Byte	Var.	...

Fig. 1-16
Sample Configuration Request Format

This frame could be variable length depending on the number of options that are requested.

PPP Encapsulation

PPP allows the peers on a given link to establish the encapsulation to be used for datagrams. Frames transmitted via PPP have three fields as shown in Figure1-7.

Protocol	Information	Padding (optional)

Fig. 1-17
PPP Encapsulation

The fields in a PPP frame are used as follows:

Protocol Field
Establishes the network protocol that sent that datagram and with regard to which it should be interpreted.

Information Field
The packet received from the network-level protocol to be transmitted over the physical medium under the control of the PPP.

Padding
Establishes the network protocol that sent that datagram and with regard to which it should be interpreted.

The Protocol Field

By default, the protocol field is two bytes in length. But, it may optionally shorten to one byte if both peers agree. It is transmitted in Big Endian (most significant byte first).

Notice that in the protocol values in the following table, the first byte is always even and the second byte is always odd. The reason for this is field compression.

Protocol field values are defined in RFC 1700, "Assigned Numbers." The following values (given in hexadecimal) are of special interest when PPP is used along with TCP and IP:

0021	Internet Protocol
002d	Van Jacobson Compressed TCP/IP
002f	Van Jacobson Uncompressed TCP/IP
8021	Internet Protocol Control Protocol
c021	Link Control Protocol
c023	Password Authentication Protocol
c025	Link Quality Report
c223	Challenge Handshake Authentication Protocol

0000-02ff	Network Layer protocols
8000-bfff	Network Control Protocols
c000-fff	Link-Layer Control Protocols

Fig. 1-18
Protocol Field Values

The Information Field

The information field contains the packet sent down by the network level. As is usual in stacked protocols, PPP encapsulates the packet without interpreting it. Unless otherwise established by peer – to – peer negotiation, the default Maximum Receive Unit length for the information field is 1500 bytes including any padding but excluding the Protocol field.

The Padding Field

The padding field supports protocols and equipment that prefer (or require) that the overall packet length be extended to a 32-bit boundary or be otherwise fixed. Its use is not mandatory except as implied by configuration options negotiated between the peers in the link.

PPP Link Operation

Before user information can be sent across a point-to-point link, each of the two endpoint systems comprising the desired link must test the link and negotiate how the link will be configured and maintained.

These functions are performed using the Link Control Protocol. The PPP software on each peer (endpoint) system creates packets for this purpose, framed with the standard PPP protocol field. Once the link has been established, each peer authenticates the other if so requested. Finally, PPP must send Network Control Protocol packets to negotiate the network-layer protocol(s) that will be supported in this link.

Once the link has been established and both peers have agreed to support a given network-layer protocol on this link, datagrams from that network-layer protocol may be sent over the link.

The link will remain available for communications until it is explicitly closed. This can happen at the LCP or NCP level, either by administrator intervention or through a time-out interrupt. Specific network-layer protocols can be enabled and disabled on the link at any time without affecting the capability of the PPP link to support other network-layer protocol transmissions.

Understanding IP Addresses

This section describes the underlying concepts if IP addresses. We will discuss the format of an IP address, the concepts of a sub-netting and super-netting, and why we use them. Let's first look at basic IP address formatting.

IP Address Format

IP addresses are a standard 32 bits long. Figure 1-19 shows the format of an IP address:

208.229.201.0

Fig. 1-19
Sample IP Address Format

The class field can be defined by the first three numbers of the IP address. In our example, the number would be *208*. There are three types of classes. They are *Class A, Class B, and Class C*. Think of an Internet as a large network like any other physical network. The difference is that the Internet is a virtual structure that is implemented entirely in software. This allows the designers to choose packet formats and sizes, addresses, delivery techniques, and so on. Nothing is dictated by hardware. For addresses, designers of TCP/IP chose a scheme similar to physical network addressing in which each host on the Internet is assigned a 32-bit integer address, called its *IP Address*. The clever part of IP addressing is that the integers are carefully chosen to make routing efficient. An IP address encodes the identification of the network to which a host attaches as well as the identification of a unique host on that network.

We can follow some guidelines to break this up and analyze the format for following the rules of IP addressing:

Class	Network	Host

In the simplest case, each host attached to an Internet is assigned a 32-bit universal identifier as its Internet address. The bits of IP addresses for all hosts on a given network share a common prefix, as mentioned earlier.

Each address is a pair: both network and host on that network.
Given an IP address, its class can be determined from the three high-order bits, with two bits being sufficient to distinguish among the three primary classes. Class A addresses, which are used for a small amount of networks that have more than 65,536 hosts, devote 7 bits to network and 24 bits to the host. Class B addresses, which are used for intermediate size networks that have between 256 and 65,536 hosts, allocate 14 bits to the network and 16 bits to the host. Class C addresses, which are used for networks with less than 256 hosts, allocate 21 bits to the network and only 8 bits to the host. Note that the IP address has been defined in such a way that it is possible to extract the host or network portions quickly. Routers, which use the network portion of an address when deciding where to send a packet, depend on efficient extraction to achieve high speed.

Network and Broadcast Addresses

We have already said that the major advantage of encoding network information in Internet addresses is that it makes efficient routing possible. Another advantage is that Internet addresses can refer to networks as well as hosts. By convention, host *0* is never assigned to an individual host. Instead, an IP address with host *0* is used to refer to the network itself.

Another significant advantage of the Internet addressing scheme is that it includes a *broadcast address* that refers to all hosts on the network. According to the standard, any host containing all *1*'s is reserved for broadcast. On many network technologies, broadcasting can be as efficient as normal transmission; on others, broadcasting is supported by the network software, but requires substantially more delay than a single transmission. Some networks do not support broadcast at all. Therefore, having an IP address does not guarantee the availability or efficiency of broadcast delivery.

Limited Broadcast

Technically, the broadcast address we just described is called a *directed broadcast address* because it contains both a valid network ID and the broadcast host ID. A directed broadcast can be interpreted unambiguously at any point on the Internet because it uniquely identifies the target network in addition to specifying broadcasts on that network. Directed broadcast addresses provide a powerful (and somewhat dangerous) mechanism that allows a remote system to send a single packet that will be broadcast on the specified network.

From an addressing point of view, the chief disadvantage of directed broadcast is that it requires knowledge of the network address. A *limited broadcast address* provides a broadcast address for the local network independent of the assigned IP address. The local broadcast address consists of 32 *1's* (it is sometimes called the all *1's* broadcast address). A host may use the limited broadcast address as part of a start-up procedure before it learns its IP address or the IP address for the local network. Once the host learns the correct IP address for the local network, it should use the directed broadcast. As a general rule, TCP/IP protocols restrict broadcasting to the smallest possible set of machines.

Drawbacks in Internet Addressing

Encoding network information in an Internet address does have some disadvantages. The most obvious disadvantage is that addresses refer to network connections, not to the host computer. Another weakness of the Internet addressing scheme is that when any Class C network grows to more than 255 hosts, it must have its address changed to a Class B address.

Note: If a host computer moves from one network to another, its IPaddress must change.

Dotted Decimal Notation

When communicating to humans, in either technical documents or through application programs, IP addresses are written as four decimal integers separated by decimal points. Each integer gives the value of one byte of the IP address. For example:

10000000 00001010 00000010 00011110

is written:

128.10.2.30

When we express IP addresses, we will use dotted decimal notation. This table summarizes the range of values for each class of IP addresses:

Class	Lowest Address	Highest Address
A	1.0.0.1	126.255.255.254
B	128.1.0.1	191.255.255.254
C	192.0.1.1	223.255.255.254
D	224.0.0.0	239.255.255.255
E	240.0.0.0	247.255.255.255

Fig. 1-20
IP Address Denotations

Loopback Address

In Figure 1-20 we see that not all-possible addresses have been assigned to classes. For example, address 127.0.0.0, a value from the class A range, is reserved for *loopback*; and is intended for use in testing TCP/IP and for inter-process communication on the local machine. When any program uses the loopback address for the destination, the computer returns the data without sending the traffic across any network. A host or router should never propagate routing or reachability information for network number *127;* it is not a network address.

Special Address Conventions

In practice, IP uses only a few combinations of *0*'s or *1*'s.

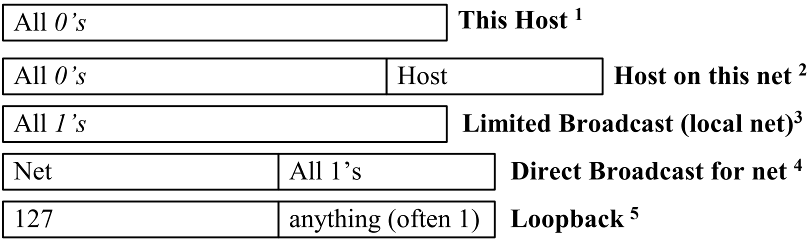


Fig. 1-21
Special Forms of IP Addresses.

As the notes in Figure 1-21 show, using all *0*'s for the network is only allowed during the bootstrap procedure. It allows a machine to communicate temporarily. Once the machine learns its correct network and IP address, it must not use network *0*.

[1] Allowed only at system startup and is never a valid destination address.
[2] Never a valid source address.
[3] Never a valid source address.
[4] Never a valid source address.
[5] Should never appear on a network.

IP Addressing Format

Let's look at the physical setup of an IP address. If we have an IP address number of:

208.229.201.66

or

11010000.11100101.11001001.01000010

This number represents the address of a particular machine on a network. This is that machines personal "identification" number. Each machine on that network will have the same first three numbers. For example, a machine with the address: 208.229.201.68, would still be on the same network and would still be a class C address. However, this would denote a different machine.

Note: When an IP address ends in 0, it denotes the physical network, or wire. For Example: 208.229.201.0

Netmasks

Netmasking allows the grouping together and breaking-up of IP address space. To illustrate this point lets look at the format of IP addressing in binary:

11010000	11100101	11001001	01000010

Fig. 1-22
Standard IP Address in Binary

110	10000	11100101	11001001	01000010

Fig. 1-23
Class Identifying Digits for an IP Address
(i.e. Class A, B, or C)

If we were to set up a netmask, we would essentially be creating an "overlay" for an IP address. It would look similar to this (in binary):

11111111	11111111	11111111	00000000
Network			Host

Fig. 1-24
Sample Netmask

11010000	11100101	11001001	01000010
11111111	11111111	11111111	00000000
Network			Host

Fig. 1-25
Netmasking Example

The all *1*'s side represents the network side. The all *0*'s side represents the host side. If you were to take this netmask and "overlay" it on the IP address format like Fig. 1.25, you can see that the default "line" between the network and the host falls between the third and fourth number.

Note: Every machine on a network wire must agree on the value of the netmask. The values must be identical for the netmask to operate correctly.

Reserved Addresses

In a netmask, there are guidelines that must be followed for correct setup with no errors. The most important thing to remember is this:

> *If the host or network portion of the IP Address (after applying the netmask) is all 1's or 0's, it cannot be assigned as a machine's IP address.*

Any combination of numbers that translate into binary as being all *1*'s or *0*'s also cannot be used. Depending on where the netmask is setup, the reserved numbers will change. For example:

192	0	0	X
128	0	X	X
127	X	X	X
0	X	X	X

Fig. 1-26
Reserved Addresses

In these examples the network portion of the IP address is *0*, except for 127.X.X.X which is reserved for loopback.

In the next section, we will see how we can "change" the position of the bits in the network section of the netmask to manipulate our network for our personal needs.

Sub-Netting & Super-Netting

Let's look at an example to explain the use of sub-netting and super-netting. Suppose that we have applied for an IP address, and we received a class C address. In our business, we have 20 machines in the shipping department, 20 machines in the engineering department, and 50 machines in the clerical department. We want all those machines to be able to access the network independently based on their departments. Instead of applying for 3 separate class C addresses, we can manipulate our current class C address to accommodate our needs. We know that we have 254 physical addresses available, but because we do not want them all on the same physical network, we need to find an alternative way to use our current address. What we need to do is essentially "break-up" our current address. We do this by what is referred to as *sub-netting.* We know that in the structure of an IP address anything that is a *1* (in binary), is referenced as the network, and anything that is a *0* is referenced as the host. We can restructure the netmask to move where the division is between the network portion and the host portion.

IP Address	208	229	201	0
Netmask	11111111	11111111	11111111	00000000
Hexadecimal	FF	FF	FF	00

IP Address	208	229	201	0
Netmask	11111111	11111111	11111111	11000000
Hexadecimal	FF	FF	FF	C0

Fig. 1-27
Illustration of Sub-Netting

In the first chart of Figure 1-27, we see the default setup for a class C netmask. We see the result of sub-netting in the second chart of Figure 1-27. We have essentially changed the location of the bar between the third and fourth integers of our IP address. We changed the location by changing the value of the netmask from:

FF FF FF 00 to FF FF FF C0

Notice that we have changed the first two bits of the last byte to represent the network address. In order to achieve *super-netting* the same procedure is followed with the exception that we move the bar between the third and fourth bytes to the left instead of to the right. It allows us to group several class X addresses together to be one network address.

The Internet Protocol (IP)

By concept, a TCP/IP Internet provides three sets of services as shown in Figure 1-28; by the way they are arranged in the figure, we can see that there seems to be a dependency among them.

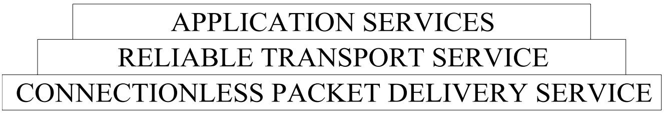

APPLICATION SERVICES
RELIABLE TRANSPORT SERVICE
CONNECTIONLESS PACKET DELIVERY SERVICE

Fig. 1-28
Three Conceptual Layers of Internet Services

At the lowest level, a connectionless delivery service provides a foundation on which everything rests. At the next level, a reliable transport service provides a higher-level platform on which applications depend. We will explore these services in more detail to see what each one provides and which protocols are associated with them.

Connectionless Packet Delivery Service

The most fundamental Internet service consists of a packet delivery system. Technically, the service is defined as an unreliable, best effort connectionless packet delivery system; similar to the service provided by network hardware that operates on a best-effort delivery pattern. The service is called unreliable because delivery is not guaranteed. The packet may be lost, duplicated, delayed, or delivered out of order. The service will not detect such conditions, nor will it inform the sender or receiver. The service is called *connectionless* because each packet is treated independently from all others. A sequence of packets sent from one computer to another may travel over different paths, or some may be lost while others are delivered. Finally, the service is referred to as *best-effort delivery* because the Internet software makes an earnest attempt to deliver packets. That is, the Internet does not discard packets arbitrarily. Unreliability arises only when resources are exhausted or underlying networks fail.

Purpose of the Internet Protocol

The protocol that defines the unreliable, connectionless delivery mechanism is called the *Internet Protocol* and is usually referred to by its initials *IP*. IP provides three important definitions. First, the IP protocol defines the basic unit of data transfer used throughout a TCP/IP Internet. Thus, it specifies the exact format of all data as it passes across a TCP/IP Internet. Second, IP software performs the *routing* function, choosing a path over which data will be sent. Third, in addition to the precise, formal specification of data formats and routing, IP includes a set of rules that embody the idea of unreliable packet delivery. The rules characterize how hosts and routers should process packets, how and when error messages should be generated, and the conditions under which packets can be discarded. IP is such a fundamental part of the design that a TCP/IP Internet is sometimes called an *IP-based technology.*

IP is designed for routing traffic between networks, or across a network of networks. An application running on a client machine generates messages or data to be sent to a machine located on another network. IP receives these messages from the transport layer software residing on a server that provides a gateway from the LAN (Local Area Network) or WAN (World Area Network) onto the Internet (or other TCP/IP network). Not all terminals on the network are end-user machines or gateways to LANs and WANs. Some terminals are devoted to routing packets along various potential pathways from the sending terminal to the receiving terminal.

The Internet Datagram

On a physical network, the unit of transfer is a frame that contains a header and data, where the header gives information such as the (physical) source and destination addresses. The Internet calls its basic transfer unit an *Internet datagram*, sometimes referred to as an *IP datagram* or merely a *datagram*. Like a typical physical network frame, a datagram is divided into header and data areas. Also like a frame, the datagram header contains the source and destination addresses and a field type that identifies the contents of the datagram. The difference is that the datagram header contains IP addresses whereas the frame header contains physical addresses.

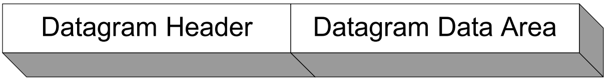


Fig. 1-29
General form of an IP datagram

IP specifies the header format including the source and destination IP addresses. IP does not specify the format of the data area; it can be used to transport arbitrary data.

Datagram Format

Now that we have described the general layout of an IP datagram, we can look at the contents in more detail. Figure 1-30 shows the arrangement fields in a datagram:

<table>
<tr><td colspan="2">0000 0000
0123 4567</td><td colspan="2">0011 1111
8901 2345</td><td colspan="2">1111 2222
6789 0123</td><td colspan="2">2222 2233
4567 8901</td></tr>
<tr><td>VER</td><td>HLEN</td><td colspan="2">TOS</td><td colspan="4">Total Length</td></tr>
<tr><td colspan="4">Identification</td><td>FLG</td><td colspan="3">Total Length</td></tr>
<tr><td colspan="2">TTL</td><td colspan="2">Protocol</td><td colspan="4">Header Checksum</td></tr>
<tr><td colspan="8">Source IP Address</td></tr>
<tr><td colspan="8">Destination IP Address</td></tr>
<tr><td colspan="8">IP Options (If Any) Followed by DATA</td></tr>
</table>

Fig. 1-30
Internet Datagram Format Basic Unit of Transfer in a TCP/IP Internet

Because datagram processing occurs in software, the contents and format are not restricted by any hardware. For instance, the first 4-bit field in a datagram *(VER)* contains the version of the IP protocol that was used to create the datagram. It is used to verify that the sender, receiver, and any routers in between them agree on the format of the datagram. All IP software is required to check the version field before processing a datagram to ensure it matches the format the software expects. If standards change, machines will reject datagrams with protocol versions that differ from theirs, preventing them from misinterpreting datagram contents according to an outdated format.

The header length field *(HLEN)*, also 4 bits, gives the datagram header length measured in 32-bit words. As we will see, all fields in the header have fixed length except for *IP OPTIONS* and corresponding *PADDING* fields. The most common header, which contains no options or padding, measures 20 bytes and has a header length field equal to *5*.

The *TOTAL LENGTH* field gives the length of the IP datagram measured in bytes, including bytes in the header and data. The size of the data area can be computed by subtracting the length of the header *(HLEN)* from the *TOTAL LENGTH*. Because the *TOTAL LENGTH* field is 16 bits long, the maximum possible size of an IP datagram is 65,535 bytes. It may become more important if future networks can carry data packets larger than 65,535 bytes.

Datagram Type of Service and Datagram Precedence

Informally called *Type of Service (TOS)*, the 8-bit *SERVICE TYPE* field specifies how the datagram should be handled and is broken down into five subfields.

Three *PRECEDENCE* bits specify datagram precedence, with values ranging from 0 (normal precedence) through 7 (network control), allowing senders to indicate the importance of each datagram. Although most host and router software ignores type of service, it is an important concept because it provides a mechanism that can allow control information to have precedence over data. For example, if all hosts and routers honor precedence, it is possible to implement congestion control algorithms that are not affected by the by the congestion they are trying to control. Bits *D, T,* and *R* specify the type of transport the datagram desires. When set, the *D* bit requests low delay, the *T* bit requests high throughput, and the *R* bit requests high reliability. Of course, it may not be possible for an Internet to guarantee the type of transport requested (i.e. it could be that no path to the destination has the requested property). Thus, we think of the transport request as a hint to the routing algorithms, not as a demand. If a router knows more than one possible route to a given destination, it can use the type of transport field to select one with characteristics closest to those desired. For example, suppose a router can select between a low capacity leased line and a high bandwidth (but high delay) satellite connection. Datagrams carrying a keystroke from a user to a remote computer could have the *D* bit set requesting that they be delivered as quickly as possible, while datagrams carrying a bulk file transfer could have the *T* bit set requesting that they travel across the high capacity satellite path.

Datagram Encapsulation

Before we can understand the next fields in the datagram, it is important to consider how datagrams relate to physical network frames. One question that you may have is "How large can a datagram be?" Unlike physical network frames that must be recognized by hardware, datagrams are handled by software. They can be any length the protocol designers choose. The current datagram format allows for 16 bits to the total length field, limiting the datagram to at most 65,535 bytes.

More fundamental limits on datagram size arise in practice. We know that as datagrams move from one machine to another, the underlying physical network must always transport them. To make internet transportation efficient, we would like to guarantee that each datagram travels in a distinct physical frame.

The idea of carrying one datagram in one network frame is called *encapsulation*. To the underlying network, a datagram is like any other message sent from one machine to another. The hardware does not recognize the datagram format, nor does it understand the IP destination address. Therefore, when one machine sends an IP datagram to another, the entire datagram travels in the data portion of the network frame.

Understanding Checksums

Introduction

A checksum allows information to follow a set of guidelines that is pertinent to the correct sending and receiving of data. Checksums are used in several different types of protocols. For example, UDP, TCP, and IP, all use variations of a checksum. A good way to understand the meaning and the use of the checksum is to analyze the format. Look at the chart below; we can see that the format of an IP header provides us with certain information.

<table>
<tr><td>VER</td><td>HLEN</td><td colspan="2">TOS</td><td colspan="4">Total Length</td></tr>
<tr><td colspan="4">Identification</td><td>FLG</td><td colspan="3">Total Length</td></tr>
<tr><td colspan="2">TTL</td><td colspan="2">Protocol</td><td colspan="4">Header Checksum</td></tr>
<tr><td colspan="8">Source IP Address</td></tr>
<tr><td colspan="8">Destination IP Address</td></tr>
<tr><td colspan="8">IP Options (If Any) Followed by DATA</td></tr>
</table>

For our example, we will plug in the following information for each of these fields:

VER (Veserion)	4
HLEN (Header Length)	5
TOS (Type of Service)	0
Total Length	14
Indentification	1
FLG (Flag)/Fragment Offset	0
TTL (Time to Live)	255
Protocol	17 (UDP)
Header Checksum	0
Source IP Address	1.2.3.4
Destination IP Address	1.2.3.5

Explanation of Checksums

If you notice, the value for the header checksum equals 0. To begin with, the value for the checksum is always set to zero and then simple addition is done.

Now if we take the values in hexadecimal and simply add them we get a result. That result is then plugged back into the original checksum computation and then sent. This is all done in the format of 16-bit words.

Computing
4500 +
000E +
0001 +
0000 +
FF11 +
0000 +
0102 +
0304 +
0102 +
0305
= 14C2D

Our result is larger than 16 bits so we must perform the carry function.

14C2D
+ 0001
= 4C2E

Now we must perform the *ones complement:*

~(4C2D)
= B3D1

This is the interesting part of what the checksum does. Once the value has been calculated, the system takes the *ones complement* of that result. The ones complement is a function that takes the value of the result in binary and essentially switches the value. If the binary unit has a value of 1, it becomes 0. And if the binary unit has a value of 0, it becomes 1. That value then becomes the value for the checksum field.

The receiver also uses the ones complement function when checking the information that was received. After the value is calculated and all the necessary carries are done, the system then takes the ones complement of the number. This number will always be equal to zero.
To illustrate this point:

Computing	
4500 +	
000E +	
0001 +	
0000 +	
FF11 +	
B3D1 +	← Checksum Field
0102 +	
0304 +	
0102 +	
0305	
= 1FFFE	

Our result is larger than 16 bits so we must perform the carry function.

1FFFE
+ 0001
= FFFF

Now we must perform the *ones complement:*

~(FFFF)
= 0000

When the checksum is configured, as we mentioned earlier, the value of the checksum is set to 0. Then after the addition is done, the result is then sent to the receiver in the checksum field. The receiver then calculates the data for itself. The result that the receiver gets should be equal to zero. If this is the case then the information will follow. After the values have been calculated, there may be an instance where the value is larger than a 16-bit word. As we mentioned earlier, the calculations are done in short word format. If the result is larger than a 16-bit word, the system performs a *carry* that allows the value to be shown as a 16-bit word. This function is can be done a maximum of 2 times.

The Internet Control Message Protocol (ICMP)

The previous section shows how the Internet Protocol software provides an unreliable connectionless datagram delivery service by arranging for each router to forward datagrams. A datagram travels from router to router until it reaches one that can deliver the datagram directly to its final destination. If a router cannot route or deliver a datagram, or if the router detects an unusual condition that affects its ability to forward the datagram, the router needs to inform the original source to take action to avoid or correct the problem. In this section we will discuss a mechanism that Internet routers and hosts use to communicate control or error information. We will see that routers use the mechanism to report problems, and the hosts use it to test whether destinations are reachable.

The Internet Control Message Protocol

In the connectionless system we have described so far, each router operates autonomously, routing or delivering datagrams that arrive without coordinating with the original sender. This system works well if all machines operate correctly all the time. Aside from communication lines or processors failing; there are a variety of conditions that will impede IP's ability to deliver datagrams: when the destination machine is temporarily or permanently disconnected from the network, when the time-to-live counter expires, or when intermediate routers become so congested that they cannot process the incoming traffic. The important difference between having a single network implemented with dedicated hardware and an internet implemented with software is that in the former, the designer can add special hardware to inform attached hosts when problems arise. In an internet, which has no such hardware mechanism, a sender cannot tell whether a delivery failure resulted from a local malfunction or a remote one. This can make debugging difficult. The IP protocol itself contains nothing to help the sender test connectivity or learn about such failures.

Designers added a special-purpose message mechanism to the TCP/IP protocols to allow routers in an internet to report errors or provide information about unexpected circumstances. This mechanism, known as the *Internet Control Message Protocol (ICMP)*, is considered a required part of IP and must be included in every IP implementation.

Like all other traffic, ICMP messages travel across the Internet in the data portion of IP datagrams. The ultimate destination of an ICMP message is not an application program or user on the destination machine, but the Internet Protocol software on that machine. That is, when an ICMP error message arrives, the ICMP software modules handle it. Of course, if ICMP determines that a particular higher-level protocol or application program has caused a problem, it will inform the appropriate module.

So in other words:

The Internet Control Message Protocol allows routers to send error or control messages to other routers or hosts; ICMP provides communication between the Internet Protocol software on one machine to the Internet Protocol software on another.

Initially designed to allow routers to report the cause of delivery errors to hosts, ICMP is not restricted to routers. Although guidelines restrict the use of some ICMP messages, an arbitrary machine can send an ICMP message to any other machine. Thus, a host can use ICMP to correspond with a router or another host. The chief advantage of allowing hosts to use ICMP is that it provides a single mechanism used for all control and information messages.

Error Reporting vs. Error Correction

Technically, ICMP is an *error reporting mechanism*. It provides a way for routers that encounter an error to report the error to the original source. Although the protocol specification outlines intended uses of ICMP and suggests possible actions to take in response to error reports, ICMP does not fully specify the action to be taken for each possible error. When a datagram causes an error, ICMP can only report the error condition back to the original source of the datagram; the source must relate the error to an individual application program or take other action to correct the problem.

Most errors stem from the original source, but some do not. Because ICMP reports problems to the original source, it cannot be used to inform intermediate routers about problems. Unfortunately, the original source has no responsibility for the problem or control over a misbehaving router. In fact, the source may not be able to determine which router caused the problem.

You might ask, “Why restrict ICMP to communication with the original source?” Well, to answer that question, we know that a datagram only contains fields that specify the original source and the ultimate destination. It does not contain a complete record of its trip through the Internet (except for unusual cases where the record route option is used). Furthermore, because routers can establish and change their own routing tables, there is no global knowledge of routes. Thus, when a datagram reaches a given router, it is impossible to know the path it has taken to get there. If the router detects a problem, it cannot know the set of intermediate machines that processed the datagram, so it cannot inform them of the problem. Instead of silently discarding the datagram, the router uses ICMP to inform the original source that a problem has occurred, and trusts that host administrators will cooperate with network administrators to locate and repair the problem.

ICMP Message Delivery

ICMP messages require two levels of encapsulation as Figure 1-31 shows. Each ICMP message travels across the Internet in the data portion of an IP datagram, which itself travels across each physical network in the data portion of a frame. Datagrams carrying ICMP messages are routed exactly like datagrams carrying information for users; there is no additional reliability or priority. Thus, error messages themselves may be lost or discarded. Furthermore, in an already congested network, the error messages may cause additional congestion. An exception is made to the error handling procedures if an IP datagram carrying an ICMP message causes an error. The exception, established to avoid the problem of having error messages about error messages, specifies that ICMP messages are not generated from errors that result from datagrams carrying ICMP error messages.

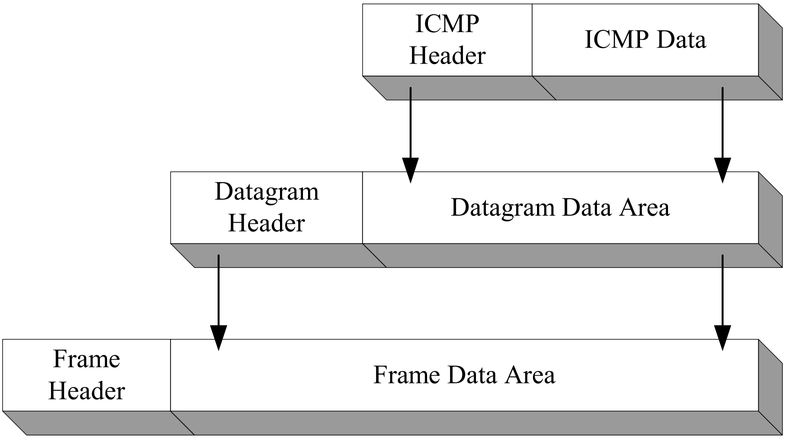


Fig. 1-31
Two Levels of ICMP Encapsulation

The ICMP message is encapsulated in an IP datagram, and then is encapsulated into a frame datagram. To identify the ICMP, the datagram protocol field contains the value *1*.

It is important to keep in mind that even though ICMP messages are encapsulated and sent using IP datagrams, it is not considered a higher-level protocol – it is a required part of IP. The reason for using IP to deliver ICMP messages is that they may need to travel across several physical networks to reach their final destination. Thus, they cannot be delivered by the physical transport alone.

ICMP Message Format

Although each ICMP message has its own format, they all begin with the same three fields. These are an 8-bit integer message *TYPE* field that identifies the message, an 8-bit *CODE* field that provides further information about the message type, and a 16-bit *CHECKSUM* field (ICMP uses the same additive checksum algorithm as IP but the ICMP checksum only covers the ICMP message). In addition, ICMP messages that report errors always include the header and the first 64 data bits of the datagram causing the problem.

The reason for returning more than the datagram header alone is to allow the receiver to determine which protocol(s) and which application program were responsible for the datagram. Higher-level protocols in the TCP/IP suite are designed so that crucial information is encoded in the first 64 bits.

The *TYPE* field defines the meaning of the message as well as its format. The types include:

Type Field	ICMP Message Type
0	Echo Reply
3	Destination Unreachable
4	Source Quench
5	Redirect (change a route)
8	Echo Request
11	Time Exceeded for a Datagram
12	Parameter Problem on a Datagram
13	Timestamp Request
14	Timestamp Reply
15	Information Request (obsolete)
16	Information Reply (obsolete)
17	Address Mask Request
18	Address Mask Reply

Fig. 1-32
ICMP Message Types

Testing Destination Reachability and Status (Ping)

TCP/IP protocols provide facilities to help network managers or users to identify network problems. One of the most frequently used debugging tools invoke the ICMP *echo request* and *echo reply* messages. A host or router sends an ICMP echo request message to a specified destination. Any machine that receives an echo request formulates an echo reply and returns it to the original sender. The request sends an optional data area; the reply contains a copy of the data sent in the request. The echo request and associated reply can be used to ensure that a destination is reachable and responding. Because both the request and reply travel in IP datagrams, successful receipt of a reply verifies that major pieces of the transport system work. First, IP software on the source computer must route the datagram. Second, intermediate routers between the source and destination must be operating and must route the datagram correctly. Third, the destination machine must be running (it must at least respond to interrupts), and both ICMP and IP software must be working. Finally, all routers along the return path must have correct routes.

On many systems, the command users invoke to send ICMP echo requests is referred to as *ping*. A sophisticated version of ping can send a series of ICMP echo requests, capture responses, and provide statistics about datagram loss. It allows the user to specify the length of the data being sent and the interval between requests. Less sophisticated versions merely send one ICMP echo request and await a reply. Figure 1-33 shows the format of an echo request and reply messages:

<table>
<tr><th>0</th><th>8</th><th>16 31</th></tr>
<tr><td>TYPE (8 or 0)</td><td>CODE (0)</td><td>CHECKSUM</td></tr>
<tr><td colspan="2">IDENTIFIER</td><td>SEQUENCE NUMBER</td></tr>
<tr><td colspan="3">OPTIONAL DATA</td></tr>
<tr><td colspan="3">...</td></tr>
</table>

Fig. 1-33
Format of an Echo Request and Reply

The field *OPTIONAL DATA* is a variable length field that contains data to return to the sender. An echo reply always returns the same data that was received in the request. The sender, to match replies to the requests, uses fields SEQUENCE NUMBER and IDENTIFIER. The value of the *TYPE* field specifies whether the message is a request *(8)* or a reply *(0)*.

Summary

Normal communication across an internet involves sending messages from an application on one host to an application on another host. Routers may need to communicate directly with the network software on a particular host to report abnormal conditions or to send the host new routing information. The Internet Control Message Protocol provides for the extra-normal communication among routers and hosts; it is an integral, required part of IP.

The User Datagram Protocol (UDP)

The *User Datagram Protocol* or UDP is the primary mechanism that application programs use to send datagrams to other application programs. UDP messages contain both destination port numbers and source port numbers in addition to the data being sent. This makes it possible for the UDP software at the destination to deliver information to the correct recipient and for the recipient to send a reply.

UDP utilizes the underlying IP to transport information from one machine to another. By using the IP format for delivery, UDP inherits the same characteristics of an IP message. It is an unreliable, connectionless datagram service. For example, it does not use acknowledgments to makes sure messages have arrived. It does not order incoming messages, and it does not provide feedback to control the rate at which information passes between machines. From this we can gather that UDP messages can be lost, duplicated, or arrive out of order. We also know that packets can be sent faster than the recipient can process them. To reiterate:

The User Datagram Protocol (UDP) provides an unreliable, connectionless delivery service using IP to transport messages between machines. It uses IP to carry messages, but utilizes the distinction among multiple destinations.

An application program that relies on UDP accepts full responsibility for the problems of reliability, including message loss, duplication, delay, and even out-of-order delivery. However, we can look at certain applications that do not need to utilize these strict options. For example, if a connection is made over the Internet that needs to transmit voice and video applications, we can see that speed is more important than reliability. Because packets are transmitted at such a high rate of speed, we are willing to sacrifice the reliability for rate of transfer.

If we were using a reliable method for voice and video, we would have to wait for two machines to exchange information to ensure that the information will not be lost, sacrificing time. However, if we were to make the same connection using UDP there would be a steady steam of packets entering the application, because there is no use of acknowledgements. If two packets were to get discarded or sent out-of-order, or even lost, the result that we might see would be very minimal. Again the important thing to decide is which is more important speed, or reliability.

UDP Message Format

Each UDP message is referred to as a *user datagram*. A user datagram consists of two parts: a UDP header and a UDP data area. As we can see in Figure 1-34, a UDP header is divided into four parts: a 16-bit field that specifies the port from which it originated, the port that it is going to, the size of the message (or message length), and the UDP checksum.

0	16 31
UDP SOURCE PORT	UDP DESTINATION PORT
UDP MESSAGE LENGTH	UDP CHECKSUM
DATA	
...	

Fig. 1-34
Format of Fields in a UDP Datagram

The *SOURCE PORT* and *DESTINATION PORT* fields contain 16-bit UDP protocol numbers, which are used as application identifiers among the processes waiting to receive them. The *SOURCE PORT* is optional. However, when it is used, it specifies the port to which replies should be sent. If the field is not used, it should be set to zero.

The *LENGTH* field contains a count of bytes in the UDP datagram, including the header and the user data. The minimum number that can be in this field is eight, the length of the header by itself.

As we mentioned earlier the *CHECKSUM* field is optional. A value of zero in the checksum field indicates that the checksum has not been computed.

UDP Pseudo-Header

To compute a checksum, UDP attaches a *pseudo-header* to the UDP datagram, and pads the datagram with one byte of zeros and then computes the checksum over the entire object. It is important to remember that the padding of zeros and the pseudo-header are ***not*** transmitted with the UDP datagram, nor are they included in the length.

The purpose of using a pseudo-header is to verify that the UDP datagram has reached the appropriate destination. The pseudo header is not transmitted to the recipient. The receiver takes the information contained in the IP header and places it into the pseudo-header format for the purpose of computing the checksum.

<table>
<tr><td colspan="3">SOURCE IP ADDRESS</td></tr>
<tr><td colspan="3">DESTINATION IP ADDRESS</td></tr>
<tr><td>ZERO</td><td>PROTOCOL</td><td>UDP LENGTH</td></tr>
</table>

Fig. 1-35
Pseudo Header Format Used to Compute a Checksum.

UDP Encapsulation

The UDP encapsulation format is one that works in the transport protocol layer. UDP lies in the layer above the Internet Protocol Layer. By concept, application programs access UDP, which use IP to send and receive datagrams. We can see an illustration of this in Figure 1-36.

Conceptual Layering
Application
User Datagram (UDP)
Internet (IP)
Network Interface

Fig. 1-36
Conceptual Layering of UDP Between Application Programs and IP

By layering UDP above IP, we can encapsulate a complete UDP message, including the UDP header and data, into an IP datagram.

As we learned earlier, encapsulation means that the protocol, UDP in this case, basically "inserts" the information into a frame, and then sends it across the network. In the example of UDP, it takes the information and attaches its own header to it and sends it to the network.

The Transport Control Protocol (TCP)

The first thing we need to remember is that TCP is a connection-oriented protocol. This means that when the TCP protocol is used, both sides need send each other some type of recognition that they have established a connection. This is different from the earlier mentioned section describing such protocols as UDP, which is a *connectionless* oriented protocol, where information is sent out without the need for acknowledgement. Commonly, TCP is known to be a very reliable protocol because of this fact.

Reliable Stream Delivery

Why do we need reliable stream delivery? To answer this question at the lowest level, computer communication networks provide unreliable packet delivery. As we have mentioned in previous sections, packets can be lost or destroyed when transmission errors interfere with data, when network hardware fails, or when networks become too heavily loaded. Networks that deliver packets dynamically, such as UDP, can deliver them out of order, deliver them after a substantial delay, or even deliver duplicates. Also, underlying network technologies may dictate optimal packet size or invoke other constraints needed to achieve efficient transfer rates.

At the highest level, application programs may need to send large volumes of data from one computer to another. Using a connectionless delivery system for large volume transfers can be tedious, and also requires programmers to incorporate error detection and recovery into each application program. All these problems and inconveniences have caused a necessity to find a general-purpose solution to providing reliable stream delivery. This will make it possible for experts to build a single instance of stream protocol software that *all* application programs use. The advantage of this is to help isolate application programs from the details of networking, and it also makes it possible to define a uniform interface for stream transfer service.

Reliability

Reliable stream delivery service guarantees to deliver a stream of data from one machine to another without duplication or loss. This statement brings up an interesting question, "How can protocol software provide reliable transfer if the underlying communication system offers only unreliable packet delivery?"

Most reliable protocols use a single fundamental technique known as *positive acknowledgement with retransmission*. This technique requires a recipient to communicate with the source by sending an *acknowledgement* (ACK) message as it receives data. The sender keeps the report of each packet it sends and then waits for an acknowledgement before sending the next packet. The sender also starts a timer when it sends a packet and retransmits a packet if the timer expires before a packet arrives. The following diagram shows how the simplest positive acknowledgement protocol transfers data.

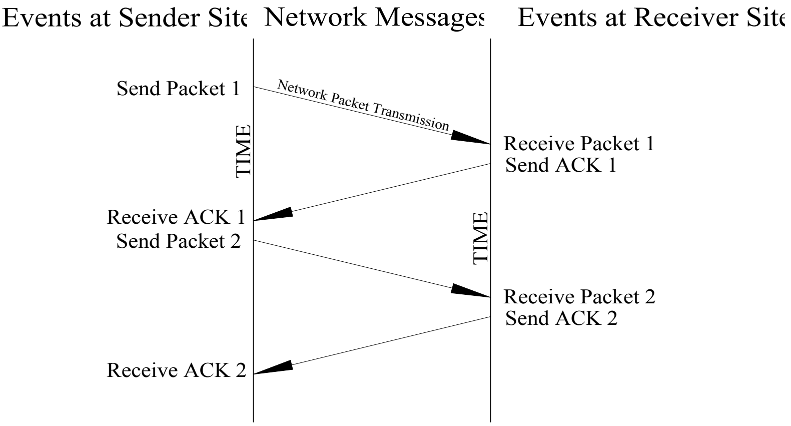


Fig. 1-37
Simple Positive Acknowledgement with Retransmission

The final reliability problem arises when an underlying packet delivery system duplicates a packet. Duplicates can also arise when networks experience high delays that cause premature retransmission. Solving duplication problems requires careful consideration because both packets *and* acknowledgements can be duplicated. Usually, reliable protocols can detect duplicate packets by assigning each packet a sequence number and requiring the receiver to remember which sequence numbers it has received. We will explore this further in an upcoming section. To avoid confusion caused by delayed or duplicate acknowledgements, positive acknowledgement protocols send sequence numbers back in acknowledgements. Therefore, the receiver can correctly associate acknowledgements with packets.

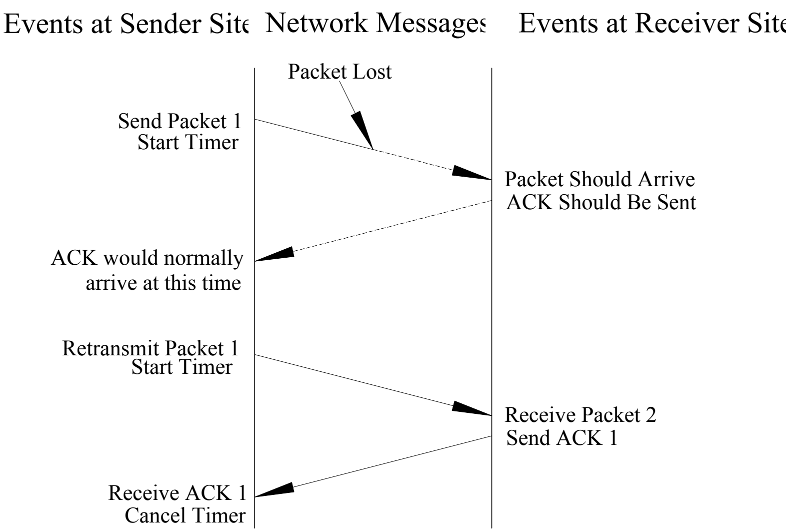


Fig. 1-38
Packet Loss and Retransmission

Sliding Windows

As we continue in the explanation of the TCP protocol suite we need to examine the concept of a *sliding window.* A sliding window enables stream transmission to be efficient. To understand this concept, we need to keep in mind Figure 1-37. It shows how a sender transmits a packet and waits for an acknowledgement from the receiver before transmitting another. Also, we can gather from this illustration that data only flows from one machine to another in one direction at any given time, even if the network is capable of simultaneous communications in both directions. The network will be completely idle during times the machines delay responses, for example, when computing a checksum.

When dealing with a network with high transmission delays, we must remember that a simple positive acknowledgement protocol wastes a substantial amount of network bandwidth. This is because it must wait to send another packet until it receives an acknowledgement for the original packet that was sent. The solution to this problem is the Sliding Window Concept.

Sliding window protocols use network bandwidth more efficiently because they allow the user to transmit multiple packets before waiting to receive an acknowledgement. To illustrate this point, look at Figure 1-39. If we think of a sequence of packets to be transmitted, the protocol places a small fixed-size *window* on the sequence and transmits all packets that lie inside of that window.

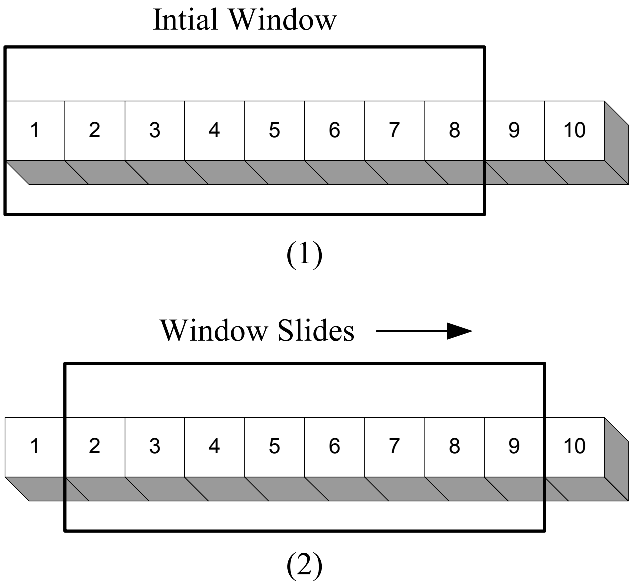


Fig. 1-39
Sliding Window Protocol

The first half of this illustration shows a *sliding window* protocol with an eight-packet window. The second part of the illustration shows the window *sliding,* so that packet 9 can be sent when an acknowledgement has been received for packet 1. Remember: Only *unacknowledged* packets are retransmitted.

When we say a packet is *unacknowledged,* it means the packet has been transmitted, but no acknowledgement has been received. Technically, the number of packets that can be unacknowledged at any given time is constrained to the window size and is limited to a small fixed number. For example, as in our illustration, a window that has the size of eight, the sender has permission to transmit 8 packets before it receives an acknowledgement. Once the sender has received an acknowledgement for the first packet, the window essentially “slides”, and sends the next packet. The window continues to slide as long as acknowledgements are received.

The performance of sliding window protocols depends on the window size and the speed at which the network accepts packets. In our next illustration, you can see an operation of a sliding window protocol when sending three packets. Note that the sender transmits all the packets before receiving any acknowledgments.

With a window size of *1*, a sliding window protocol is the same as our simple positive acknowledgement protocol. By increasing our window size, it is possible to eliminate network idle time completely. Simply put, the sender can transmit packets as fast as the network can transfer them. We need to remember that a finely tuned sliding window protocol keeps the network completely saturated with packets

The timer is an important part of this protocol. By design, the sliding window protocol remembers which packets have been acknowledged and keeps a separate timer for each unacknowledged packet. If a packet is lost, the timer expires and the sender retransmits that packet. When a sliding window slides, it moves past all acknowledged packets. On the receiving end, the protocol software keeps its own window, accepting and acknowledging packets as they arrive. Thus, the window breaks the packets up into three sets: packets to the left of the window that have been successfully transmitted, received and acknowledged, packets to the right of the window that have not yet been sent, and packets that lie in the window that are being transmitted.

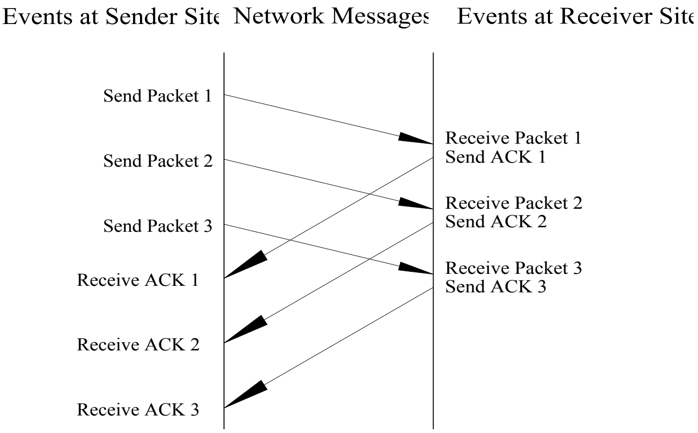


Fig. 1-40
Three Packets Transmitted Using a Sliding Window Protocol

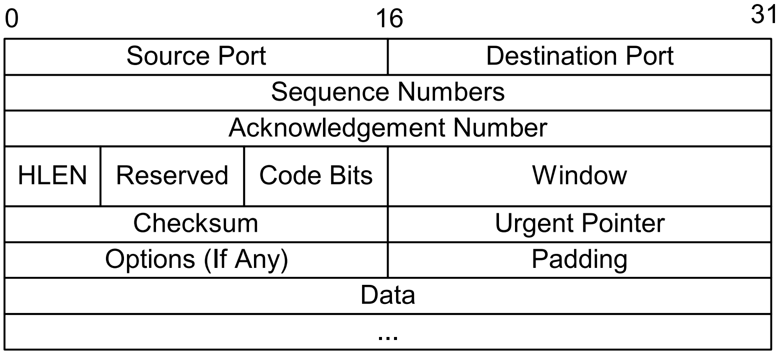
Transmission Control Protocol

Now that we have discussed and understand the sliding window principle, we can look at the reliable stream service provided by the TCP/IP Internet protocol suite. The service is defined by the *Transmission Control Protocol* or *TCP*. This reliable stream service is so important that the entire protocol suite is referred to as TCP/IP. It is important to remember that TCP is not a piece of software, it is a communication protocol. People seem to encounter TCP software much more than they do the communication protocol, so it is natural that to think of a particular implementation as the standard

TCP is both a connection-oriented protocol, and a byte stream oriented protocol. What this means is that TCP transmits a set of bytes at a time. These are called TCP segments. Until now, we have discussed only packet-oriented delivery of information. For example, if we were to make a call to the stack with two 100-byte pieces of information, using TCP, it would send the information as a 200-byte piece. Unlike other protocols, it does not break the information into packets, and then send it. There is no packet delineation. By using this method, data transfer can be likened to a modem connection. The method is essentially the same but the commands we use are different. We use the *connect* function from the sender. The receiver uses the *listen* function to receive incoming data.

TCP Header

We say that TCP is very reliable. The way TCP achieves reliability is in the format of the header. As we mentioned earlier, the unit of transfer between two machines using TCP software is called a *segment*. Segments are exchanged to establish connections, to transfer data, to send acknowledgements, to advertise window sizes, and to close connections. Let's look at the format of a TCP segment including the TCP header:



<table>
<tr><th>0</th><th></th><th></th><th>16</th><th>31</th></tr>
<tr><td colspan="3">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="5">Sequence Numbers</td></tr>
<tr><td colspan="5">Acknowledgement Number</td></tr>
<tr><td>HLEN</td><td>Reserved</td><td>Code Bits</td><td colspan="2">Window</td></tr>
<tr><td colspan="3">Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="3">Options (If Any)</td><td colspan="2">Padding</td></tr>
<tr><td colspan="5">Data</td></tr>
<tr><td colspan="5">...</td></tr>
</table>

Fig. 1-41
Format of a TCP Segment Including the TCP Header

Each segment is divided into two parts, a header followed by data. The header, also called the TCP header, carries the expected identification and control information. The Fields, *SOURCE PORT* and *DESTINATION PORT* contain the TCP port numbers that identify application programs at the ends of the connection. The *SEQUENCE NUMBER* field identifies the sender's byte stream of data in the segment. The *ACKNOWLEDGEMENT NUMBER* specifies the number of bytes the source expects to receive next. The sequence number refers to the stream flowing in the same direction as the segment while the acknowledgement field refers to the stream flowing in the opposite direction. The following chart shows examples of currently assigned port numbers:

Decimal	Keyword	Description
7	ECHO	Echo
9	DISCARD	Discard
21	FTP	File Transfer Protocol
23	TELNET	Terminal Connection

Fig. 1-42
Example of Well Known Ports

The *HLEN* field contains an integer value that specifies the length of the segment header in 32-bit multiples. It is needed because the *OPTIONS* field varies in length, depending on which options have been included. The 6-bit field marked as *RESERVED* is reserved for future use.

Some segments carry only acknowledgements while others carry data. Others carry requests to establish or close a connection. TCP software uses the field *CODE BITS* to determine the purpose and contents of the segment. These six bits tell how to interpret other fields in the header according to the table in Figure 1-43.

Bit (Left to Right)	Meaning if Bit is set to 1
URG	Urgent pointer field is valid
ACK	Acknowledgement field is valid
PSH	This segment requests a push
SYN	Reset the connection
FIN	Sender has reached the end of its byte stream

Fig. 1-43
Components of the Code Bits Header Field

TCP software advertises how much data it is willing to accept every time it sends a segment by specifying its buffer size in the *WINDOW* field. This field contains a 16-bit integer in network standard byte order. Window advertisements accompany all segments, including those carrying data, as well as those carrying only an acknowledgement.

TCP/IP and Client/Server Relationships

The TCP/IP protocol stack is designed around the concept of client machines that receive service from other machines on a network. For example, a personal computer that accesses the Internet through a LAN (Local Area Network) gateway relies on that gateway server to "talk TCP/IP" across the Internet on its behalf.

With that in mind, we can analyze the TCP/IP stack and find that each successively higher layer is a client to the layer beneath it.

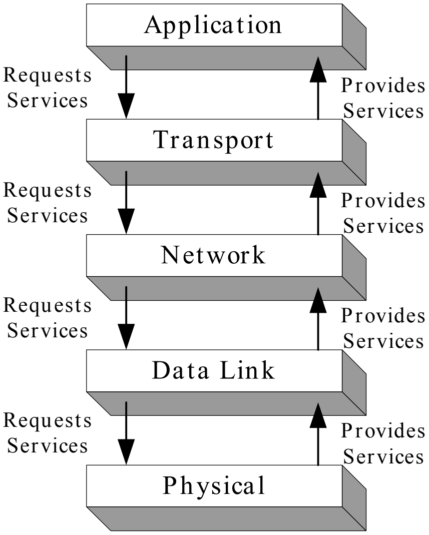


Fig. 1-44
Client/Server Relationships in TCP/IP

IP is a client of the data link layer software. It uses that software's services to achieve its physical transmission of packets. UDP and TCP are clients of IP. They use they use the IP routing mechanisms to move messages across a switched network.

Summary

Anything that depends on the transmission of data from one terminal to another can efficiently use TCP/IP to get it there reliably. When ARPANET was established, the need for reliable transmission was a must. By today's standards, TCP/IP has become the leading means of getting information from one place to another.

Introduction to BSD Sockets

Intro to BSD Sockets

The Berkeley Sockets 4.4 API (Applications Programmer Interface) is a set of standard function calls made available at the application level. These functions allow programmers to include Internet communications capabilities in their products.

The Berkeley Sockets API (also frequently referred to as simply 'sockets') was originally released with 4.2BSD in 1983. Enhancements have continued through the 4.4BSD systems. Berkeley-based code can be found in many different operating systems, both commercial and public domain, such as BSD/OS, FreeBSD, NetBSD, OpenBSD, and UnixWare 2.x. Other popular operating systems such as Solaris and Linux employ the standard sockets interface, though the code was written from scratch.

Other sockets APIs exist, though Berkeley Sockets is generally regarded as the standard. Two of the most common APIs are Winsock and TLI. Winsock (Windows Sockets) was developed for the Microsoft Windows platform in 1993, and is based significantly on the BSD interface. A large subset of the BSD API is provided, with most of the exceptions being platform-specific to BSD systems. TLI (Transport Layer Interface) was developed by AT&T, and has the capability to access TCP/IP and IPX/SPX transport layers. XTI (X/Open Transport Interface, developed by X/Open Company Ltd.) is an extension of TLI that allows access to both TCP/IP and NetBios.

Overview of How Sockets Works

BSD Sockets generally relies upon client/server architecture. For TCP communications, one host listens for incoming connection requests. When a request arrives, the server host will accept it, at which point data can be transferred between the hosts. UDP is also allowed to establish a connection, though it is not required. Data can simply be sent to or received from a host.

The Sockets API makes use of two mechanisms to deliver data to the application level: **ports** and **sockets**. Ports and sockets are one of the most misunderstood concepts in sockets programming.

All TCP/IP stacks have 65,536 ports for both TCP and UDP. There is a full compliment of ports for UDP (numbered 0-65535) and another full compliment, with the same numbering scheme, for TCP. The two sets do not overlap. Thus, communication over both TCP and UDP can take place on port 15 (for example) at the same time.

A port is not a physical interface – it is a concept that simplifies the concept of Internet communications for humans. Upon receiving a packet, the protocol stack directs it to the specific port. If there is no application listening on that port, the packet is discarded and an error may be returned to the sender. However, applications can create sockets, which allow them to attach to a port. Once an application has created a socket and bound it to a port, data destined to that port will be delivered to the application. This is why the term socket is used – it is the connection mechanism between the outside world (the ports) and the application.

A common misunderstanding is that sockets-based systems can only communicate with other sockets-based systems. This is not true. TCP/IP or UDP/IP communications are handled at the port level – the underlying protocols do not care what mechanisms exist above the port. Any Internet host can communicate with any other, be it Berkeley Sockets, WinSock, or anything else. Sockets is just an API that allows the programmer to access Internet functionality – it does not modify the manner in which communications occur.

Let us use the example of an office building to illustrate how sockets and ports relate to each other. The building itself is analogous to an Internet host. Each office represents a port, the receptionist is a socket, and the business itself is an application.

Suppose you are a visitor to this building, looking for a particular business. You wander in, and get directed to the appropriate office. You enter the office, and speak with the receptionist, who then relays your message to the business itself. If there is nobody in the office, you leave.

To rephrase the above in sockets terminology: A packet is transmitted to a host. It eventually gets to the correct port, at which point the socket conveys the packet's data to the application. If there is no socket at the destination port, the packet is discarded.

Byte-Ordering Functions

Because TCP/IP has to be a universal standard, allowing communications between any platforms, it is necessary to have a method of arranging information so that big-endian and little-endian machines can understand each other. Thus, there are functions that take the data you give them and return them in network byte-order. On platforms where data is already correctly ordered, the functions do nothing and are frequently macro'd into empty statements. Byte-ordering functions should always be used as they do not impact performance on systems that are already correctly ordered and they promote code portability.

The four byte-ordering functions are *htons*, *htonl*, *ntohs*, and *ntohl*. These stand for host to network short, host to network long, network to host short, and network to host long, respectively.

htons translates a short integer from host byte-order to network byte-order. *htonl* is similar, translating a long integer. The other two functions do the reverse, translating from network byte-order to host byte-order.

Data Structures

Before venturing into the realm of actual API functions, one must understand a few structures. The most important of these is **sockaddr_in**. It is defined as follows:

```
struct sockaddr_in
{
    short          sin_family;
    u_short        sin_port;
    struct in_addr sin_addr;
    char           sin_zero[8];
};
```

The structure **in_addr** that is used in **sockaddr_in** is defined as:

```
struct in_addr
{
    u_long s_addr;
};
```

These are the most important data structures used in sockets. The second consists of an unsigned long integer that contains the IP address that will be associated with the socket. The first has two other important fields – sin_family and sin_port. sin_family tells sockets which protocol family to use. For IPv4, the constant AF_INET should always be passed in. sin_port tells what port number will be associated with the socket.

sockaddr_in is a modification of the standard **sockaddr** structure:

```
struct sockaddr
{
    u_char sa_len;
    u_char sa_family ;
    char   sa_data[14] ;
};
```

Socket calls expect the standard **sockaddr** structure. However, for IPv4 communications, it is proper to pass in a **sockaddr_in** structure that has been cast to a **sockaddr**.

Common Sockets Calls

This section lists the most commonly used socket calls and describes their uses. This is purely introductory material. For a more complete description of how the calls work, please see the Programmer's Reference section of this manual.

socket

A socket, in the simplest sense, is a data structure used by the Sockets API. When the user calls this function, it creates a socket and returns reference a number for that socket. That reference number, in turn, must be used in future calls.

bind

This call allows a user to associate a socket with a particular local port and IP address. In the case of a server (see **listen** and **accept** below), it allows the user to specify which port and IP address incoming connections must be addressed to. For outgoing connection requests (see **connect** below), it allows the user to specify which port the connection will come from when viewed by the other host. Note: **bind** is unnecessary for sockets that are not going to be set up to accept incoming connections. In this case, the stack will pick an appropriate IP address and a random port (known as an ethereal port).

listen

This function prepares the given socket to accept incoming TCP requests. It must be called before **accept**.

accept

This function detects incoming connection requests on the listening socket.
In blocking mode, this call will cause a task to sleep until a connection request is received. In non-blocking mode, this call will return TM_EWOULDBLOCK indicating that no connection request is present and that **accept** must be called again. If the user calls **accept** and a connection request is pending, **accept** creates another socket based on the properties of the listening socket. If the call is successful, the socket descriptor of the newly created and connected socket is returned. The new socket is created to allow communications with multiple clients from a single port on the server (think of web servers, which listen on port 80 by default and are capable of communicating with thousands of hosts at the same time). Each time the user calls **accept** and there is a connection requests pending, it creates a new socket.

connect

When a user issues a **connect** command, the stack creates a connection with another host. Before connect can instruct the stack to establish a connection, the user must pass a socket and a **sockaddr_in** structure containing the destination IP address and port. In TCP, an actual connection is negotiated. In UDP, however, no packets are exchanged.

send

This call allows a user to send data over a connected socket. Unlike **sendto**, this socket *must* be connected. Because the socket is already connected, it is not necessary to specify the destination address (the destination address was set in **accept** or **connect**). **send** can be used for either UDP or TCP data.

sendto

Unlike **send**, **sendto** requires users to specify the destination port and address. This is useful for UDP communications only, as TCP requires a pre-existing connection. **sendto** may be used on either connected or unconnected UDP sockets. In the case that a UDP socket is already connected, the destination address provided to **sendto** will override the default established on the socket with **connect**.

recv

This function allows the user to receive data on the connected socket. **recv** can be used for either TCP or UDP.

recvfrom

This function allows the user to receive data from a specified UDP socket (whether or not it is connected). It may not be used for TCP sockets, as they require a connection.

close

This function closes (read: deletes) a socket that has been allocated with the **socket** call. If the socket is connected, it closes the connection before deleting it. Because the **close** call is frequently used for more than one purpose (closing open files, for example), it is renamed **tfClose** in the Turbo Treck stack to avoid conflicts with the preexisting function.

Example Code

The following are simplified examples of using the Sockets API to create Internet connectivity in an application. They are all available on the Turbo Treck protocols CD, in the examples\ directory. Four examples are given: UDP Client, UDP Server, TCP Client, and TCP Server. All of the examples are coded in blocking-mode.

UDP Client

This first example shows how to code a UDP client. A socket is created, and **sendto** is called the specified number of times. Note that **bind** is never called. For outgoing connections, bind is not necessary, as the stack will pick a random port and an appropriate IP address.

```
#include <trsocket.h>

#define TM_BUF_SIZE         1400
#define TM_PACKETS_TO_SEND  10
#define TM_DEST_ADDR        "10.0.0.1"
#define TM_DEST_PORT        9999

char   testBuffer[TM_BUF_SIZE];
char * errorStr;

int udpClient(void)
{
    int                testSocket;
    unsigned int       counter;
    struct sockaddr_in destAddr;
    int                errorCode;
    int                returnVal;

    counter = 0;
    returnVal = 0;

/* Specify the address family */
    destAddr.sin_family = AF_INET;
/* Specify the destination port */
    destAddr.sin_port = htons(TM_DEST_PORT);
/* Specify the destination IP address */
    destAddr.sin_addr.s_addr = inet_addr(TM_DEST_ADDR);
/* Create a socket */
    testSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

/*
 * Verify the socket was created correctly. If not, return
 * immediately
 */
    if (testSocket == TM_SOCKET_ERROR)
    {
        returnVal = tfGetSocketError(testSocket);
        errorStr = tfStrError(returnVal);
        goto udpClientEnd;
    }
/* While we haven’t yet sent enough packets... */
    while (counter < TM_PACKETS_TO_SEND)
    {
/* Send another packet to the destination specified above */
        errorCode = sendto(testSocket,
                           testBuffer,
                           TM_BUF_SIZE,
                           0,
                           &destAddr,
                           sizeof(destAddr));

/*
 * Check if there was an error while sending. If so, break from the
 * loop
 */
        if (errorCode < 0)
        {
            returnVal = tfGetSocketError(testSocket);
            errorStr = tfStrError(returnVal);
            break;
        }

/* Increment the number of packets sent by 1 */
        counter++;

    }

udpClientEnd:
/* Make sure the socket exists before we close it */
    if (testSocket != -1)
    {
/* Close the socket */
        tfClose(testSocket);
    }

    return(returnVal);
}
```

UDP Server

This code is a very simple UDP server. It creates a socket, binds it to the desired port (it is not necessary to supply an IP address… the stack will pick one. This is very useful for making code portable), and then receives data. Upon receipt of the data, the sourceAddr structure is filled out with the originating IP Address and port of the incoming packet.

```
#include <trsocket.h>

#define TM_BUF_SIZE    1500
#define TM_DEST_PORT   9999

char   testBuffer[TM_BUF_SIZE];
char * errorStr;

int udpServer(void)
{
    int                testSocket;
    struct sockaddr_in sourceAddr;
    struct sockaddr_in destAddr;
    int                errorCode;
    int                addrLen;
    int                returnVal;

    returnVal = 0;

/* Specify the address family */
    destAddr.sin_family = AF_INET;
/*
 * Specify the dest port (this being the server, the destination
 * port is the one we'll bind to)
 */
    destAddr.sin_port = htons(TM_DEST_PORT);
/*
 * Specify the destination IP address (our IP address). Setting
 * this value to 0 tells the stack that we don't care what IP
 * address we use - it should pick one. For systems with one IP
 * address, this is the easiest approach.
 */
    destAddr.sin_addr.s_addr = 0;

/*
 * The third value is the specific protocol we wish to use. We pass
 * in a 0 because the stack is capable of figuring out which
 * protocol to used based on the second parameter (SOCK_DGRAM =
 * UDP, SOCK_STREAM = TCP)
 */
    testSocket = socket(AF_INET, SOCK_DGRAM, 0);

/* Make sure the socket was created successfully */
    if (testSocket == TM_SOCKET_ERROR)
    {
        returnVal = tfGetSocketError(testSocket);
        errorStr = tfStrError(returnVal);
        goto udpServerEnd;
    }

/*
 * Bind the socket to the port and address at which we wish to
 * receive data
 */
    errorCode = bind(testSocket, &destAddr, sizeof(destAddr));

/* Check for an error in bind */
    if (errorCode < 0)
    {
        returnVal = tfGetSocketError(testSocket);
        errorStr = tfStrError(returnVal);
        goto udpServerEnd;
    }
 /* Do this forever... */
    while (1)
    {
/* Get the size of the sockaddr_in structure */
        addrLen = sizeof(sourceAddr);
 /*
 * Receive data. The values passed in are:
 * We receive said data on testSocket.
 * The data is stored in testBuffer.
 * We can receive up to TM_BUF_SIZE bytes.
 * There are no flags we care to set.
 * Store the IP address/port the data came from in sourceAddr
 * Store the length of the data stored in sourceAddr in addrLen.
 *The length that addrLen is set to when it’s passed in is
 *used to make sure the stack doesn’t write more bytes to
 *sourceAddr than it should.
 */
        errorCode = recvfrom(testSocket,
                             testBuffer,
                             TM_BUF_SIZE,
                             0,
                             &sourceAddr,
                             &addrLen);
/* Make sure there wasn’t an error in recvfrom */
        if (errorCode < 0)
        {
            returnVal = tfGetSocketError(testSocket);
            errorStr = tfStrError(returnVal);
            break;
        }
    }
udpServerEnd:
/* Make sure we have an actual socket before we try to close it */
    if (testSocket != -1)
    {
/* Close the socket */
        tfClose(testSocket);
    }
     return(returnVal);
}
```

TCP Client

This code is very much like the UDP client. Unlike UDP, however, it must call **connect** before actually transferring data, as TCP requires a negotiated connection. It then calls send the specified number of times. An interesting observation is that the number of TCP data packets actually sent out on the wire will very probably not equal the number defined in this code. TCP is stream-based rather than datagram based, so it will buffer data and attempt to send it in the most convenient size packets (generally, maximum sized packets).

```
#include <trsocket.h>

#define TM_BUF_SIZE         1400
#define TM_PACKETS_TO_SEND  10
#define TM_DEST_ADDR        “10.129.36.52”
#define TM_DEST_PORT        9999

char   testBuffer[TM_BUF_SIZE];
char * errorStr;

int tcpClient(void)
{
    int                testSocket;
    unsigned int       counter;
    struct sockaddr_in destAddr;
    int                errorCode;
    int                sockOption;
    int                returnVal;

    returnVal = 0;
    counter = 0;

/* Specify the address family */
    destAddr.sin_family = AF_INET;
/* Specify the destination port */
    destAddr.sin_port = htons(TM_DEST_PORT);
/* Specify the destination IP address */
    destAddr.sin_addr.s_addr = inet_addr(TM_DEST_ADDR);

/* Create a socket */
    testSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
/*
 * Verify the socket was created correctly. If not, return
 * immediately
 */
    if (testSocket == TM_SOCKET_ERROR)
    {
        returnVal = tfGetSocketError(testSocket);
        errorStr = tfStrError(returnVal);
        goto tcpClientEnd;
    }
/* Connect to the server */
    errorCode = connect(testSocket, &destAddr, sizeof(destAddr));
/* Verify that we connected correctly */
    if (errorCode < 0)
    {
        returnVal = tfGetSocketError(testSocket);
        errorStr = tfStrError(returnVal);
        goto tcpClientEnd;
    }

/* While we haven’t yet sent enough packets... */
    while (counter < TM_PACKETS_TO_SEND)
    {
/* Send another packet to the destination specified above */
        errorCode = send(testSocket,
                         testBuffer,
                         TM_BUF_SIZE,
                         0);

/*
 * Check if there was an error while sending. If so, break from the
 * loop
 */
        if (errorCode < 0)
        {
            returnVal = tfGetSocketError(testSocket);
            errorStr = tfStrError(returnVal);
            break;
        }

/* Increment the number of packets sent by 1 */
        counter++;

    }

tcpClientEnd:
/* Make sure we have a socket before closing it */
    if (testSocket != -1)
    {
/* Close the socket */
        tfClose(testSocket);
    }

    return(returnVal);
}
```

TCP Server

This is the most complicated of the examples. It creates a socket, binds that socket to a port, and configures it as a listening socket. This allows it to receive incoming connections. It then calls accept, which will block until an incoming connection request is received. When **accept** returns, the sourceAddr structure will have been filled out with the originating IP Address and port of the incoming connection request. **accept** creates a new socket, which is then used to receive data until the connection is closed by the other side. When this happens, the application goes back to waiting for an incoming connection request.

```
#include <trsocket.h>

#define TM_BUF_SIZE    1400
#define TM_DEST_PORT   9999

char   testBuffer[TM_BUF_SIZE];
char * strError;

int tcpServer(void)
{
    int                listenSocket;
    int                newSocket;
    struct sockaddr_in sourceAddr;
    struct sockaddr_in destAddr;
    int                errorCode;
    int                addrLen;
    int                returnVal;

    returnVal = 0;

/* Specify the address family */
    destAddr.sin_family = AF_INET;
/*
 * Specify the dest port (this being the server, the destination
 * port is the one we'll bind to
 */
    destAddr.sin_port = htons(TM_DEST_PORT);
/*
 * Specify the destination IP address (our IP address). Setting
 * this value to 0 tells the stack that we don't care what IP
 * address we use - it should pick one. For systems with one IP
 * address, this is the easiest approach.
 */
    destAddr.sin_addr.s_addr = inet_addr(0);

/* Create a socket */
    listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

/* Make sure the socket was created successfully */
    if (listenSocket == TM_SOCKET_ERROR)
    {
        returnVal = tfGetSocketError(listenSocket);
        errorStr = tfStrError(returnVal);
        goto tcpServerEnd;
    }

/*
 * Bind the socket to the port and address at which we wish to
 * receive data
 */
    errorCode = bind(listenSocket, &destAddr, sizeof(destAddr);

/* Check for an error in bind */
    if (errorCode < 0)
    {
        returnVal = tfGetSocketError(listenSocket);
        errorStr = tfStrError(returnVal);
        goto tcpServerEnd;
    }

/* Set up the socket as a listening socket */
    errorCode = listen(listenSocket, 10);

/* Check for an error in listen */
    if (errorCode < 0)
    {
        returnVal = tfGetSocketError(listenSocket);
        errorStr = tfStrError(returnVal);
        goto tcpServerEnd;
    }

/* Do this forever... */
    while (1)
    {
/* Get the size of the sockaddr_in structure */
        addrLen = sizeof(sourceAddr);

/*
 * Accept an incoming connection request. The address/port info for
 * the connection’s source is stored in sourceAddr. The length of
 * the data written to sourceAddr is stored in addrLen. The
 * initial value of addrLen is checked to make sure too many
 * bytes are not written to sourceAddr
 */
        newSocket = accept(listenSocket, &sourceAddr, &addrLen);

/* Check for an error in accept */
        if (newSocket < 0)
        {
            returnVal = tfGetSocketError(listenSocket);
            errorStr = tfStrError(returnVal);
            goto tcpServerEnd;
        }

/* Do this forever... */
        while (1)
        {
```

```
/* Receive data on the new socket created by accept */
            errorCode = recv(newSocket,
                             testBuffer,
                             TM_BUF_SIZE,
                             0);

/* Make sure there wasn’t an error */
            if (errorCode < 0)
            {
                tfClose(newSocket);
                returnVal = tfGetSocketError(newSocket);
                errorStr = tfStrError(returnVal);
                goto tcpServerEnd;
            }
/*
 * Receiving 0 bytes of data means the connection has been closed.
 * If this happens, close the new socket and break out of this
 * (the inner) loop.
 */
            if (errorCode == 0)
            {
                tfClose(newSocket);
                break;
            }
        }
    }

tcpServerEnd:
/* Make sure there’s a socket there before closing it */
    if (listenSocket != -1);
    {
/* Close the listening socket */
        tfClose(listenSocket);
    }

    return(returnVal);
}
```

Turbo Treck Systems

Turbo Treck Real-Time TCP/IP Systems

Turbo Treck TCP/IP is designed to be easy to integrate into your environment. It is also designed to be scalable to your needs. In other words it is designed to allow you to use as much or as little of Turbo Treck TCP/IP as you need. For example, if you decide to make an `int` type 16 bits, you can use up to 32000 sockets with very little penalty in performance. When using the Turbo Treck TCP/IP system we want you to think of it as a "Black Box". We believe that there should be no reason for you as an integrator to be intimately familiar with the internals of Turbo Treck TCP/IP. We have designed a series of API's that you can use to hook Turbo Treck to any environment. These API's include a kernel interface, timer interface, driver interface, sockets interface, and miscellaneous API's to allow you to configure Turbo Treck TCP/IP to your environment. Built inside of Turbo Treck is a locking system, buffer system, and timer system. These different systems are described in this chapter so that you know how Turbo Treck uses the API's that are described in the next two chapters. You will also need to know what the requirements are to integrate Turbo Treck Real-Time TCP/IP into your environment.

Locking System

Turbo Treck Real-Time TCP/IP can be used in many different models. You do not need an operating system in order to use Turbo Treck in your environment. However, if you have an operating system, we will of course use that operating system to allow us to work best for you. Operating systems seem to come in two different types. The simplest operating system is a non-preemptive operating system. The most common variety of an operating system is a preemptive operating system. The difference between the two types is crucial when you decide to integrate Turbo Treck Real-Time TCP/IP into your environment. Preemption is when a task is interrupted by another task of higher priority, or when a fixed time interval has occurred. With a non-preemptive operating system, the user does not need to set up any critical sections. A critical section is a section of code that is protected from preemption by disabling interrupts while working on a shared data area or a shared resource. Since a non-preemptive operating system is by definition non-preemptive, we do not require that protection in this environment. If you have other real-time needs, this may not be the operating system for you.

In a preemptive operating system environment, we do need to concern ourselves with the possibility that we may be preempted or swapped out by another task or interrupt service routine (ISR). In embedded environments we cannot have long critical sections, otherwise this would inhibit our ability to operate in real time. This is what our locking system is concerned with. With our locking system, we are able to protect a section of code over long periods of time without having the interrupts disabled on the system for long periods of time.

With networking it is impossible to achieve 100 percent re-entrancy of the code without locking. An example of this would be an application task trying to receive data at the same time as a higher priority task is trying to store data in the receive queue. We are forced to protect the application task while it is updating its pointers to the receive queue, otherwise the user may be returned a corrupt pointer or even worse the corrupt pointer is put into the socket entry. For a non-preemptive operating system this is never an issue because each task is operating at the same priority. One task completes everything it needs to before another task is swapped in.

Another important item to consider when operating in embedded environments is that care must be taken with how we call a networking stack from an interrupt service routine. If we were to receive a packet inside of an interrupt service routine and pass that packet to the networking stack, it is possible that the packet may generate a response to the network. This would increase the time that we stay inside of our interrupt service routine. Most embedded systems have a requirement that the interrupt service routine time is kept to a minimum. You will find with the Turbo Treck TCP/IP, a very small set of function calls are supported within an interrupt service routine. These calls are limited in the amount of work that they do. One of these would be to update the timer. However, this is not a recommended method for updating the timer system. We would in fact prefer that the update and execute of the timer system be done in either a main line loop or a separate timer task. Another call from an ISR that is supported is notification of a send complete or a received packet. Note that we do not support passing a packet back to the stack from an ISR. All we do is notify the stack that a packet has been received by the network hardware.

As we can see from what we have discussed thus far, we only need the locking system when we are using Turbo Treck with a preemptive operating system and we are using Turbo Treck as a shared library in this environment. What we have done with the locking system is to use the most atomic feature found in most embedded operating systems to achieve single threaded access to a data area. We use a counting semaphore to guarantee that only one thread or task of a system can access certain shared data areas at one time. By using a counting semaphore we allow the operating system to properly context switch to a higher priority task at the end of the locked section.

In our Turbo Treck TCP/IP we use many different locks. By operating in this fashion we avoid stopping other tasks that have no interest in our protected data area. If there is no contention for the same data area, tasks will still operate without blocking. We do not call the operating system unless there is contention for a shared data area.

There is a large difference between a critical section and locking in Turbo Treck TCP/IP. A critical section is used to protect a shared data area in a very small amount of code. This code is typically less than five assembly instructions in most instances. On the other hand, a lock is used to protect a shared data area or resource from reentrancy for longer periods. This shared data area could be a socket entry, routing entry, or ARP entry. Even with locks we attempt to keep the time as short as possible to avoid having contention for the shared data area or resource. Counting semaphores are also used to allow us to wait for a resource to become available, such as received data. When we the call socket function **recv** and we are allowing blocking to occur, if there is no data to receive, we will pend on a counting semaphore until there is data for us to receive or the socket has been closed.

Blocking means that we will wait until some event has occurred that we want to happen. A task that is blocked will allow other tasks to still run. By default, Sockets operate in blocking mode. In order to achieve blocking when it needs to occur, we rely on a counting semaphore. By using counting semaphores for blocking, we do not need to call the operating system in a critical section. This is extremely important for embedded systems. In fact, some operating systems will not allow you to call them inside of a critical section. If we did not use a counting semaphore, then there could be a window of opportunity where a task never exits the blocking state should the event that we are waiting for occurred during that small window.

Most operating systems have the concept of a counting semaphore. Not all operating systems offer a counting semaphore. If your RTOS does not provide a counting semaphore, but does have an event flag, then use the counting semaphore implementation in *kernel\trcousem.c.*

A counting semaphore is a mechanism that allows you to wait for a resource indefinitely. It also allows you to release the resource before you actually wait. If the event occurs before you wait, your task will not wait for the resource. This mechanism is very important to the operation of Turbo Treck TCP/IP. Protocol stacks need this mechanism in order to work in an asynchronous mode properly. Now there is another question that you must be asking yourself “Do I actually need a counting semaphore?” The answer is simple. If you are not using a preemptive operating system, and you do not make any blocking calls, then you do not need a counting semaphore. We only use the counting semaphore for blocking and locking.

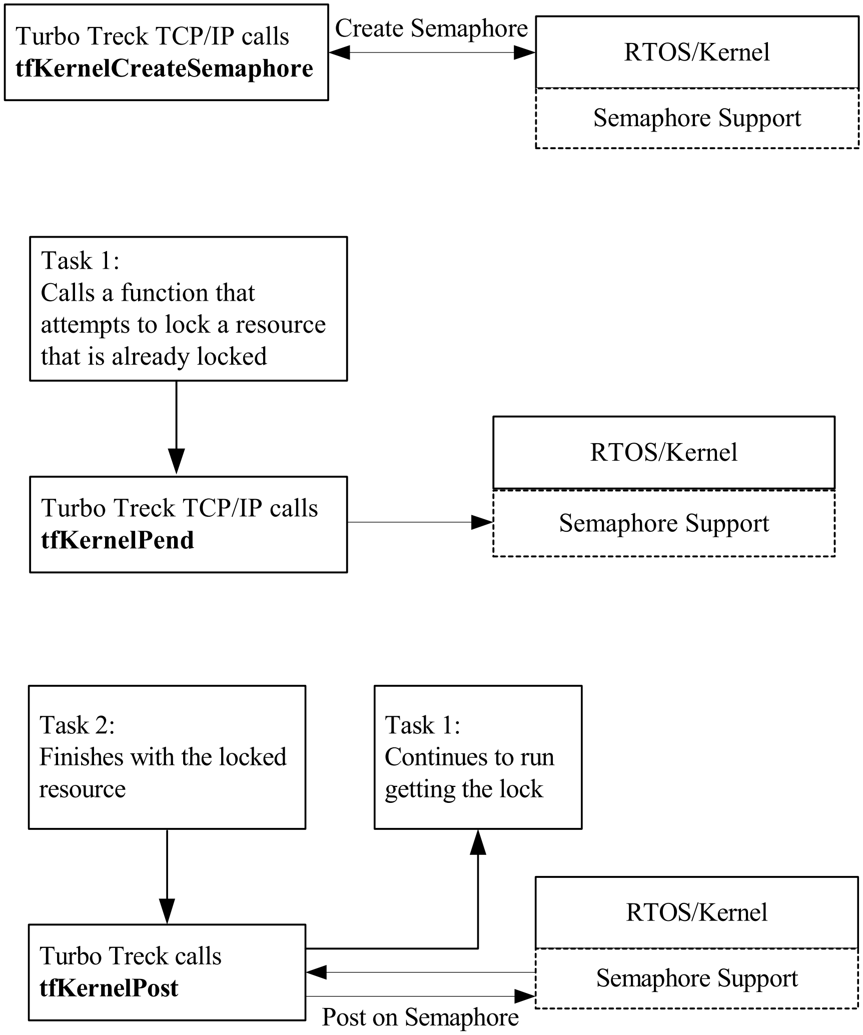


Figure 3-1
RTOS allows Task 1 to Continue

Buffer System

In Turbo Treck TCP/IP we have an internal buffer management system for performance reasons. We have found that if we need to call the operating system every time that we need a buffer and again have to call the operating system in order to free that buffer, there is a severe penalty on performance. What we do with Turbo Treck TCP/IP is allow the system to dynamically allocate memory as needed. This means the memory pool may grow over time. After Turbo Treck TCP/IP is used actively over a period of about one-minute, the maximum pool that Turbo Treck TCP/IP will be using is already allocated from the operating system.

The pool that Turbo Treck is using will only grow as more performance demands are placed on the networking system. By allocating buffers in this fashion, Turbo Treck eliminates the guesswork that is involved in trying to setup a system.

We allow the protocol stack to only use the memory that is required for your performance demands. You do not need to set aside an area of memory of a fixed size. We have found that if you are forced to set aside a fixed size memory block that will be used by the TCP/IP stack to dynamically allocate memory from, you will be forced to set aside more memory than you will use in order to prevent “out of memory” errors from occurring. We dynamically allocate almost all of our data structures. This is done to keep the static memory area as small as possible. Also by doing this, we allow you to change the configuration of the protocol stack without recompiling.

By this time you must be asking yourself the question “So, how much RAM will I use?” We have found that a single TCP socket implementation with a small number of receive buffers will use less than 20k bytes of RAM when the highest performance demands are placed upon the protocol stack. The amount of RAM used in your system will depend on many different factors. These factors include the size of your sockets send queue and receive queue as well as the number of pre-allocated receive buffers for the device. Most of the RAM is used for your data. The internal data structures are very small. For example: a single UDP socket entry requires less than 200 bytes for the internal data structures. If we are using TCP on a socket, then the internal data structure grows to less than 400 bytes. A socket entry is only allocated between the socket call and the close call. This means that the socket data structures are not in use when the socket is not active. This allows you to have a high number of sockets defined on a system and use a small amount of RAM while they are not in use. Maximum RAM usage is also dependent on the maximum number of concurrent sockets that you have open at any given time. The interface to your operating system or “C” compiler from the buffer system is a simple malloc and free call.

Timer System

The timer system is the most simple of the systems to understand. We need a fixed time interval notification into the protocol stack. This is very similar to the crystal that is attached to your microprocessor. We use this fixed time notification to let us know how much time has elapsed. This is very important for a protocol stack. We must know how long a TCP packet takes to get to a remote system. We must also know how much time an ARP entry has been in the ARP queue. In addition, the timer system also measures routing entries and is used for Turbo Treck PPP.

You don’t deal with all of the individual timers that Turbo Treck TCP/IP manages. Since we only use a single notification to let us know how much time is elapsed, the timer system will manage all of the different timers used by the Turbo Treck TCP/IP system.

Turbo Treck TCP/IP is flexible in the way that you provide this notification. You have the option of using an ISR or a task to do the timer notification. The notification is done in a single function call. You can then decide when you wish to execute the timers. The execution of the timers is done with a single function

Integrating Turbo Treck Real-Time Protocols Into Your Environment

Integrating Treck Real-Time Protocols Into Your Environment

Now that you have a basic understanding of TCP/IP and the Treck Systems, you are probably wondering how to integrate Treck protocols into your system. You have looked at the product and discovered there are many files and API's that you may have to deal with. To aid you in your integration effort, we will break this task down into several key elements:

1) Deciding how you will use the protocols in your system
2) Setting up the TRSYSTEM.H file for various compile time switches
3) Creating the Build Command (.BAT) files for a DOS Environment and Building the Library
4) Creating an RTOS Interface
5) Hooking in the Timer
6) Key things you need to do in order to start using Treck protocols
7) Testing the new library with a simple loop-back test
8) Using Ethernet or PPP
9) Adding a new device driver
10) Testing your new device driver

If we have some or all of these elements setup for you, you will (of course) be able to skip some of these steps. In our design, we have made every effort to help the integration process be as simple as possible.

TIP: Don't be overwhelmed by what looks like a lot of work for you to do. Most of these steps can be performed in one or two days. You will find that you will spend most of your time in the place where the hardware meets the software (the device driver). Just go through these steps and take them one at a time and you will have the integration completed before you know it.

Step 1- Determining How to Use the Protocols in Your System

To help in making this decision, you will need to ask yourself a few questions.

Am I using a Real-Time Operating System (RTOS)?

Is my processor Big or Little Endian?

I do not have an RTOS/Kernel to Interface to

If you do not have an RTOS/Kernel to which to interface, you will still need a minimum RTOS interface (which we provide). If you decide not to use an RTOS, then remember that most of the calls into the Treck Protocol Stack will be done from a main line loop. The biggest disadvantage in not using an RTOS is that you *cannot* make blocking calls (i.e. make recv () wait for data), otherwise you will stop all processing in your system. Another disadvantage is that you cannot prioritize events. If you are thinking about this, but are still undecided, we suggest that you look at using uC/OS as your RTOS. This is included with the base Treck protocols. If you are dead set against using an RTOS, then we will help you understand the limitations in the Kernel configuration section of this chapter.

I have a RTOS/Kernel to Interface to, but it does not allow any blocking/ pending calls

An example of this would be an event scheduler with all the calls made from a main loop. This falls into the "I do not have an RTOS/Kernel to interface to" category.

I have a RTOS/Kernel to Interface to

You will need to decide if your RTOS is preemptive or non-preemptive. If you do not know, you can always assume preemptive (which is the safest). Preemptive RTOS/Kernels need to have locking of shared resources.

TIP: A Preemptive operating system is one where a timer tick or other event causes a context switch to a new task. Tasks may have different priorities. Higher priority tasks may interrupt lower priority tasks.

A non-preemptive operating system is one where all tasks have the same priority, run in a round robin fashion, and explicitly give up the CPU through an RTOS call. This is usually called a "Round Robin Scheduler"

Next, you will need to decide how you are going to use Treck Real-Time TCP/IP with your RTOS.

Do you want to have Treck run as its own task or in the context of other tasks as a *shared library?*

This choice depends on how you want your system to run. Most embedded systems choose to use the protocol stack as a shared library because it gives you the most flexibility on how you want to prioritize the different processes in using a protocol stack. If you decide to use Treck protocols as a single task, then there is also the performance hit of having a thin layer between the application and the protocol stack. Most people, who want to have a separate task, decide to do it to maintain modularity of the protocol stack. In doing so, they allow the protocol stack to be a "plug-in" to an existing application.

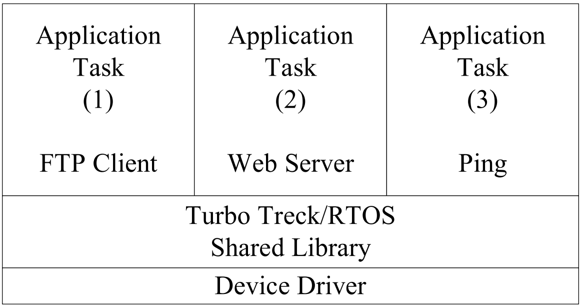


Fig. 4-1
Example of Using Treck Protocols as a Shared Library

Step 2 - Setting TRSYSTEM.H for Various Compile Time Switches

The TRSYSTEM.H file is used to configure the compile time switches for the Treck protocol stack. It should be considered the master configuration file. In this file, you will find macros to define things such as optimization, endian, and defaults for run time parameters used by the protocol stack. You may find that you do not need to modify this file if your environment is already defined in the TRSYSTEM.H file. *When you are integrating to your hardware platform and hooking the protocol stack up to your kernel, this will be the only file that you need modify.* If you decide to turn on macros that are defined in this file, you can do it at the top of the file. The following are options that you may decide to turn *on* or *off*:

Performance Macros

TM_BYPASS_ETHER_LL

This performance macro is used to “in-line” the Ethernet processing. When this macro is enabled, it allows us to process Ethernet packets much faster than sending them to a separate link layer. The disadvantage to using this macro is that it costs a little more code space to do the Ethernet processing. If you are not using Ethernet, then you do not want to have this macro defined.

TM_IP_FRAGMENT

If this macro definition is removed from *trsystem.h*, the IP fragmentation code is not compiled in. If you expect to send IP fragments, you need to keep this macro in *trsystem.h*. Otherwise, you can remove the TM_IP_FRAGMENT macro from *trsystem.h*, thereby reducing code size.

TM_IP_REASSEMBLY

If this macro definition is removed from *trsystem.h*, the IP reassembly code is not compiled in. If you expect to receive IP fragments, you need to keep this macro in *trsystem.h*. Otherwise, you can remove the TM_IP_REASSEMBLY macro from *trsystem.h*, thereby reducing code size.

TM_IP_FRAGMENT_NO_COPY

If you define TM_IP_FRAGMENT and your device driver supports scattered send, then define TM_IP_FRAGMENT_NO_COPY to avoid an extra internal data copy when IP datagrams need to be fragmented.

TM_DISABLE_PMTU_DISC

By default, the TCP Path MTU Discovery code is compiled in and turned on. Adding this macro will prevent the compilation of the TCP Path MTU Discovery code, thereby reducing code size.

TM_DISABLE_TCP_SACK

By default, the TCP Selective Acknowledgement code is compiled in and turned on. Adding this macro will prevent the compilation of the TCP Selective Acknowledgement code, thereby reducing code size.

TM_TCP_FACK

Define the TM_TCP_FACK macro to enable the Forward Acknowledgement algorithm. It an only be enabled if TM_DISABLE_TCP_SACK is not defined. Enabling TM_TCP_FACK will allow the TCP Sack to retransmit more than one SACKed segment (*wireless TCP option only*).

TM_USE_TCP_PACKET

Enable this macro if you wish to enable code that modifies TCP behavior and forces TCP to send data on user send packet boundaries. Note that this will only enable the code. You must also set the TM_TCP_PACKET option at the IPPROTO_TCP level as described in the **setsockopt** API page.
Warning! Uncommenting this macro will make TCP less efficient and will disable Path MTU discovery and TCP Selective Acknowledgements.

TM_DISABLE_DYNAMIC_MEMORY

By default, when the Treck stack allocates blocks of memory of size less than 4096 bytes, it keeps them in its internal buffer management system, when they are freed, instead of releasing them to the heap. Adding this macro will prevent the compilation of the Treck internal buffer management system code, thereby reducing code size. However, this is ***not recommended***, as this will reduce performance, since this will force the Treck stack to call the Operating system every time it needs to allocate or free a block of memory. If the user has not disabled the Treck Dynamic Memory (i.e. not defined this macro), then the user can free all the unused dynamic memory held by the Treck stack at any time, by calling the API ***tfFreeDynamicMemory***.

TM_ARP_UPDATE_ON_RECV

Enable this macro if you want the stack to update the ARP cache for every packet that is being received. This will prevent the ARP cache entry from timing out, and will prevent the stack from having to send an ARP request every 10 minutes, but at the cost of code size, and speed, since every incoming packet needs to be checked, and the ARP entry updated then.

TM_OPTIMIZE_SPEED

This performance macro is used to optimize the code in favor of speed over size. When using this macro you will notice that the code is much larger than without it. However, you will see a noticeable increase in speed. Typically, this macro is used to in-line functions that are used repeatedly.

TM_OPTIMIZE_SIZE

This macro is used to optimize the code in favor of size over speed. When using this macro you will get the smallest code size available. You will notice that the performance of the system is degraded when you use this macro. This macro is to be used when the performance of the system does not matter but code size does.

TM_ERROR_CHECKING

This macro is used to turn on error checking for the protocol stack. It is used for debugging purposes; it is not intended for use in your released product. The error checking will send messages to the functions ***tfKernelError*** and ***tfKernelWarning*** for you to log. There is a performance and size impact when using this macro.

TM_THREAD_STOP

Thread stop is used to stop a thread which has had an unrecoverable error.
By default, it is defined as a forever loop.
You can define this macro to supersede the default definition.

TM_PROTO_EXTERN

By default the prototypes declared in the Treck header files are not declared extern. If your linker dictates that they should be declared extern, then add the following macro:`#define TM_PROTO_EXTERN extern.`

TM_LOOP_TO_DRIVER

If this macro definition is added to trsystem.h, it will allow the user to loop back application data all the way to the device driver when sending to an interface configured IP address. By default, this macro is not defined in trsystem.h, and this causes the Treck TCP/IP stack to loop back application data above the link layer when the user is sending to an interface configured IP address. This is useful when debugging a device driver, or link layer.

TM_USE_DRV_ONE_SCAT_SEND

This macro is commented out by default.
Define the TM_USE_DRV_ONE_SCAT_SEND macro if you wish to use a single call to the device driver, passing the packet handle, even when sending a frame with scattered buffers. Note: to enable this feature, this macro must be added and **tfUseInterfaceOneScatSend** must be called on the interface that supports it.

TM_USE_DRV_SCAT_RECV

This macro is commented out by default.
Define the TM_USE_DRV_SCAT_RECV macro, if you want to allow the device driver recv function to pass back a frame to the stack in scattered buffers ("Gather Read"). Note that to enable this feature, this macro needs to be added, and **tfUseInterfaceScatRecv** needs to be called on the interface that supports it.

TM_INDRV_INLINE_SEND_RECV

This macro is defined by default.
Undefine/delete the TM_INDRV_INLINE_SEND_RECV macro to test loop back/intra machine driver with a separate recv task. In that case, the intra machine received data will no longer be processed in the send path, but will be processed when **tfRecvInterface/tfRecvScatInterface** is called. In the examples directory, the txscatlp.c, and txscatdr.c modules contain sample code that uses this feature.

TM_DISABLE_TCP_ACK_PUSH

This macro is uncommented by default. When this macro is commented out, Treck TCP will ACK every TCP segment that it receives that has the PUSH bit set. Comment out this macro if you need the Treck stack to interoperate well with a peer that runs Windows 2000, because Internet Explorer waits for the ACK response to the PUSH bit being set before sending any new data.

TM_SINGLE_INTERFACE_HOME

If this macro definition is added to trsystem.h, it will reduce code size by approximately 4.5 kilobytes by removing support for multiple interfaces and multi-homing. Note that when this macro is enabled, you can only have a single interface and a single IP address configured on that interface.

Warning: Defining this will prevent addition of the loop back device.

1. Packets sent to the single interface IP address will be sent all the way to the driver, instead of being loop back by the stack.
2. Packets cannot be sent to the IP loop back address, 127.0.0.1.

TM_MULTIPLE_CONTEXT

This macro is commented out by default. Uncomment this macro if you want to run multiple instances of the Treck TCP/IP stack. Running multiple instances of the Treck stack is further described in appendix A of this manual.

TM_DISABLE_ANSI_LINE_FILE

This macro is only relevant when TM_ERROR_CHECKING is #define'd, in which case the tm_assert macro uses the ANSI __LINE__ and __FILE__ macros (if available) to print the source line number and source file name identifying where in the code an assertion failure occurred. Uncomment the TM_DISABLE_ANSI_LINE_FILE macro if your compiler does not support the ANSI__LINE__ and __FILE__ macros.

TM_DISABLE_TCP_RFC2414

Enable this macro if you want to disable the TCP Initial Send Window Increase as described in RFC-2414.

TM_DISABLE_TCP_RFC2581

Enable this macro if you want to remove the TCP congestion control update to RFC2001 as described in RFC2581 (*wireless TCP option only*).

TM_DISABLE_TCP_RFC3042

Enable this macro if you want to remove the Limited Transmit Algorithm that enhances TCP's Loss Recovery as described in RFC3042 (*wireless TCP option only*).

TM_DEV_SEND_OFFLOAD

Enable this macro if you want to use the device driver TCP segmentation/ checksum offload capabilities.

TM_DEV_RECV_OFFLOAD

Enable this macro if you want to use the device driver recv packet checksum offload capabilities.

TM_USER_OCS

Enable this macro to use a user defined checksum routine, e.g. in Assembly. Rather than the standard C routine.

TM_PC_LINT

Enable this macro if you are using Gimpel Software's PC-lint to check the Treck stack code.

TM_TCP_ANVL

Uncomment this macro when testing the Treck stack against Ixia's Automated Network Validation Library (ANVL).

TM_USE_AUTO_IP

This macro is commented out by default. Uncomment this macro, if you want to use Auto IP Configuration, or want to add collision detection.

TM_USE_RAW_SOCKET

Enable this macro if you want to use raw sockets (tfRawSocket) which allows you to send data above the IP layer or to send data with an IP header, and to receive data with an IP header.

TM_RAW_SOCKET_INPUT_ICMP_REQUESTS

Enable this macro if you want ICMP echo requests, and ICMP address mask requests to be delivered to a raw ICMP socket.

TM_USE_REUSEADDR_LIST

Enable this macro if you want to use the SO_REUSEADDR socket level option with setsockopt(), which allows you to bind the same port number to multiple sockets using different local IP addresses.

TM_USE_PPP_MSCHAP

This macro is commented out by default. Uncomment this macro if you want to use MS-CHAP for PPP authentication.

TM_PPP_LQM

Enable this macro if you want to use PPP Link Quality Monitoring.

TM_USE_EAP

Enable this macro to enable Extensible Authentication Protocol with PPP.

TM_USE_IPHC

Enable this macro to enable IP header compression (RFC-2507) with PPP.

TM_USE_VCHAN

Enable this macro to support splitting out IPv4 and IPv6 traffic in the send path to separate virtual channels to simulate multiple hosts as multiple IP aliases. Note that this is incompatible with the default mode of IPv6 operation, therefore IPv6 prefix discovery must be disabled.

TM_OPTIMIZE_MANY_MHOMES

Enable this macro to speed up lookup in the recieve-path of IP aliases configured on the interface. This is intended for situations where the user has configured many (i.e. >50) IP aliases on a single interface.

TM_USE_NETSTAT

Define this macro, if you want to use the netstat tool for for outputting the routing entries, ARP entries and socket entries. The netstat tool is described in the Application Reference Section.

TM_USE_SNIFF

if TM_USE_SNIFF is defined, interface drivers will dump the packets to the file system in libpcap format, to be opened by tcpdump, windump, ethereal, or other libpcap file readers.Models for Running Treck

TM_TRECK_NO_KERNEL

This macro is used to compile the stack so that it will not use an RTOS/Kernel. When you do not have a real time operating system, you will not need to use the locking system that the Treck protocol stack provides. Note that you will not be able to make any calls that will cause the system to block. You must ensure that any call into the Treck protocol stack will not block. Critical sections for the driver interface are still enabled when you are using Treck without a kernel.

TM_TRECK_NONPREEMPTIVE_KERNEL

This macro is used to compile the stack so that it will use a non-preemptive kernel. You must ensure that your kernel is not preemptive before you define this macro. This macro disables the locking system, which is not needed by a non-preemptive kernel. When you are using a non-preemptive kernel you may still make calls to the Treck protocol stack that are either blocking or non-blocking. Critical sections for the driver interface are still enabled when you are using Treck with a non-preemptive kernel.

TM_TRECK_TASK

This macro is used to compile the stack so that it can be used as an independent task. When using Treck in this manner, no ISR may call any Treck function. In this model, there are no critical sections and no blocking may occur. This model is typically used when the application and the Treck code are in the same task and the device driver sends messages through the operating system to the task. It is similar to using Treck without a kernel. It may be used with either a preemptive or a non-preemptive kernel.

TM_TRECK_PREEMPTIVE_KERNEL

This macro is used to compile the stack so it can be used with a preemptive kernel. In this model, all critical sections are enabled. You may also feel free to make blocking or non-blocking calls in this environment. This model is the most popular for systems integrating Treck into their environment.

TM_TASK_RECV

This macro is used to notify the stack that a separate blocking receive task will be used. It is not required to use a separate receive task. However, it is usually efficient to do so. The receive task will be responsible for processing all received packets from a given interface. If more than one interface is defined on the system, more than one receive task may be used. If you wish to make the decision to use a blocking receive task at run time, then you should define this macro. A separate blocking receive task can only be used if you are using a real-time operating system, and you are using the Treck protocols as a shared library.

Note: It is possible to use a polling (i.e. non blocking) separate receive task, in which case you do not need to define TM_TASK_RECV.

TM_TASK_XMIT

This macro is used to notify the stack that a separate blocking transmit task will be used to send packets to the device driver. A separate blocking transmit task can only be used if you are using a real time operating system, and you are using the Treck protocols as a shared library. If more than one interface is defined on the system, more than one transmit task may be used. It is not required to use a separate blocking transmit task. If no transmit task is used, then the packets are queued to the device driver send queue, and are sent to the driver in the context of the sending thread. The sending thread could be a user application task sending data, or the receive task sending a TCP acknowledgment, or a PING echo reply, or the timer task re-transmitting data. If a separate transmit task is used, then packets are queued to the device send queue in the context of the sending thread. Then, when a context switch occurs, the packets are sent to the driver in the context of the transmit task. In most cases it is inefficient to use a separate blocking transmit task because it requires a context switch on nearly every packet sent by the sending thread.

Note: It is possible to use a polling (i.e. non blocking) separate transmit task, in which case you do not need to define TM_TASK_XMIT.

TM_TASK_SEND

This macro is used to notify the stack that a separate blocking send complete task will be used. Notice that this is not a send task but a send complete task. The user calls the send complete function in the context of the send complete task, and this allows the user to adjust the priority of that task. In most cases, it is inefficient to use a separate send complete task. The most efficient means of processing send completes is in the context of the task that is calling the actual driver send routine. The reason that it is inefficient is because a separate send complete task requires a context switch on nearly every packet that was sent.

Note: It is possible to use a polling (i.e. non blocking) separate send complete task, in which case you do not need to define TM_TASK_SEND.

Timer Updates

TM_TICK_LENGTH

This macro is used to define the default value that will be used to specify the interval between timer updates for the protocol stack. This is a default value and can be changed at run-time. The value specified is in milliseconds. The accuracy of this time is not critical. This value should be the average time in between updates to the Treck Timer System. The typical values used for this macro are between 10 and 100 milliseconds.

Word Order

You do need to know if your processor is big endian or little endian. "Endian" refers to the byte order of integers (both long and short) on the processor. For more information on the difference between big and little endian, please see the chapter "Introduction to TCP/IP".

TM_LITTLE_ENDIAN

This macro is used to let the stack know at compile time that the processor is a little endian processor. If you choose to use this macro on a big endian processor, you will see your packets with the word fields reversed.

TM_BIG_ENDIAN

This macro is used to let the stack know at compile time that the processor is a big endian processor. If you choose to use this macro on a little endian processor, you will see your packets with the word fields reversed.

Memory Allocation

TM_USE_SHEAP

If the user defines this macro, the stack allocates its blocks of memory from the Treck simple heap static array using the simple heap allocation routine, ***tfSheapMalloc***, instead of calling the RTOS ***tfKernelMalloc***. Similarly the Treck stack will release an allocated block of memory (of size bigger than 4096 bytes) by calling the Treck simple heap free routine ***tfSheapFree***, instead of calling the RTOS ***tfKernelFree***. The user will define this macro if the user's RTOS does not provide heap management routines, or if the user does not want the Treck stack to allocate its blocks of memory from the RTOS heap. Note that if you define this macro, and thereby do use the Treck stack simple heap, you cannot disable the Treck stack dynamic memory allocation (see TM_DISABLE_DYNAMIC_MEMORY above), since the Treck simple heap can only handle freeing blocks that are bigger than 4096 bytes. For the same reason, you cannot call ***tfFreeDynamicMemory*** if you define this macro.

TM_SHEAP_SIZE

If you do define TM_USE_SHEAP, then you also have to define TM_SHEAP_SIZE, to give the size in bytes of the Treck simple heap array.

TM_DYNAMIC_CREATE_SHEAP

Normally, the Treck simple heap is implemented as a static array. However, you can override this behavior by defining the macro TM_DYNAMIC_CREATE_SHEAP in your trsystem.h file, in which case you decide how you want to implement it (i.e. dynamic memory allocation specific to your RTOS) by customizing the API tfKernelSheapCreate.

Complier Qualification

TM_FAR

This macro is used to allow access to far data on an Intel processor (normally running in real mode). It should be defined to far if you need to access far data. For other processors, it should not be defined.

TM_CODE_FAR

This macro is used to allow access to far code sections on an Intel processor. If the stack is compiled in small model, you may have functions that are in other code segments that are called by the Treck protocol stack. In this case, you should define this macro to be far. For other processors, it should not be defined.

TM_INTERRUPT interrupt

If you have a different keyword for interrupt, then TM_INTERRUPT here to be that keyword.

TM_GLOBAL_QLF

Default qualifier for global variables. If you would like to put all global variables into one memory type (e.g. far, huge, near), then #define TM_GLOBAL_QLF here to be that qualifier and leave TM_GLOBAL_NEAR undefined.

TM_GLOBAL_NEAR

Default qualifier for frequently used global variables.
If you would like to put the frequently used global variables into one memory type (e.g. near) and the rest of them into a different memory type (e.g. far, huge), then the memory type for frequently used global variables here as TM_GLOBAL_NEAR and leave TM_GLOBAL_QLF undefined

NOTE about TM_GLOBAL_QLF and TM_GLOBAL_NEAR:
Only one of them can be redefined to have a value, otherwise compiler errors will occur.

TM_CONST_QLF

Default qulifier for constants. If you would like to put all constants into one memory type (e.g. far, huge, near), the TM_CONST_QLF here to be that qualifier.

Data Alignment

TM_ETHER_HW_ALIGN

Various Ethernet chips exist with the ability to make use of DMA. Many of these chips also come with the requirement that their buffers be aligned on a certain byte-boundary (4-byte and 16-byte being most common). TM_ETHER_HW_ALIGN defines how many bytes each Ethernet buffer obtained with ***tfGetEthernetBuffer*** will be aligned to. This macro defaults to 4.

TM_NEED_ETHER_32BIT_ALIGNMENT

Uncomment TM_NEED_ETHER_32BIT_ALIGNMENT and uncomment TM_USE_DRV_SCAT_RECV, (without using ***tfUseInterfaceScatRecv)***, if you want ***tfRecvInterface*** to make the TCP/IP header aligned on a 4-byte boundary, if it is not aligned coming from a single buffer in the Ethernet device driver receive function.

Predefined Processor Macros

TM_INTEL_X86
TM_MOTOROLA_CPU32
TM_MOTOROLA_68K
TM_MOTOROLA_PPC
TM_TMS320_C3
TM_TMS320_C3_BIG_ENDIAN
(For TI C3 like DSP with byte order reversed)
TM_TMS320_C5
TM_TMS320_C6
TM_ARM7
TM_EZ80

These macros are used to specify the processor you are using and your environment. They simply set up the proper word order or endian for the processor.

Compiler Specification

These optional macros are used to define the compiler that you are using. They are used to allow in-line assembly for the critical sections and assembly cores for one's compliment checksums on some platforms. Not all of the supported compilers are listed here, because some of them are automatically determined during compilation (see the TRMACRO.H file).

TM_COMPILER_GHS_ARM

This is used for the Green Hills ARM compiler

TM-COMPILER_GHS_PPC

This is used for the Green Hills PowerPC compiler

TM_COMPILER_SDS

This is used for the SDS 68K compiler

TM_COMPILER_DDI_PPC

This is used for the Diab Data Power PC compiler

TM_COMPILER_MRI_68K

This is used for the Microtec Research (Mentor Graphics) 68K compiler

RTOS/Kernel Environments

These macros are used for various predefined kernels. With these macros, you can use one of the real time operating systems listed below and most of the above settings will be set for you. Feel free to use these as examples if you decide to add a different RTOS.

TM_KERNEL_ELX_86

This is used for the ELX-86 RTOS

TM_KERNEL_UCOS_X86

This is used for uC/OS for the Intel Platform

TM_KERNEL_UCOS_PPC

This is used for uC/OS for the Motorola Power PC Platform

TM_KERNEL_UCOS_CPU32

This is used for uC/OS for the Motorola CPU32 Core (683xx)

TM_KERNEL_UCOS_MIPS

This is used for uC/OS for the MIPS processor

TM_KERNEL_UCOS_XSCALE

This is used for uC/OS for the XSCAL processor

TM_KERNEL_UCOS_SH2

This is used for uC/OS for the SH2 platform

TM_KERNEL_AMX_CPU32

This is used for AMX for the Motorola CPU32 Core (683xx)

TM_KERNEL_THREADX_ARM7
This is used for ThreadX for the ARM7 platform

TM_KERNEL_AMX_X86
This is used for AMX for the Intel Platform

TM_KERNEL_THREADX_CPU32
This is used for ThreadX for the Motorola CPU32 Core (683xx)

TM_KERNEL_DOS_X86
This is used for DOS for the Intel Platform

TM_KERNEL_CMX_C16X
This is used for CMX RTOS for the Siemens C16X processors

TM_KERNEL_NONE_EZ80
This is used for Zilog EZ80 target board, no OS

TM_KERNEL_NONE_H8S
This is used for Hitachi H8S platform, no OS

TM_KERNEL_RZK_EZ80
This is used for RZK RTOS for Zilog EZ80 target board

TM_KERNEL_REALOS_FR
This is used for REALOS RTOS for the Fujitsu MBxxxxx platform

TM_KERNEL_TI_DSP_BIOS
This is used for DSO_BIOS RTOS for the TI DSPs

TM_KERNEL_WIN32_X86
This is used for Win32

TM_KERNEL_VISUAL_X86
This is used for testing under VC++ environment, single threaded, for automated testing.

TM_KERNEL_LINUX_APP
This is used for Linux

Step 3 - Creating the Build Command (.BAT) Files for a DOS Environment and Building the Library

Instead of using make files to build the Treck TCP/IP library, We have put together simple batch files for a DOS build environment. These are designed to be easy to understand and design for compilers that we do not support yet. The other feature of using batch files for DOS is that they can be easily converted to shell scripts for the UNIX environment. It is best to think of the batch files as a three-tiered system for building your project.

TIP: Before running any batch files to build the system, you should run the batch file "SETUP.BAT" located in the TRECKBIN directory to setup the proper path to the TurboTreck directories.

Tier 1: Compiler Primitives and Library Utility Primitives
Tier 2: Compile/Library all the Treck code
Tier 3: Automated Build System

Let's look at each of these tiers individually.

Tier 1: Setting up the compiler and library utility primitives

Here you simply create a batch file (.BAT) that contains the compiler command line. The first parameter (%1) is the file to compile (without the .C extension). The second parameter is optional but is typically used for a #define that is passed to the compiler when building the Treck code.

An example of this is as follows:

File bcl86cc.bat

```
@echo off
if "%2"=="" goto noopt
bcc -v -ml -1 -c -w9999 -I%trhome%\include -D%2 %1.c > %1.err
goto end
:noopt
bcc -v -ml -1 -c -w9999 -I%trhome%\include %1.c > %1.err
:end
```

This file could have been as simple as this:

```
bcc -v -ml -1 -c -w9999 -I%trhome%\include %1.c > %1.err
```

That is all there is to the compile batch file, but we still need a library batch file. We do the same thing for the library utility. Notice that the library utility does not use the second parameter (%2) and the first parameter is the same as in the compile command line.

An example of this is as follows:

File bc86lib.bat

```
@echo off
tlib treck +%1.obj
```

An important thing to note is that our command line only adds a single object to the library.

You should place both of these files into the `TRECKBIN` directory.

Now that you have completed these two tasks, you can build the Treck Library.

Tier 2: Compile/Library all the Treck code

The batch files that are used for this tier are called `BUILD.BAT`, and `BUILDLIB.BAT`. You do not need to modify this file (unless you are removing protocols from the build). This file can be called directly from the DOS command line like this.

C> build bcl86cc

Note: Your current directory MUST be the SOURCE directory in order to build the Treck objects.

Once the build completes (and it should without warnings or errors if your paths are set correctly), you are ready to create your library like this.

C> buildlib bc86lib

If you have done everything correctly to this point, you should have a library. If you get any compile errors, please make sure that they are not due to any changes to the `TRSYSTEM.H` file. If you still have errors (or warnings), please contact technical support.

Tier 3: Setting up the Automated Build System

There is one final batch file that you can add/modify. It is the automation that builds the “C” code into objects and inserts the objects into a library. As you can imagine, it is fairly simple. It calls BUILD.BAT to compile first, then BUILDLIB.BAT to put the objects into a library. In our Tier 3 batch files, we have them pass an extra #define into the build so we don’t have to modify TRSYSTEM.H. We also delete the library before we build to make sure that it is always a clean library. Because we support more than one compiler, the tier 3 batch files are a little more complex than you would design for your system.

An example of one of our build batch files is as follows:

File ucosx86l.bat

```
@echo off
if “%1”==”” goto usage
del treck.lib
call build %1l86cc TM_KERNEL_UCOS_X86
call buildlib %186lib
goto end
:usage
echo %0 builds Treck TCP/IP for the Intel x86 Real Mode Processor
echo running under uC/OS v1.1 (LARGE MODEL)
echo .
echo Usage:
echo %0 compiler
echo .
echo Compilers supported:
echo Microsoft “C” 6.0: MC
echo Borland “C” 5.0:   BC
echo .
echo Example: %0 MC
:end
```

Your build batch file does not need to be nearly this complex. It can be as simple as:

```
@echo off
del TRECK.LIB
call build bcl86cc
call buildlib bc86lib
```

You then use this file to build the TCP/IP library to link with your application, kernel, and device driver. This way you do not need to call build and buildlib from the DOS command prompt.

Step 4 - Creating an RTOS/Kernel Interface

This section describes how to setup the RTOS/Kernel Interface. Treck Real-Time TCP/IP is designed so that it can use nearly any Real Time Operating System. If you do not have an RTOS, then you will still need to read this section so that you can understand how the “No RTOS” version of the kernel interface is put together. There are already pre-configured RTOS/kernel Interfaces. These are located below the SOURCE directory in the following locations:

RTOS	Processor Mfg.	Processor Type	Compiler	Location
uC/OS	Motorola	CPU32	SDS	source\kernel\ucos \motorola\cpu32\sds
uC/OS	Motorola	CPU32	MRI	source\kernel\ucos \motorola\cpu32\mri
uC/OS	Motorola	PowerPC	Diab	source\kernel\ucos \motorola\ppc\diab
uC/OS	Intel	x86 Real	Borland v4.5/5.0	source\kernel\ucos\ intelx86\real\borland
uC/OS	Intel	x86 Real	Microsoft v6.0	source\kernel\ucos intelx86\real\msc60
AMX	Motorola	CPU32	SDS	source\kernel\amx\cj
AMX	Intel	x86 Real	Borland	source\kernel\amx\aj
ThreadX	Motorola	PowerPC	Green Hills	source\kernel\motorola\ threadx\ppc\ghs

The file name for the kernel interface is normally named as follows:
trXXX.c. Where XXX is the name of the kernel. For example the uC/OS interface is named trucos.c. In each of these files you will find the RTOS/kernel interface functions. You will find the TM_FAR macro used in these interface functions. This macro is defined to be far on the Intel processor running in real/small model. You will also notice that these files all include `TRSOCKET.H`. This single file contains all of the types that you will need to build the RTOS/Kernel interface.

The kernel interface is broken down into the following elements.

- Initialization
- Memory Allocation and Free
- Critical Section Handling
- Error Logging
- Warning Information Logging
- Task Suspend and Resume
- ISR Interface

Initialization

The function ***tfKernelInitialize*** is used to initialize the Treck/kernel interface. It is not needed for all RTOS/Kernels. The system calls this function prior to any other kernel calls. You should think of this function as a way to get things initialized for your environment. If this function is not needed, it should be a stub function.

Example:

```
void tfKernelInitialize (void)
{
/* Initialize the Treck to Kernel Interface */
    return;
}
```

Memory Allocation and Free

The functions ***tfKernelMalloc*** and ***tfKernelFree*** are used to allocate and free memory that is to be used by the Treck protocol stack. They are the same definition as the ANSI malloc and free functions. We have provided these functions to give you the flexibility to use the memory allocation that you want for your environment. You can use your RTOS malloc and free. These functions are used to pass memory blocks to the Treck Memory System. Unless you have disabled the dynamic memory feature, the Treck Memory System will not free these blocks unless they are larger than the Treck Memory System keeps track of. By definition, there will be more mallocs than frees, however, as time progresses there will be fewer and fewer mallocs.

Examples:

```
void TM_FAR *tfKernelMalloc (unsigned size)
{
    return(malloc(size));
}

void tfKernelFree (void TM_FAR *memoryBlock)
{
    free(memoryBlock);
}
```

Treck Simple Heap

TM_USE_SHEAP

If the user defines this macro, the stack allocates its blocks of memory from the Treck simple heap static array using the simple heap allocation routine, ***tfSheapMalloc***, instead of calling the RTOS ***tfKernelMalloc***. Similarly the Treck stack will release an allocated block of memory (of size bigger than 4096 bytes) by calling the Treck simple heap free routine ***tfSheapFree***, instead of calling the RTOS ***tfKernelFree***. The user will define this macro if the user's RTOS does not provide heap management routines, or if the user does not want the Treck stack to allocate its blocks of memory from the RTOS heap.

TM_SHEAP_SIZE

If you do define TM_USE_SHEAP, then you also have to define TM_SHEAP_SIZE, to give the size in bytes of the Treck simple heap array.

TM_SHEAP_MIN_BLOCK_SIZE 4

Minimum block size in bytes when using the simple heap, either when allocating a new block, or when refragmenting a freed block.
If not defined here, default value is 4.

TM_DYNAMIC_CREATE_SHEAP

Normally, the Treck simple heap is implemented as a static array. However, you can override this behavior by defining the macro TM_DYNAMIC_CREATE_SHEAP in your trsystem.h file, in which case you decide how you want to implement it (i.e. dynamic memory allocation specific to your RTOS) by customizing the API ***tfKernelSheapCreate***.
If TM_DYNAMIC_CREATE_SHEAP is defined then ***tfKernelSheapCreate*** will be called once per page. By default there is one page of memory, but that can be changed as shown in the TM_SHEAP_NUM_PAGE section below.

TM_SHEAP_NUM_PAGE 1

Number of memory pages. Instead of a unique array, the simple heap can be divided into several ones. Uncomment TM_SHEAP_NUM_PAGE and indicate the number of pages if you are using the simple heap with a system that has paged memory and that one page is too small to hold the whole simple heap memory (static or preallocated by ***tfKernelSheapCreate***).
In this case the simple heap will be divided in several non-contiguous pages. If you do define TM_SHEAP_NUM_PAGE to a number bigger than 1, then you will likely need to define TM_DYNAMIC_CREATE_SHEAP also. The default for TM_SHEAP_NUM_PAGE is 1. Each page of memory must have the same size. The default page size is TM_SHEAP_SIZE/ TM_SHEAP_NUM_PAGE.

If the page size is lower or equal to 4Kb (TM_BUF_Q6_SIZE), you cannot use dynamic memory and must define TM_DISABLE_DYNAMIC_MEMORY.

tfKernelSheapCreate

This finction dynamically allocates the Treck simple heap. **tfKernelSheapCreate** is only called when TM_DYNAMIC_CREATE_SHEAP has been #define'd. The size of the simple heap allocated must be TM_SHEAP_SIZE, with the start of the simple heap (i.e. the pointer returned) being aligned on a 32-bit boundary. Please see **tfKernelSheapCreate** in the programmer's reference section of this manual.

Critical Section Handling

The functions ***tfKernelSetCritical*** and ***tfKernelReleaseCritical*** are used to set critical sections around code that is NOT reentrant. The critical sections are used to prevent ANY other calls into the protocol stack. These are used to prevent one task from updating a pointer while another task is accessing it. Without critical sections, there is the chance of corrupting pointers if a pointer update is interrupted by another task or an ISR. The Treck protocol stack is designed to keep these critical sections to a minimum (usually about five assembly instructions).

Example for an Intel x86:

```
void tfKernelSetCritical (void)
{
    _asm(“cli”);
}

void tfKernelReleaseCritical (void)
{
    _asm(“sti”);
}
```

TIP: If you can guarantee that no preemption of Treck code will occur by another task or ISR calling a TurboTreck function (this includes the tfNotifyInterfaceIsr function), then the critical sections are not needed. It is recommended that you continue to use the critical sections unless you are absolutely sure that no preemption of Treck code will occur.

You can also in-line these functions by defining tm_kernel_set_critical and tm_kernel_release_critical to be the assembly for these functions to save the extra function call in the TRSYSTEM.H file. You can find examples in TRMACRO.H. This will enhance the data transfer speed.

Error Logging

The function ***tfKernelError*** is used to report unrecoverable error messages and cause a restart of the system. Errors that are reported include corrupt pointers and invalid memory. The Treck protocol stack *must not* continue once this function has been called.

Example:

```
void  tfKernelError (char TM_FAR *functionName,
                    char TM_FAR *errorMessage)
{
    printf("Fatal Error in Function %s: %s \n",
           functionName, errorMessage);
    while(1)
    {
/*
 * LOOP Forever until the watchdog timer fires and
 * reboots
 */
        ;
    }
}
```

Warning Information Logging

The function ***tfKernelWarning*** is used to convey to the user any abnormalities that occur during the normal operation of the Treck protocol stack. These would include such items as bad checksums. These are not fatal errors and the stack should continue to operate normally. This function is provided to help the user diagnose network problems.

Example:

```
void tfKernelWarning (char TM_FAR *functionName,
                      char TM_FAR *warningMessage)
{
    printf("Warning in Function %s: %s \n",
           functionName, warningMessage);
}
```

Task Suspend and Resume

This set of functions is the most complex portion of the RTOS/Kernel Interface. For task suspend and remove, we rely on a primitive that is available in most RTOS/Kernels. This primitive is a counting semaphore with an associated counter initialized to zero. The features that we are looking for in the primitive are:

- It can be posted to before it is pended on
- Only one task waiting on the primitive will be re-scheduled when a post occurs
- The primitive is not tied to a specific task

Note: If your RTOS does not provide a counting semaphore, but does have an event flag, please use the implementation provided in *kernel\trcousem.c****, or*** *kernel\trctsem2.c****. Follow the instructions given in the file you picked, and in "Appendix B".***

An example of why we need this feature is as follows:

1) The user calls the socket API recv in blocking mode.
2) There is no data to receive so the Treck stack starts to call the pend routine.
3) A context switch occurs for a receive task, before the user task finished the pend call.
4) The receive task processes the incoming packet and queues it to the socket the user had opened, and calls post.
5) A context switch then occurs back to the user task that will then finish calling pend.

Without the ability to post before the pend, the user task calling recv will wait forever. The other option is to call pend in a critical section (which is VERY undesirable for a real-time system). A counting semaphore with an associated counter initialized to zero solves this problem, so we use this primitive for task suspend and resume.

We must first look at the data type that is used to carry the counting semaphore. For this we use a union of all basic types. Because we use this to hold the counting semaphore, we are able to integrate to any RTOS/Kernel. This union is defined in the file TRSOCKET.H, as follows:

```
typedef union tuUserGenericUnion
{
    unsigned long   gen32bitParm;
    long            genSlongParm;
    void TM_FAR    *genVoidParmPtr;
    unsigned short  gen16BitParm;
    unsigned int    genUintParm;
    int             genIntParm;
    unsigned char   gen8BitParm;
    char            genCharParm;
} ttUserGenericUnion;
```

By using the ***ttUserGenericUnion*** data type, you will be able to pass your counting semaphore between Treck protocols and your RTOS. A set of three functions provides access between the Treck protocol stack and your RTOS. These functions are used to create, pend on, and post on a counting semaphore that will be used by the protocol stack. Because a task cannot pend more than once at any given time, and because the Treck stack re-uses unused counting semaphores dynamically, the maximum number of counting semaphores that will be needed would be equal to the number of tasks that are calling Treck. The RTOS functions below are only examples. You will need to map these calls into your own RTOS. If you are not using an RTOS, then these functions should be stubs.

The ***tfKernelCreateCountSem*** function is used to create a counting semaphore that is provided by the operating system with an associated counter initialized to zero. If the counting semaphores are created at compile time with your operating system, then you simply pass one of these back to Treck. An example of this function is as follows:

```
int tfKernelCreateCountSem (ttUserGenericUnionPtr
                            countingSemaphore)
{
    int retCode;
/* Create and Initialize the semaphore to zero */
    countingSemaphore->genVoidParmPtr = RTOSSemCreate(0);
    if (countingSemaphore->genVoidParmPtr != (void TM_FAR *)0)
    {
        retCode = 0;
    }
    else
    {
        retCode = -1;
    }
    return(retCode);
}
```

The function ***tfKernelPendCountSem*** is used to pend or wait on a counting semaphore. This function must wait indefinitely until the post occurs. We pass back in the counting semaphore in the union that was used for the create. An example of this is as follows:

```
int tfKernelPendCountSem (ttUserGenericUnionPtr countingSemaphore)
{
    int retCode;

    if((RTOSSemPend(countingSemaphore>genVoidParmPtr)) ==
                                                       RTOS_NO_ERR)
    {
        retCode = 0;
    }
    else
    {
        retCode = -1;
    }
    return(retCode);
}
```

The function ***tfKernelPostCountSem*** is used to wake up another task or thread that has called ***tfKernelPendCountSem***. An example of this function is as follows:

```
int tfKernelPostCountSem (ttUserGenericUnionPtr countingSemaphore)
{
    int retCode;
    if ((RTOSSemPost(countingSemaphore->genVoidParmPtr)) ==
                                                       RTOS_NO_ERR)
    {
        retCode = 0;
    }
    else
    {
        retCode = -1;
    }
    return(retCode);
}
```

ISR Interface

The ISR interface is used to provide an interface for device drivers (as device drivers are the only items that need an interrupt service routine). In the ISR interface, we need to be able to install an interrupt service routine and provide a mechanism to notify the protocol stack about ISR events.

The function ***tfKernelInstallIsrHandler*** is used to install an ISR handler. Device drivers written by Elmic USA normally call this function. It is not called directly by the Treck protocol stack. If you do not use an Elmic device driver, then you will not need this function in your interface. If you are using an Elmic driver, then this function is used to install the device drivers interrupt handler. Our drivers do not do any packet processing within the ISR. They simply notify task level routines that an event has occurred.

```
void tfKernelInstallIsrHandler(ttUserIsrHandlerPtr funcPtr,
                               unsigned long       offSet)
{
/*
 * funcPtr is the Function To Be Called as the Handler
 * offSet is the offset into the Interrupt Vector Table
 */
    RTOSInstallISR((void TM_FAR *)funcPtr, offSet);
}
```

Device Interface Routines

The set of four functions ***tfKernelCreateEvent***, ***tfKernelPendEvent***, ***tfKernelIsrPostEvent***, ***tfKernelTaskPostEvent*** are used internally by the Treck device interface routines. They are only needed if the user wishes to use, and to block in, the receive task, or the transmit task, or the send complete task. ***tfKernelTaskYield*** is only used if the user uses a trasnmit task, and wishes to call the Treck function ***tfInterfaceSpinLock*** inside the device driver send function to let the other tasks run, while waiting for the device driver to be ready to transmit.

No RTOS or event scheduler with main loop:

If you do not have any RTOS, or if you have an event scheduler with a main loop implementation, then you have neither defined TM_TRECK_PREEMPTIVE_KERNEL, nor TM_TRECK_NONPREMTIVE_KERNEL in your *trsystem.h*. In that case, then the first four functions are not needed, because they will not be called, since in that case the user is not allowed to block.

Preemptive or non-preemptive kernel:

If you are using a preemptive kernel, or a non-preemptive kernel, then you have defined either TM_TRECK_PREEMPTIVE_KERNEL or TM_TRECK_NON_PREEMPTIVE_KERNEL in your *trsystem.h.* In that case, you will need some or all of these first 4 functions, as shown in the following table, depending on which of the following additional macros you have defined in your *trsystem.h*.

TM_TASK_RECV	TM_TASK_XMIT	TM_TASK_SEND
tfKernelCreateEvent	tfKernelCreateEvent	tfKernelCreateEvent
tfKernelPendEvent	tfKernelPendEvent	tfKernelPendEvent
tfKernelIsrPostEvent		tfKernelIsrPostEvent
	tfKernelTaskPostEvent	

Note that if you have not defined any of these macros, you do not need to provide any of these 4 functions, since they will not be called.

tfKernelCreateEvent

The function ***tfKernelCreateEvent*** is used to create a counting semaphore, or binary semaphore, of event flag. It is only called from task level. One event will be created per interface, and per TM_TASK_XXXX macro defined. For example, if you have configured 2 interfaces, and have defined TM_TASK_RECV, TM_TASK_XMIT, and TM_TASK_SEND, then you will need 6 events. At most one task will be waiting on that kernel primitive.

```
void tfKernelCreateEvent (ttUserGenericUnionPtr eventPtr)
{
    void *semaphorePtr;

 /* Initialize the semaphore to zero */
    semaphorePtr = RTOSSemCreate(0);
    if (semaphorePtr != (void *)0)
    {
        eventPtr->genVoidParmPtr = (void *)semaphorePtr;
    }
    else
    {
        tfKernelError (“tfKernelCreateEvent”,
                      “Unable to Create Semaphore”);
        tm_thread_stop;
    }
}
```

tfKernelPendEvent

The function ***tfKernelPendEvent*** is used to pend on an event that will be posted to. The task calling this function must wait indefinitely while waiting for the post event call. It is only called from task level, either from ***tfWaitReceiveInterface***, or from ***tfWaitXmitInterface***, or from ***tfWaitSentInterface***.

```
/*
 * Wait on an Event
 */
void tfKernelPendEvent (ttUserGenericUnionPtr eventPtr)
{
    int kernelError;

    RTOSSemPend(eventPtr->genVoidParmPtr, 0, &kernelError);
    if (kernelError != RTOS_NO_ERR)
    {
        tfKernelError (“tfKernelPendEvent”,
                      “Unable to Pend on Semaphore”);
        tm_thread_stop;
    }
}
```

tfKernelIsrPostEvent

The function ***tfKernelIsrPostEvent*** is used to signal that an event has occurred and resume tasks that are waiting via ***tfKernelPendEvent***. This is called from ***tfNotifyInterfaceIsr***, itself called from an interrupt handler.

Note: Most RTOS have special calls or wrappers when a system call is called from an interrupt handler. Please consult your RTOS manual for the proper way to post from an interrupt handler.

```
void tfKernelIsrPostEvent (ttUserGenericUnionPtr eventPtr)
{
    int kernelError;

    kernelError = RTOSSemPost(eventPtr->genVoidParmPtr);
    if (kernelError != RTOS_NO_ERR)
    {
        tfKernelError (“tfKernelIsrPostEvent”,
                      “Unable to Post on Semaphore”);
    }
}
```

tfKernelTaskPostEvent

The function ***tfKernelTaskPostEvent*** is used to signal that an event has occurred and resume the transmit task waiting via ***tfKernelPendEvent***. This call is only called from the Treck stack in the context of a task. This call is needed if the user has defined TM_TASK_XMIT macro in *trsytem.h,* or
if the user calls ***tfNotifyInterfaceTask*** and has defined either TM_TASK_RECV or TM_TASK_SEND.

```
void tfKernelTaskPostEvent (ttUserGenericUnionPtr eventPtr)
{
    int kernelError;

    kernelError = RTOSSemPost(eventPtr->genVoidParmPtr);
    if (kernelError != RTOS_NO_ERR)
    {
        tfKernelError (“tfKernelPostEvent”,
                      “Unable to Post on Semaphore”);
    }
}
```

tfKernelTaskYield

The function ***tfKernelTaskYield*** is used to yield the CPU, and let other tasks run. This function is called only if the user uses a transmit task, and calls the Treck function ***tfInterfaceSpinLock*** from within the device driver send function to let other tasks run and access the device driver routines, while waiting for the device driver to be ready to transmit.

```
void tfKernelTaskYield (void)
{
     RTOSYield ();
}
```

This completes the RTOS interface. After you have made these functions map onto your RTOS, you are ready to move onto the next phase. We suggest that you compile the code that you have written in this phase.

Step 5 - Hooking in the Timer

Now that you have compiled the library and created an RTOS interface, you are ready to consider how you must hook in the timer. Notice that the timer interface is not part of the RTOS interface. This is because the timer interface can be hooked up in different ways. These different methods do not depend on the RTOS interface. There are two different methods to hook up the timer. You must not call any timer functions prior to calling *tfStartTreck*.

Method 1: Use a Timer Task to Update and Execute the Timers.

Method 2: Use a Timer ISR to Update the Timers, and Execute from either a main line loop or a task.

These methods are described in more detail below:

Method 1: A Timer Task to Update and Execute Timers

In this model, you will have a simple loop in a separate timer task. In this loop you will update the timers and execute the timers that have expired. You will then pause the task for some fixed interval. This interval MUST match the tick count that you have setup for the system (see trsystem.h or *tfInitTreckOptions*). This is the most popular method to use when you have a real time operating system. An example of this method is as follows:

```
#include <trsocket.h>
void timerTask(void)
{
    while(1)
    {
/* Update the Timers */
        tfTimerUpdate ();
/* Execute the Timers */
        tfTimerExecute ();
/* Call the RTOS to delay for 1 clock tick */
        RTOSTimeDelay(1);
    }
}
```

If your RTOS does not allow you to start a task at run-time, then you should use a simple flag or semaphore to prevent the calls to the timer prior to calling ***tfStartTreck***.

An example of this is as follows:

```
#include <trsocket.h>
static int treckStarted=0;

void mainTask(void)
{
    int errCode;
    errCode=tfStartTreck ()
    if (errCode=TM_ENOERROR)
    {
        treckStarted=1;
/* Other initialization code follows ... */
    }
}


void timerTask(void)
{
    while(1)
    {
/* Check to make sure that Treck has started */
        if (treckStarted)
        {
/* Update the Timers */
            tfTimerUpdate ();
/* Execute the Timers */
            tfTimerExecute ();
        }
/* Call the RTOS to delay for 1 clock tick */
        RTOSTimeDelay(1);
    }
}
```

Tip: A good timer interval is between 10ms and 100ms. It is not necessary that the timer be VERY accurate between calls to tfTimerUpdate, but it should be accurate on average over time. It is usually best to set the timer task to the highest priority of tasks operating with the protocol stack, but it is not necessary.

Method 2: A Timer ISR to Update Timers and Execute from Either a Main Line Loop or a Task.

In this method, you simply update the timers from an ISR routine using the call ***tfTimerUpdateIsr***. Note that this is a different call than from the call made at task level (***tfTimerUpdate***). This call is designed to be able to be called from an interrupt handler. It is not recommended to use this method if you have an RTOS. You must NOT call the function ***tfTimerUpdateIsr*** before you call the function **tfStartTreck**. In this regard, it has the same rules as ***tfTimerUpdate***. An example of using this method for a main line loop is as follows:

```
#include <trsocket.h>
static int treckStarted=0;

void timerIsrHandler(void)
{
/* Check to make sure that Treck has started */
    if (treckStarted)
    {
/* Update the Timers */
        tfTimerUpdateIsr ();
    }
}

void main(void)
{
    int errCode;
    errCode=tfStartTreck ();
    treckStarted=1;
/* Other initialization code follows ... */
    if (errCode == TM_ENOERROR)
    {
        while(1)
        {
            tfTimerExecute ();
/* Application code follows ... */
        }
    }
}
```

Tip: A good timer interval is between 10ms and 100ms. It is not necessary that the timer be VERY accurate between calls to tfTimerUpdateIsr, but it should be accurate on average over time.

Step 6- Key Things to Start Using Treck

Now that you have a library and an idea of how to build your timer interface, you are probably wondering how to start using the protocols on your platform. You should at this point, try to use the loopback support built into the protocol stack. This will let you know if there is a problem with the RTOS or the Timer interface. There is one call that you **MUST** make into the protocol stack before you can start using it. That function is called ***tfStartTreck***. If you call this function (which does not take any parameters), it will initialize all the stack parameters to defaults, if you do not call ***tfInitTreckOptions*** first. If you have not set a tick length value in trsystem.h or with ***tfInitTreckOptions***, you will get a kernel warning. You will get a kernel error, if you compile for the wrong byte order of your machine (big endian versus little endian). **The only call that you can make into the protocol stack, prior to calling *tfStartTreck*, is *tfInitTreckOptions*.** All of the functions mentioned here could be found in the "Programmers Reference" section of the manual. As we mentioned above you can use the function ***tfInitTreckOptions*** to change any of the default parameters used for the stack. It is not recommend that you change any parameters at this point, as the stack should work fine with the defaults that are set up.

The following is an example of how easy it is to start using the protocol stack.

```
#include <trsocket.h>

void main(void)
{
/* Define the variables that I need to Startup */
    int             errorCode;
    int             sd;

/* Start the protocol stack */
    errorCode=tfStartTreck ();

    if (errorCode == TM_ENOERROR)
    {
/* Now do my loopback code */
        sd=socket (PF_INET,SOCK_DGRAM,0);
                        .
                        .
                        .
```

Step 7 - Testing the New Library with a Loopback Test

Now you are ready to start using Treck protocols in loopback mode. The loopback address is defined as 127.0.0.1. You should setup a simple client/server to make sure that packets traverse all the way down the stack and back up. The loopback driver sits below TCP/IP so it is a good test to see if you have things working thus far. *If you are not familiar with "sockets" programming, then you should review the section titled "Introduction to BSD Sockets". Other good reference material on this subject would include, "Introduction to TCP/IP Volume 3 (BSD Edition)" by Douglas E. Comer, and "UNIX Network Programming Volume 1", by W. Richard Stevens*. After you have completed the loopback test, you are ready to move onto the next section.

The reason that we suggest that you complete the loopback test first is to reduce the amount of items that you are attempting to debug. It is VERY difficult to debug your RTOS interface, Timer interface, Device Driver, and Application all at the same time. From our experience, we have found that most people have trouble with the device driver (because it is where the software meets the hardware). If you test first with loopback, you can isolate your debugging to the RTOS, Timer, and Loopback application code first and debug the device driver last. One interesting item about Treck protocols is that you do not need to recompile the protocol stack in order to add your device driver later, so performing this task should not be too much of a burden. If you are not familiar with sockets programming, the loopback interface is a great place to learn.

Tip: You should not attempt to add your device driver until you are confident that the stack is working correctly in loopback mode.

We have included an example of using the loopback interface. It is not intended to be an example of using the protocol stack for high performance. It is written to be easy to understand and follow. Please feel free to use this code as a guide in writing your loopback test program.

Example Loop Back Application:

```
#include <trsocket.h>
#include <stdio.h> /* for printf */

#define TM_PORT_LOOPBACK_TEST htons (10)

char bufferArray[1024];

/*
 * Example of a UDP socket loop back send and receive.
 */
void main (void)
{
    int                errorCode;
    int                failed;
    struct sockaddr_in addr;
    unsigned long      ipAddr;
    int                sd;
    int                len;
    int                recvdLength;
    int                sentLength;

    failed = 0;
    sd = TM_SOCKET_ERROR;
    recvdLength = TM_SOCKET_ERROR;
    sentLength = TM_SOCKET_ERROR;
    ipAddr = 0UL;
/* Amount of data to send, and receive */
    len = sizeof(bufferArray);
/* Example setting the tick length at 10 milliseconds */
    errorCode = tfInitTreckOptions (TM_OPTION_TICK_LENGTH,
                                    (unsigned long)10);
    if (errorCode != TM_ENOERROR)
    {
        printf(“tfInitTreckOptions failed %d\n”,
                errorCode);
        failed = 1;
    }
    if (failed == 0)
    {
/* Start the Treck initialization */
        errorCode = tfStartTreck ();
    }
    if (errorCode != TM_ENOERROR && failed == 0)
    {
        printf(“tfStartTreck failed %d\n”,
                errorCode);
        failed = 1;
    }
    if (failed == 0)
    {
/* Open a UDP socket */
        sd = socket (PF_INET, SOCK_DGRAM, IP_PROTOUDP);
    }
    if ((sd == TM_SOCKET_ERROR) && (failed == 0))
    {
/* Retrieve the socket error for failed socket call */
        errorCode = tfGetSocketError (sd);
        printf(“socket failed %d\n”,
               errorCode);
        failed = 1;
    }
    if (failed == 0)
    {
/* Bind the UDP socket to a well known UDP port */
        addr.sin_family = PF_INET;
        addr.sin_port = TM_PORT_LOOPBACK_TEST;
        addr.sin_addr.s_addr = 0;
        errorCode = bind (sd,
                          (struct sockaddr TM_FAR *)&addr,
                          sizeof(struct sockaddr_in));
    }
    if ((errorCode == TM_SOCKET_ERROR) && (failed == 0))
    {
/* Retrieve the socket error for failed bind call */
        errorCode = tfGetSocketError (sd);
        printf(“bind failed %d\n”,
                errorCode);
        failed = 1;
    }
    if (failed == 0)
    {
/* Send some loop back data to our own socket */
        addr.sin_addr.s_addr = inet_addr (“127.0.0.1”);
        sentLength = sendto (sd,
                             bufferArray,
                             len,
                             0,
                             (struct sockaddr TM_FAR *)&addr,
                             sizeof(struct sockaddr_in));
        if ((sentLength == TM_SOCKET_ERROR) && (failed == 0))
        {
/* Retrieve the socket error for failed sendto call */
            errorCode = tfGetSocketError (sd);
            printf(“sendto failed %d\n”,
                   errorCode);
            failed = 1;
        }
        if (failed == 0)
        {
            printf(“%d sent successfully\n”, sentLength);
/* Receive some loop back data from our own socket */
            recvdLength = recvfrom (sd,
                                    bufferArray,
                                    len,
                                    0,
                                    (struct sockaddr TM_FAR *)0,
                                    0);
        }
        if ((recvdLength == TM_SOCKET_ERROR) && (failed == 0))
        {
/* Retrieve the socket error for failed recvfrom call */
            errorCode = tfGetSocketError (sd);
            printf(“recvfrom failed %d\n”,
                   errorCode);
            failed = 1;
        }
        if (failed == 0)
        {
            printf(“%d received succesfully\n”, recvdLength);
        }
    }
    if (sd != TM_SOCKET_ERROR)
    {
/* All done. Close the socket */
        (void)tfClose (sd);
        if (failed == 0)
        {
            printf(“UDP LOOP BACK Test success\n”);
        }
    }
}
```

Step 8 - Using Ethernet or PPP

Now that you are comfortable that the stack and interfaces to it are working correctly, you are ready to include the link layer that you wish to use. For this operation, you will need to define a handle to store the desired link layer into. We call this a link layer handle and it is defined as a ***ttUserLinkLayer*** type. Adding (or using) a link layer is a simple process. You simply call link layer USE function to pull in the appropriate link layer from the library. These functions are called ***tfUseEthernet***, ***tfUseAsyncPpp***, ***tfUseAsyncServerPpp***, and ***tfUseNullLinkLayer***. The null link layer is used to pass raw IP datagrams directly to/from the device driver interface. This way you can use the device driver interface to hookup a new link layer. In the base TCP/IP package, you will receive both the Ethernet and Null link layers.

Examples of this follow:

```
{
    ttUserLinkLayer ethernetHandle;
    ttUserLinkLayer pppServerHandle;
    ttUserLinkLayer pppClientHandle;
    ttUserLinkLayer nullLinkHandle;

/* Select the Ethernet DIX protocol */
    ethernetHandle=tfUseEthernet ();

/* Select the PPP server Link Layer Protocol */
    pppServerHandle =
            tfUseAsyncServerPpp (notifyServerFuncPtr);

/* Select the PPP Client Link Layer Protocol */
    pppClientHandle =
            tfUseAsyncPpp (notifyClientFuncPtr);

/* Select the NULL Link Layer */
    nullLinkHandle=tfUseNullLinkLayer();
}
```

Note: PPP is an optional protocol, and if you did not purchase PPP and attempt to call tfUseAsyncPpp or tfUseAyncServerPpp functions, you will get an error.

Step 9 - Adding a New Device Driver

This step is the most complicated because the device driver interface is so versatile. The device driver interface is the same for all link layers (including PPP). You will find existing device drivers in the drivers directory. If we have provided a device driver, chances are that you will still need to modify it for your hardware platform. Device drivers can almost never be moved between different hardware platform types without modification.

To allow you to hook up your driver in many different ways, we provide these additional functions that you would call from your driver and task level to allow for a receive task, transmit task, and send complete task or to poll the device driver. The use of these functions is optional (depending on your environment):

- tfNotifyInterfaceIsr
- tfNotifyInterfaceTask (may be used in addition to tfNotifyInterfaceIsr)
- tfCheckReceiveInterface
- tfInterfaceSetOptions
- tfCheckXmitInterface
- tfCheckSendInterface
- tfWaitReceiveInterface (only if TM_TASK_RECV is defined)
- tfWaitXmitInterface (only if TM_TASK_XMIT is defined)
- tfWaitSentInterface (only if TM_TASK_SEND is defined)

tfRecvInterface is used to move received data into the stack, and ***tfSendCompleteInterface*** is used to notify the stack that data has been sent. The use of these functions is required:

- tfRecvInterface
- tfSendCompleteInterface

tfXmitInterface, and ***tfInterfaceSpinLock*** are used only if the user uses a separate transmit task to send packets in the context of the transmit task, and to let the separate transmit task yield the CPU from the device driver send function while waiting for the device to be ready to transmit.

- tfXmitInterface
- tfInterfaceSpinLock

tfInterfaceSetOptions is used by the user to set interface specific options, such as turning on a transmit task for that interface, or making ***tfRecvInterface*** copy device driver buffers whose size are below a specified option threshold., as outlined below

- tfInterfaceSetOptions

tfUseInterfaceXmitQueue, and ***tfIoctlInterface*** are called by the user when he wishes to use a transmit queue to queue the buffers to a Treck device driver transmit buffer queue, as opposed to using the overhead of a transmit task. Note that the 2 methods are exclusive. You need to decide whether you want to use a transmit queue, or a transmit task, or none of the above. In addition, you cannot use a device driver transmit queue for a serial device interface (i.e with SLIP, or PPP).

- tfUseInterfaceXmitQueue (only for non serial devices, and with no transmit task)
- tfIoctlInterface

Here is a summary of functions that you may have to write in your device driver:

- driverOpen (Optional)
- driverClose (Optional)
- driverIoctl (Optional)
- driverGetPhysicalAddress (Only for Ethernet)
- driverSend
- driverReceive
- driverFreeReceiveBuffer (Optional)
- driverIsrHandler (Optional)

For Ethernet device drivers, The Treck stack provides the following additional optional functions:

- tfGetEthernetBuffer
- tfFreeDriverBuffer

For all device drivers, the Treck stack provides the following additional functions:

- tfGetDriverBuffer (More general than tfGetEthernetBuffer)
- tfFreeDriverBuffer

Some Ethernet chips (for example the Crystal LAN (cs8900)) won't dismiss a receive interrupt, until the data is copied. If this is the case, you will need to get a buffer from within the ISR, to copy the data into. The Treck stack does provide a set of tools to allow you to get a Treck buffer from within the ISR:

- tfPoolCreate
- tfPoolIsrGetBuffer
- tfPoolReceive
- tfPoolDelete

Note: Calling tfPoolIsrGetBuffer() effectively removes a Treck buffer from the available pool of buffers. A call to tfRecvInterface() will return that buffer to the pool, but only if the flag TM_POOL_REFILL_IN_LINE was specified in the call to tfPoolCreate() when allocating the pool. Otherwise, all of the Treck pool buffers will get used up, and your device driver won't be able to receive any more packets. You can also refill the Treck buffer pool by periodically calling tfIoctlInterface() and specifying the flag TM_DEV_IOCTL_REFILL_POOL_FLAG. It is recommended that when using the Treck buffer pool, you use both methods to refill the pool.

Tip: *You will only be using a subset of the functions listed on this page. You will find that the hardest part about the device driver having to read and interpret the device data sheet. A key thing to remember here is that it is easier than it looks. Just take it one step at a time, and before you know it you will have your device driver up and running!*

tfNotifyInterfaceIsr, tfNotifyInterfaceTask, tfCheckReceiveInterface, tfWaitReceiveInterface, and tfRecvInterface Functions

This group of functions is used to allow the user to notify the stack that a receive complete event has occurred, so that the event can be processed at main/ task level instead of from the ISR. In other words, the first four functions are used to tell the user when to call ***tfRecvInterface***. ***tfRecvInterface*** **may *NOT* be called directly from an interrupt handler**. The ***tfNotifyInterfaceIsr*** function is used to notify the stack from an ISR that there is data waiting to be received. The ***tfNotifyInterfaceTask*** function is used to notify the stack from a task that there is data waiting to be received. Usually, only ***tfNotifyInterfaceIsr*** is needed. On some occasions, there is a need to let the stack know that there is more data to be received in the context of a task (instead of an ISR), and this is the reason ***tfNotifyInterfaceTask*** is provided. The ***tfCheckReceiveInterface*** function is used to poll the device layer for receive events. The ***tfWaitReceiveInterface*** function is used to "block" a task until the event has occurred. You must choose if you want a task to block until data has been received or poll to see if data has been received. The choice that you make defines which of the functions (***tfWaitReceiveInterface*** or

tfCheckReceiveInterface) that you will use.

Note: tfNotifyInterfaceIsr (and/or tfNotifyInterfaceTask) need to be used whether the user chooses to use tfCheckReceiveInterface (polling method), or tfWaitReceiveInterface (pending method). If you do not have an RTOS you cannot use fWaitReceiveInterface. If you do have an RTOS, you can use tfWaitReceiveInterface, but only if you define the TM_TASK_RECV macro in **trsystem.h.**

Note: Our device driver interface does NOT allow you to process packets directly from an ISR. The major reason that we do not allow this is because our stack is designed for real-time systems. Under real-time constraints, we need to keep interrupt latency to a minimum.

It is not required to use the functions ***tfNotifyInterfaceIsr***, (or/and ***tfNotifyInterfaceTask***), ***tfCheckReceiveInterface*** and ***tfWaitReceiveInterface*** as long as you can guarantee that the ***tfRecvInterface*** call is made when there are packets to be received from the device driver. Remember, ***tfRecvInterface*** may not be called from an interrupt handler.

An example of using these calls for receive processing from a separate task is as follows:

```
void recvTask(void)
{
    while(1)
    {
/* Wait for the receive event from an ISR */
        tfWaitReceiveInterface (myInterfaceHandle);
/* Move the data into the stack */
        tfRecvInterface (myInterfaceHandle);
    }
}
```

An example of using these calls for receive processing from a main line loop is as follows:

```
void main(void)
{
   ttUserInterface interfaceHandle;
   int             errorCode;

   errorCode = tfStartTreck();
   treckStarted = 1;
/* Other main processing, like adding, and opening an interface */
                    .
                    .
                    .
    for (;;)
   {
/* Check if Treck timers have expired */
        tfTimerExecute();
/* Check for received packets */
        if (tfCheckReceiveInterface(myInterfaceHandle) ==
            TM_ENOERROR)
        {
/*
 * Call the stack to move the data from the driver
 * and process it
 */
            tfRecvInterface (myInterfaceHandle);
        }
/* Application code */
    }
}
```

An example ISR handler for both of the methods would look something like this:

```
deviceIsrHandler(void)
{
    int receivedPacketCount;
/*
 * Store number of packets ready to be received in
 * receivedPacketCount.
 */
/*
 * Notify the stack that data is waiting to be
 * processed
 */
    tfNotifyInterfaceIsr(myInterfaceHandle,
                         receivedPacketCount, 0, 0, 0UL);
}
```

tfRecvInterface, and tfInterfaceSetOptions functions
The Treck ***tfRecvInterface*** function will process incoming data. If incoming data is a PING echo request, this function will generate, and send the reply in the context of the receive task. If incoming data is for a user application socket, then the ***tfRecvInterface*** function will process the incoming data, and then will queue the buffer (containing the user application data) in the application socket receive queue. The problem is that on a non-serial device, the device driver receive buffers will be tied up in the socket receive queue, until the application actually ***recv*** the data. Typically a device driver will have pre-allocated maximum size Ethernet packet, and if there is little data in each Ethernet frame, this could end up consuming a lot of memory. To fix this, two techniques are used:

1. *Copy at the socket receive queue level:*
If there is no data in the application socket receive queue, the device driver buffer will be queued. If there is a buffer already queued in the application socket receive queue, by default the Treck stack will attempt to copy to a new buffer (for UDP), or to append to the previous buffer (for TCP), and free the device driver buffer. By default the Treck stack will attempt to do so only if the ratio of the allocated block size to the buffer size is at or above 4 (i.e if the data uses less than, or 25 % of the allocated block). However, if the device driver recv function gives back a driver buffer, which is not a Treck buffer, then the Treck stack won't know the block allocation size. In that case, the Treck stack

will assume that the block allocation size is the same as the data size, and no copy will take place, unless the user changes the default ratio to 1. The user can change this ratio, using the per socket ***setsockopt*** function. For example to change the ratio to 2

```
int optionValue;
optionValue = 2;
errorCode = setsockopt(socketDescriptor, SOL_SOCKET, TM_SO_RCVCOPY,
                       &optionValue, sizeof(int));
```

2. *Copy inside tfRecvInterface before processing the data (disallowed for PPP or SLIP interfaces):*

The user can set an interface option, so that the device driver buffer can get copied into a new buffer, if the device driver buffer data size is below a configurable threshold. To set that option to 64 bytes for example:

```
short optionValue;
optionValue = 64; /* Copy only if size is below or at 64 bytes */
errorCode = tfInterfaceSetOptions(myInterfaceHandle,
                                  TM_DEV_OPTIONS_RECV_COPY,
                                  &optionValue,
                                  sizeof(short));
```

Note: This option is not allowed on a PPP, or SLIP interface, since PPP, or SLIP will copy the buffer into a new buffer while processing the incoming data. So there is no need to copy the data an extra time.

tfInterfaceSetOptions, tfCheckXmitInterface, tfWaitXmitInterface, tfXmitInterface, and tfInterfaceSpinLock Functions

This group of functions is used to allow synchronization between the user application sending threads, and a separate Treck user transmit task. ***They are not needed if the user does not wish to use a dedicated TurboTreck user transmit task to send data to the device driver***. The ***tfCheckXmitInterface***, or ***tfWaitXmitInterface*** function is used to tell the user when to call ***tfXmitInterface.*** The ***tfCheckXmitInterface*** function is used to poll for packets queued to the device send queue. The ***tfWaitXmitInterface*** function is used to “block” the transmit task until one or more packets have been queued to the device send queue. You must choose if you want a task to block until, or to poll to see if a packet has been queued to the send queue. The choice that you make defines which of the functions (***tfWaitXmitInterface*** or ***tfCheckXmitInterface***) that you will use.

Note: After having initialized the stack, the user need to call tfInterfaceSetOptions once with the TM_DEV_OPTIONS_XMIT_TASK option, to let the Treck stack know that a separate transmit task is used, whether the user chooses to use tfCheckXmitInterface (polling method), or tfWaitXmitInterface (pending method). If you do not have an RTOS you cannot use tfWaitXmitInterface. If you do have an RTOS, you can use tfWaitXmitInterface but only if you define the TM_TASK_XMIT macro in trsystem.h.

The user will call the ***tfXmitInterface*** function, from the Treck user transmit task to send the next packet in the device send queue to the device driver. The device driver send is only called by ***tfXmitInterface***, and thereby in the context of the Treck user transmit task. The user will call ***tfInterfaceSpinLock*** from his device driver send in a loop while waiting for the device to be ready to transmit the buffer that the Treck stack wishes to send. ***tfInterfaceSpinLock*** will release the Treck device driver lock, and yield the CPU, allowing other tasks to run and allowing, for example, the recv task to get the Treck device driver lock, and therefore allowing the recv task to access the device driver recv routine. Note that ***tfInterfaceSpinLock*** can only be called from the device driver send, and only when a transmit task is used.

An example of using these calls to transmit packets from a separate task is as follows:

```
ttUserInterface myInterfaceHandle;

 void main(void)
{
 int              errorCode;
   short            optionValue;

   errorCode = tfStartTreck();
   treckStarted = 1;
/* Other main processing, like adding an interface */
                     .
                     .

   if (errorCode == TM_ENOERROR)
   {
       optionValue = 1; /* turn option on */
       errorCode = tfInterfaceSetOptions(myInterfaceHandle,
                                         TM_DEV_OPTIONS_XMIT_TASK,
                                         &optionValue,
                                         sizeof(short));
   }
/* Start a timer task */
/* Start the receive task for my interface handle */
/* Start the transmit task for my interface handle */
                     .
                     .
/* Other main processing, like opening the interface */
                     .
                     .
}
```

```
void xmitTask(void)
{
    while(1)
    {
/* Wait for the post from the Treck stack */
        tfWaitXmitInterface (myInterfaceHandle);
/* Move the data from the stack to the device driver send
 * function
 */
        tfXmitInterface (myInterfaceHandle);
    }
}
```

An example of using these calls to transmit the packets from a main line loop is as follows:

```
void main(void)
{
   ttUserInterface interfaceHandle;
   int             errorCode;
   short           optionValue;

   errorCode = tfStartTreck();
   treckStarted = 1;
/* Other main processing, like adding an interface */
                     .
                     .
   if (errorCode == TM_ENOERROR)
   {
       optionValue = 1; /* turn option on */
       errorCode = tfInterfaceSetOptions(interfaceHandle,
                                         TM_DEV_OPTIONS_XMIT_TASK,
                                         &optionValue,
                                         sizeof(short));
   }
/*
 * Other main processing, like installing a timer ISR to make
 * sure that tfTimerUpdateIsr is called periodically, and opening
 * the interface.
 */
                     .
                     .
    for (;;)
   {
/* Check if Treck timers have expired */
        tfTimerExecute();
/* Check for received packets */
        if (tfCheckReceiveInterface (myInterfaceHandle) ==
            TM_ENOERROR)
        {
/*
 * Call the stack to move the data from the driver
 * and process it
 */
            tfRecvInterface (myInterfaceHandle);
        }

/* Check for packets to send */
        if (tfCheckXmitInterface (myInterfaceHandle) ==
            TM_ENOERROR)
        {
/*
 * Call the stack to move the data from the Treck stack to
 * the device driver send function.
 */
            tfXmitInterface (myInterfaceHandle);
        }
    }
}
```

tfUseInterfaceXmitQueue, tfIoctlInterface Functions (only for non serial devices)

If you do not want the overhead of a transmit task to send data to the device driver, then you can use a Treck device transmit queue interface. This method allows the driver to return an error, in case the chip is not ready to transmit. In that case a pointer to the buffer that could not be transmitted (along with its length, and flag) is stored in an empty slot, in the Treck device transmit queue. The device transmit queue should be big enough to hold pointers to all the buffers that will be sent by the application. The size of the data sent by an application is limited by the socket send queue size. So, an interface transmit queue should be big enough to hold pointers to buffers sent from all the application sockets through that particular interface. The space allocation overhead for a x entries device transmit queue is 12 + x * 8 bytes. So for a device transmit queue containing 1000 slots, the overhead is 8012 bytes.

What happens if the interface transmit queue is not big enough?

If the device transmit queue is not big enough to hold all the buffer pointers that the device driver could not send, the Treck stack will drop the packets corresponding to the buffers that could not be queued. If the user uses the device driver scattered send capability, some Ethernet frames could therefore be partially sent. In that case, the Treck stack will ensure that at least a minimum Ethernet size packet will be sent.

Re-transmission of buffers that have been queued to the device transmit queue.

If a new buffer is being sent, and the device transmit queue is not empty, the Treck stack will first try and empty the device transmit queue. If it fails to empty it completely, then it will try to queue the current buffer to the device transmit queue, storing the buffer pointer, data length and flag in the next available slot at the end of the Treck device transmit queue. Also when ***fSendCompleteInterface*** is called, the Treck stack will try and empty the device transmit queue, but only if ***tfSendCompleteInterface*** is not called from the device driver send function to avoid recursion. Also the user should call

tfIoctlInterface periodically with the TM_DEV_IOCTL_EMPTY_XMIT_FLAG flag to try and flush the Treck device transmit queue.

Note: Using a Treck device transmit queue is not allowed for a SLIP or PPP serial device interface, because the SLIP or PPP link layer functions send data to the device driver in a unique per interface buffer. Since the same per interface buffer is being re-used, a pointer to it cannot be kept in the device transmit queue.

Example on using a device transmit queue of a 1000 entries for a given interface:

```
void main(void)
{
   ttUserInterface interfaceHandle;
   int             errorCode;
   short           optionValue;

   errorCode = tfStartTreck();
   treckStarted = 1;
/* Other main processing, like adding an interface */
                     .
                     .

   if (errorCode == TM_ENOERROR)
   {
       errorCode = tfUseInterfaceXmitQueue(interfaceHandle, 1000);
   }
/* Other main processing, like opening the interface */
                     .
                     .
}
```

Also every time ***tfTimerExecute*** is called, then the ***tfIoctlInterface*** function should be called as follows:

```
errorCode = tfIoctlInterface(interfaceHandle,
                             TM_DEV_IOCTL_EMPTY_XMIT_FLAG,
                             (void TM_FAR *)0, 0);
```

	Advantages	Disadvantages
Transmit Task	Calling the device driver send is done in the context of a separate transmit task, not in the context of the user application, or the recv task, or the timer task. The transmit task can wait inside the device driver send function for the device to be ready to transmit, and can call tfInterfaceSpinLock, allowing other tasks to access the other device driver functions. For example the recv task could access the driver recv function.	In blocking mode, extra context switch on every packet sent.In polling mode, extra processing to check on packets ready to be sent.
Tranmit Queue	The device driver send can return an error if it is not ready to transmit the current buffer. Hence the sending thread (i.e user application task, or recv task, or timer task) does not have to wait inside the device driver send function for the device to be ready to transmit.	Extra memory required.One extra function call required in the send path.Cannot be used with a SLIP or PPP interface.
No Transit Task, No Transmit Queue	No overhead.	The sending thread (i.e user application task, or recv task, or timer task) has to wait in a busy loop inside the device driver send function for the device to be ready to transmit. This is minimal with an Ethernet device. While the sending thread is waiting, no other task can access any other device driver function. In particular the recv task could not access the driver recv function.

tfNotifyInterfaceIsr, tfCheckSentInterface, tfWaitSentInterface, and tfSendCompleteInterface Functions

This group of functions is used to allow the user to notify the stack that a send complete event has occurred, so that the event can be processed at main/task level instead of from the ISR. In other words, the first three functions are used to tell the user when to call ***tfSendCompleteInterface***. ***tfSendCompleteInterface*** may *not* be called directly from an interrupt handler. The ***tfNotifyInterfaceIsr*** function is used to notify the stack from an ISR that the data buffer is not in use by the device driver anymore. The ***tfCheckSentInterface*** function is used to poll the device layer for "send complete" notify events. The ***fWaitSentInterface*** function is used to "block" a task until the send complete event has been notified. You must choose if you want a task to block until data has been sent or poll to see if data has been sent. The choice that you make defines which of the functions (***tfCheckSentInterface*** or ***tfWaitSentInterface***) that you will use.

Note: tfNotifyInterfaceIsr need to be used whether the user chooses to use tfCheckSentInterface (polling method), or tfWaitSentInterface (pending method). If you do not have an RTOS you cannot use fWaitSentInterface. If you do have an RTOS, you can use tfWaitSentInterface, but only if you define the TM_TASK_SEND macro in trsystem.h.

Tip: It is not required to use the functions tfNotifytInterfaceIsr, tfCheckSentInterface and tfWaitSentInterface as long as you can guarantee that the tfSendCompleteInterface call is made when the device driver has sent the data. Remember that tfSendCompleteInterface may not be called from an interrupt handler.

Now let's look at send complete processing. Send complete means that the sending frame is not in use by the driver anymore. Here is send complete processing from a separate task:

```
void sendCompleteTask(void)
{
    while(1)
    {
/* Wait for the send complete event from an ISR */
        tfWaitSentInterface (myInterfaceHandle);
/*
 * Tell the stack that the driver is done with the
 * current frame
 */
        tfSendCompleteInterface(myInterfaceHandle,
                                TM_DEV_SEND_COMPLETE_APP);
    }
}
```

An example of using these calls for send complete processing from a main line loop is as follows:

```
void main(void)
{
/*
 * Other main processing, like initialization, adding, opening an
 * interface.
 */
    for (;;)
    {
/* Other main processing, like timer, recv */
                   .
/* Check for sent packets */
        if (tfCheckSentInterface (myInterfaceHandle) ==
            TM_ENOERROR)
        {
/*
 * Call the stack to tell it that it owns the frame
 * now
 */
            tfSendCompleteInterface (myInterfaceHandle,
                                     TM_DEV_SEND_COMPLETE_APP);
        }
   }
}
```

An example ISR handler for both of the methods would look something like this:

```
void deviceIsrHandler(void)
{
   int           receivedPacketCount;
   int           sendCompletePacketCount
   unsigned long totalBytesSent;

/*
 * Store number of packets ready to by received in
 * receivedPacketCount.
 */
/* Store number of send complete packets in sendCompletePacketCount
 * and store total number of bytes sent in totalBytesSent.
/*
 * Notify the stack that data is waiting to be
 * processed, and that the driver has transmitted
 * some complete frames.
 */
    tfNotifyInterfaceIsr(myInterfaceHandle,
                         receivedPacketCount,
                         sendCompletePacketCount,
                         totalBytesSent,
                         0UL);
}
```

Our experience tells us that it is usually best NOT to use a send complete task that is the highest priority. The reason is that a send complete task will cause a context switch on every sent packet. Let's look at a couple of methods to avoid using the send complete task.

Methods to avoid using a send complete task.
When one of the two methods outlined here is used, then tfNotifyInterfaceIsr, tfCheckSentInterface, or tfWaitSentInterface need not be called.

Method 1: Copy the Data.
This method is less efficient than the extra context switch on large packets. For drivers that use I/O ports to communicate with the chip, the data is already being copied to the data to the I/O port inside the device driver send function anyways. An example of this method is as follows:

```
int driverSend(ttUserInterface interfaceHandle,
               char TM_FAR    *dataPtr,
               int             dataLength,
               int             flag)
{
    while (dataLength)
    {
/* Send to I/O Port */
        outb(*dataPtr++);
        dataLength–;
    }
/* We are done with the data pointer */
    if (flag==TM_USER_BUFFER_LAST)
    {
        tfSendCompleteInterface (interfaceHandle,
                                 TM_DEV_SEND_COMPLETE_DRIVER);
    }
    return TM_DEV_OKAY;
}
```

Method 2: In-line the Send Complete Call
In this method, in your device driver send function, look for previous sent packets that have completed transmission, and call send complete for those packets. This method is a little trickier. One must be careful when using this method to make sure to also look for sent packets periodically (outside of the driver send call), in the driver ioctl function. Otherwise, if the Treck stack creates packets faster than the chip can process them, then a deadlock condition can occur, where the send complete never gets called. The *trquicc.c* driver contains this method.

Device Driver Functions that You May Need to Provide

In this section, we will describe and provide examples for the functions that you may need to provide. In all of these functions, you can be guaranteed of single threaded access to them (provided that you do not call any of these functions directly). We use the internal Treck locking system to provide this facility. Because of this, you should not need any critical sections in the device driver code, except to protect data area that you set, or access in the ISR. Success in all of these calls is returned to the stack via the macro TM_DEV_OKAY while failure is indicated by TM_DEV_ERROR.

1. deviceOpen

The stack calls this optional routine to initialize the hardware (and optionally install the ISR handler. If your driver requires pre-allocated buffers to be given to the chip, you could pre-allocate your buffers in this function. If your hardware initializes correctly, then you should return a success value (TM_DEV_OKAY), otherwise you should return TM_DEV_ERROR. This routine is optional if you perform your hardware initialization elsewhere. Example:

```
int myDeviceOpen(ttUserInterface interfaceHandle)
{
/* Initialize the Hardware */
                        .
                        .
                        .
/* Install the ISR Handler Routine */
    tfKernelInstallIsrHandler(myHandlerFunctionPtr,
                              myHandlerIsrLocation);
    return (TM_DEV_OKAY);
}
```

In addition, some chips (for example the Crystal LAN (cs8900)) won't dismiss a receive interrupt, until the data is copied. If this is the case, you must get a buffer from within the ISR. The Treck stack provides a set of tools that allow you to get a Treck buffer from within the ISR. If you want to use these Treck tools, you must to call **tfPoolCreate** in the **deviceOpen** function. For example: we pre-allocate a pool of ten 128-bytes small buffers, and we pre-allocate a pool of five 1518-byte big buffers. The TM_POOL_REFILL_IN_LINE flag, indicates that we want the buffers to be re-allocated in the receive task thread.

```
#define TM_ID_RECV_BIG_BUFFERS      5
#define TM_ID_RECV_SMALL_BUFFERS    10
#define TM_ID_SMALL_BUFFER_SIZE     128
```

```
int myDeviceOpen(ttUserInterface interfaceHandle)
{
   int errorCode;

/* Initialize the Hardware */
                        .
                        .
                        .
/* Install the ISR Handler Routine */
    tfKernelInstallIsrHandler(myHandlerFunctionPtr,
                              myHandlerIsrLocation);
/* Example where we pre-allocate a pool of 10 128-bytes
 * small buffers, and 5 maximum Ethernet size big buffers.
 * The TM_POOL_REFILL_IN_LINE flag, indicates that we want
 * the buffers to be re-allocated in the receive task thread.
 */
    errorCode = tfPoolCreate( interfaceHandle,
                              TM_ID_RECV_BIG_BUFFERS,
                              TM_ID_RECV_SMALL_BUFFERS,
                              TM_ETHER_MAX_PACKET_CRC,
                              TM_ID_SMALL_BUFFER_SIZE,
                              0,
                              TM_POOL_REFILL_IN_LINE);
    return errorCode;
}
```

2. deviceClose

This optional routine is called by the stack, to turn off Transmit and Receive. It can also be used to remove the ISR. Most people do not need a close for ethernet since the device typically stays connected. It is very useful for PPP devices.

Example:

```
int myDeviceClose(ttUserInterface interfaceHandle)
{
/* Turn off Ethernet reception, Disable Ethernet transmission,
 * Uninstall the ISR handler.
 */
    return (TM_DEV_OKAY);
}
```

If you had called **tfPoolCreate** in your **deviceOpen** function, then you will need to call **tfPoolDelete** in your **deviceClose** function.

Example:

```
int myDeviceClose(ttUserInterface interfaceHandle)
{
   int errorCode;
/* Turn off Ethernet reception, Disable Ethernet transmission,
 * Uninstall the ISR handler.
 */

    errorCode = tfPoolDelete(interfaceHandle);
    return errorCode;
 }
```

3. driverIoctl

This optional routine is used as a pass through routine in the stack in most cases. It is normally only called when the user calls ***tfIoctlInterface***. The flags that are passed to ***tfIoctlInterface*** are the same as those passed to ***driverIoctl***. The only exception to this rule is when multicast support is needed. In this case, the Treck stack calls this function with Treck reserved flags: TM_DEV_SET_MCAST_LIST to give the list of multicast addresses the Ethernet chip should receive, or TM_DEV_SET_ALL_MCAST so that the Ethernet chip receives all multicast addresses. ***Normally this routine is used to periodically refresh the receive pool and perform send completes***.

Note: that flag values bigger than, or equal to 0x1000 are reserved by the Treck stack.

Example:

```
int myDeviceIoctl(ttUserInterface interfaceHandle,
                  int             flag,
                  void    TM_FAR *optionPtr,
                  int             optionLen)
{
    int errorCode;
    switch (flag)
    {
        case REFILL_RECEIVE:
            myDriverReceiveRefill();
            errorCode=TM_DEV_OKAY;
            break;
        default:
            errorCode=TM_DEV_ERROR;
            break;
    }
    return (errorCode);
}
```

If you are using the Treck stack ISR pool functions, i.e., if you had called **tfPoolCreate** in your **deviceOpen** function, then you need to call the following function periodically:

```
errorCode =tfIoctlInterface(interfaceHandle,
                            TM_DEV_IOCTL_REFILL_POOL_FLAG,
                            (void TM_FAR *)0,
                            0);
```

Note that in that case ***tfIoctlInterface*** will not call your device driver ioctl function. The pool refill will be done internally by the Treck stack.

4. driverGetPhysicalAddress

This routine is used to return the physical address of the device to the stack. Currently it is only called for Ethernet devices. ***The protocol stack needs the physical address to formulate an Ethernet frame.***

Example:

```
int myDeviceGetPhyAddr(ttUserInterface interfaceHandle,
                       char  TM_FAR *  physicalAddress)
{
/*
 * Get the physical address from hardware or firmware
 * Save it in Network Byte Order
 */
   tfMemCpy ( physicalAddress, deviceAddress,
            TM_ETHERNET_PHY_ADDR_LEN);
   return(TM_DEV_OKAY);
}
```

5. driverSend

This routine is used to send the data out the network device. Since Treck protocols support “scatter send” for devices, there is a flag that gets passed into the device driver send routine. There are two possible values for this flag:

TM_USER_BUFFER_MORE
TM_USER_BUFFER_LAST

The flag value TM_USER_BUFFER_MORE means that there is more data to follow for this frame.

The flag value TM_USER_BUFFER_LAST means that this is the last piece of this frame.

If your device does not support “scatter send”, then the flag will always be set to TM_USER_BUFFER_LAST with the exception of PPP or Slip serial devices.

Ethernet devices

The Treck Ethernet, or Treck Null link layer code, will only send scattered data if the device supports "scatter send". If the device supports "scatter send", the only Ethernet scattered data frames sent by the Treck stack are TCP packets. An Ethernet device driver does not need to copy the Treck data, and can keep a pointer to it. The data will not be freed until ***tfSendCompleteInterface*** is called later on by the user, when the user knows that the packet has been transmitted.

tfSendCompleteInterface Notes

tfSendCompleteInterface may be called only once per complete frame which is denoted by the TM_USER_BUFFER_LAST flag. Please note that this function will operate on the frames in the order that they were delivered to your driver send routine. If your device driver send function returns an error for a given buffer, which has the TM_USER_BUFFER_LAST flag set, then you do not need to call ***tfSendCompleteInterface*** for that buffer. In that case, if a Treck device transmit queue is used, the Treck stack will try and queue the buffer to the device transmit queue, and try to send it later. If it fails to queue the buffer to the device transmit queue, or if there is no device transmit queue, then the Treck stack will remove the corresponding packet from the send queue, and free it.

PPP or SLIP serial devices

The Treck PPP link layer code, and Treck SLIP link layer code will send scattered data to SLIP or PPP serial devices. This is because of the PPP asynchronous byte stuffing or because of SLIP escaping special characters. The Treck PPP link layer code, and Treck SLIP link layer code will copy the stuffed bytes, or escaped bytes, along with the packet bytes, into a single intermediate buffer. That single intermediate buffer will be repeatedly sent to the driver when it is full, or when the end of the packet has been reached. By default, the intermediate buffer size is one byte. Note that the Treck PPP link layer code, or Treck SLIP link layer code, will re-use the same intermediate buffer, so in a serial device driver send, you need to copy the buffer data immediately. If your serial device driver can handle more than one byte at a time, you can change the size of the intermediate buffer being sent, with ***tfPppSetOption*** for a PPP link layer, or ***tfSlipSetOptions*** for a SLIP link layer. You can change the size of the SLIP intermediate buffer at any time, and the change will take effect immediately. But you need to change the size of the PPP intermediate buffer before opening the interface.

For example to change the intermediate SLIP send buffer size to 1500, you can call:

```
unsigned short optionValue;
optionValue = 1500;
errorCode = tfSlipSetOptions(interfaceHandle,
                             TM_SLIP_OPT_SEND_BUF_SIZE,
                             (void TM_FAR *)&optionValue,
                             sizeof(unsigned short));
```

To change the intermediate PPP send buffer size to 1500 for example, you need to call (before calling ***tfOpenInterface***):

```
unsigned short optionValue;
optionValue = 1500;
errorCode = tfPppSetOption(interfaceHandle,
                           TM_PPP_PROTOCOL, 0
                           TM_PPP_SEND_BUFFER_SIZE,
                           (const char TM_FAR *)&optionValue,
                           sizeof(unsigned short));
```

Note: You must only call send completes for the piece of data that has the TM_USER_BUFFER_LAST flag set to guarantee that only ONE send complete per frame is issued!

Example:

```
int myDeviceSend(ttUserInterface interfaceHandle,
                   char TM_FAR  *dataPtr,
                   int           dataLength,
                   int           flag)
{
    while (dataLength)
    {
/* Send to I/O Port */
        outb(MY_DEVICE_PORT, *dataPtr++);
        dataLength–;
    }

/* We are done with the frame */
    if (flag == TM_USER_BUFFER_LAST)
    {
         tfSendCompleteInterface (interfaceHandle,
                                  TM_DEV_SEND_COMPLETE_DRIVER);
    }
    return TM_DEV_OKAY;
}
```

6. driverReceive

In this routine, a received packet is passed back into the protocol stack. The stack calls this routine to retrieve a frame from the device driver. You can use one of our buffers to store the data from the driver into, or you can use your own buffer.

Special Ethernet Considerations for the driverReceive routine:
You MUST return an entire frame, as the stack does not support "Gather Read".

On RISC processors, you should be careful that the IP header will be on a four byte boundary. Since the Ethernet header is 14 bytes long, this implies that the start of the Ethernet buffer should be on a 2-bytes boundary, but not 4-byte boundary. If you use our function ***tfGetEthernetBuffer*** you are guaranteed that this will be the case.

If you decide not to use ***tfGetEthernetBuffer*** with an Ethernet device, you should (for optimum performance) make sure the Ethernet buffer is aligned on a two-byte (NOT FOUR BYTE) boundary.

Ethernet Header Offsets (When Long Word Aligned)

0	6	12	14
Destination Addr	Source Addr	Type	IP Header

Notice that the IP Header does not start on a long word boundary. This would not work on most RISC processors. On other processors this may result in poor performance, however, processors such as the M68EN360 require that the received data appear on a long word boundary. This is the why we subtract 2 bytes from the pointer returned by ***tfGetEthernetBuffer*** in our quicc driver.

Ethernet Header Offsets (When Short Word Aligned)

2	8	14	16
Destination Addr	Source Addr	Type	IP Header

tfGetEthernetBuffer does this short word alignment for you. If you do not use ***tfGetEthernetBuffer*** and allocate your own buffers, then you should verify that they are short word aligned.

We will show examples of using your own buffer or using ***tfGetEthernetBuffer***.

Special PPP Considerations:
For PPP you do not need to pass an entire frame back to the protocol stack. You simply pass as much or as little as you wish, as the PPP Link Layer will determine the framing automatically.

An Example of using tfGetEthernetBuffer to create a buffer to store the incoming data into:

```
int myDeviceReceive(ttUserInterface       interfaceHandle,
                    char TM_FAR * TM_FAR *dataPtr,
                    int TM_FAR           *dataLength,
                    ttUserBufferPtr       bufHandlePtr)
{
    int errorCode;
/* Get a buffer to store the data into */
    *dataPtr=tfGetEthernetBuffer (bufHandlePtr);
    if (*dataPtr == (char TM_FAR *)0)
    {
/* No memory so return an error */
        errorCode=TM_DEV_ERROR;
    }
    else
    {
/* Copy from the device driver into the buffer */
        tfMemCpy (*dataPtr,deviceRecvDataPtr,
                 deviceDataLength);
/* Save the length */
        *dataLength=deviceDataLength;
/* Good return value */
        errorCode=TM_DEV_OKAY;
    }
    return(errorCode);
}
```

An Example of using the driver's buffer to receive a frame and passing it directly to the protocol stack:

Notice that the *bufHandlePtr is NULL. This is how the stack knows who created the buffer being received.

```
int myDeviceReceive(ttUserInterface       interfaceHandle,
                    char TM_FAR * TM_FAR *dataPtr,
                    int TM_FAR           *dataLength,
                    ttUserBufferPtr       bufHandlePtr)
{
/* Save the pointer to the beginning of the data */
    *dataPtr=deviceRecvDataPtr;
/* Save the length */
    *dataLength=deviceDataLength;
/* (IMPORTANT) NULL OUT THE BUFFER HANDLE */
    *bufHandlePtr=(ttUserBufferPtr)0;
    return(TM_DEV_OKAY);
}
```

Tip: Typically the user calls tfGetEthernetBuffer, or tfGetDriverBuffer routine to pre-allocate the receive buffers to receive into. This way the driver (if it is capable), can store the data directly into a protocol stack buffer. This eliminates the extra call to the driver to free the receive buffer that the driver passed it. The trquicc.c driver contains this method.

Special case when the user uses the Treck pool to be able to get a Treck buffer, and to copy the received data to it within the receive ISR.
Recall that in that case, the user has called ***tfPoolCreate*** in the device driver open function. The driver receive function is very simple. The driver receive function could either be ***tfPoolReceive*** (i.e be the driver receive function pointer parameter in ***tfAddInterface***), or the driver receive function could call ***tfPoolReceive***. For example:

```
int myDeviceReceive(ttUserInterface       interfaceHandle,
                    char TM_FAR * TM_FAR *dataPtr,
                    int TM_FAR           *dataLength,
                    ttUserBufferPtr       bufHandlePtr)
{
       return tfPoolReceive(interfaceHandle,
                            dataPtr,
                            dataLength,
                            bufHandlePtr);
}
```

7. driverFreeReceiveBuffer

This function is used ONLY if you use your own buffer allocation for the received packets (i.e. do not use ***tfGetEthernetBuffer***, nor ***tfGetDriverBuffer***, nor the Treck stack recv ISR pool functions). Since you pass a buffer to the stack (the stack is zero copy), you will need to know when the buffer is not in use anymore. When this happens, we call your ***driverFreeReceiveBuffer*** function to let you free or reuse the buffer.

Example:

```
int myDeviceFreeReceiveBuffer(
                   ttUserInterface interfaceHandle,
                   char TM_FAR    *dataPtr)
{
/* Free the Data here */
    return(TM_DEV_OKAY);
}
```

8. driverIsrHandler

The interrupt service routine only need to call the ***tfNotifyInterfaceIsr*** function, if you use the Check or Wait functions described earlier. Normally you need to dismiss the interrupt and notify the stack of the event that occurred. The only call that you can make from a device interrupt handler into the protocol stack is one ***tfNotifyInterfaceIsr*** function.

Tip: you should call tfNotifyInterfaceIsr only once per ISR, because some RTOS on some CPU's will re-enable interrupt when posting on an event, therefore causing our counter update in tfNotifyInterfaceIsr to become non-re-entrant.

Example:

```
void myDeviceIsrHandler(void)
{
              int recvPacketCount;
              int sendCompletePacketCount;
    unsigned long totalBytesSent;

    recvPacketCount = 0;
    sendCompletePacketCount = 0;
    totalBytesSent = 0;
/* Check for receive interrupt */
    if (receivedData)
    {
/* Accumulate number of packets ready to be received */
        recvPacketCount++;
    }
/* Check for send complete interrupt */
    if (sendComplete)
    {
 /*Accumulate number of packets that have been transmitted */
        sendCompletePacketCount++;
/*Accumulate sent packet data sizes */
        totalBytesSent += packetDataSize;
    }
/* Call this function only once */
    tfNotifyInterfaceIsr(myInterfaceHandle,
                         receivedPacketCount,
                         sendCompletePacketCount,
                         totalBytesSent,
                         0UL);

/* Dismiss the interrupt */
}
```

Special case when the user uses the Treck pool to be able to get a Treck buffer, and to copy the received data to it within the receive ISR.

Recall that in that case, the user has called ***tfPoolCreate*** in the device driver open function. The user calls ***tfPoolIsrGetBuffer*** inside the ISR, to get a pre-allocated Treck buffer from the Treck pool, so that the incoming network data can be copied inside the ISR.

Example:

```
/* myInterfaceHandle initialized in deviceOpen */
static ttUserInterface myInterfaceHandle;

void myDeviceIsrHandler(void)
{
              int recvPacketCount;
              int recvPacketLength;
              int sendCompletePacketCount;
    unsigned long totalBytesSent;
    char TM_FAR * dataPtr;

    recvPacketCount = 0;
    sendCompletePacketCount = 0;
    totalBytesSent = 0;
/* Check for receive interrupt */
    if (receivedData)
    {
/* Retrieve the packet length into recvPacketLength */
/* Get a Treck buffer from the ISR */

        dataPtr = tfPoolIsrGetBuffer( myInterfaceHandle,
                                      recvPacketLength);
        if (dataPtr != (char TM_FAR *)0)
        {
/* Copy the data into the Treck buffer pointed to by dataPtr */
/* Accumulate number of packets ready to be received */
           recvPacketCount++;
        }
        else
        {
/* Copy the data into a scratch buffer */
        }
    }
/* Check for send complete interrupt */
    if (sendComplete)
    {
 /*Accumulate number of packets that have been transmitted */
        sendCompletePacketCount++;
/*Accumulate sent packet data sizes */
        totalBytesSent += packetDataSize;
    }
/* Call this function only once */
    tfNotifyInterfaceIsr(myInterfaceHandle,
                         receivedPacketCount,
                         sendCompletePacketCount,
                         totalBytesSent,
                         0UL);

/* Dismiss the interrupt */
}
```

Further Device Driver Modifications to allow a device driver to be shared by several Ethernet Interfaces

Modify your device driver as follows, to allow a device driver to be shared by several Ethernet inter

Device Driver API	**Where Used**
int tfDeviceStorePointer (ttUserInterface interfaceHandle, ttvoidPtr deviceDriverPtr);	Device driver open function
int tfDeviceStorePointer (ttUseInterface interfaceHandle);	Device driver close function
ttVoidPtr ttDeviceGetPointer (ttUserInterface interfaceHandle);	Any device driver function

Summary of Device Driver API’s that are provided to allow a device driver to be shared by several Ethernet interfaces

Device driver open function

First, make sure that you move all device driver local variables to a structure. In the open function, allocate such a structure, and give the pointer to the Treck TCP/IP stack, so that it can be stored on the interface, i.e.,

```
errorCode = tfDeviceStorePointer(interfaceHandle,
deviceDriverPointer );
```

Device driver close function

In the device driver close function, call **tfDeviceClearPointer** to dissociate the device driver structure from the interface handle.

```
deviceDriverPointer = tfDeviceClearPointer(interfaceHandle);
```

Then, if the returned deviceDriverPointer is non-null, free the allocated structure pointed to by deviceDriverPointer.

Any device driver function

When the device driver needs access to local structure, **tfDeviceGetPointer** should be called. Given an interface handle, **tfDeviceGetPointer** will retrieve the pointer to the device driver structure:

```
deviceDriverPointer = tfDeviceGetPointer(interfaceHandle);
```

It will return a non-zero pointer on success.

Alternatively the following macro can be used to retrieve the pointer:

```
deviceDriverPointer =
tm_device_get_pointer(interfaceHandle);
```

Device driver ISR Handler

The user should keep a global mapping between an interrupt vector, and corresponding interface handle so that the user can retrieve the corresponding device driver pointer, with the **tfDeviceGetPointer** API.

Adding and Configuring your New Device Driver

Now that you have your device driver code, you need to tell the protocol stack about it. There are two calls to inform the stack about your driver. These calls should be made after you have called ***tfStartTreck*** and before any sockets calls. These two calls are comprised of an add (***tfAddInterface***) and open (***tfOpenInterface***).

The following example shows how to inform the protocol stack of the new device driver.

```
{
/* Location to store Link Layer Handle into */
    ttUserLinkLayer     ethernetLinkLayer;

/* Location to save Interface Handle into */
    ttUserInterface myInterfaceHandle;

    ethernetLinkLayer=tfUseEthernet ();

    myInterfaceHandle = tfAddInterface (
/* name of the device */
                  “MYDEVICE.001”,
/* Link Layer to use */
                    ethernetLinkLayer,
/* Open Function */
                    myDeviceOpen,
/* Close Function */
                    myDeviceClose,
/* Send Function */
                    myDeviceSend,
/* Receive Function */
                    myDeviceReceive,
/* Free a Receive Buffer Function */
                    myDeviceFreeReceiveBuffer,
/* IOCTL Function */
                    myDeviceIoctl,
/* Get Physical Address Function */
                    myDeviceGetPhysicalAddress,
/* INT to store error (if one is returned */
                    &errorCode);
/* Now open/configure the device */
    errorCode = tfOpenInterface (

/* The handle from tfAddInterface */
                myInterfaceHandle,
/* Our IP Address */
                inet_addr (“192.1.1.2”),
/* Out Netmask (Super or subnet) */
                inet_addr (“255.255.255.0”),
/* Special Flags Enable Scatter Send */
                TM_SCATTER_SEND_ENB,
/* Max buffers per frame */
                5 )
}
```

tfAddInterface Notes

For any functions that you did not implement for your device driver (because they were not needed), you can pass a NULL pointer into ***tfAddInterface*** instead of having a stub routine.

tfOpenInterface Notes

Note: tfConfigInterface has been deprecated. Please use tfOpenInterface. tfConfigInterface will still function in your code, and may be used to configure additional IP addresses on the same interface (multi homing).

Scattered send

Note that the flag TM_SCATTER_SEND_ENB is used to inform the protocol stack that the device can support "Scatter Send". If we support scatter send, we have to tell the stack what is the maximum number of pieces that the driver can handle in scatter send mode. If you do not use Scatter Send, then you can pass a 0 flags value and set Max buffers per frame to be one (1).

What if I do not know my IP address / netmask, and want to retrieve them from the net?

You can use the Treck stack BOOTP, or DHCP protocols.

Note: Please, refer to the BOOTP, or DHCP section in "Appendix B", and to BOOTP, or DHCP function reference calls in "Appendix A".

What if I do not know my IP address / netmask, and want to retrieve them from the net, but do not wish to use the *Treck* BOOTP or DHCP protocols?

To be able to open an interface without setting an IP address in the routing table, call ***tfOpenInterface***, using the TM_DEV_IP_USER_BOOT flag as follows:

```
 errorCode = tfOpenInterface( interfaceHandle,
                              0UL,
                              UL,
                              TM_DEV_IP_USER_BOOT,
                              1);
```

Note that if you device driver support scatter send, you can OR that flag to the TM_DEV_IP_USER_BOOT flag, and change the last parameter accordingly. You can then open a socket, and try and send data through that interface, calling the function ***tfSendToInterface***. ***tfSendToInterface*** is identical to ***sendto***, but takes 2 extra arguments: the interfaceHandle as returned by ***tfAddInterface***, and a multi home index, 0 in our case:

```
toAddress.sin_addr.s_addr = 0xFFFFFFFFUL;
len = tfSendToInterface( desc, buf, 512, 0,
                         (struct sockaddr TM_FAR *)(&toAddress),
                         sizeof(struct sockaddr),
                         interfaceHandle, 0);
```

You can also use the regular ***recvfrom*** on that socket. Once you have retrieved your IP address, and netmask from the net, you can insert those values in the device and routing table using the ***tfFinishOpenInterface*** function:

```
errorCode = tfFinishOpenInterface(interfaceHandle, IPAddr, mask);
```

Note: For more information on functions used in this section, please refer to the "Programmers Reference" section of this manual.

Single Send Call Send per Frame, Out of Order Send

Description

Single call to the driver send per scattered frame

By default, when the stack sends a frame scattered among different buffers, and the device driver supports scattered send, the stack will make multiple calls to the device driver send function (one per scattered buffer). This is inefficient. This section describes additional APIs that have been added to the stack in order to support a single call to the device driver send API per frame, even when sending scattered data.

Out of Order Frame Transmission

Also, because the user might want to order frames for transmission in a different order as transmitted by the stack, the user might want to signal out of order frame send completion. This is not allowed with the default device driver send interface, because the frame handle is not given to the user as a parameter, and therefore cannot be given as a parameter to **tfSendCompleteInterface**.

The modified device driver send API, described in this section, now takes the frame handle as a parameter. A new API (**tfSendCompletePacketInterface**) has been added to allow the user to specify which frame has been transmitted. So even if the user device driver does not support scattered send, a user might still want to use the modified device driver send API described in this section, if the user needs to signal out of order frame send completion.

Note: The modified driver send interface is not supported for point to point link layers (such as PPP, or SLIP), and is not supported in conjunction with a transmit queue.

TM_USE_DRV_ONE_SCAT_SEND

First, in order to allow a single call to the driver send routine for a scattered frame, TM_USE_DRV_ONE_SCAT_SEND need to be defined in trsystem.h.

Modified driverSend

The user driverSend function API must be modified to support a single call to the driver send routine for a scattered frame. The modified driverSend function now takes only two parameters. The first parameter is still the interfaceHandle as before. The second parameter is a pointer to a structure containing the information needed to access the scattered data in a frame.

```
int  devOneScatSendFunc  (ttUserInterface
                          interfaceHandle,
                           ttUserPacketPtr
```

```
                        packetUPtr );
```

where ttUserPacketPtr is a pointer to a ttUserPacket structure.
ttUserPacket and ttUserPacketPtr are defined as follows:

```
typedef struct tsUserPacket

      {
          struct tsUserPacket *  pktuLinkNextPtr;
          tt8BitPtr              pktuLinkDataPtr;
          ttPktLen               pktuLinkDataLength;
          ttPktLen               pktuChainDataLength;
          int                    pktuLinkExtraCount;
      } ttUserPacket;

typedef ttUserPacket * ttUserPacketPtr;
```

where:

ttUserPacket fields	**Description**
ptkuLinkNextPtr	points to the next ttUserPacket structure
pktuLinkDataPtr	points to the data in the current link
pktuLinkDataLength	contains the length of the data in the current link
pktuLinkChainDataLength	contains the total length of the scattered data. Its value is ony valid in the first link
pktuLinkExtraCount	contains the number of extra links besides the first one. Its value is only valid in the first link.

The modified user device driver send function will loop through all the links of the scattered frame in order to send a complete frame.

tfUseInterfaceOneScatSend

The Treck stack need to be made aware that the modified driver send function need to be used instead of the default driver send function.
So, after the call to **tfAddInterface**, and before the call to **tfOpenInterface**, the user need to call **tfUseInterfaceOneScatSend**:

```
int                   tfUseInterfaceOneScatSend
(
ttUserInterface       interfaceHandle,
ttDevOneScatSendFunc  devOneScatSendFunc
);
```

Where devOneScatSendFunc type is defined as follows:

```
typedef int (*ttDevOneScatSendFuncPtr)
(
ttUserInterface interfaceHandle,
ttUserPacketPtr packetUPtr
);
```

After the interface is configured, this new device driver send function will be called, where the first parameter to the device driver send function is the interface handle as before, and the second parameter, is a pointer to the ttUserPacket structure as described above.

Note: Once tfUseInterfaceOneScatSend has been called successfully on an interface, the stack will always call the modified device driver send function passed as a second parameter to tfUseInterfaceOneScatSend for that interface.

Example

Modified driverSend function to support per-frame single call scattered send:

```
int devOneScatSendFunc( ttUserInterface interfaceHandle,
                       ttUserPacketPtr packetUPtr );
```

User calls

The user calls **tfAddInterface** specifying a null device driver send function to add the interface:

```
   interfaceHandle=tfAddInterface(
            "QUICC.SCC1",
            linkLayerHandle,
            tfDevEtherOpen,
            tfDevEtherClose,
            (ttDevSendFuncPtr)0,
            tfDevEtherReceive,
            (ttFreeRecvBufferFuncPtr)0,
            tfDevIoctl,
            tfDevGetPhyAddr,
            &errorCode
            );
```

Next the user calls the new **tfUseInterfaceOneScatSend** API to specify the modified device driver send function:

```
errorCode = tfUseInterfaceOneScatSend(
            ttUserInterface  interfaceHandle,
            ttDevOneScatSendFunc devOneScatSendFunc
            );
```

If the call does not fail, then the user can now call **tfConfigInterface**()/ **tfOpenInterface** to configure an IP address on the interface.
Note that **tfConfigInterface**/**tfOpenInterface** is unchanged.

```
    if (errorCode == TM_ENOERROR)
    {
       ipAddress=inet_addr("208.229.201.110");
       netMask=htonl(0xffffff00); /* 255.255.255.192
*/
/* Config the Interface */
        errorCode=tfOpenInterface(
            interfaceHandle,
            ipAddress,
            netMask,
            TM_DEV_SCATTER_SEND_ENB,
            5,
            0);
    }
```

tfSendCompletePacketInterface

The default device driver interface does not allow the user to specify which frame have been sent by the device driver. The stack assumes that the frames have been sent in the order of transmission to the device driver send function. When using a the modified single call device driver send interface, the user has the choice of either calling **tfSendCompleteInterface** for each frame that has been sent out, or **tfSendCompletePacketInterface** specifying the frame that has just been sent out. The later choice is useful for device driver where frames might not be sent out in the order they were transmitted.

tfSendCompletePacketInterface is similar to tfSendCompleteInterface, but takes the frame handle passed to the modified device driver send function as a parameter:

```
void                tfSendCompletePacketInterface
(
ttUserInterface     interfaceHandle,
ttUserPacketPtr     packetPtr,
int                 devDriverLockFlag
);
```

Limitations

The single scattered send call is not supported on point to point link layers such as PPP, or SLIP.

Device Driver Scattered recv ("Gather Read")

Description

By default the stack expects all data within a frame given by the user device driver recv function to be contiguous. Some device drivers support receiving data within a frame in scattered buffers, because it is more efficient.
The interface to the device driver recv interface can be optionally modified to allow it.

TM_USE_DRV_SCAT_RECV

First, in order to allow a scattered device driver recv,
TM_USE_DRV_SCAT_RECV need to be defined in trsystem.h.

Modified driver recv routine

Description

The user driverRecv function API need to be modified to support giving scattered data within a frame to the stack. As in the case of the non scattered driver recv API, the user can choose to either use pre-allocated stack buffers, or non-stack buffers to store the data into.

The scattered driverRecv function takes four parameters as shown below.

```
int devScatRecvFunc( ttUserInterface  interfaceHandle,
                     ttDruBlockPtrPtr uDevBlockPtrPtr,
                     int            * uDevBlockCountPtr,
                     int            * flagPtr );
```

devScatRecvFunc Parameters

The modified user device driver recv function will be responsible for initializing,

```
(*uDevBlockPtrPtr)
(*uDevBlockCountPtr)
(*flagPtr)
```

Parameters	**Meaning**
interfaceHandle	Initialized by the caller, as before
(*uDevBlockPtrPtr)	Upon return from the device driver scattered recv function, points to an array of user block data of type ttDruBlock
(*uDevBLockCountPtr)	Upon return from the device driver scattered recv contains the number of such elements.

(*flagPtr)	Upon return from the device driver scattered recv indicates whether the stack owns the buffers (TM_DEV_SCAT_RECV_STACK_BUFFER), or whether the device driver owns the buffers (TM_DEV_SCAT_RECV_USER_BUFFER).

uDevLockPtrPtr

Upon retrun from the device driver scattered recv function, *uDevBlockPtrPtr points to an array of ttDruBlock. There is one ttDruBlock per scattered buffer in the received frame. Each ttDruBlock element contains a pointer to a user buffer, a pointer to the beginning of the user data in the user buffer, and the user data length in the user buffer.

ttDruBlock Fields	**Description**
druDataPtr	points to beginning of data
druDataLength	indicates the data length
druBufferPtr	pointer to be passed to the device driver free function if user sets the flag to TM_DEV_SCAT_RECV_USER_BUFFER in the driver recv call function. (See *flagPtr below.)
druStackBufferPtr	pointer to stack pre-allocated user buffer if user sets the flag to TM_DEV_SCAT_RECV_STACK_BUFFER in the driver recv call function. (See *flagPtr below.)

If the device driver scattered recv function sets *flagPtr to TM_DEV_SCAT_RECV_STACK_BUFFER, then druStackBufferPtr should point to a buffer pointed to by first parameter of either tfGetEthernetBuffer, tfGetDriverBuffer, and the stack will be responsible for freeing that buffer, when the stack is done processing that buffer.

If the device driver scattered recv function sets *flagPtr to TM_DEV_SCAT_RECV_USER_BUFFER, then druBufferPtr points to a user allocated buffer. When the stack is done processing that buffer, the stack will call
the device driver free function (as set in tfAddInterface).

The user is responsible for managing the memory containing the array of ttDruBlock. It is guaranteed that when the stack calls the modified driver recv function for an interface, the array of ttDruBlock previously given to the stack by a previous call to the driver recv function for the same interface will not be accessed anymore. So it is safe for the user to re-use the array itself, then. On the other hand, it is only safe to re-use a user allocated buffer after the device driver free function is called.

uDevBlockCountPtr
Upon retrun from the device driver scattered recv function,
*uDevBlockCountPtr should be set to the number of ttDruBlock in the arrary of ttDruBlock given to the stack.

flagPtr
Upon retrun from the device driver scattered recv function,
*flagPtr should be set to
TM_DEV_SCAT_RECV_STACK_BUFFER if the user used pre-allocated stack buffers,
that the stack will be responsible for freeing.
or
TM_DEV_SCAT_RECV_USER_BUFFER if the user used its own buffers,

tfUseInterfaceScatRecv

The Treck stack need to be made aware that the modified driver recv function need to be used instead of the default driver recv function.
So, after the call to **tfAddInterface**, and before the call to **tfOpenInterface**, the user need to call **tfUseInterfaceScatRecv**:

```
int tfUseInterfaceScatRecv(
                ttUserInterface      interfaceHandle,
                ttDevScatRecvFunc    devScatRecvFunc
);
```

tfRecvScatInterface

Instead of calling **tfRecvInterface**, the user needs to call **tfRecvScatInterface**. The modified device driver recv function is itself called from within **tfRecvScatInterface**.

Scattered recv contiguous length threshold used in tfRecvScatInterface

Description
tfRecvScatInterface will automatically copy up to a per device configurable recv contiguous header length, if the data received in the first link of a device driver scattered recv is below this threshold. In that case, a new buffer is allocated by the stack, and the threshold number of bytes (but not more than the total length of the buffer) is copied.

Compile time value: TM_DEV_DEF_RECV_CONT_HDR_LENGTH
Default threshold value is given by the
TM_DEV_DEF_RECV_CONT_HDR_LENGTH macro
and is by default defined to 68 for IPv4, and 88 for IPv6. It can be defined in trsystem.h to overwrite the default value.

Run time modification using tfInterfaceSetOptions
That default threshold value can be changed at run time using the tfInterfaceSetOptions API, with the
TM_DEV_OPTIONS_SCAT_RECV_LENGTH option.

Dealing with non contiguous network protocol headers in scattered recv buffers, TM_RECV_SCAT_MIN_INCR_BUF

When the stack processes a received scattered buffer, it checks that the network protocol header is contiguous at a given layer. It if is not, then it means that the header straddles between the first buffer and consecutive buffers in the frame

If there is enough room at the end of the first of those consecutive buffers, then the end of the header is copied there. If there is not enough room, then a new buffer is allocated and replaces the first buffer. Data from the first buffer, plus the end of the header is copied into the first buffer. To prevent numerous re-allocation of the first buffer while processing the network headers, we allocate the maximum of TM_RECV_SCAT_MIN_INCR_BUF, and size of data that needs to be contiguous. By default TM_RECV_SCAT_MIN_INCR_BUF is set to 128.
It can be defined to a different value in trsystem.h.

```
#define TM_RECV_SCAT_MIN_INCR_BUF    128
```

Example

New User device driver recv function:

```
int   tfDevEtherScatRecvFunc(
                    ttUserInterface interfaceHandle,
                       ttDruBlockPtrPtr uDevBlockPtrPtr,
                    int          * uDevBlockCountPtr,
                    int          * flagPtr );
```

User calls:

User calls **tfAddInterface** specifying a null device driver recv function:

```
/* Add the Interface */
  interfaceHandle=tfAddInterface(
            "QUICC.SCC1",
            linkLayerHandle,
            tfDevEtherOpen,
            tfDevEtherClose,
            tfDevEtherSend,
            (ttDevRecvFuncPtr)0,
            tfDevFreeRecvBuffer,
            tfDevIoctl,
            tfDevGetPhyAddr,
            &errorCode);
```

Next User calls the new **tfUseInterfaceScatRecv** API to specify the device driver scattered buffer recv function:

```
errorCode = tfUseInterfaceScatRecv
            (
            ttUserInterface     interfaceHandle,
            ttDevScatRecvFunc   tfDevEtherScatRecvFunc
            );
```

If the call does not fail, then the user can now call **tfConfigInterface**/ **tfOpenInterface** to configure an IP address on the interface. Note that **tfConfigInterface**/**tfOpenInterface** is unchanged.

```
    if (errorCode == TM_ENOERROR)
    {
        ipAddress=inet_addr("208.229.201.110");
        netMask=htonl(0xffffff00); /* 255.255.255.0 */
/* Config the Interface */
        errorCode=tfConfigInterface(
            interfaceHandle,
            ipAddress,
            netMask,
            TM_DEV_SCATTER_SEND_ENB,
            5,
            0);
    }
```

Next, the user calls **tfRecvScatInterface** instead of **tfRecvInterface**.

No copy loop back driver

For testing purposes, a loop back device driver has been added below link layer that uses single scattered device driver send calls, and scattered device driver recv calls.

```
ttUserInterface tfUseScatIntfDriver
                  (
                  char                * namePtr,
                  ttUserLinkLayer     linkLayerHandle,
                  int                 * errorCode
                  );
```

In order to use this no copy loop back driver:

Add the following macros to trsystem.h:

```
#define TM_LOOP_TO_DRIVER
#define TM_USE_DRV_SCAT_RECV
#define TM_USE_DRV_ONE_SCAT_SEND
```

replace **tfAddInterface** with **tfUseScatIntfDriver**

In the examples directory txscatlp.c and txscatdr.c, use the no copy loop back driver.

Step 10 - Testing Your New Device Driver

If you wrote your own device driver (or modified ours), you are now ready to begin testing on the network. You will need some tools to do this correctly if you do not already have them. Your most important tool when testing a device driver is a network analyzer. There are many to choose from. The cost ranges from a few hundred dollars to tens of thousands of dollars. If you are in a hurry and are on a tight budget, we suggest that you visit the following web site.
http://www.klos.com/

At Klos Technologies, they have both Ethernet and PPP analyzers that run on a PC under DOS for a few hundred dollars.

When testing your device driver, your analyzer is your best friend (next to the support engineer at Treck Inc).

Things to test for:

- ✓ Make sure that you are sending packets. If you are not, then make sure that your driver’s send routine is being called. Set a break point with your debugger and step through your drivers send routine.
- ✓ Make sure receive processing is happening correctly by making sure that the incoming IP application data arrives at the socket.
- ✓ Try UDP before trying TCP. It is a simpler test and helps isolate problems.

If you are having problems:

- ✓ Ensure the IP address and Netmask are set correctly.
- ✓ Make sure your hardware is operating correctly.
- ✓ Make sure your hardware address is correct and valid.
- ✓ Make sure that tfTimerUpdate/tfTimerUpdateIsr and tfTimerExecute are being called.
- ✓ Make sure your loopback test still runs.
- ✓ Make sure that you have interpreted the device user’s manual correctly.
- ✓ Trying sending packets by directly calling your driver’s send routine.
- ✓ Make sure that you are passing the correct parameters to the Treck functions.
- ✓ Look for compiler warnings in your code.
- ✓ Make sure that you actually check the return codes of the Treck functions that you call to ensure that an error did not occur.
- ✓ Watch out for padding by the compiler. Treck data structures are already padded and aligned on 32 bit boundaries.

Please see Appendix C for detailed driver testing and debugging information.

Programmer’s Reference

Programmer's Reference Function List

BSD 4.4 Socket API

accept
bind
connect
getpeername
getsockname
getsockopt
htonl
htons
inet_addr
inet_aton
inet_ntoa
listen
ntohl
ntohs
readv
recv
recvfrom
rresvport
select
send
sendto
setsockopt
shutdown
socket
tfClose
tfIoctl
tfRead
tfWrite
writev

Socket Extension Calls

tfBindNoCheck
tfBlockingState
tfFlushRecvQ
tfFreeDynamicMemory
tfFreeZeroCopyBuffer
tfGetOobDataOffset
tfGetSendCompltBytes
tfGetSocketError
tfGetWaitingBytes
tfGetZeroCopyBuffer
tfInetToAscii
tfIpScatteredSend
tfRawSocket
tfRecvFromto
tfRegisterIPForwCB
tfResetConnection
tfSendToFrom
tfSendToInterface
tfSocketArrayWalk
tfSocketScatteredSendTo
tfZeroCopyRecv
tfZeroCopyRecvFrom
tfZeroCopySend
tfZeroCopySendTo
tfZeroCopyUserBufferSend
tfZeroCopyUserBufferSendto

Call Back Function Registration

tfRegisterSocketCB
tfRegisterSocketCBParam

Treck Initialization

tfInitTreckOptions
tfSetTreckOptions
tfStartTreck

Device/Interface API

tfAddInterface
tfAddInterfaceMhomeAddress
tfCheckReceiveInterface
tfCheckSentInterface
tfCheckXmitInterface
tfCloseInterface
tfConfigInterface
tfDeviceClearPointer
tfDeviceGetPointer
tfDeviceStorePointer
tfFinishOpenInterface
tfFreeDriverBuffer
tfGetDriverBuffer
tfGetBroadcastAddress
tfGetIfMtu
tfGetIpAddress
tfGetNetMask
tfInterfaceGetVirtualChannel
tfInterfaceSetOptions
tfInterfaceSetVirtualChannel
tfInterfaceSpinLock
tfIoctlInterface
tfNotifyInterfaceIsr
tfNotifyInterfaceTask
tfNotifyReceiveInterfaceIsr
tfNotifySentInterfaceIsr
tfOpenInterface
tfPoolCreate
tfPoolDelete
tfPoolIsrFreeBuffer
tfPoolIsrGetBuffer
tfPoolReceive
tfRecvInterface
tfRecvScatInterface
tfSendCompleteInterface
tfSendCompletePacketInterface
tfSetIfMtu
tfUnConfigInterface
tfUseInterfaceOneScatSend
tfUseInterfaceScatRecv

tfUseInterfaceXmitQueue
tfUseIntfDriver
tfUseScatIntfDriver
tfWaitReceiveInterface
tfWaitSentInterface
tfWaitXmitInterface
tfXmitInterface

Null Link Layer API

tfUseNullLinkLayer

Ethernet API

tfGetEthernetBuffer
tfUseEthernet

SLIP API

tfGetSlipPeerIpAddress
tfSetSlipPeerIpAddress
tfSlipSetOptions
tfUseSlip

ARP/Routing TableAPI

tfAddArpEntry
tfAddDefaultGateway
tfAddMcastRoute
tfAddProxyArpEntry
tfAddStaticRoute
tfDelArpEntryByIpAddr
tfDelArpEntryByPhysAddr
tfDelDefaultGateway
tfDelProxyArpEntry
tfDelStaticRoute
tfDisablePathMtuDisc
tfGetArpEntryByIpAddr
tfGetArpEntryByPhysAddr
tfGetDefaultGateway
tfUseRip

Timer Interface API

tfTimerExecute
tfTimerUpdate
tfTimerUpdateIsr

Kernel/RTOS Interface

tfKernelCreateCountSem
tfKernelCreateEvent
tfKernelDeleteCountSem
tfKernelError
tfKernelFree
tfKernelInitialize
tfKernelInstalIsrHandler
tfKernelIsrPostEvent
tfKernelMalloc
tfKernelPendCountSem
tfKernelPendEvent
tfKernelPostCountSem
tfKernelReleaseCritical
tfKernelSetCritical
tfKernelSheapCreate
tfKernelTaskPostEvent
tfKernelTaskYield
tfKernelWarning

Compiler Library Replacement Functions

tfMemCpy
tfMemSet
tfQsort
tfSPrintF
tfSScanF
tfStrCat
tfStrChr
tfStrCmp
tfStrCpy
tfStrCSpn
tfStrError
tfStrLen
tfStrNCmp
tStrRChr
tfStrStr
tfStrToL
tfStrToUl
tfVSPrintF
tfVSScanF

BSD 4.4 Socket API

accept

include <trsocket.h>

```
Int                  accept
(
Int                  socketDescriptor,
struct sockaddr *  addressPtr,
int *                addressLengthPtr
);
```

Function Description

The argument *socketDescriptor* is a socket that has been created with **socket,** bound to an address with **bind**, and that is listening for connections after a call to **listen**. **accept** extracts the first connection on the queue of pending connections, creates a new socket with the properties of *socketDescriptor*, and allocates a new socket descriptor for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, **accept** blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are present on the queue, **accept** returns an error as described below. The accepted socket is used to **send** and **recv** data to and from the socket that it is connected to. It is not used to accept more connections. The original socket remains open for accepting further connections. **accept** is used with connection-based socket types, currently with SOCK_STREAM.

Using **select** (prior to calling **accept**):

It is possible to **select** a listening socket for the purpose of an **accept** by selecting it for a read. However, this will only indicate when a connect indication is pending; it is still necessary to call **accept**.

Using **tfRegisterSocketCB** (prior to calling **accept**):

Alternatively, the user could issue a **tfregisterSocketCB** call on the listening socket with a TM_CB_ACCEPT event flag to get an asynchronous notification of an incoming connection request. Again, this will only indicate that a connection request is pending; it is still necessary to call **accept** after having received the asynchronous notification.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor that was created with **socket** and bound to with **bind** and is listening for connections with **listen**
addressPtr	The structure to write the incoming address into.
addressLengthPtr	Initially, it contains the amount of space pointed to by *addressPtr.* On return it contains the length in bytes of the address returned.

Returns

New Socket Descriptor or –1 on error.
If **accept** fails, the errorCode can be retrieved with **tfGetSocketError** *socketDescriptor)* which will return one of the following error codes:

TM_EBADF	The socket descriptor is invalid.
TM_EINVAL	addressPtr was a null pointer.
TM_EINVAL	addressLengthPtr was a null pointer.
TM_EINVAL	The value of addressLengthPtr was too small.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_EPERM	Cannot call **accept** without calling **listen** first.
TM_EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
TM_EWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

bind

```
#include <trsocket.h>

int                    bind
(
int                    socketDescriptor,
const struct sockaddr *addressPtr,
int                    addressLength
);
```

Function Description

bind assigns an address to an unnamed socket. When a socket is created with **socket**, it exists in an address family space but has no address assigned. **bind** requests that the address pointed to by *addressPtr* be assigned to the socket. Clients do not normally require that an address be assigned to a socket. However, servers usually require that the socket be bound to a "well known" address. The port number must be less than 32768 (TM_SOC_NO_INDEX), or could be 0xFFFF (TM_WILD_PORT). Binding to the TM_WILD_PORT port number allows a server to listen for incoming connection requests on all the ports. Multiple sockets cannot bind to the same port with different IP addresses (as might be allowed in UNIX).

Parameters

Parameter	Description
socketDescriptor	The socket descriptor to assign an IP address and port number to.
addressPtr	The pointer to the structure containing the address to assign.
addressLength	The length of the address structure.

Returns

Value	Meaning
0	Success
-1	An error occurred

bind can fail for any of the following reasons:

TM_EADDRINUSE	The specified address is already in use. TM_EBADF *socketDescriptor* is not a valid descriptor.
TM_EINVAL	One of the passed parameters is invalid or socket is already bound.

connect

```
#include <trsocket.h>

int                          connect
(
int                          socketDescriptor,
const struct sockaddr *      addressPtr,
int                          addressLength
);
```

Function Description

The parameter *socketDescriptor* is a socket. If it is of type SOCK_DGRAM, **connect** specifies the peer with which the socket is to be associated; this address is the address to which datagrams are to be sent if a receiver is not explicitly designated; **it is the only address from which datagrams are to be received**. If the socket *socketDescriptor* is of type SOCK_STREAM, **connect** attempts to make a connection to another socket (either local or remote). The other socket is specified by *addressPtr*. *addressPtr* is a pointer to the IP address and port number of the remote or local socket. If *socketDescriptor* is not bound, then it will be bound to an address selected by the underlying transport provider. Generally, stream sockets may successfully **connect** only once; datagram sockets may use **connect** multiple times to change their association. Datagram sockets may dissolve the association by connecting to a null address.

Note that a non-blocking **connect** is allowed. In this case, if the connection has not been established, and not failed, connect will return TM_SOCKET_ERROR, and **tfGetSocketError** will return TM_EINPROGRESS error code indicating that the connection is pending. connect should never be called more than once. Additional calls to **connect** will fail with TM_EALREADY error code returned by **tfGetSocketError**.

Non-blocking connect and select:

After issuing one non-blocking **connect**, the user can call **select** with the write mask set for that socket descriptor to check for connection completion. When select returns with the write mask set for that socket, the user can call **getsockopt** with the SO_ERROR option name. If the retrieved pending socket error is TM_ENOERROR, then the connection has been established, otherwise an error occurred on the connection, as indicated by the retrieved pending socket error.

Non-blocking connect and tfRegisterSocketCB:

Alternatively, the user could have issued a **tfRegisterSocketCB** call with TM_CB_CONNECT_COMPLT|TM_CB_SOCKET_ERROR event flags prior to issuing a **non-blocking connect**, to get an asynchronous notification of the completion of the **connect** call. If the TM_CB_SOCKET_ERROR flag is set when the call back function is called, the user can retrieve the pending socket error by calling **getsockopt** with the SO_ERROR option name.

Non-blocking connect and polling:
Alternatively, after the user issues a non-blocking connect call that returns TM_SOCKET_ERROR, the user can poll for completion, by calling tfGetSocketError until tfGetSocketError no longer returns TM_EINPROGRESS.

If **connect** fails, the socket is no longer usable, and must be closed. **connect** cannot be called again on the socket.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to assign a name (port number) to.
addressPtr	The pointer to the structure containing the address to connect to for TCP. For UDP it is the default address to send to and the only address to receive from.
addressLength	The length of the address structure.

Returns

Value	**Meaning**
0	Success
-1	An error occurred

connect can fail for any of the following reasons:

TM_EADDRINUSE	The socket address is already in use.
TM_EADDRNOTAVAIL	The specified address is not available on the remote / local machine.
TM_EPFNOSUPPORT	Addresses in the specified address family cannot be used with this socket
TM_EINPROGRESS	The socket is non-blocking and the current connection attempt has not yet been completed.
TM_EALREADY	**connect** has already been called on the socket. Only one connect call is allowed on a socket.

TM_EBADF	*socketDescriptor* is not a valid descriptor.
TM_ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should close the socket descriptor, and issue another **socket** call to obtain a new descriptor before attempting another **connect** call.
TM_EPERM	Cannot call **connect** after **listen** call.
TM_EINVAL	One of the parameters is invalid
TM_EHOSTUNREACH	No route to the host we want to connect to. The calling program should close the socket descriptor, and should issue another socket call to obtain a new descriptor before attempting another connect call.
TM_ETIMEDOUT	Connection establishment timed out, without establishing a connection. The calling program should close the socket descriptor, and issue another **socket** call to obtain a new descriptor before attempting another **connect** call.

getpeername

#include <trsocket.h>

```
int                 getpeername
(
int                 socketDescriptor,
Struct sockaddr *   fromAddressPtr,
int *               addressLengthPtr
);
```

Function Description

This function returns the IP address / Port number of the remote system to which the socket is connected.

Parameters

Parameter	Description
socketDescriptor	The socket descriptor that we wish to obtain information about.
fromAddressPtr	A pointer to the address structure that we wish to store this information into.
addressLengthPtr	The length of the address structure.

Returns

Value	Meaning
0	Success
-1	An error occurred

getpeername can fail for any of the following reasons:

TM_EBADF	*socketDescriptor* is not a valid descriptor.
TM_ENOTCONN	The socket is not connected.
TM_EINVAL	One of the passed parameters is not valid.

getsockname

#include <trsocket.h>

```
int                   getsockname
(
int                   socketDescriptor,
struct sockaddr *     myAddressPtr,
int *                 addressLengthPtr
);
```

Function Description
This function returns to the caller the **Local** IP Address/Port Number that we are using on a given socket.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor that we wish to inquire about.
myAddressPtr	The pointer to the address structure where the address information will be stored.
*addressLengthPt*r	The length of the address structure.

Returns

Value	**Meaning**
0	Success
-1	An error occurred

getsockname can fail for any of the following reasons:

TM_EBADF	*socketDescriptor* is not a valid descriptor.
TM_EINVAL	One of the passed parameters is not valid.

getsockopt

#include <trsocket.h>

```
int                 getsockopt
(
int                 socketDescriptor,
int                 protocolLevel,
int                 optionName,
char *              optionValuePtr,
int *               optionLengthPtr
);
```

Function Description

getsockopt is used retrieve options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level. When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *protocolLevel* is specified as SOL_SOCKET. To manipulate options at any other level, *protocolLevel* is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the TCP protocol, *protocolLevel* is set to the TCP protocol number. For **getsockopt**, the parameters *optionValuePtr* and *optionLengthPtr* identify a buffer in which the value(s) for the requested option(s) are to be returned. For **getsockopt**, *optionLengthPtr* is a value-result parameter, initially containing the size of the buffer pointed to by *optionValuePtr*, and modified on return to indicate the actual size of the value returned. *optionName* and any specified options are passed un-interpreted to the appropriate protocol module for interpretation. The include file <trsocket.h> contains definitions for the options described below. Options vary in format and name. Most socket-level options take an int for *optionValuePtr*. SO_LINGER uses a *struct linger* parameter that specifies the desired state of the option and the linger interval (see below). struct linger is defined in <trsocket.h>. *struct linger* contains the following members:

l_onoff	on = 1/off = 0
l_linger	linger time, in seconds

The following options are recognized at the socket level:

SOL_SOCKET ***protocolLevel* options**	**Description**
SO_ACCEPTCON	Enable/disable listening for connections. **listen** turns on this option.
SO_DONTROUTE	Enable/disable routing bypass for outgoing messages. **Default 0**.

SO_ERROR

When an error occurs on a socket, the Turbo Treck stack internally sets the error code on the socket. It is called the pending error for the socket. If the user had called select for either readability or writability, select returns with either or both conditions set. If the user had registered a call back function with the TM_CB_SOCKET_ERROR flag, the user would be notified. In both cases, the user can then retrieve the pending socket error, by calling **getsockopt** with this option name at the SOL_SOCKET level, and the Turbo Treck stack will reset the internal socket error. Alternatively if the user is waiting for incoming data, **read** or other **recv** APIs can be called. If there is no data queued to the socket, the **read**/**recv** call returns TM_SOCKET_ERROR, the Turbo Treck stack resets the internal socket error, and the pending socket error can be returned if the user calls **tfGetSocketError** (equivalent of errno). Note that the SO_ERROR option is useful when the user uses **connect** in non-blocking mode, and **select**.

SO_KEEPALIVE

Enable/disable keep connections alive. **Default 0 (disable)**

SO_OOBINLINE

Enable/disable reception of out-of-band data in band. **Default is 0**.

SO_REUSEADDR

Enable this socket option to bind the same port number to multiple sockets using different local IP addresses. Note that to use this socket option, you also need to uncomment TM_USE_REUSEADDR_LIST in trsystem.h. **Default 0 (disable)**.

SO_RCVLOWAT

The low water mark for receiving.

SO_SNDLOWAT

The low water mark for sending.

SO_RCVBUF

The buffer size for input. **Default is 8192 bytes**.

SO_SNDBUF

The buffer size for output. **Default is 8192 bytes**.

TM_SO_RCVCOPY

TCP socket: fraction use of a receive buffer below which we try and append to a previous receive buffer in the socket receive queue. *UDP socket*: fraction use of a receive buffer below which we try and copy to a new receive buffer, if there is already at least a buffer in the receive queue. This is to avoid keeping large pre-allocated receive buffers, which the user has not received yet, in the socket receive queue.**Default value is 4 (25%).**

TM_SO_SNDAPPEND

TCP socket only. Threshold in bytes of ‘send’ buffer below, which we try and append, to previous ‘send’ buffer in the TCP send queue. Only used with **send**, not with **tfZeroCopySend**. This is to try to regroup lots of partially empty small buffers in the TCP send queue waiting to be ACKED by the peer; otherwise we could run out of memory, since the remote TCP will delay sending ACKs. Note that care should be taken not to use **tfZeroCopySend** when sending small buffers, since we do not try to regroup small buffers with **tfZeroCopySend**.
Default value is 128 bytes.

SO_UNPACKEDDATA

TI C3x and C5x DSP platforms only: If this option is enabled, all socket data will be sent and received in byte unpacked format. If this option is disabled, all socket data will be sent in a byte packed format, as received from the network. **Default 0 (disable)**

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a **bind** call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages (every 2 hours) on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address. SO_LINGER controls the action taken when unsent messages are queued on a socket and a close on the socket is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close of the socket attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the **setsockopt** call when SO_LINGER is requested). If SO_LINGER is disabled and a close on the socket is issued, the system will process the close of the socket in a manner that allows the process to continue as quickly as possible. The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with **recv** call without the MSG_OOB flag.   SO_SNDBUF and SO_RCVBUF are options that adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data. The Internet protocols place an absolute limit of 64 Kbytes on these values for UDP and TCP sockets (in the default mode of operation).

The following options are recognized at the IP level.

IP_PROTOIP

***protocolLevel* options**	**Description**
IPO_MULTICAST_IF	Get the configured IP address that uniquely identifies the outgoing interface for multicast datagrams sent on this socket. A zero IP address parameter indicates that we want to reset a previously set outgoing interface for multicast packets sent on that socket.
IPO_MULTICAST_TTL	Get the default IP TTL for outgoing multicast datagrams.
IPO_SRCADDR	Get the IP source address for the connection.
IPO_TOS	IP type of service. **Default 0**
IPO_TTL	IP Time To Live in seconds. **Default 64**

The following options are recognized at the TCP level.

IP_PROTOTCP

***protocolLevel* options** **Description**

TCP_KEEPALIVE

Get the idle time in seconds for a TCP connection before it starts sending keep alive probes. Note that keep alive probes will be sent only if the SO_KEEPALIVE socket option is enabled. **Default 7,200 seconds**.

TCP_MAXRT

Get the amount of time in **seconds** before the connection is broken once TCP starts retransmitting, or probing a zero window when the peer does not respond. A TCP_MAXRT value of 0 means the system default, and -1 means retransmit forever. Note that unless the TCP_MAXRT value is –1 (wait forever), the connection can also be broken if the number of maximum retransmission TM_TCP_MAX_REXMIT has been reached. See TM_TCP_MAX_REXMIT below. **Default 0**.

(which means use system default of TM_TCP_MAX_REXMIT times network computed round trip time for an established connection. For a non established connection, since there is no computed round trip time yet, the connection can be broken when either **75 seconds** or when TM_TCP_MAX_REXMIT times default network round trip time have elapsed, whichever occurs first).

TCP_MAXSEG

Get the maximum TCP segment size sent on the network. Note that the TCP_MAXSEG value is the maximum amount of data (including TCP options, but not the TCP header) that can be sent per segment to the peer. i.e. the amount of user data sent per segment is the value given by the TCP_MAXSEG option minus any enabled TCP option (for example 12 bytes for a TCP time stamp option).

Default is IP MTU minus 40 bytes.

TCP_NODELAY
If this option value is non-zero, the Nagle algorithm that buffers the sent data inside the TCP is disabled. Useful to allow client's TCP to send small packets as soon as possible (like mouse clicks). **Default 0**.

TCP_NOPUSH
If this option value is non-zero, then TCP delays sending any TCP data until a full sized segment is buffered in the TCP buffers. Useful for applications that send continuous big chunks of data and know that more data will be sent such as FTP. (Normally, the TCP code sends a non full-sized segment, only if it empties the TCP buffer). **Default 0**.

TCP_STDURG
If this option value is zero, then the urgent data pointer points to the last bye of urgent data + 1, like in Berkeley systems. **Default 1** (urgent pointer points to last byte of urgent data as specified in RFC1122).

TM_TCP_PACKET
If this option value is non-zero, then TCP behaves like a message-oriented protocol (i.e. respects packet boundaries) at the application level in both send and receive directions of data transfer. Note that for the receive direction to respect packet boundaries, the TCP peer which is sending must also implement similar functionality in its send direction. This is useful as a reliable alternative to UDP. Note that preserving packet boundaries with TCP will not work correctly if you use out-of-band data. **Default 0.**

*TM_TCP_SEL_ACK
If this option value is zero, then TCP selective Acknowledgment options are disabled. **Default 1**.

*TM_TCPWND_SCALE
If this option value is non-zero, then the TCP window scale option is enabled. **Default 1**.

*TM_TCP_TS
If this option value is non-zero, then the TCP time stamp option is enabled.

Default 1.

TM_TCP_SLOW_START
If this option value is non-zero, then the TCP slow start algorithm is enabled. **Default 1**.

TM_TCPDELAY_ACK
Get the TCP delay ack time in milliseconds. **Default 200 milliseconds**.

TM_TCPMAX_REXMIT
Get the maximum number of retransmissions without any response from the remote before TCP gives up and aborts the connection. See also TCP_MAXRT above. **Default 12**.

TM_TCP_KEEPALIVE_CNT
Get the maximum number of keep alive probes without any response from the remote before TCP gives up and aborts the connection. See also TCP_KEEPALIVE above. **Default 8**.

TM_TCPFINWT2TIME
Get the maximum amount of time TCP will wait for the remote side to close after it initiated a close. **Default 600 seconds**.

TM_TCP2MSLTIME
Get the maximum amount of time TCP will wait in the TIME WAIT state once it has initiated a close of the connection. **Default 60 seconds**.

TM_TCP_RTO_DEF
Get the TCP default retransmission timeout value in milliseconds. Used when no network round trip time has been computed yet. **Default 3,000 milliseconds**.

TM_TCP_RTO_MIN
Get the minimum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TM_TCP_RTO_MIN and TM_TCP_RTO_MAX. **Default 100 milliseconds**.

TM_TCPRTO_MAX
Get the maximum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TM_TCPRTO_MIN and TM_RTO_MAX. **Default 64,000 milliseconds**.

TM_TCPPROBE_MIN
Get the minimum window probe timeout

interval in milliseconds. The network computed window probe timeout is bound by TM_TCP_PROBE _MIN and TM_TCP_PROBE _MAX. **Default 500 milliseconds**.

TM_TCP_PROBE_MAX
Get the maximum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TM_TCP_PROBE _MIN and TM_TCP_PROBE _MAX. **Default 60,000 milliseconds**.

TM_TCP_KEEPALIVE_INTV
Get the interval between Keep Alive probes in seconds. See TM_TCP_KEEPALIVE_CNT. **Default 75 seconds**.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to get the option from.
protocolLevel	The protocol to get the option from. See below.
optionName	The option to get. See above and below.
optionValuePtr	The pointer to a user variable into which the option value is returned. User variable is of data type described below.
optionLengthPtr	Pointer to the size of the user variable, which is the size of the option data type, described below. It is a value-result parameter, and the user should set the size prior to the call.

ProtocolLevel	**Description**
SOL_SOCKET	Socket level protocol
IP_PROTOIP	IP level protocol
IP_PROTOTCP	TCP level protocol

ProtocolLevel	OptionName	Option data type	Option value
SOL_SOCKET	SO_ACCEPTCON	int	0 or 1
	SO_DONTROUTE	int	0 or 1
	SO_ERROR	int	
	SO_KEEPALIVE	int	0 or 1
	SO_LINGER	struct linger	
	SO_OOBINLINE	int	0 or 1
	SO_RCVBUF	unsigned long	
	SO_RCVLOWAT	unsigned long	
	SO_REUSEADDR	int	0 or 1
	SO_SNDBUF	unsigned long	
	SO_SNDLOWAT	unsigned long	
	TM_SO_RCVCOPY	unsigned int	
	TM_SO_SNDAPPEND	unsigned int	
	SO_UNPACKEDDATA	int	0 or 1
IP_PROTOIP	IPO_MULTICAST_IF	struct in_addr	
	IPO_MULTICAST_TTL	unsigned char	
	IPO_TOS	unsigned char	
	IPO_TTL	unsigned char	
	IPO_SRCADDR	ttUserIpAddress	
IP_PROTOTCP	TCP_KEEPALIVE	int	
	TCP_MAXRT	int	
	TCP_MAXSEG	int	
	TCP_NODELAY	int	0 or 1
	TCP_NOPUSH	int	0 or 1
	TCP_STDURG	int	0 or 1
	TM_TCP_2MSLTIME	int	
	TM_TCP_DELAY_ACK	int	
	TM_TCP_FINWT2TIME	int	
	TM_TCP_KEEPALIVE_CNT	int	
	TM_TCP_KEEPALIVE_INTV	int	
	TM_TCP_MAX_REXMIT	int	
	TM_TCP_PACKET	int	0 or 1
	TM_TCP_PROBE_MAX	unsigned long	
	TM_TCP_PROBE_MIN	unsigned long	
	TM_TCP_RTO_DEF	unsigned long	
	TM_TCP_RTO_MAX	unsigned long	
	TM_TCP_RTO_MIN	unsigned long	
	TM_TCP_SEL_ACK	int	0 or 1
	TM_TCP_SLOW_START	int	0 or 1
	TM_TCP_TS	int	0 or 1
	TM_TCP_WND_SCALE	int	0 or 1

Returns

Value	**Meaning**
0	Successful set of option
-1	An error occurred

getsockopt will fail if:

TM_EBADF	The socket descriptor is invalid
TM_EINVAL	One of the parameters is invalid
TM_ ENOPROTOOPT	The option is unknown at the level indicated

htonl

```
#include <trsocket.h>

unsigned long     htonl
(
unsigned long     longValue
);
```

Function Description

This function converts a long value from host byte order to network byte order.

Parameters

Parameter	**Description**
longValue	The value to convert

Returns

The converted value

htons

```
#include <trsocket.h>

unsigned short    htons
(
unsigned short    shortValue
);
```

Function Description

This function converts a short value from host byte order to network byte order.

Parameters

Parameter	**Description**
shortValue	The value to convert

Returns

The converted value

inet_addr

```
#include <trsocket.h>

unsigned long     inet_addr
(
char *            ipAddressDottedStringPtr
);
```

Function Description

This function converts an IP address from the decimal dotted notation to an unsigned long.

Parameters

Parameter	**Description**
ipAddressDottedStringPtr	The dotted string (i.e. "208.229.201.4")

Returns

Value	**Meaning**
-1	Error
Other	The IP Address in Network Byte Order

inet_aton

```
#include <trsocket.h>

unsigned long     inet_aton
(
Char *            ipAddressDottedStringPtr
);
```

Function Description

This function converts an IP address from the decimal dotted notation to an unsigned long.

Parameters

Parameter	**Description**
ipAddressDottedStringPtr	The dotted string (i.e. "208.229.201.4")

Returns

Value	**Meaning**
0	Error
Other	The IP Address in Network Byte Order

inet_ntoa

```
#include <trsocket.h>

Char *              inet_ntoa
(
struct in_addr      inAddr
);
```

Function Description

This function converts an IP address structure (the *sin_addr* element of the *sockaddr_in* structure) to an ASCII string in dotted decimal notation.

Note: inet_ntoa is not reentrant. It is only provided for BSD support. Users should use the equivalent reentrant function called tfInetToAscii.

Parameters

Parameter	**Description**
inetAddr	Structure containing the address to convert.

Returns

Value	**Meaning**
0	Error
Other	ASCII string of the IP address in dotted decimal notation.

listen

```
#include <trsocket.h>

int                 listen
(
int                 socketDescriptor,
int                 backLog
);
```

Function Description

To accept connections, a socket is first created with **socket** a backlog for incoming connections is specified with **listen** and then the connections are accepted with **accept**. The **listen** call applies only to sockets of type SOCK_STREAM. The *backLog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, and the underlying protocol supports retransmission, the connection request may be ignored so that retries may succeed. For AF_INET sockets, the TCP will retry the connection. If the backlog is not cleared by the time the TCP times out, **connect** will fail with TM_ETIMEDOUT.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to listen on.
backlog	The maximum number of outstanding connections allowed on the socket.

Returns

Value	**Meaning**
0	Success
-1	An error occurred

listen can fail for the following reason:

TM_EADDRINUSE	The address is currently used by another socket.
TM_ EBADF	The socket descriptor is invalid.
TM_ EOPNOTSUPP	The socket is not of a type that supports the operation **listen**.

ntohl

```
#include <trsocket.h>

unsigned long    ntohl
(
unsigned long    longValue
);
```

Function Description

This function converts a long value from network byte order to host byte order.

Parameters

Parameter	**Description**
longValue	The value to convert

Returns

The converted value

ntohs

```
#include <trsocket.h>

unsigned short    ntohs
(
unsigned short    shortValue
);
```

Function Description

This function converts a short value from network byte order to host byte order.

Parameters

Parameter	**Description**
shortValue	The value to convert

Returns

The converted value

readv

```
#include <trsocket.h>

int                 readv
(
int                 socketDescriptor,
struct iovec *      iov,
int                 iovcnt
);
```

Function Description

readv functions as a scatter read because the received data can be placed in multiple buffers. **readv** attempts to read data from the socket *socketDescriptor* and places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1].

The iovec structure contains the following members:

caddr_t iov_base;

int iov_len;

Each iovec entry specifies the base address and length of an area in memory where data should be placed. **readv** always fills one buffer completely before proceeding to the next. On success, **readv** returns the number of bytes actually read and placed in the buffer; this number may be less than the total of all of the iov_len values. A value of 0 is returned when an end-of-file has been reached.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor from which to read data
iov	The list of buffers to put the received data
iovcnt	The number of buffers in the list

Returns

Value	**Meaning**
>0	Number of bytes actually read from the socket.
0	EOF
-1	An error occurred

readv will fail if:

TM_EBADF	The socket descriptor is invalid
TM_EINVAL	The iovcnt is 0 or less than 0. The sum of the iov_len values overflowed an integer
TM_ENOBUFS	There was insufficient user memory available to complete the operation
TM_ EWOULDBLOCK	The socket is marked as non-blocking and no data is available to be read.
TM_ESHUTDOWN	The peer has closed the connection and there is no more data to read (TCP socket only)
TM_ENOTCONN	Socket is not connected

recv

```
#include <trsocket.h>

int                 recv
(
int                 socketDescriptor,
char *              bufferPtr,
int                 bufferLength,
int                 flags
);
```

Function Description

recv is used to receive messages from another socket. **recv** may be used only on a *connected* socket (see **connect**, **accept**). *socketDescriptor* is a socket created with **socket** or **accept**. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see **socket**). The length of the message returned could also be smaller than *bufferLength* (this is not an error). If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking, or the MSG_DONTWAIT flag is set in the *flags* parameter, in which case -1 is returned with socket error being set to TM_EWOULDBLOCK.

Out-of-band data not in the stream (urgent data when the SO_OOBINLINE option is not set (default)) (TCP protocol only).

A single out-of-band data byte is provided with the TCP protocol when the SO_OOBINLINE option is not set. If an out-of-band data byte is present, **recv** with the MSG_OOB flag *not set* will not read past the position of the out-of-band data byte in a single **recv** request. That is, if there are 10 bytes from the current read position until the out-of-band byte, and if we execute a **recv** specifying a bufferLength of 20 bytes, and a flag value of 0, **recv** will only return 10 bytes. This forced stopping is to allow us to execute the SOIOCATMARK **tfIoctl** to determine when we are at the out-of-band byte mark. Alternatively, **tfGetOobDataOffset** can be used instead of **tfIoctl** to determine the offset of the out-of-band data byte. When we are at the mark, **recv** with the MSG_OOB flag set can read the out-of-band data byte. Note that the user needs to either use **select** or **tfRegisterSocketCB** in order to know when out-of-band data has arrived, or is arriving.

Out-of-band data (when the SO_OOBINLINE option is set (see setsockopt)).

(TCP protocol only)

If the SO_OOBINLINE option is enabled, the out-of-band data is left in the normal data stream and is read without specifying the MSG_OOB. More than one out-of-band data bytes can be in the stream at any given time. The out-of-band byte mark corresponds to the final byte of out-of-band data that was received. In this case, the MSG_OOB flag cannot be used with **recv**. The out-of-band data will be read in line with the other data. Again, **recv** will not read past the position of the out-of-band mark in a single **recv** request. Again, **tfIoctl** with the SOIOCATMARK, or **tfGetOobDataOffset** can be used to determine where the last received out-of-band byte is in the stream. Note that the user needs to either use **select** or **tfRegisterSocketCB** in order to know when out-of-band data has arrived, or is arriving.

select may be used to determine when more data arrives, or/and when out-of-band data arrives.

tfRegisterSocketCB may be used to asynchronously determine when more data arrives, or/and when out-of-band data arrives.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor from which to receive data.
bufferPtr	The buffer into which the received data is put.
bufferLength	The length of the buffer area that *bufferPtr* points to.
flags	See below.

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Do not wait for data, but rather return immediately
MSG_OOB	Read any "out-of-band" data present on the socket rather than the regular "in-band" data
MSG_PEEK	"Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data

Returns

Value	**Meaning**
>0	Number of bytes actually received from the socket.
0	EOF
-1	An error occurred

recv will fail if:

TM_EBADF	The socket descriptor is invalid
TM_ENOBUFS	There was insufficient user memory available to complete the operation
TM_ EMSGSIZE	The socket requires that message be received atomically, and bufferLength was too small
TM_EWOULDBLOCK	The socket is marked as non-blocking or the MSG_DONTWAIT flag is used and no data is available to be read, or the MSG_OOB flag is set and the out of band data has not arrived yet from the peer
TM_ESHUTDOWN	The remote socket has closed the connection, and there is no more data to be received (TCP socket only)
TM_EINVAL	One of the parameters is invalid, or the MSG_OOB flag is set and, either the SO_OOBINLINE option is set, or there is no out of band data to read or coming from the peer
TM_ENOTCONN	Socket is not connected
TM_EHOSTUNREACH	No route to the connected host

recvfrom

```
#include <trsocket.h>

int                recvfrom
(
int                socketDescriptor,
char *             bufferPtr,
int                bufferLength,
int                flags,
struct sockaddr *  fromPtr,
int *              fromLengthPtr
);
```

Function Description

recvfrom is used to receive messages from another socket. **recvfrom** may be used to receive data on a socket whether it is in a connected state or not but not on a TCP socket. *socketDescriptor* is a socket created with **socket**. If *fromPtr* is not a NULL pointer, the source address of the message is filled in. *fromLengthPtr* is a value-result parameter, initialized to the size of the buffer associated with *fromPtr*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see **socket**). If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking, or the MSG_DONTWAIT flag is set in the *flags* parameter, in which case -1 is returned with socket error being set to EWOULDBLOCK.

select may be used to determine when more data arrives, or/and when out-of-band data arrives.

tfRegisterSocketCB may be used to asynchronously determine when more data arrives, or/and when out-of-band data arrives.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to receive data from.
bufferPtr	The buffer to put the received data
bufferLength	The length of the buffer area that *bufferPtr* points to
flags	See Below.
fromPtr	The socket from which the data is (or to be) received.
fromLengthPtr	The length of the data area the *fromPtr* points to then upon return the actual length of the from data

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Do not wait for data, but rather return immediately
MSG_PEEK	"Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

Returns

Value	**Meaning**
>0	Number of bytes actually received from the socket.
0	EOF
-1	An error occurred

recvfrom will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_EINVAL	One of the parameters is invalid.
TM_ EMSGSIZE	The socket requires that message be received atomically, and bufferLength was too small.
TM_EPROTOTYPE	TCP protocol requires usage of **recv**, not **recvfrom**.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_EWOULDBLOCK	The socket is marked as non-blocking and no data is available to be read.

rresvport

#include <trsocket.h>

```
int                 rresvport
(
int *               portToReservePtr
);
```

Function Description

rresvport is used to create a TCP socket and bind a reserved port to the socket starting with the port to reserve given by the user. The *portToReservePtr* parameter is a value result parameter. The integer pointed to by *portToReservePtr* is the first port number that the function attempts to bind to. The caller typically initializes the starting port number to IPPORT_RESERVED – 1. (IPPORT_RESERVED is defined as 1024.) If the bind fails because that port is already used, then **rresvport** decrements the port number and tries again. If it finally reaches IPPORT_RESERVEDSTART (defined as 600) and finds it already in use, it returns –1 and set the socket error to TM_EAGAIN. If this function successfully binds to a reserved port number, it returns the socket descriptor to the user and stores the reserved port that the socket is bound to in the integer cell pointed to by *portToReservePtr.*

Parameters

Parameter	**Description**
portToReservePtr	Pointer to the port number to reserve, and to the port number reserved on success.

Returns

Value	**Meaning**
>= 0	Valid socket descpriptor
-1	An error occurred

If an error occured, the socket error can be retrieved by calling **tfGetSocketError** and using TM_SOCKET_ERROR as the socket descriptor parameter.

rresvport will fail if:

TM_AGAIN	The TCP/IP stack could not find any port number available between IPPORT_RESERVEDSTART and the port number to reserve.
TM_EINVAL	Bad parameter; pointer is null or port number to reserve is less than IPPORT_RESERVEDSTART (600).

select

```
#include <trsocket.h>

int                 select
(
int                 numberSockets,
fd_set *            readSocketsPtr,
fd_set *            writeSocketsPtr,
fd_set *            exceptionSocketsPtr,
struct timeval *    timeOutPtr
);
```

Function Description

select examines the socket descriptor sets whose addresses are passed in *readSocketsPtr*, *writeSocketsPtr*, and *exceptionSocketsPtr* to see if any of their socket descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. Out-of-band data is the only exceptional condition. The *numberSockets* argument specifies the number of socket descriptors to be tested. Its value is is the maximum socket descriptor to be tested, plus one. The socket descriptors from 0 to *numberSockets* -1 in the socket descriptor sets are examined. On return, **select** replaces the given socket descriptor sets with subsets consisting of those socket descriptors that are ready for the requested operation. The return value from the call to **select** is the number of ready socket descriptors. The socket descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such file descriptor sets:

FD_ZERO(&fdset);	Initializes a socket descriptor set (*fdset)* to the null set.
FD_SET(fd, &fdset);	Includes a particular socket descriptor *fd* in *fdset*.
FD_CLR(fd, &fdset);	Removes *fd* from *fdset*.
FD_ISSET(fd, &fdset);	Is non-zero if *fd* is a member of *fdset*, zero otherwise.

Note the term "fd" is used for BSD compatibility since select is used on both file systems and sockets under BSD Unix.

The timeout parameter specifies a length of time to wait for an event to occur before exiting this routine. s*truct timeval* contains the following members:

tv_sec	Number of seconds to wait
tv_usec	Number of microseconds to wait

If the total time is less than one millisecond, *select* will return immediately to the user. The resolution of this timer is equal to the system tick length (the amount of time between calls to *tfTimerUpdate* / *tfTimerUpdateIsr*).

Parameters

Parameter	**Description**
numberSockets	Biggest socket descriptor to be tested, plus one.
readSocketsPtr	The pointer to a mask of sockets to check for a read condition.
writeSocketsPtr	The pointer to a mask of sockets to check for a write condition.
exceptionSocketsPtr	The pointer to a mask of sockets to check for an exception condition: Out of Band data.
timeOutPtr	The pointer to a structure containing the length of time to wait for an event before exiting.

Returns

Value	**Meaning**
>0	Number of sockets that are ready
0	Time limit exceeded
-1	An error occurred

If an error occurred, the socket error can be retrieved by calling **tfGetSocketError** and using TM_SOCKET_ERROR as the socket descriptor parameter.

select will fail if:

TM_EBADF	One of the socket descriptors is bad.

send

```
#include <trsocket.h>

int                 send
(
int                 socketDescriptor,
char *              bufferPtr,
int                 bufferLength,
int                 flags
);
```

Function Description

send is used to transmit a message to another transport end-point. **send** may be used only when the socket is in a *connected* state. *socketDescriptor* is a socket created with **socket**.

If the message is too long to pass atomically through the underlying protocol (non TCP protocol), then the error TM_EMSGSIZE is returned and the message is not transmitted.

A return value of -1 indicates locally detected errors only. A positive return value does not implicitly mean the message was delivered, but rather that it was sent.

Blocking socket send: if the socket does not have enough buffer space available to hold the message being sent, **send** blocks.

Non blocking stream (TCP) socket send: if the socket does not have enough buffer space available to hold the message being sent, the send call does not block. It can send as much data from the message as can fit in the TCP buffer and returnes the length of the data sent. If none of the message data fits, then –1 is returned with socket error being set to TM_EWOULDBLOCK.

Non blocking datagram socket send: if the socket does not have enough buffer space available to hold the message being sent, no data is being sent and -1 is returned with socket error being set to TM_EWOULDBLOCK.

The **select** call may be used to determine when it is possible to send more data.

Sending Out-of-Band Data:

For example, if you have remote login application, and you want to interrupt with a ^C keystroke, at the socket level you want to be able to send the ^C flagged as special data (also called out-of-band data). You also want the TCP protocol to let the peer (or remote) TCP know as soon as possible that a special character is coming, and you want the peer (or remote) TCP to notify the peer (or remote) application as soon as possible. At the TCP level, this mechanism is called TCP urgent data. At the socket level, the mechanism is called out-of-band data. Out-of-band data generated by the socket layer, is implemented at the TCP layer with the urgent data mechanism. The user application can send one or several out-of-band data bytes. With TCP you cannot send the out-of-band data ahead of the data that has already been buffered in the TCP send buffer, but you can let the other side know (with the urgent flag, i.e the term urgent data) that out-of-band data is coming, and you can let the peer TCP know the offset of the current data to the last byte of out-of-band data. So with TCP, the out-of-band data byte(s) are not sent ahead of the data stream, but the TCP protocol can notify the remote TCP ahead of time that some out-of-band data byte(s) exist. What TCP does, is mark the byte stream where urgent data ends, and set the Urgent flag bit in the TCP header flag field, as long as it is sending data before ,or up to, the last byte of out-of-band data.

In your application, you can send out-of-band data, by calling the **send** function with the MSG_OOB flag. All the bytes of data sent that way (using **send** with the MSG_OOB flag) are out-of-band data bytes. Note that if you call **send** several times with out-of-band data, TCP will always keep track of where the last out-of-band byte of data is in the byte data stream, and flag this byte as the last byte of urgent data. To receive out-of-band data, please see the **recv** section of this manual.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to use to send data
bufferPtr	The buffer to send
bufferLength	The length of the buffer to send
flags	See below

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Do not wait for data send to complete, but rather return immediately.
MSG_OOB	Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support "out-of-band" data. Only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data.
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the operation. Only diagnostic or routing programs use it.

Returns

Value	**Meaning**
>=0	Number of bytes actually sent on the socket
-1	An error occurred

send will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_EINVAL	One of the parameters is invalid.the *bufferPtr* is NULL, the *bufferLength* is <= 0 or an unsupported flag is set.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_EHOSTUNREACH	Non-TCP socket only. No route to destination host.
TM_ EMSGSIZE	The socket requires that message to be sent atomically, and the message was too long.
TM_EWOULDBLOCK	The socket is marked as non-blocking and the send operation would block.
TM_ENOTCONN	Socket is not connected.
TM_ESHUTDOWN	User has issued a write **shutdown** or a **tfClose** call (TCP socket only).

sendto

#include <trsocket.h>

```
int                  sendto
(
int                  socketDescriptor,
char *               bufferPtr,
int                  bufferLength,
int                  flags,
const struct sockaddr* toPtr,
int                  toLength
);
```

Function Description

sendto is used to transmit a message to another transport end-point. **sendto** may be used at any time (either in a connected or unconnected state), but not for a TCP socket. *socketDescriptor* is a socket created with **socket**. The address of the target is given by *to* with *toLength* specifying its size.

If the message is too long to pass atomically through the underlying protocol, then –1 is returned with the socket error being set to TM_EMSGSIZE, and the message is not transmitted.

A return value of -1 indicates locally detected errors only. A positive return value does not implicitly mean the message was delivered, but rather that it was sent.

If the socket does not have enough buffer space available to hold the message being sent, and is in blocking mode, **sendto** blocks. If it is in non-blocking mode or the MSG_DONTWAIT flag has been set in the *flags* parameter, –1 is returned with the socket error being set to TM_EWOULDBLOCK.

The **select** call may be used to determine when it is possible to send more data.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to use to send data.
bufferPtr	The buffer to send.
bufferLength	The length of the buffer to send.
toPtr	The address to send the data to.
toLength	The length of the to area pointed to by *toPtr.*
flags	See below

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Don't wait for data send to complete, but rather return immediately.
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the operation. Only diagnostic or routing programs use it.

Returns

Value	**Meaning**
>=0	Number of bytes actually sent on the socket
-1	An error occurred
TM_EHOSTDOWN	Destination host is down

sendto will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_EINVAL	One of the parameters is invalid: the *bufferPtr* is NULL, the *bufferLength* is <= 0, an unsupported flag is set, *toPtr* is NULL or *toLength* contains an invalid length.
TM_EHOSTUNREACH	No route to destination host.
TM_ EMSGSIZE	The socket requires that message be sent atomically, and the message was too long.
TM_EPROTOTYPE	TCP protocol requires usage of **send** not **sendto**.
TM_EWOULDBLOCK	The socket is marked as non-blocking and the send operation would block.

setsockopt

```
#include <trsocket.h>

int                 setsockopt
(
int                 socketDescriptor,
int                 protocolLevel,
int                 optionName,
const char *        optionValue,
int                 optionLength
);
```

Function Description

setsockopt is used manipulate options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level. When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *protocolLevel* is specified as SOL_SOCKET. To manipulate options at any other level, *protocolLevel* is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the TCP protocol, *protocolLevel* is set to the TCP protocol number. The parameters *optionValuePtr* and *optionlength* are used to access option values for **setsockopt**. *optionName* and any specified options are passed un-interpreted to the appropriate protocol module for interpretation. The include file <trsocket.h> contains definitions for the options described below. Most socket-level options take an int pointer for *optionValuePtr*. For **setsockopt**, the integer value pointed to by the *optionValuePtr* parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a struct linger parameter that specifies the desired state of the option and the linger interval (see below). struct linger is defined in <trsocket.h>. struct linger contains the following members:

l_onoff	on = 1/off = 0
l_linger	linger time, in seconds

The following options are recognized at the socket level

SOL_SOCKET

***protocolLevel* options**	**Description**
SO_DONTROUTE	Enable/disable routing bypass for outgoing messages. **Default 0**.
SO_KEEPALIVE	Enable/disable keep connections alive. **Default 0**.
SO_LINGER	Linger on close if data is present. **Default is on with linger time of 60 seconds**.
SO_OOBINLINE	Enable/disable reception of out-of-band data in band. **Default 0**.
SO_REUSEADDR	Enable this socket option to bind the same port number to multiple sockets using different local IP addresses. Note that to use this socket option, you also need to uncomment TM_USE_REUSEADDR_LIST in trsystem.h. **Default 0 (disable)**.
SO_RCVLOWAT	The low water mark for receiving data.
SO_SNDLOWAT	The low water mark for sending data.
SO_R CVBUF	Set buffer size for input. **Default 8192 bytes**.
SO_SNDBUF	Set buffer size for output. **Default 8192 bytes**.
TM_SO_RCVCOPY	*TCP socket*: fraction use of a receive buffer below which we try and append to a previous receive buffer in the socket receive queue. *UDP socket*: fraction use of a receive buffer below which we try and copy to a new receive buffer, if there is already at least a buffer in the receive queue. This is to avoid keeping large pre-allocated receive buffers, which the user has not received yet, in the socket receive queue.**Default value is 4 (25%)**

TM_SO_SNDAPPEND	*TCP socket only*. Threshold in bytes of ‘send’ buffer below, which we try and append, to previous ‘send’ buffer in the TCP send queue. Only used with **send**, not with **tfZeroCopySend**. This is to try and regroup lots of partially empty small buffers in the TCP send queue waiting to be ACKED by the peer; otherwise we could run out of memory, since the remote TCP will delay sending ACKs. Note that care should be taken not to use **tfZeroCopySend** when sending small buffers, since we do not try and regroup small buffers with **tfZeroCopySend**. **Default value is 128 bytes.**
SO_UNPACKEDDATA	TI C3x and C5x DSP platforms only: If this option is enabled, all socket data will be sent and received in byte unpacked format. If this option is disabled, all socket data will be sent in a byte packed format, as received from the network. **Default 0 (disable)**

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a **bind** call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on a socket and a close on the socket is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close of the socket attempt until it is able to transmit the data or decides it is unable to deliver the information. A timeout period, termed the linger interval, is specified in the **setsockopt** call when SO_LINGER is requested. If SO_LINGER is disabled and a close on the socket is issued, the system will process the close of the socket in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with **recv** call without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options that adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data. The Internet protocols place an absolute limit of 64 Kbytes on these values for UDP and TCP sockets (in the default mode of operation).
The following options are recognized at the IP level:

IP_PROTOIP

***protocolLevel* options**	**Description**
IPO_HDRINCL	This is a toggle option used on Raw Sockets only. If the value is non-zero, it instructs the Turbo Treck stack that the user is including the IP header when sending data. **Default 0**.
IPO_TOS	IP type of service. **Default 0**.
IPO_TTL	IP Time To Live in seconds. **Default 64**.
IPO_SRCADDR	Our IP source address. **Default: first multi-home IP address on the outgoing interface**.
IPO_MULTICAST_TTL	Change the default IP TTL for outgoing multicast datagrams
IPO_MULTICAST_IF	Specify a configured IP address that will uniquely identify the outgoing interface for multicast datagrams sent on this socket. A zero IP address parameter indicates that we want to reset a previously set outgoing interface for multicast packets sent on that socket
IPO_ADD_MEMBERSHIP	Add group multicast IP address to given interface (see struct ip_mreq data type below)
IPO_DROP_MEMBERSHIP	Delete group multicast IP address

from given interface (see struct ip_mreq data type below)

ip_mreq structure definition

```
struct          ip_mreq
{
struct in_addr  imr_multiaddr;
struct in_addr  imr_interface
};
```

ip_mreq structure Members

Member	**Description**
imr_multiaddr	IP host group address that the user wants to join/leave
imr_interface	IP address of the local interface that the host group address is to be joined on, or is to leave from. If *imr_interface* is zero, then the default local interface selected with **tfSetMcastInterface** will be used instead.

The following options are recognized at the TCP level. Options marked with an asterix can only be changed prior to establishing a TCP connection.

IP_PROTOTCP ***protocolLevel* options**	**Description**
TCP_KEEPALIVE	Sets the idle time in seconds for a TCP connection, before it starts sending keep alive probes. It cannot be set below the default value. Note that keep alive probes will be sent only if the SO_KEEPALIVE socket option is enabled. **Default 7,200 seconds**.
TCP_MAXRT	Sets the amount of time in **seconds** before the connection is broken, once TCP starts retransmitting, or probing a zero window, when the peer does not respond. A TCP_MAXRT value of 0 means to use the system default, and -1 means to retransmit forever. If a positive value is specified, it may be rounded-up to the connection next retransmission time.

Note that unless the TCP_MAXRT value is –1 (wait forever), the connection can also be broken if the number of maximum retransmissions has been reached (TM_TCP_MAX_REXMIT).
See TM_TCP_MAX_REXMIT below.
Default 0 (Which means use system default of TM_TCP_MAX_REXMIT times network computed round trip time for an established connection; for a non established connection, since there is no computed round trip time yet, the connection can be broken when either **75 seconds**, or when TM_TCP_MAX_REXMIT times default network round trip time have elapsed, whichever occurs first).

TCP_MAXSEG

Sets the maximum TCP segment size sent on the network. Note that the TCP_MAXSEG value is the maximum amount of data (including TCP options, but not the TCP header) that can be sent per segment to the peer., i.e the amount of user data sent per segment is the value given by the TCP_MAXSEG option minus any enabled TCP option (for example 12 bytes for a TCP time stamp option) . The TCP_MAXSEG value can be decreased or increased prior to a connection establishment, but it is not recommended to set it to a value higher than the IP MTU minus 40 bytes (for example 1460 bytes on Ethernet), since this would cause fragmentation of TCP segments. ***Note: setting the TCP_MAXSEG option will inhibit the automactic computation of that value by the system based on the IP MTU (which avoids fragmentation), and will also inhibit Path Mtu Discovery***.
After the connection has started, this value cannot be changed. Note also that the TCP_MAXSEG value cannot be set below 64 bytes. **Default value is IP MTU minus**

40 bytes.

TCP_NODELAY
Set this option value to a non-zero value, to disable the Nagle algorithm that buffers the sent data inside the TCP. Useful to allow client’s TCP to send small packets as soon as possible (like mouse clicks). **Default 0**

TCP_NOPUSH
Set this option value to a non-zero value, to force TCP to delay sending any TCP data until a full sized segment is buffered in the TCP buffers. Useful for applications that send continuous big chunks of data like FTP, and know that more data is coming. (Normally the TCP code sends a non full-sized segment, only if it empties the TCP buffer). **Default 0**

TCP_STDURG
Set this option value to a zero value, if the peer is a Berkeley system since Berkeley systems set the urgent data pointer to point to last byte of urgent data+1. **Default 1** (urgent pointer points to last byte of urgent data as specified in RFC1122).

TM_TCP_PACKET
Set this option value to a non-zero value to make TCP behavelike a message-oriented protocol (i.e. respect packet boundaries) at the application level in both send and receive directions of data transfer. Note that for the receive direction to respect packet boundaries, the TCP peer which is sending must also implement similar functionality in its send direction. This is useful as a reliable alternative to UDP. Note that preserving packet boundaries with TCP will not work correctly if you use out-of-band data. TM_USE_TCP_PACKET must be defined in trsystem.h to use the TM_TCP_PACKET option. **Default 0**

*TM_TCP_SEL_ACK
Set this option value to a non-zero value, to enable sending the TCP selective Acknowlegment option. **Default 1**

*TM_TCP_WND_SCALE
Set this option value to a non-zero value, to enable sending the TCP window scale option. **Default 1**

*TM_TCP_TS	Set this option value to a non-zero value, to enable sending the Time stamp option. **Default 1**
TM_TCP_SLOW_START	Set this option value to zero, to disable the TCP slow start algorithm. **Default 1**
TM_TCP_DELAY_ACK	Sets the TCP delay ack time in milliseconds. **Default 200 milliseconds**
TM_TCP_MAX_REXMIT	Sets the maximum number of retransmissions without any response from the remote, before TCP gives up and aborts the connection. See also TCP_MAXRTabove. **Default 12**
TM_TCP_KEEPALIVE_CNT	Sets the maximum numbers of keep alive probes without any response from the remote, before TCP gives up and aborts the connection. See also TCP_KEEPALIVE above. **Default 8**
TM_TCP_FINWT2TIME	Sets the maximum amount of time TCP will wait for the remote side to close, after it initiated a close. **Default 600 seconds**
TM_TCP_2MSLTIME	Sets the maximum amount of time TCP will wait in the TIME WAIT state, once it has initiated a close of the connection. **Default 60 seconds**
TM_TCP_RTO_DEF	Sets the TCP default retransmission timeout value in milliseconds, used when no network round trip time has been computed yet. **Default 3,000 milliseconds**
TM_TCP_RTO_MIN	Sets the minimum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TM_TCP_RTO_MIN and TM_TCP_RTO_MAX. **Default 100 milliseconds**
TM_TCP_RTO_MAX	Sets the maximum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TM_TCP_RTO_MIN and TM_RTO_MAX. **Default 64,000 milliseconds**

TM_TCP_PROBE_MIN	Sets the minimum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TM_TCP_PROBE_MIN and TM_TCP_PROBE_MAX. **Default 500 milliseconds**
TM_TCP_PROBE_MAX	Sets the maximum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TM_TCP_PROBE_MIN and TM_TCP_PROBE_MAX. **Default 60,000 milliseconds**
TM_TCP_KEEPALIVE_INTV	Sets the interval between Keep Alive probes in seconds. See TM_TCP_KEEPALIVE_CNT. This value cannot be changed after a connection is established, and cannot be bigger than 120 seconds. **Default 75 seconds**

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to set the options on..
protocolLevel	The protocol to set the option on. See below.
optionName	The name of the option to set. See below and above.
optionValuePtr	The pointer to a user variable from which the option value is set. User variable is of data type described below.
optionLength	The size of the user variable. It is the size of the option data type described below.

ProtocolLevel	**Description**
SOL_SOCKET	Socket level protocol.
IP_PROTOIP	IP level protocol.
IP_PROTOTCP	TCP level protocol

ProtocolLevel	OptionName	Option data type	Option value
SOL_SOCKET	SO_DONTROUTE	int	0 or 1
	SO_KEEPALIVE	int	0 or 1
	SO_LINGER	struct linger	
	SO_OOBINLINE	int	0 or 1
	SO_RCVBUF	unsigned long	
	SO_RCVLOWAT	unsigned long	
	SO_REUSEADDR	int	0 or 1
	SO_SNDBUF	unsigned long	
	SO_SNDLOWAT	unsigned long	
	TM_SO_RCVCOPY	unsigned int	
	TM_SO_SNDAPPEND	unsigned int	
	SO_UNPACKEDDATA	int	0 or 1
IP_PROTOIP	IPO_TOS	unsigned char	
	IPO_TTL	unsigned char	
	IPO_SRCADDR	ttUserIpAddress	
	IPO_MULTICAST_TTL	unsigned char	
	IPO_MULTICAST_IF	struct in_addr	
	IPO_ADD_MEMBERSHIP	struct ip_mreq	
	IPO_DROP_MEMBERSHIP	struct ip_mreq	
IP_PROTOTCP	TCP_KEEPALIVE	int	
	TCP_MAXRT	int	
	TCP_MAXSEG	int	
	TCP_NODELAY	int	0 or 1
	TCP_NOPUSH	int	0 or 1
	TCP_STDURG	int	0 or 1
	TM_TCP_PACKET	int	0 or 1
	TM_TCP_2MSLTIME	int	
	TM_TCP_DELAY_ACK	int	
	TM_TCP_FINWT2TIME	int	
	TM_TCP_KEEPALIVE_CNT	int	
	TM_TCP_KEEPALIVE_INTV	int	
	TM_TCP_MAX_REXMIT	int	
	TM_TCP_PROBE_MAX	unsigned long	
	TM_TCP_PROBE_MIN	unsigned long	
	TM_TCP_RTO_DEF	unsigned long	
	TM_TCP_RTO_MAX	unsigned long	
	TM_TCP_RTO_MIN	unsigned long	
	TM_TCP_SEL_ACK	int	0 or 1
	TM_TCP_SLOW_START	int	0 or 1
	TM_TCP_TS	int	0 or 1
	TM_TCP_WND_SCALE	int	0 or 1

Returns

Value	**Meaning**
0	Successful set of option
-1	An error occurred

setsockopt will fail if:

TM_EBADF	The socket descriptor is invalid
TM_EINVAL	One of the parameters is invalid
TM_ ENOPROTOOPT	The option is unknown at the level indicated.
TM_EPERM	Option cannot be set after the connection has been established.
TM_EPERM	IPO_HDRINCL option cannot be set on non-raw sockets.
TM_ENETDOWN	Specified interface not yet configured.
TM_EADDRINUSE	Multicast host group already added to the interface.
TM_ENOBUF	Not enough memory to add new multicast entry.
TM_ENOENT	Attempted to delete a non-existent multicast entry on the specified interface.

shutdown

```
#include <trsocket.h>

int                 shutdown
(
int                 socketDescriptor,
int                 howToShutdown
);
```

Function Description

Shutdown a socket in read, write, or both directions determined by the parameter *howToShutdown.*

Parameters

Parameter	**Description**
socketDescriptor	The socket to shutdown
howToShutdown	Direction: 0 = Read 1 = Write 2 = Both

Returns

Value	**Meaning**
0	Success
-1	An error occurred

shutdown will fail if:

TM_EBADF	The socket descriptor is invalid
TM_EINVAL	One of the parameters is invalid
TM_EOPNOTSUPP	Invalid socket type - can only shutdown TCP sockets.
TM_ESHUTDOWN	Socket is already closed or is in the process of closing.

socket

```
#include <trsocket.h>

int                socket
(
int                family,
int                type,
int                protocol
);
```

Function Description

socket creates an endpoint for communication and returns a descriptor. The *family* parameter specifies a communications domain in which communication will take place; this selects the protocol family that should be used. The protocol family is generally the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file <trsocket.h>. If *protocol* has been specified, but no exact match for the tuplet family, type, and protocol is found, then the first entry containing the specified family and type with zero for protocol will be used. The currently understood format is PF_INET for ARPA Internet protocols. The socket has the indicated *type*, which specifies the communication semantics.

Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW

A SOCK_STREAM type provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism is supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length); a SOCK_DGRAM user is required to read an entire packet with each **recv** call or variation of **recv** call, otherwise an error code of TM_EMSGSIZE is returned. *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, multiple protocols may exist, in which case, a particular protocol must be specified in this manner.

The protocol number to use is particular to the "communication domain" in which communication is to take place. If the caller specifies a protocol, then it will be packaged into a socket level option request and sent to the underlying protocol layers. Sockets of type SOCK_STREAM are full-duplex byte streams. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with **connect** on the client side. On the server side, the server must call **listen** and then **accept**. Once connected, data may be transferred using **recv** and **send** calls or some variant of the **send** and **recv** calls. When a session has been completed, a **close** of the socket should be performed. The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated.

If a piece of data (for which the peer protocol has buffer space) cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with (-1) return value and with TM_ETIMEDOUT as the specific socket error. The TCP protocols optionally keep sockets "warm" by forcing transmissions roughly every two hours in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (for instance 5 minutes). SOCK_DGRAM or SOCK_RAW sockets allow datagrams to be sent to correspondents named in **sendto** calls. Datagrams are generally received with **recvfrom** which returns the next datagram with its return address. The operation of sockets is controlled by socket level *options*. These options are defined in the file <trsocket.h>. **setsockopt** and **getsockopt** are used to set and get options, respectively.

Parameters

Parameter	**Description**
family	The protocol family to use for this socket (currently only PF_INET is used).
type	The type of socket.
protocol	The layer 4 protocol to use for this socket.

Family	***Type***	***Protocol***	**Actual protocol**
PF_INET	SOCK_DGRAM	IPPROTO_UDP	**UDP**
PF_INET	SOCK_STREAM	IPPROTO_TCP	**TCP**
PF_INET	SOCK_RAW	IPPROTO_ICMP	**ICMP**
PF_INET	SOCK_RAW	IPRPTOTO_IGMP	**IGMP**

Returns

New Socket Descriptor or –1 on error. If an error occured, the socket error can be retrieved by calling **tfGetSocketError** and using TM_SOCKET_ERROR as the socket descriptor parameter

socket will fail if:

TM_ EMFILE	No more sockets are available
TM_ENOBUFS	There was insufficient user memory available to complete the operation
TM_ EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this family.

tfClose

```
#include <trsocket.h>

int                 tfClose
(
int                 socketDescriptor
);
```

Function Description

This function is used to close a socket. It is not called **close** to avoid confusion with an embedded kernel file system call.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to close

Returns

Value	**Meaning**
0	Operation completed successfully
-1	An error occurred

tfClose can fail for the following reasons:

TM_EBADF	The socket descriptor is invalid.
TM_EALREAY	A previous **tfClose** call is already in progress.
TM_ETIMEDOUT	The linger option was on with a non-zero timeout value, and the linger timeout expired before the TCP close handshake with the remote host could complete (blocking TCP socket only).

tfIoctl

```
#include <trsocket.h>

int                 tfIoctl
(
int                 socketDescriptor,
unsigned long       request,
int *               argumentPtr
);
```

Function Description

This function is used to set/clear nonblocking I/O, to get the number of bytes to read, or to check whether the specified socket's read pointer is currently at the out of band mark. It is not called **ioctl** to avoid confusion with an embedded kernel file system call.

Request	**Description**
FIONBIO	Set/clear nonbllocking I/O: if the int cell pointed to by *argumentPtr* contains a non-zero value, then the specified socket non-blocking flag is turned on. If it contains a zero value, then the specified socket non-blocking flag is turned off. See also **tfBlockingState**.
FIONREAD	Stores in the int cell pointed to by *argumentPtr* the number of bytes available to read from the socket descriptor. See also **tfGetWaitingBytes.**
SIOCATMARK	Stores in the int cell pointed to by *argumentPtr* a non-zero value if the specified socket's read pointer is currently at the out-of-band mark, zero otherwise. See **revc** call for a description of out-of-band data. See also **tfGetOobDataOffset**.

Example

Given a valid socketDescriptor, the following code will turn on the socket’s nonblocking I/O flag.

```
int argValue;
argValue = 1;
tfIoctl(socketDescriptor, FIONBIO, &argValue);
```

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor we want to perform the ioctl request on.
request	FIONBIO, FIONREAD, or SIOCATMARK
argumentPtr	A pointer to an int cell in which to store the request parameter or request result.

Returns

Value	**Meaning**
0	Success.
-1	An error has occured.

tfioctl can fail for the following reasons:

TM_EBADF	The socket descriptor is invalid.
TM_EINVAL	Request is not one of FIONBIO, FIONREAD, or SOIOCATMARK.

tfRead

```
#include <trsocket.h>

int                 tfRead
(
int                 socketDescriptor,
char *              bufferPtr,
int                 bufferLength
);
```

Function Description

tfRead is used to receive messages from another socket. It is not called **read** to avoid confusion with an embedded kernel file system call. It operates identically to **recv**, except that it does not have any flag parameter, and hence does not support out-of-band data, or overwriting the blocking state of the socket for the duration of the call.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to receive data from.
bufferPtr	The buffer to put the received data into
bufferLength	The length of the buffer area that *bufferPtr* points to

Returns

Value	**Meaning**
>0	Number of bytes actually received from the socket
0	EOF
-1	An error occurred

tfread will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_ EMSGSIZE	The socket requires that message be received atomically, and bufferLength was too small.
TM_EWOULDBLOCK	The socket is marked as non-blocking and no data is available to be read.
TM_ESHUTDOWN	The remote socket has closed the connection, and there is no more data to be read (TCP socket only).
TM_ENOTCONN	Socket is not connected.
TM_EINVAL	One of the parameter is invalid.

tfWrite

```
#include <trsocket.h>

int                 tfWrite
(
int                 socketDescriptor,
char *              bufferPtr,
int                 bufferLength
);
```

Function Description

tfWrite is used to transmit a message to another transport end-point. It is not called **write** to avoid confusion with an embedded kernel file system call. It operates identically to **send**, except that it does not have any flags parameter, and hence does not support sending out-of-band data, or overwriting the blocking state of the socket for the duration of the call.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to use to send data.
bufferPtr	The buffer to send.
bufferLength	The length of the buffer to send.

Returns

Value	**Meaning**
>=0	Number of bytes actually sent on the socket.
-1	An error occurred.

tfWrite will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_EHOSTUNREACH	Non-TCP socket only. No route to destination host.
TM_ EMSGSIZE	The socket requires that message to be sent atomically, and the message was too long.
TM_EWOULDBLOCK	The socket is marked as non-blocking and the write operation would block.

writev

```
#include <trsocket.h>

int                    writev
(
int                    socketDescriptor,
const struct iovec     *iov,
int                    iovcnt
);
```

Function Description

writev functions as a scatter write because the written data can be placed in multiple buffers. **writev** attempts to write data to the socket *socketDescriptor* by gathering the data from into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1].

The iovec structure contains the following members:

caddr_tiov_base;
intiov_len;

Each iovec entry specifies the base address and length of an area in memory from where data should be gathered. **writev** always reads one buffer completely before proceeding to the next. On success, **writev** return the number of bytes actually written; this number may be less than the total of all of the iov_len values if there is not enough space on the send queues.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to write data to.
iov	The list of buffers to gather and send the data from.
iovcnt	The number of buffers in the list.

Returns

Value	**Meaning**
>=0	Number of bytes actually written
-1	An error occurred

writev will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_EINVAL	The iovcnt is 0 or less than 0. The sum of the iov_len values overflowed an integer.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_ EMSGSIZE	The socket requires that message to be sent atomically, and the message was too long.
TM_ EWOULDBLOCK	The socket is marked as non-blocking and all the data could not be written.
TM_ENOTCONN	The socket is not connected.
TM_ESHUTDOWN	The user has issued a write **shutdown** call or a **tfClose** call (TCP socket only).

Socket Extension Calls

tfBindNoCheck

```
#include <trsocket.h>

int                    tfBindNoCheck
(
int                    socketDescriptor,
const struct sockaddr  *addressPtr,
int                    addressLength
);
```

Function Description

tfBindNoCheck assigns an address to an unnamed socket. When a socket is created with **socket**, it exists in an address family space but has no address assigned. **tfBindNoCheck** requests that the address pointed to by *addressPtr* be assigned to the socket. Clients do not normally require that an address be assigned to a socket. However, servers usually require that the socket be bound to a "well known" address. The port number must be less than 32768 (TM_SOC_NO_INDEX), or could be 0xFFFF (TM_WILD_PORT). Binding to the TM_WILD_PORT port number allows a server to listen for incoming connection requests on all the ports.

This function is similar to **bind**, except that **bind** checks that the address that the user wants to bind to is a valid configured address on an interface; **tfBindNoCheck** does not check for that, and allows the user to bind to any IP address.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to assign an IP address and port number to.
addressPtr	The pointer to the structure containing the address to assign.
addressLength	The length of the address structure.

Returns

Value	**Meaning**
0	Success
-1	An error occurred

tfBindNoCheck can fail for any of the following reasons:

TM_EADDRINUSE	The specified address is already in use.
TM_EBADF	*socketDescriptor* is not a valid descriptor.
TM_EINVAL	One of the passed parameters is invalid, or socket is already bound.
TM_EINPROGRESS	**tfBindNoCheck** is already running.

tfBlockingState

#include <trsocket.h>

```
int                 tfBlockingState
(
int                 socketDescriptor,
int                 blockingState
);
```

Function Description

This function is used to set blocking or non-blocking on a socket as the default mode of operation. This can be overridden with the MSG_DONTWAIT flags on subsequent calls. The **tfIoctl** call with the *FIONBIO* request can be used instead of the **tfBlockingState** call.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to set the non-blocking flag on.
blockingState	One of the following:
	TM_BLOCKING_OFF
	TM_BLOCKING_ON

Returns

Value	**Meaning**
0	Success
-1	Error

tfBlockingState can fail for the following reason:

TM_ EBADF	The socket descriptor is invalid.
TM_EINVAL	*blockingState* parameter is invalid.

tfFlushRecvQ

```
#include <trsocket.h>

int                 tfFlushRecvQ
(
int                 socketDescriptor
);
```

Function Description

This function flushes the socket receive buffer for the specified socket. This can be useful with UDP sockets to flush queued packets that are too big for the application to process (i.e. when the call to **recvfrom** returns TM_EMSGSIZE). Note that this function flushes **all** packets queued to be received on the specified socket.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to flush the socket receive buffer for.

Returns

Value	**Meaning**
TM_ENOERROR	This function always returns TM_ENOERROR.

tfFreeDynamicMemory

```
#include <trsocket.h>

int                  tfFreeDynamicMemory
(
void
);
```

Function Description

This function frees all memory allocated dynamically by the Turbo Treck internal memory management system. Called by the user, when the user does not want to use the Turbo Treck stack anymore, and wants all currently unused memory to be returned to the user's system.

Note: The user cannot free the Turbo Treck Dynamic memory if the user uses the Turbo Treck simple heap.

Returns

Value	**Meaning**
0	Success
TM_ENOENT	The user cannot free the dynamic memory either because the user had disabled the dynamic memory by defining the following macro: TM_DISABLE_DYNAMIC_MEMORY in *trsystem.h*, or because the user uses the Turbo Treck simple heap.

tfFreeZeroCopyBuffer

```
#include <trsocket.h>

int                 tfFreeZeroCopyBuffer
(
ttUserMessage       bufferHandle
);
```

Function Description

This function is used to free a zero copy buffer that was allocated via **tfGetZeroCopyBuffer**, **tfZeroCopyRecv**, or **tfZeroCopyRecvFrom**.

Parameters

Parameter	**Description**
bufferHandle	The buffer handle of the buffer to free.

Returns

Value	**Meaning**
0	Success
-1	Error. The buffer did not belong to the user.

tfGetOobDataOffset

```
#include <trsocket.h>

int                tfGetOobDataOffset
(
int                socketDescriptor
);
```

Function Description

This function is used to get the offset of the out of band data sitting in the receive queue. This allows the user to determine where the out of band data is located in the stream. The **tfIoctl** call with the *SOIOCATMARK* request can be used instead of the **tfGetOobDataOffset** call, but the **tIfoctl** call only tells us whether we are at the out-of-band byte mark, whereas the **tfGetOobDataOffset** call gives us the offset to the out-of-band byte. See **recv** call for out-of-band data description.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to get the number of bytes in the data stream to the out-of-band data offset.

Returns

Number of out of band offset, -1 if call failed

tfGetOobDataOffset can fail for the following reasons:

TM_ EBADF	The socket descriptor is invalid.
TM_EINVAL	No Out of Band Data is waiting to be read.

tfGetSendCompltBytes

```
#include <trsocket.h>

int                 tfGetSendCompltBytes
(
int                 socketDescriptor
);
```

Function Description

This function is used to get the number of sent bytes that have been acked since the last call to **tfGetSendCompltBytes** call by the peer on a TCP socket.

Parameters

Parameter	**Description**
socketDescriptor	The socket number to check on the amount of bytes acked by the peer.

Returns

Value	**Meaning**
>=0	Number of bytes that have been sent to, and acked by the TCP peer.
-1	An error has occurred

tfGetSendCompltBytes can fail for the following reasons:

TM_ EBADF	The socket descriptor is invalid.
TM_EOPNOTSUPP	The socket is not a TCP socket.

tfGetSocketError

```
#include <trsocket.h>

int                 tfGetSocketError
(
int                 socketDescriptor
);
```

Function Description

This function is used when any socket call fails (TM_SOCKET_ERROR), to get the error value back. This call has been added to allow for the lack of a per-process errno value that is lacking in most embedded real-time kernels.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to get the error on.

Returns

The last errno value for a socket

tfGetWaitingBytes

```
#include <trsocket.h>

int                 tfGetWaitingBytes
(
int                 socketDescriptor
);
```

Function Description

This function is used to get the number of bytes waiting to be read on a socket. The **tfIoctl** call with the *FIONREAD* request can be used instead.

Parameters

Parameter	Description
socketDescriptor	The socket number to check on the amount of waiting bytes

Returns

Value	Meaning
>=0	Number of bytes waiting to be read
-1	An error has occurred.

tfGetWaitingBytes can fail for the following reason:

TM_ EBADF	The socket descriptor is invalid.

tfGetZeroCopyBuffer

#include <trsocket.h>

```
ttUserMessage     tfGetZeroCopyBuffer
(
int               size,
char **           dataPtrPtr
);
```

Function Description

This function is used to get a Zero Copy Buffer. This buffer can then be used with **tfZeroCopySend** and **tfZeroCopySendTo** for zero copy send functions. This is a TRUE zero copy if the device driver supports DMA sending.

Parameters

Parameter	**Description**
size	The size of the buffer to be used for the zero copy.
dataPtrPtr	Pointer to a pointer that is the beginning of the user's data area. This is where the user can store the data to be sent.

Returns

The Zero Copy buffer or (ttUserMessage *)0 if not successful

tfGetZeroCopyBuffer will fail for the following reason:

There was insufficient user memory availbale to complete the operation.

tfInetToAscii

```
#include <trsocket.h>

void                  tfInetToAscii
(
unsigned long         ipAddress,
char *                outputBuffer
);
```

Function Description

This function is used to convert an IP address to the dotted string notation.

Parameters

Parameter	**Description**
ipAddress	The IP address to convert
outputBuffer	A character buffer to store the result into. It is the caller's responsibility to ensure that there is ample room in the buffer for the null terminated string. The output will never be longer than 16 bytes.

Returns

Nothing

tfIpScatteredSend

```
#include <trsocket.h>

int                 tfIpScatteredSend
(
ttUserInterface     interfaceHandle,
ttUserBlockPtr      userBlockPtr,
int                 userBlockCount,
ttUserFreeFuncPtr userFreeFunctionPtr
);
```

Function Description

This function allows the user to send an IP datagram directly to the stack without using a socket interface, without the Turbo Treck stack changing any IP header field (except if IP fragmentation is needed), and directly from user owned scattered data buffers. Even the IP header itself could be scattered among several data buffers. The user passes a pointer to an array of user block data of type ttUserBlock and the number of elements in the array. Each ttUserBlock element contains a pointer to a user buffer, a pointer to the beginning of the user data in the user buffer, and the user data length in the user buffer. Upon return from this routine, the user can reuse the ttUserBlock array, but the Turbo Treck stack owns the user buffers that were pointed to by the ttUserBlock elements. The userFreeFunction will be called by the Turbo Treck stack for each user buffer, when the data has been sent out on the network and the device driver no longer needs to access the data in the user buffer. An example of **tfIpScatteredSend()** usage is shown in the loop back test module **txscatlp.c**.

If the user uses a preemptive kernel, i.e. TM_TRECK_PREEMPTIVE_KERNEL is defined in trsystem.h, it is the user's responsibility to ensure that the user free function is re-entrant, as it could potentially be called from different threads.

Parameters

Parameter	**Description**
interfacehandle	If the user knows the interface the packet is to be sent out to, then this field is non-null.
userBlockPtr	Pointer to the first element (data type ttUserBlock) of the user array that contains information about the user scattered data.
UserBlockCount	Number of elements in the above array

ttUserBlock data type:

```
typedef struct tsUserBlock
{
char * userBufferPtr;    Pointer to buffer (passed to the free routine)
char * userDataPtr;      Pointer to beginning of data
int    userDataLength;   Data length
} ttUserBlock;
```

Function prototype for the user supplied user data free call back function:

```
int userFreeFunc (char TM_FAR * bufferPtr);
```

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EINVAL	One of the parameters is bad, i.e. userBlockPtr is null, or userBlockCount is less or equal to 0 or userFreeFunction is null, or interface handle is invalid
TM_ENOBUFS	Could not allocate Turbo Treck headers to send the data or could not allocate an ARP entry
TM_EFAULT	Bad packet. The IP header length is bigger than the total data length in the user scattered buffers or IP header size is bigger than the maximum allowed by the RFC, i.e. 60, or it less than the minimum, 20
TM_EMSGSIZE	Data length of the IP datagram is bigger than the IP MTU, and fragmentation is not allowed
TM_EHOSTUNREACH	No route to host
TM_EDESTADDRREQ	Destination IP address in the IP header is zero.
TM_ENOENT	No interface could be found to send the packet out (only if destination IP address is limited broadcast)
TM_ENXIO	Outgoing interface is closed
TM_EIO	Interface transmit queue if full (only if interface transmit queue is used).
Other error code	Returned by device driver send function

tfRawSocket

#include <trsocket.h>

```
int                 tfRawSocket
(
ttUserIpAddress     ipAddress,
int                 protocol
);
```

Function Description

This function creates a raw socket. A raw socket enables the user to either send data above the IP layer, or to send data with the IP header. In the latter case, the user must set the IPO_HDRINCL option at the IP level in the **setsockopt** call. When the user receives data on the raw socket, the IP header is always included. Given a transport layer protocol and IP address, **tfRawSocket** returns a raw socket bound to TM_RAW_IP_PORT and ipAddress. Note that ipAddress can be a zero IP address if the user does not want to bind to a specific IP address. The user can use this function only if TM_USE_RAW_SOCKET has been defined in **trsystem.h**. An example of **tfRawSocket** usage is shown in the loop back test module **txscatlp.c**.

Parameters

Parameter	**Description**
ipAddress	User IP address to bind the raw socket with.
Protocol	Protocol above IP, for example: IPPROTO_IGMP IPPROTO_ICMP IPPROTO_OSPF

Returns

Value	**Meaning**
>= 0	Success
TM_SOCKET_ERROR	Call failed. The user can retrieve the error code with the **tfGetSocketError** (TM_SOCKET_ERROR) API. See below for table of possible error codes.

tfRawSocket will fail if:

TM_EINVAL
Invalid protocol for raw socket send/receive.

The following protocols are disallowed:

IPPROTO_UDP

IPPROTO_TCP

TM_EINVAL
There is no interface configured with the specified IP address (only if ipAddress parameter is not zero).

TM_EADDRINUSE
A raw socket has already been opened for this protocol.

tfRecvFromTo

#include <trsocket.h>

```
int                           tfRecvFromTo
(
int                           socketDescriptor,
Char *                        bufferPtr,
int                           bufferLength,
int                           flags,
struct sockaddr *             fromAddressPtr,
int *                         addressLengthPtr,
struct sockaddr *             toAddressPtr
);
```

Function Description

tfRecvFromTo behaves the same as **recvfrom** except for the use of the parameter *toAddressPointer.* This parameter specifies the address to which data was sent. **tfRecvFromTo** is used to receive messages from another socket. **tfRecvFromTo** may be used to receive data on a socket whether it is in a connected state or not, but not on a TCP socket. *socketDescriptor* is a socket created with **socket**. If *fromAddressPtr* is not a NULL pointer, the source address of the message is filled in. If *toAddressPtr* is not a NULL pointer, the destination address of the message is filled in. *addressLengthPtr* is a value-result parameter, initialized to the size of the buffer associated with *fromAddressPtr*, and with *toAddressPtr*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see **socket**). If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking, or the MSG_DONTWAIT flag is set in the *flags* parameter, in which case -1 is returned with socket error being set to EWOULDBLOCK.

select may be used to determine when more data arrives, or/and when out-of-band data arrives.

tfRegisterSocketCB may be used to asynchronously determine when more data arrives, or/and when out-of-band data arrives.tfRegisterIpForwCB

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to use to receive data
bufferPtr	The buffer to store the received data
bufferLength	The length of the buffer to store the received data
flags	MSG_DONTWAIT: Don't wait for data 0: wait for data to come in
fromAddressPtr	Where to store the address the data came from
addressLengthPtr	Length of the area pointed to by *fromAddressPtr*, or *toAddressPtr*
toAddressPtr	Where to store the address the data was sent to

Returns

Value	**Meaning**
>=0	Number of bytes actually sent on the socket
-1	An error occurred, error can be retrieved with **tfGetSocketError**

tfRecvFromTo can fail for the following reasons:

TM_EBADF	The socket descriptor is invalid
TM_ENOBUFS	There was insufficient user memory available to complete the operation
TM_EMSGSIZE	The message was too long
TM_EPROTOTYPE	TCP protocol requires usage of **recv**, not **tfRecvFromTo**
TM_EWOULDBLOCK	The socket is marked as non-blocking and the **tfRecvFromTo** operation would block

tfRegisterIpForwCB

```
#include <trsocket.h>

int                     tfRegisterIpForwCB
(
ttUserIpForwCBFuncPtr  ipForwCBFuncPtr
);
```

Function Description

Used to register a function for the Turbo Treck stack to call when a packet cannot be forwarded. The call back function parameters will indicate the source IP address, and destination IP address of the packet in network byte order. This is useful to let the user know that a packet cannot be forwarded because a dial up interface is closed for example. In that case the user call back function could trigger a dial-up on demand to allow forwarding of subsequent packets. If the user does not want to enable or configure the interface, then the call back function should return a non-zero error code. In that case the stack will send a host unreachable ICMP error message as if no call back function had been registered. If the user wants to enable or configure the interface, then the call back function should return TM_ENOERROR. In that case the Turbo Treck stack will silently drop the packet without sending an ICMP error message, allowing the sender to try and send more data.

Parameters

Parameter	**Description**
ipForwCBFuncPtr	Pointer to user call back function that returns an integer as described above, and that takes two parameters of type ttUserIpAddress, the first one being the source IP address in network byte order of the IP datagram to be forwarded, and the second one being its destination IP address in network byte order.

Function prototype for the user supplied IP forward call back function:

```
int ipForwCBFunc ( ttUserIpAddress srcIpAddress,
                   ttUserIpAddress destIpAddress );
```

Returns

Value	**Meaning**
TM_ENOERROR	This function always returns TM_ENOERROR

tfResetConnection

#include <trsocket.h>

```
int     tfResetConnection
(
int     socketDescriptor
);
```

Function Description

This function is used to abort a connection on a socket. It only works with TCP (STREAM) sockets and sends a RST and discards all outstanding data.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to Reset

Returns

Value	**Meaning**
0	Operation completed successfully
-1	An error occurred

tfResetConnection can fail for the following reasons:

TM_EBADF	The socket descriptor is invalid
TM_EOPNOTSUPP	The socket is not a STREAM socket

tfSendToFrom

#include <trsocket.h>

```
int                 tfSendToFrom
(
int                 socketDescriptor,
char *              bufferPtr,
int                 bufferLength,
int                 flags,
const struct sockaddr *toAddressPtr,
int                 addressLength,
const struct sockaddr *fromAddressPtr
);
```

Function Description

tfSendToFrom behaves the same as sendto except for the use of the parameter *fromAddressPointer*. This parameter specifies the address from where data is to be sent. tfSendToFrom is used to transmit a message to another transport endpoint. tfSendToFrom may be used at any time (either in a connected or unconnected state), but not for a TCP socket. *socketDescriptor* is a socket created with socket. The address of the target is given by *toAddressPtr,* with *addressLength* specifying its size.

If the message is too long to pass atomically through the underlying protocol, then –1 is returned with the socket error being set to TM_EMSGSIZE, and the message is not transmitted.

A return value of -1 indicates locally detected errors only. A positive return value does not implicitly mean the message was delivered, but rather that it was sent.

If the socket does not have enough buffer space available to hold the message being sent, and is in blocking mode, **tfSendToFrom** blocks. If it is in non-blocking mode or the MSG_DONTWAIT flag has been set in the *flags* parameter, –1 is returned with the socket error being set to TM_EWOULDBLOCK. The **select** call may be used to determine when it is possible to send more data.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to use to send data
bufferPtr	The buffer to send
bufferLength	The length of the buffer to send
flags	MSG_DONTWAIT: Don't wait for room in the socket send queue 0: wait for room in the socket send queue
toAddressPtr	The address to send the data to
addressLength	The length of the area pointed to by *toAddressPtr*, or *fromAddressPtr*
fromAddressPtr	The address to send the data from

Returns

Value	**Meaning**
>=0	Number of bytes actually sent on the socket
-1	An error occurred, error can be retrieved with **tfGetSocketError**

tfSendToFrom can fail for the following reasons:

TM_EBADF	The socket descriptor is invalid
TM_ENOBUFS	There was insufficient user memory available to complete the operation
TM_EMSGSIZE	The message was too long
TM_EPROTOTYPE	TCP protocol requires usage of **send**, not **tfSendToFrom**
TM_EWOULDBLOCK	The socket is marked as non-blocking and the send operation would blocktfSendToInterface

tfSendToInterface

```
#include <trsocket.h>

int                        tfSendToInterface
(
int                        socketDescriptor,
char *                     bufferPtr,
int                        bufferLength,
int                        flags,
const struct sockaddr*     toPtr,
int                        toLength,
ttUserInterface            interfaceHandle,
unsigned char              mhomeIndex
);
```

Function Description

tfSendToInterface is used to transmit a message to another transport end-point. **tfSendToInterface** may be used at any time (either in a connected or unconnected state), but not for a TCP socket. *socketDescriptor* is a socket created with **socket**. The address of the target is given by *to* with *toLength* specifying its size.

If the message is too long to pass atomically through the underlying protocol, then –1 is returned with the socket error being set to TM_EMSGSIZE, and the message is not transmitted.

A return value of -1 indicates locally detected errors only. A positive return value does not implicitly mean the message was delivered, but rather that it was sent.

If the socket does not have enough buffer space available to hold the message being sent, and is in blocking mode, **tfSendToInterface** blocks. If it is in non-blocking mode or the MSG_DONTWAIT flag has been set in the *flags* parameter, –1 is returned with the socket error being set to TM_EWOULDBLOCK.

The **select** call may be used to determine when it is possible to send more data.

Note: If tfSendToInterface is used on an Ethernet interface with the interface configured temporarily with a zero IP address (TM_DEV_IP_USER_BOOT flag), then the only allowed destination IP addresses are either limited broadcast (i.e. all F’s), or multicast addresses.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to use to send data.
bufferPtr	The buffer to send.
bufferLength	The length of the buffer to send.
toPtr	The address to send the data to.
toLength	The length of the to area pointed to by *toPtr*.
flags	See below
InterfaceHandle	Interface to send the data through
mhomeIndex	Multihome index on the configured interface.

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Don’t wait for data send to complete, but rather return immediately.
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the operation. Only diagnostic or routing programs use it.

Returns

Value	**Meaning**
>=0	Number of bytes actually sent on the socket
-1	An error occurred

tfSendToInterface will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_EHOSTUNREACH	No route to destination host.
TM_ EMSGSIZE	The socket requires that message be sent atomically, and the message was too long.
TM_EPROTOTYPE	TCP protocol requires usage of **send** not **tfSendToInterface**.
TM_EWOULDBLOCK	The socket is marked as non-blocking and the send operation would block.
TM_EINVAL	One of the parameters is bad: the interface handle or multihome index is invalid, or the bufferPtr is null, or bufferLength is zero.

tfSocketArrayWalk

```
#include <trsocket.h>

int                 tfSocketArrayWalk
(
ttWalkCBFuncPtr     callBackFuncPtr,
ttUserVoidPtr       argPtr
);
```

Function Description

This function is used to walk the list of open sockets, calling the user supplied call back function, passing the socket descriptor, and user argument *argPtr*, to it until we reach the end of the list, or the user call back function returns an error whichever comes first. Return error value from the call back function to the user.

Parameters

Parameter	Description
callBackFuncPtr	The function pointer to the user function to call for each open socket. See below for function prototype.
argPtr	User pointer to be passed back to the call back function.

Function prototype for the user supplied call back function:

```
int                          callBackFunc
(
int                          socketDescriptor,
ttUserVoidPtr                argPtr
);
```

Returns

Value	Meaning
0	Success
TM_EINVAL	The callback function pointer is null.
Other	As returned by the user call back function.

tfSocketScatteredSendTo

```
#include <trsocket.h>

int                           tfSocketScatteredSendTo

(
int                           socketDescriptor,
ttUserBlockPtr                userBlockPtr,
int                           userBlockCount,
ttUserFreeFuncPtr             userFreeFunction,
int                           flags,
const struct sockaddr TM_FAR* toAddressPtr,
int                           toAddressLength
);
```

Function Description

This function allows the user to send data on a non-TCP socket directly from user owned scattered data buffers. The user passes a pointer to an array of user block data of type ttUserBlock, and the number of elements in the array. Each ttUserBlock element contains a pointer to a user buffer, a pointer to the beginning of the user data in the user buffer, and the user data length in the user buffer. Upon return from this routine, the user can reuse the array of ttUserBlock, but the Turbo Treck stack owns the user buffers that were pointed to by the ttUserBlock elements. The only exception is when TM_SOCKET_ERROR is returned and the errorCode retrieved with **tfGetSocketError()** is TM_EWOULDBLOCK. In that case, the user still owns the buffers and should try to resend the same buffers later on. The userFreeFunction will be called by the Turbo Treck stack for each user buffer when the data has been sent out on the network and the device driver no longer needs to access the data in the user buffer. An example of **tfSocketScatteredSendTo()** usage is shown in the loop back test module **txscatlp.c**

If the user uses a preemptive kernel, i.e. TM_TRECK_PREEMPTIVE_KERNEL is defined in trsystem.h, it is the user's responsibility to ensure that the user free function is re-entrant, as it could potentially be called from different threads.

Parameters

Parameter	**Description**
socketDescriptor	As returned by a socket() call
UserBlockPtr	Pointer to the first element of the user array that contains information about the user scattered data.
userBlockCount	number of elements in the above array
userFreeFunction	Pointer to the user free function that will be called for each user buffer when the data in the user buffer no longer need to be accessed. This function is called by the Turbo Treck stack with the pointer to the user buffer as a parameter.
flags	0, or MSG_DONTWAIT. If MSG_DONTWAIT, the call is non-blocking for the duration of the call. If flag is 0, then it is blocking if the socket is in blocking mode, non-blocking otherwise.
toAddressPtr	The address to send the data to. If the user had called connect on the socket, then toAddressPtr should be null.
toAddressLength	The length of the address

tUserBlock data type:

```
typedef struct tsUserBlock
{
 char * userBufferPtr;    Pointer to buffer (passed to the free
 routine)
 char * userDataPtr;      Pointer to beginning of data
 int    userDataLength;   Data length
} ttUserBlock;
```

Function prototype for the user supplied user data free call back function:

int userFreeFunc (char TM_FAR * bufferPtr);

Returns

Value	**Meaning**
> 0	Number of bytes transmitted/queued. Turbo Treck will free the user buffers.
TM_SOCKET_ERROR	Call failed. The user can retrieve the error code with **the tfGetSocketError** (socketDescriptor) API. See below for table of possible error codes.

tfSocketScatteredSendTo will fail if:

TM_EWOULDBLOCK	Not enough room to queue the data in the socket send queue. User still owns the data user buffers, and is responsible for resending them later on.
TM_EINVAL	One of the parameters is bad, i.e. userBlockPtr is null, or userBlockCount is less or equal to 0 or userFreeFunction is null, or socket is a TCP socket with toAddressPtr being non-zero, or flags is not zero, or MSG_DONTWAIT.
TM_ENOBUFS	Not enough memory to allocate the Turbo Treck headers
TM_EBADF	The socket descriptor is invalid
TM_EMSGSIZE	The data length was bigger than the send queue, or the data length caused the IP datagram to be bigger than the IP MTU, and IP fragmentation is not allowed.
TM_EPROTOTYPE	Attempt to send on a connected TCP socket
TM_EHOSTUNREACH	No route to host
TM_ENOENT	No interface could be found to send the packet out (limited broadcast destination IP address.)
TM_ENXIO	Outgoing interface is closed
TM_EIO	Interface transmit queue if full (only if interface transmit queue is used).
Other error code	Returned by device driver send function.

tfZeroCopyRecv

#include <trsocket.h>

```
int                 tfZeroCopyRecv
(
int                 socketDescriptor,
ttUserMessage *     bufferHandlePtr,
char **             dataPtrPtr,
int                 maxBufferLength,
int                 flags
);
```

Function Description

This function is used to recv a zero copy message. It operates identically to **recv** with the exception that it takes a zero copy buffer handle as a parameter and puts the beginning of the data into a passed pointer. However, it does not support the MSG_OOB flag, since there is only one byte of out-of-band data, and it would be inefficient to use the **tfZeroCopyRecv** in that case. To recv out-of-band data, the **recv** call should be used instead.

Note: The user must call tfFreeZeroCopyBuffer in order to free the buffer after the user has finished with it.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to send data to.
bufferHandlePtr	The zero copy buffer that contains the message
dataPtrPtr	A pointer to a char pointer where the start of the data is stored.
maxBufferLength	The length of the message
flags	See below.

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT — Don't wait for data, but rather return immediately

MSG_PEEK — "Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

MSG_SCATTERED — (Non TCP sockets). If the receive data is scattered, it will be passed to the user as is, and will not get copied into a new buffer to make the data contiguous. Upon return, bufferHandlePtr will point to a scattered buffer, as explained below.

MSG_SCATTERED

Description

When the MSG_SCATTERED bit is set in the flags parameters, then upon successful return from **tfZeroCopyRecv**, the bufferHandlePtr parameter should be cast to a pointer to a ttUserPacket structure, where ttUserPacket is defined as follows:

```
typedef struct tsUserPacket
{
   struct tsUserPacket    * pktuLinkNextPtr;
   tt8BitPtr                pktuLinkDataPtr;
   ttPktLen                 pktuLinkDataLength;
   ttPktLen                 pktuChainDataLength;
   int                      pktuLinkExtraCount;
} ttUserPacket;
```

ttUserPacket fields	**Description**
ptkuLinkNextPtr	points to the next ttUserPacket structure
pktuLinkDataPtr	points to the data in the link
pktuLinkDataLength	contains the length of that data
pktuLinkChainDataLength	contains the total length of the scattered data. Its value is ony valid in the first link
pktuLinkExtraCount	Contains the number of extra links besides the first one. Its value is only valid in the first link.

Example

The following code extract shows how to navigate a scattered recv buffer, and copies it into bufferArray[].

```
#define TM_BUF_ARRAY_SIZE     3000

char bufferArray[TM_BUF_ARRAY_SIZE];


      .....
      retCode = tfZeroCopyRecv( ssd,
                   &recvMessageHandle,
                   &dataPtr,
                   sizeof(bufferArray),
                   MSG_SCATTERED );
      if (retCode > 0)
      {
        recvdLength += retCode;
        tm_assert(testzUdpToFrom, retCode == sendDataLength);
        scatteredRecvCopyNCheck( recvMessageHandle,
                     (ttUser8BitPtr)dataPtr,
                     (ttPktLen)retCode );
        (void)tfFreeZeroCopyBuffer(recvMessageHandle);
      }

void scatteredRecvCopyNCheck( ttUserMessage recvMessageHandle,
                ttUser8BitPtr dataPtr,
                ttPktLen      msgLength )
{
  ttUserPacketPtr     recvPacketUPtr;
  char          * bufferPtr;
  int           extraCount;

  extraCount = -1;
  recvPacketUPtr = (ttUserPacketPtr)recvMessageHandle;
```

```
    bufferPtr = &bufferArray[0];
    tm_assert(testzUdp, msgLength == recvPacketUPtr->pktuChainDataLength);
    tm_assert(testzUdp, msgLength <= sizeof(bufferArray));
    tm_assert(testzUdp, dataPtr == recvPacketUPtr->pktuLinkDataPtr);
    do
    {
        tm_assert(testzUdp,
                  msgLength >= recvPacketUPtr->pktuLinkDataLength);
        tm_assert(testzUdp,  recvPacketUPtr->pktuLinkDataPtr
                          != (ttUser8BitPtr)0 );
        tm_bcopy(recvPacketUPtr->pktuLinkDataPtr,
                 bufferPtr,
                 recvPacketUPtr->pktuLinkDataLength);
        bufferPtr += recvPacketUPtr->pktuLinkDataLength;
        msgLength -= recvPacketUPtr->pktuLinkDataLength;
        recvPacketUPtr = recvPacketUPtr->pktuLinkNextPtr;
        extraCount++;
    }
    while (    (msgLength != (ttPktLen)0)
            && (recvPacketUPtr != (ttUserPacketPtr)0) );
    tm_assert(testzUdp, msgLength == 0);
    recvPacketUPtr = (ttUserPacketPtr)recvMessageHandle;
    tm_assert(testzUdp, extraCount == recvPacketUPtr->pktuLinkExtraCount);
}
```

Note: See also tfZeroCopySend

Returns

Value	**Meaning**
0	EOF
>0	Number of bytes received
-1	An error has occurred

tfZeroCopyRecv will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.

TM_ EMSGSIZE	The socket requires that message be received atomically, and bufferLength was too small.
TM_EWOULDBLOCK	The socket is marked as non-blocking or the MSG_DONTWAIT flag is used and no data is available to be read.
TM_ESHUTDOWN	The remote socket has closed the connection, and there is no more data to be received. (TCP socket only).
TM_EINVAL	One of the parameter is invalid.
TM_ENOTCONN	Socket is not connected.

tfZeroCopyRecvFrom

#include <trsocket.h>

```
int                     tfZeroCopyRecvFrom
(
int                     socketDescriptor,
ttUserMessage *         bufferHandlePtr,
char **                 dataPtrPtr,
int                     maxBufferLength,
int                     flags,
struct sockaddr *       fromAddressPtr,
int *                   fromAddressLength
);
```

Function Description

This function is used to recv a zero copy message. It operates identically to **recvfrom** with the exception that it takes a zero copy buffer handle as a parameter and puts the beginning of the data into a passed pointer.

Note: The user must call tfFreeZeroCopyBuffer in order to free the buffer after the user has finished with it.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to send data to
bufferHandlePtr	The zero copy buffer that contains the message
dataPtrPtr	A pointer to a char pointer where the start of the data is stored
maxBufferLength	The maximum length of the message
flags	See below
fromAddressPtr	The address to packet came from
fromAddressLength	The length of the address

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Do not wait for data, but rather return immediately
MSG_PEEK	"Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.
MSG_SCATTERED	If the receive data is scattered, it will be passed to the user as is, and will not get copied into a new buffer to make the data contiguous. Upon return, bufferHandlePtr will point to a scattered buffer, as explained below.

MSG_SCATTERED

Description

When the MSG_SCATTERED bit is set in the flags parameters, then upon successful return from **tfZeroCopyRecvFrom**, the bufferHandlePtr parameter should be cast to a pointer to a ttUserPacket structure, where ttUserPacket is defined as follows:

```
typedef struct tsUserPacket
{
   struct tsUserPacket    * pktuLinkNextPtr;
   tt8BitPtr                pktuLinkDataPtr;
   ttPktLen                 pktuLinkDataLength;
   ttPktLen                 pktuChainDataLength;
   int                      pktuLinkExtraCount;
} ttUserPacket;
```

ttUserPacket fields	**Description**
ptkuLinkNextPtr	Points to the next ttUserPacket structure
pktuLinkDataPtr	Points to the data in the link
pktuLinkDataLength	Contains the length of that data
pktuLinkChainDataLength	Contains the total length of the Scattered data. Its value is ony valid in the first link
pktuLinkExtraCount	Contains the number of extra links besides the first one. Its value is only valid in the first link.

Example

The following code extract shows how to navigate a scattered recv buffer, and copies it into bufferArray[].

```
#define TM_BUF_ARRAY_SIZE     3000

char bufferArray[TM_BUF_ARRAY_SIZE];

        .....
        retCode = tfZeroCopyRecvFrom( ssd,
                          &recvMessageHandle,
                          &dataPtr,
                          sizeof(bufferArray),
                          MSG_SCATTERED,
                          tempSockAddrFrom.sockPtr,
                          &fromAddressLength );
        if (retCode > 0)
        {
          recvdLength += retCode;
          tm_assert(testzUdpToFrom, retCode == sendDataLength);
          scatteredRecvCopyNCheck( recvMessageHandle,
                            (ttUser8BitPtr)dataPtr,
                            (ttPktLen)retCode );
          (void)tfFreeZeroCopyBuffer(recvMessageHandle);
        }

void scatteredRecvCopyNCheck( ttUserMessage recvMessageHandle,
                     ttUser8BitPtr dataPtr,
                     ttPktLen      msgLength )
{
  ttUserPacketPtr     recvPacketUPtr;
  char             * bufferPtr;
  int              extraCount;

  extraCount = -1;
  recvPacketUPtr = (ttUserPacketPtr)recvMessageHandle;
  bufferPtr = &bufferArray[0];
  tm_assert(testzUdp, msgLength == recvPacketUPtr->pktuChainDataLength);
  tm_assert(testzUdp, msgLength <= sizeof(bufferArray));
  tm_assert(testzUdp, dataPtr == recvPacketUPtr->pktuLinkDataPtr);
  do
  {
    tm_assert(testzUdp,
          msgLength >= recvPacketUPtr->pktuLinkDataLength);
    tm_assert(testzUdp,  recvPacketUPtr->pktuLinkDataPtr
               != (ttUser8BitPtr)0 );
    tm_bcopy(recvPacketUPtr->pktuLinkDataPtr,
         bufferPtr,
         recvPacketUPtr->pktuLinkDataLength);
    bufferPtr += recvPacketUPtr->pktuLinkDataLength;
    msgLength -= recvPacketUPtr->pktuLinkDataLength;
    recvPacketUPtr = recvPacketUPtr->pktuLinkNextPtr;
    extraCount++;
  }
  while (    (msgLength != (ttPktLen)0)
        && (recvPacketUPtr != (ttUserPacketPtr)0) );
  tm_assert(testzUdp, msgLength == 0);
  recvPacketUPtr = (ttUserPacketPtr)recvMessageHandle;
  tm_assert(testzUdp, extraCount == recvPacketUPtr->pktuLinkExtraCount);
}
```

Note: See also tfZeroCopySendTo

Returns

Value	**Meaning**
0	EOF
>0	Number of bytes received
-1	An error has occurred.

tfZeroCopyRecvFrom will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_EINVAL	One of the parameters is invalid.
TM_ EMSGSIZE	The socket requires that message be received atomically, and bufferLength was too small.
TM_EPROTOTYPE	TCP protocol requires usage of **tfZeroCopyRecv**, not **tfZeroCopyRecvFrom**.
TM_EWOULDBLOCK	The socket is marked as non-blocking and no data is available to be read.

tfZeroCopySend

```
#include <trsocket.h>

int               tfZeroCopySend
(
int               socketDescriptor,
ttUserMessage     bufferHandle,
int               bufferLength,
int               flags
);
```

Function Description

This function is used to send a zero copy message. It operates identically to **send** with the exception that it takes a zero copy buffer handle as a parameter, and with the exception that it sends either the whole buffer or nothing.

Note: Once tfZeroCopySend is called, the caller does NOT own the buffer and it is freed by the TCP/IP stack, except when if fails with a TM_EWOULDBLOCK error code, in which case the user can either try and send the buffer at a later time, or free the zero copy buffer using tfFreeZeroCopyBuffer.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to send data to
bufferHandle	The zero copy buffer obtained from **tfGetZeroCopyBuffer**
bufferLength	The length of the message to send
flags	See below

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Don't wait for data send to complete, but rather return immediately.
MSG_OOB	Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support "out-of-band" data. Only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data.
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the

	operation. Only diagnostic or routing programs use it.
MSG_SCATTERED	(Non TCP sockets). Set this flag when sending a scattered zero copy recv buffer received with **tfZeroCopyRecv**. (See **tfZeroCopyRecv**.) Note that to send a user scattered buffer (not obtained via a call to **tfZeroCopyRecv**), then **tfSocketScatteredSendTo** can be used.

Returns

Value	**Meaning**
>=0	Number of bytes sent
-1	An error has occurred

tfZeroCopySend will fail if:

TM_EBADF	The socket descriptor is invalid
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_EHOSTUNREACH	Non-TCP socket only. No route to destination host.
TM_ EMSGSIZE	The socket requires that message to be sent atomically, and the message was too long.
TM_EWOULDBLOCK	The socket is marked as non-blocking and the buffer length exceeds the available space left in the send queue.
TM_ENOTCONN	Socket is not connected.
TM_ESHUTDOWN	User has issued a write **shutdown** or a **tfClose** call. (TCP socket only)
TM_EFAULT	*bufferHandle* does not correspond to a zero copy buffer.
TM_EINVAL	One of the parameters is bad: the bufferLength <= 0, or a flag not listed above is set.

tfZeroCopySendTo

#include <trsocket.h>

```
int                          tfZeroCopySendTo
(
int                          socketDescriptor,
ttUserMessage                bufferHandle,
int                          bufferLength,
int                          flags,
const struct sockaddr *      toAddressPtr,
int                          toAddressLength
);
```

Function Description

This function is used to send a zero copy message. It operates identically to **sendto** with the exception that it takes a zero copy buffer handle as a parameter.

Note: Once tfZeroCopySendTo is called, the caller does NOT own the buffer and it is freed by the TCP/IP stack, except when if fails with a TM_EWOULDBLOCK error code, in which case the user can either try and send the buffer at a later time, or free the zero copy buffer using tfFreeZeroCopyBuffer.

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to send data to
bufferHandle	The zero copy buffer obtained from **tfGetZeroCopyBuffer**
bufferLength	The length of the message to send
flags	Special flags for this device.
toAddressPtr	The address to send the packet to
toAddressLength	The length of the address

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Don't wait for data send to complete, but rather return immediately.
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the operation. Only diagnostic or routing programs use it.

MSG_SCATTERED	Set this flag when sending a scattered zero copy recv buffer received with **fZeroCopyRecvFrom**. (See **tfZeroCopyRecvFrom**.) Note that to send a user scattered buffer (not obtained via a call to **tfZeroCopyRecvFrom**), then **tfSocketScatteredSendTo** can be used.

Returns

Value	**Meaning**
>=0	Number of bytes sent
-1	An error has occurred

tfZeroCopySendTo will fail if:

TM_EBADF	The socket descriptor is invalid
TM_EHOSTUNREACH	No route to destination host
TM_ EMSGSIZE	The socket requires that message be sent atomically, and the message was too long.
TM_EPROTOTYPE	TCP protocol requires usage of **tfZeroCopySend** not **tfZeroCopySendTo**.
TM_EWOULDBLOCK	The socket is marked as non-blocking and the send operation would block.
TM_EFAULT	*bufferHandle* does not correspond to a zero copy buffer.
TM_EINVAL	One of the parameters is bad: the bufferLength <= 0, or a flag not listed above is set, or toAddressPtr is null, or toAddressLength does not equal the sizeof(struct sockaddr_in).

tfZeroCopyUserBufferSend

```
#include <trsocket.h>

int                          tfZeroCopyUserBufferSend
(
 int                         socketDescriptor
char                         * userBufferPtr
char                         * userDataPtr
 int                         userDataLength
ttUserFreeFuncPtr            userFreeFunction
 int                         flags
);
```

Function description

Data Socket Send. Allows the user to send data on a socket directly from user owned data buffers without having to copy the data.

Upon return from this routine, the user no longer owns its buffer.The only exception is when TM_SOCKET_ERROR is returned, and the errorCode retrieved with ***tfGetSocketError*** is TM_EWOULDBLOCK. In that case the user still owns its buffer, and should try and re-send the same buffer later on.

The userFreeFunction will be called by the Turbo Treck stack for each user buffer if an error other than TM_EWOULDBLOCK occurs, or when the data has been sent out on the network, and the device driver no longer need to access the data in the user buffer. It is the user's responsibility to ensure that the user free function is re-entrant as it could potentially be called from different threads, unless the user uses a different free function per user application thread, and the user does not use a Turbo Treck transmit task, and does not use a Turbo Treck send (complete) task.

Example of ***tfZeroCopyUserBufferSend*** usage is shown in the loop back test module txscatlp.c.

Parameters

Parameter	**Description**
socketDescriptor	As returned by a ***socket*** call
userBufferPtr	pointer to user buffer to be used when calling the user free function
userDataPtr	pointer to user data
userDataLength	size of user data
userFreeFunction	User free function called by the stack when the buffer is no longer accessed by the stack.
flags	MSG_DONTWAIT MSG_DONTROUTE MSG_WAITALL MSG_OOB or 0

Returns

Value	**Meaning**
> 0	Number of bytes transmitted/ queued. Treck will free the user buffers
TM_SOCKET_ERROR	An error occurred The user can retrieve the error code with the ***tfGetSocketError***(socketDescriptor)

Error codes

V**alue**	**Meaning**
TM_EWOULDBLOCK	Not enough room to queue the data in the socket send queue. User still owns his buffer and is responsible for re-sending later on.
TM_EINVAL	One of the parameters is bad i.e userBufferPtr or userDataPtr or userDataLength or user free function is 0 userDataLength is -1 or flags has a bit that is not allowed
TM_ENOBUFS	Not enough memory to allocate the Turbo Treck headers
TM_EBADF	The socket descriptor is invalid
TM_EMSGSIZE	The data length was bigger than the send queue
TM_EHOSTUNREACH	No route to host

tfZeroCopyUserBufferSendTo

```
#include <trsocket.h>

int                      tfZeroCopyUserBufferSendTo
(
int                      socketDescriptor
char                     * userBufferPtr
char                     * userDataPtr
int                      userDataLength
ttUserFreeFuncPtr        userFreeFunction
int                      flags
const struct sockaddr    * toAddressPtr
int                      toAddressLength
);
```

Function description

Data Socket Send. Allows the user to send data on a socket directly from user owned data buffers without having to copy the data.

Upon return from this routine, the user no longer owns its buffer.The only exception is when TM_SOCKET_ERROR is returned, and the errorCode retrieved with ***tfGetSocketError*** is TM_EWOULDBLOCK. In that case the user still owns its buffer, and should try and re-send the same buffer later on.

The userFreeFunction will be called by the Turbo Treck stack for each user buffer if an error other than TM_EWOULDBLOCK occurs, or when the data has been sent out on the network, and the device driver no longer need to access the data in the user buffer. It is the user's responsibility to ensure that the user free function is re-entrant as it could potentially be called from different threads, unless the user uses a different free function per user application thread, and the user does not use a Turbo Treck transmit task, and does not use a Turbo Treck send (complete) task.

Example of **tfZeroCopyUserBufferSendTo** ***usage is shown in the loop back test module txscatlp.c.***

Parameters

Parameter	**Description**
socketDescriptor	As returned by a ***socket*** call
userBufferPtr	pointer to user buffer to be used when calling the user free function
userDataPtr	pointer to user data
userDataLength	size of user data
userFreeFunction	User free function called by the stack when the buffer is no longer accessed by the stack.
flags	MSG_DONTWAIT MSG_DONTROUTE MSG_WAITALL or 0

Returns

Value	**Meaning**
> 0	Number of bytes transmitted/ queued. Treck will free the user buffers.
TM_SOCKET_ERROR	An error occurred. The user can retrieve the error code with the ***tfGetSocketError***(socketDescriptor) API:

Error codes

Value	**Meaning**
TM_EWOULDBLOCK	Not enough room to queue the data in the socket send queue. User still owns his buffer and is responsible for re-sending later on.
TM_EINVAL	One of the parameters is bad userBufferPtr or userDataPtr or userDataLength or user free function is 0 or userDataLength is -1 or flags has a bit set that is not allowed.
TM_ENOBUFS	Not enough memory to allocate the Turbo Treck headers
TM_EBADF	The socket descriptor is invalid
TM_EMSGSIZE	The data length was bigger than the send queue.
TM_EHOSTUNREACH	No route to host

API Call Back Function Registration

tfRegisterSocketCB

#include <trsocket.h>

```
int                         tfRegisterSocketCB
(
int                         socketDescriptor,
ttUserSocketCBFuncPtr       socketCBFuncPtr,
int                         eventFlags
);
```

Function Description
This function is used to register a function call upon completion of one of more socket events. This allows the user to issue non-blocking calls, and get a call back upon completion of one or more socket events as described in the table below. The call back function will be called repeatedly each time one or more of the events registered for occur until the user cancel the event(s) with another call to **tfRegisterSocketCB** with a new event flag value. For example, if the user register for a recv event (TM_CB_RECV), then the call back function will be called, in the context of the receive task, every time a packet is received from the network and queued in the socket receive queue.

Note that most of the call back calls will be made in the context of the receive task. TM_CB_SOCKET_ERROR, TM_CB_RESET, TM_CB_CLOSE_COMPLETE events call backs could be made in the context of an application task or the receive task. In all the other events, call backs will be made in the context of the receive task. Therefore processing should be kept at a minimum in the call back function. In a multi-tasking environment, the user call back function should set a flag or increase a counter and signal the user application task.

Function prototype for the user supplied socket call back function:

```
void   SocketCBFunc
(
int    socketDescriptor,
int    eventFlags
);
```

Example
To register for incoming data, incoming out-of-band data and remote close event notifications on socket descriptor *sd:*

retCode = tfRegisterSocketCB (sd, socketCBFunc,
TM_CB_RECV|TM_CB_RECV_OOB|TM_CB_REMOTE_CLOSE);

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to register the call back for.
sockettCBFuncPtr	The function pointers of the user function to call when one or more of the registered events occur. See below for function prototype.
eventFlags	One or more of the flags described below and OR'ed together.

EventFlags	**Description**
TM_CB_CONNECT_COMPLT	Register for a connection complete call back (TCP socket only).
TM_CB_ACCEPT	Register for a call back when a remote host has established a connection to our listening server (TCP socket only).
TM_CB_RECV	Register for a call back when incoming data has arrived from our peer.
TM_CB_RECV_OOB	Register for a call back when out-of-band data has arrived from our peer (TCP socket only).
TM_CB_SEND_COMPLT	Register for a call back when the data that we are sending has been acked by the peer host (TCP socket only).
TM_CB_REMOTE_CLOSE	Register for a call back when our peer has shutdown the connection (TCP socket only).
TM_CB_SOCKET_ERROR	Register for a call back when an error occured on the connection.
TM_CB_RESET	Register for a call back when the peer has sent a reset on the connection (TCP socket only).
TM_CB_CLOSE_COMPLT	Register for a call back when the user issued close has completed (TCP socket only).
TM_CB_WRITE_READY	Indicates that there is more room on the send queue. The user can now send more data on the connection given by the *socketDescriptor*.

When one or more of these events occur, the Turbo Treck stack calls the user supplied call back function ***socketCBFunc(****socketDescriptor, eventFlags***)**, where *socketDescriptor* is the socket descriptor as given by the user in **tfRegisterSocketCB**, and *eventFlags* contain one or more of the events specified by the user, that have occured. Described below are the action(s) to be taken by the user in the user supplied call back function upon receiving any one of the socket events:

EventFlags	**Description**
TM_CB_CONNECT_COMPLT	Non-blocking **connect** issued earlier by the user has now completed, and the connection is established with the peer on the *socketDescriptor*. The user is now able to send and recv data on the connection given by the *socketDescriptor*.
TM_CB_ACCEPT	A remote host has established a connection to the listening server *socketDescriptor*. The user can now issue an **accept** call to retrieve the socket descriptor of the newly established connection.
TM_CB_RECV	Incoming data has arrived on the connection given by *socketDescriptor*. The user can now issue any of the allowed **recv** calls for the protocol associated with the socket to retrieve the incoming data.
TM_CB_RECV_OOB	Incoming out-of-band data has arrived on the connection given by *socketDescriptor*. The user can use the appropriate method to retrieve the out of band data as described in the **recv** section above.

TM_CB_SEND_COMPLT	Some sent data has been received, and acked by the peer. The user can issue **tfGetSendCompltBytes** to retrieve the actual amount of bytes that have been received and acked since the last call to **tfGetSendCompltBytes**.
TM_CB_REMOTE_CLOSE	Our peer has shutdown the connection (sent a FIN). No more new data will be coming. The user needs to empty its receive queue using any of the **recv** calls and then close the connection (using **tfClose**).
TM_CB_WRITE_READY	Indicates that there is more room on the send queue. The user can now send more data on the connection given by the *socketDescriptor*.
TM_CB_SOCKET_ERROR	An error has occured on the connection. The user can issue a **send** or **recv** call to retrieve the error as described in the **send** and **recv** sections. Note that **recv** will return all outstanding incoming data before returning the error. The user should then issue **tfClose** to close the connection.
TM_CB_RESET	The peer has sent a RESET. The user needs to issue **tfClose** to close the socket.
TM_CB_CLOSE_COMPLT	The user issued **tfClose** has now completed.

Returns

Value	**Meaning**
0	Success
-1	An error has occurred

tfRegisterSocketCB will fail if:

TM_EBADF	The socket descriptor is invalid.
TM_EINVAL	*socketCBFuncPtr* is NULL and *eventFlags* is non-zero.

tfRegisterSocketCBParam

#include <trsocket.h>

```
int                           tfRegisterSocketCBParam
(
int                           socketDescriptor,
ttUserSocketCBParamFuncPtr    socketCBParamFuncPtr,
void *                        socketUserPtr,
int                           eventFlags
);
```

Function Description

This function is similar to **tfRegisterSocketCB**, and like **tfRegisterSocketCB**, it is used to register a function call upon completion of one of more socket events. It also allows the user to specify a user parameter that will be passed as a parameter to the call back function.

Function prototype for the user supplied socket call back function:

```
void                  socketCBParamFunc

(                     int
socketDescriptor,     void *
socketUserPtr,        int
eventFlags            );
```

Parameters

Parameter	**Description**
socketDescriptor	The socket descriptor to register the call back for
sockettCBFuncPtr	The function pointers of the user function to be called when one or more of the registered events occur. See below for function prototype.
socketUserPtr	A user pointer, that will be passes by the call back function.
eventFlags	One or more of the flags as described in the **tfRegisterSocketCB** section and OR'ed together.

When one or more of these events occur, the Turbo Treck stack calls the user supplied call back function ***socketCBParamFunc*** (*socketDescriptor, socketUserPtr, eventFlags*)**,** where *socketDescriptor* is the socket descriptor as given by the user in **tfRegisterSocketCBParam,** *socketUserPtr* is the user pointer as given by the user in **tfRegisterSocketCBParam**, and *eventFlags* contain one or more of the events specified by the user, that have occurred. The action(s) to be taken by the user in the user supplied call back function upon receiving any one of the socket events are described in the **tfRegisterSocketCB** section.

Returns

Value	**Meaning**
0	Success
-1	An error has occurred.

tfRegisterSocketParamCB will fail if

TM_EBADF	The socket descriptor is invalid.
TM_EINVAL	*socketCBFuncPtr* is NULL and *eventFlags* is non-zero.

Turbo Treck Initialization Functions

tfInitTreckOptions

```
#include <trsocket.h>

int                 tfInitTreckOptions
(
int                 optionName,
unsigned long       optionValue
);
```

Function Description

This call is used to set various options that are used by the Turbo Treck TCP/IP stack.

Note: This function should be called only before calling tfStartTreck.

Parameters

Parameter	**Description**
optionName	The option to change (see below)
optionValue	The new value to change it to

Note: The option names marked with an asterisk * can only be set using tfInitTreckOptions. All the other options are common with tfSetTreckOptions

Option Name	**Meaning**
*TM_OPTION_TICK_LENGTH	The time in Milliseconds between calls to tfTimerUpdate or tfTimerUpdateIsr. Cannot be set to 0. **DEFAULT in TRSYSTEM.H**
*TM_OPTION_SOCKETS_MAX	The maximum number of sockets allowed on the system. Allowed values are between 1 and 0x7FFF - 1. **DEFAULT: 64**
TM_OPTION_ARP_MAX_RETRY	The maximum number of ARP retries before going into the ARP quiet time state. **DEFAULT: 6**
TM_OPTION_ARP_TIMEOUT_TIME	The amount of time in seconds between ARP retries. **DEFAULT: 1**
TM_OPTION_ARP_QUIET_TIME	The length of the ARP quiet time state in seconds. **DEFAULT: 20**

TM_OPTION_ARP_TTL	The length of time that an ARP entry should be kept in the ARP cache in seconds. (To disable ARP aging, set to TM_RT_INF.) **DEFAULT: 600.**
TM_OPTION_ARP_MAX_ENTRIES	Maximum number of ARP entries in the ARP cache. Each entry consumes about 100 bytes. Lower this value if your heap space is low. **DEFAULT: 64**
TM_OPTION_ARP_SMART	Boolean to indicate whether the ARP logic should store all ARP mappings broadcast on the local network, even if we were not waiting for a reply, or if the request was not for us. Set this option value to 0, if your heap space is low. **DEFAULT: 1 (on)**
TM_OPTION_ROUTE_MAX_ENTRIES	Maximum number of dynamic route entries in the routing table. **DEFAULT: 10**
TM_OPTION_RIP_ENABLE	Boolean to enable/disable the RIPv2 Listener once it has been started. **DEFAULT**: **0**. To start **RIP**, the user should call **tfUseRip.**
TM_OPTION_RIP_SEND_MODE	The mode used to send RIP packets. See below. **Default:** TM_RIP_2_BROADCAST
TM_OPTION_RIP_RECV_MODE	The mode used to receive RIP packets (see below). **DEFAULT:TM_RIP_1 |TM_RIP_2**
TM_OPTION_IP_FORWARDING	A boolean to enable IP forwarding **DEFAULT: 0**
TM_OPTION_IP_DBCAST_FORWARD	A boolean to enable directed broadcast forwarding **DEFAULT: 0**
TM_OPTION_IP_FRAGMENT	A Boolean to enable IP fragmentation **DEFAULT: 1**
TM_OPTION_IP_TTL	The initial time-to-live for IP datagrams **DEFAULT: 64**

TM_OPTION_IP_TOS	The default Type-Of-Service for IP datagrams **DEFAULT: 0**
TM_OPTION_IP_FRAG _TTL	Fragment re-assembly timeout value in seconds. **DEFAULT: 64**
TM_OPTION_ICMP_ADDR_MASK_AGENT	A boolean to enable/ disable the ICMP address mask agent. **DEFAULT: 0**
TM_OPTION_UDP_CHECKSUM	A boolean to enable/disable UDP checksums on outgoing packets. **DEFAULT: 1**
TM_OPTION_PMTU_DECREASED_TTL	The length of time (in seconds) before a path MTU estimate increase is tried, after having received an ICMP Datagram Too Big Message error. **DEFAULT: 600 (seconds)**
TM_OPTION_PMTU_LARGER_TTL	The length of time (in seconds) before another path MTU estimate increase is tried, after a previous path MTU estimate increase has been successful. **DEFAULT: 1200 (seconds)**
TM_OPTION_TIMER_MAX_EXECUTE	The maximum number of timers to process in a single call to tfTimerExecute. **Default 0**, which means that there is no maximum. To minimize the latency of the tfTimerExecute call, set to 1.

TM_OPTION_RIP_SEND_MODE	**Meaning**
TM_RIP_NONE	Ignore requests from the TCP/IP stack to send RIP requests.
TM_RIP_1	Send RIP version 1 packets
TM_RIP_2	Multicast RIP version 2 packets.
TM_RIP_2_BROADCAST	Broadcast RIP version 2 packets.

TM_OPTION_RIP_RECV_MODE	**Meaning**
TM_RIP_NONE	Ignore incoming RIP packets.
TM_RIP_1	Accept RIP version 1 packets.
TM_RIP_2	Accept RIP version 2 packets.
TM_RIP_1|TM_RIP_2	Accept both RIP version 1 and version 2 packets.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	The option or its value is invalid.

tfSetTreckOptions

```
#include <trsocket.h>

int                 tfSetTreckOptions
(
int                 optionName,
unsigned long       optionValue
);
```

Function Description

This call is used to set various options that are used by the Turbo Treck TCP/IP stack.

Note: This function should be called only after having called tfStartTreck.

Parameters

Parameter	**Description**
optionName	The option to change (see below).
optionValue	The new value to change it to.

Option Name	**Meaning**
TM_OPTION_ARP_MAX_RETRY	The maximum number of ARP retries before going into the ARP quiet time state. **DEFAULT: 6**
TM_OPTION_ARP_TIMEOUT_TIME	The amount of time in seconds between ARP retries. **DEFAULT: 1**
TM_OPTION_ARP_QUIET_ TIME	The length of the ARP quiet time state in seconds. **DEFAULT: 20**
TM_OPTION_ARP_TTL	The length of time that an ARP entry should be kept in the ARP cache in seconds. (To disable ARP aging, set to TM_RT_INF.) **DEFAULT: 600.**
TM_OPTION_ARP_MAX_ENTRIES	Maximum number of ARP entries in the ARP cache. Each entry consumes about 100 bytes. Lower this value if your heap space is low. **DEFAULT: 64**
TM_OPTION_ARP_SMART	Boolean to indicate whether the ARP logic should store all ARP mappings broadcast on the local network, even if we were not waiting for a reply, or if the request

	was not for us. Set this option value to 0, if your heap space is low. **DEFAULT: 1 (on)**
TM_OPTION_ROUTE_MAX_ENTRIES	Maximum number of dynamic route entries in the routing table. **DEFAULT: 10**
TM_OPTION_ROUTER_AGE_LIMIT	The maximum of time in seconds that a Router entry is kept. **DEFAULT: 600**
TM_OPTION_RIP_ENABLE	Boolean to enable/disable the RIPv2 Listener once it has been started. **DEFAULT: 0**. To start **RIP**, the user should call **tfUseRip** described below.
TM_OPTION_RIP_SEND_MODE	The mode used to send RIP packets (see below). **Default: TM_RIP_2_BROADCAST**
TM_OPTION_RIP_RECV_MODE	The mode used to receive RIP packets (see below). **DEFAULT: TM_RIP_1|TM_RIP_2**
TM_OPTION_IP_FORWARDING	A boolean to enable IP forwarding. **DEFAULT: 0**
TM_OPTION_IP_DBCAST_FORWARD	A boolean to enable directed broadcast forwarding **DEFAULT: 0**
TM_OPTION_IP_FRAGMENT	A boolean to enable IP fragmentation. **DEFAULT: 1**
TM_OPTION_IP_TTL	The initial time-to-live for IP datagrams. **DEFAULT: 64**
TM_OPTION_IP_TOS	The default Type-Of-Service for IP datagrams. **DEFAULT: 0**
TM_OPTION_IP_FRAG _TTL	Fragment re-assembly timeout value in seconds. **DEFAULT: 64**
TM_OPTION_ICMP_ADDR_MASK_AGENT	A boolean to enable/disable the ICMP address mask agent **DEFAULT: 0**
TM_OPTION_UDP_CHECKSUM	A boolean to enable/disable UDP checksums on outgoing packets **DEFAULT: 1**

TM_OPTION_PMTU_DECREASED_TTL The length of time (in seconds) before a path MTU estimate increase is tried, after having received an ICMP Datagram Too Big Message error. **DEFAULT: 600 (seconds)**

TM_OPTION_PMTU_LARGER_TTL The length of time (in seconds) before another path MTU estimate increase is tried, after a previous path MTU estimate increase has been successful. **DEFAULT: 1200 (seconds)**

TM_OPTION_SEND_TRAILER_SIZE Number of bytes of extra space to include at the end of outgoing packets. This could be used, for instance, if the user needs space to copy their own trailer onto outgoing packets. **DEFAULT: 0**

TM_OPTION_TIMER_MAX_EXECUTE The maximum number of timers to process in a single call to tfTimerExecute. **Default 0**, which means that there is no maximum. To minimize the latency of the tfTimerExecute call, set to 1.

TM_OPTION_RIP_SEND_MODE	**Meaning**
TM_RIP_NONE	Ignore requests from the TCP/IP stack to send RIP requests.
TM_RIP_1	Send RIP version 1 packets
TM_RIP_2	Multicast RIP version 2 packets
TM_RIP_2_BROADCAST	Broadcast RIP version 2 packets

TM_OPTION_RIP_RECV_MODE	**Meaning**
TM_RIP_NONE	Ignore incoming RIP packets
TM_RIP_1	Accept RIP version 1 packets
TM_RIP_2	Accept RIP version 2 packets
TM_RIP_1|TM_RIP_2	Accept both RIP version 1 and version 2 packets

Returns

Value	**Meaning**
0	Success
TM_EINVAL	The option or its value is invalid.

tfStartTreck

```
#include <trsocket.h>

int                  tfStartTreck
(
void
);
```

Function Description

tfStartTreck is used by the user to initialize the Turbo Treck protocol stack. This involves initializing all the global variables and getting system resources.

Note: The only Turbo Treck call that can be made prior to calling tfStartTreck is tfInitTreckOptions.

Parameters

None

Returns

Value	**Meaning**
0	Success
TM_ENOBUFS	No memory to complete the operation.
TM_EPERM	The timer tick length has not been initialized. Use the **tfInitTreckOptions** call to initialize it.
TM_EFAULT	Code is compiled with the wrong network byte order switch. If you had the TM_LITTLE_ENDIAN switch defined in *trsystem.h,* undefine it. If you had not defined it, define it.

Device/Interface API

tfAddInterface

```
#include <trsocket.h>

TtUserInterface        tfAddInterface
(
char *                 namePtr,
ttUserLinkLayer        linkLayerHandle,
ttDevOpenCloseFuncPtr  drvOpenFuncPtr,
ttDevOpenCloseFuncPtr  drvCloseFuncPtr,
TtDevSendFuncPtr       drvSendFuncPtr,
TtDevRecvFuncPtr       drvRecvFuncPtr,
ttDevFreeRecvFuncPtr   drvFreeRecvBufFuncPtr,
TtDevIoctlFuncPtr      drvIoctlFuncPtr,
ttDevGetPhyAddrFuncPtr drvGetPhyAddrFuncPtr,
int *                  *drvAddErrorPtr
);
```

Function Description

This function is the main function to use when adding an interface to the Turbo Treck TCP/IP system. Note: Interfaces are added in the CLOSED state. An example would be as follows:

```
myInterfaceHandle=tfAddInterface(
          “NE2000.001”,
          etherLinkLayerHandle,
          ne2kOpen,
          ne2kClose,
          ne2kSend,
          ne2kReceive,
          (ttDevFreeRecvFuncPtr)0,
          ne2kIoctl,
          ne2kGetPhyAddr,
          &errorCode);
```

Parameters

Parameter	**Description**
namePtr	The callers name for the device (each call to tfAddInterface must use a unique name). The name length cannot exceed TM_MAX_DEVICE_NAME –1 (13 bytes).
linkLayerHandle	The link layer (layer 2) protocol handle that this interface will use. This is returned by **tfUseEthernet**, **tfUseAsyncPpp**, **tfUseAsyncServerPPP**, or **tfUseSlip**.
drvOpenFuncPtr	A function pointer to the device drivers open function.
drvCloseFuncPtr	A function pointer to the device drivers close function.
drvSendFuncPtr	A function pointer to the device driver’s send routine.
drvRecvFuncPtr	A function pointer to the device driver’s recv routine.
drvFreeRecvBufFuncPtr	A function pointer a device driver function that will free a receive buffer (OPTIONAL).
drvIoctlFuncPtr	A function pointer to an IOCTL routine. This function is called by **tfIoctlInterface.**
drvGetPhyAddrFuncPtr	A function pointer to the device driver that will return the physical address for the device.
drvAddErrorPtr	A pointer to an int that will contain an error (if one occurred).

The following table is a list of the prototypes for the device driver functions

```
int   drvOpenFunc(ttUserInterface  interfaceHandle);
int   drvCloseFunc(ttUserInterface interfaceHandle);
int   drvSendFunc(ttUserInterface  interfaceHandle,
            char            *dataPtr,
            int             dataLength,
            int             flag);
int   drvRecvFunc(
      ttUserInterface interfaceHandle,
      char  TM_FAR **dataPtr,
      int   TM_FAR  *dataLength,
      ttUserBufferPtr userBufferHandlePtr
     );
int drvFreeRecvFunc(ttUserInterface interfaceHandle,

            char            *dataPtr)
int   drvIoctlFunc(ttUserInterface interfaceHandle,
            int             flag,
            void           *optionPtr,
            int             optionLen);   int
drvGetPhysAddrFunc(
      ttUserInterface interfaceHandle,
      char          *physicalAddress);
```

Returns

An Interface Handle or a NULL handle when an error occurs. If an error occurs, the error is stored in **drvAddErrorPtr.*

ErrorCode	**Description**
TM_EINVAL	One of the parameters is invalid
TM_EALREADY	Device has already been added.
TM_ENOBUFS	Not enough buffers to allocate a device entry.

tfAddInterfaceMhomeAddress

#include <trsocket.h>

```
int                tfAddInterfaceMhomeAddress
(
ttUserInterface    interfaceHandle,
ttUserIpAddress    ipAddress,
unsigned char      multiHomeIndex
);
```

Function Description

Add a multihome IP address to the interface without adding it to the routing table. This allows the Turbo Treck stack, for example, to assume the IP address identity of another host on another network, and therefore to allow a server application on the Turbo Treck stack to get the packets whose destination is for that other host, and offload some of the work from that other host server. If the interface uses the Ethernet link layer, then the user should also add a proxy arp entry for the IP address it is assuming.

An example would be:

```
            errorCode = tfAddInterfaceMhomeAddress (
                        myInterfaceHandle,
                        inet_addr (“208.229.201.61”),
                        (unsigned char)1);
```

Parameters

Parameter	**Description**
interfaceId	The device entry. This is returned by **tfAddInterface**.
ipAddress	Extra multihome IP Address.
multiHomeIndex	The index for this IP address for multihoming. Because we are adding an additional IP address without really configuring the interface, the interface should already have been configured, and this multihomeIndex should be bigger than 0.

Returns

Value	**Meaning**
0	Success
TM_EADDRNOTAVAIL	Attempt to configure the device with a broadcast address.
TM_ENOBUFS	Not enough memory to complete operation
TM_EINVAL	Bad parameter, i.e bad interface handle, or multi home index out of boundaries (not between 0, and TM_MAX_IPS_PER_IF)
TM_EALREADY	The interface has already been configured at this mult-home index.

tfCheckReceiveInterface

```
#include <trsocket.h>

int                 tfCheckReceiveInterface
(
ttUserInterface     interfaceHandle
);
```

Function Description

This function is used to check if the data is waiting to be received on a particular device/interface. This call can be used in environments where it is preferable to poll the device for received data (i.e. a main loop). If you are using a separate task or thread to receive data, then you should use the **tfWaitReceiveInterface** call. The call is used is used in conjunction with **tfNotifyInterfaceIsr** and **tfRecvInterface** to poll the device driver for received data. Upon a successful return, the user should call **tfRecvInterface** to send the data from the device driver to the protocol stack.

Note: There must be a one to one correspondence from the number of received packets parameter in tfNotifyInterfaceIsr to the number of tfRecvInterface calls.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle to poll to see if data needs to be received from the device driver

Returns

Value	**Meaning**
0	There is data waiting in the device driver to be received (the driver has called **tfNotifyInterfaceIsr**).
TM_EWOULDBLOCK	There is no data waiting to be received from the device driver.

tfCheckSentInterface

```
#include <trsocket.h>

int                 tfCheckSentInterface
(
ttUserInterface     interfaceHandle
);
```

Function Description

This function is used in conjunction with **tfNotifyInterfaceIsr** and **tfSendCompleteInterface**, to poll the device driver to see if the packet sent to the device driver has been sent. This function will return 0, when the accumulated numberBytesSent notified by **tfNotifyInterfaceIsr** has reached TM_NOTIFY_SEND_LOW_WATER (2048 by default). When **tfCheckSentInterface** returns 0, the caller should then call **tfSendCompleteInterface** to allow the stack to free the zero copy buffers used by the protocol stack. This call is used when the user does not have a separate send complete task or thread.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle to check to see if data has been sent.

Returns

Value	**Meaning**
0	Send Complete has occurred (the driver has called tfNotifyInterfaceIsr from the send complete interrupt service routine, and the TM_NOTIFY_SEND_LOW_WATER mark has been reached).
TM_EWOULDBLOCK	No data has been sent yet, or threshold has not been reached.

tfCheckXmitInterface

```
#include <trsocket.h>

int                 tfCheckXmitInterface
(
ttUserInterface     interfaceHandle
);
```

Function Description

TfCheckXmitInterface is used to check if data is ready to be sent on a particular device/interface (data queued to the interface send queue). This call can be used in environments where it is preferable to poll the device for data ready to be transmitted (i.e. a main loop). If you are using a separate task or thread to transmit data, then you should use the **tfWaitXmitInterface** call. Upon a successful return, the user should call **tfXmitInterface** to send the data ready to be transmitted from the bottom of the Turbo Treck stack to the device driver.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle to poll to see if data needs to be received from the device driver

Returns

Value	**Meaning**
0	There is data waiting to be transmitted to the device driver.
TM_EWOULDBLOCK	There is no data waiting to be transmitted to the device driver.

tfCloseInterface

```
#include <trsocket.h>

int                tfCloseInterface
(
ttUserInterface    interfaceId
);
```

Function Description

This function is used to close the interface that is passed in. It simply calls the link layer close routine, the driver close routine, removes the interface from the local routing table and returns the error code that the link layer, or driver returns.

Parameters

Parameter	**Description**
interfaceId	The device driver entry that contains the close routine

Returns

Value	**Meaning**
0	Success
TM_EINVAL	Parameter is invalid
TM_EALREADY	The device is already closed
TM_EINPROGRESS	If the connection is in the process of closing the connection (as in PPP). The user doesn't need to do anything, and will be notified by the PPP link layer when the interface is actually closed.

tfConfigInterface

```
#include <trsocket.h>

int                 tfConfigInterface
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ipAddress,
ttUserIpAddress     netMask,
int                 flag,
int                 buffersPerFrameCount,
unsigned char       multiHomeIndex
);
```

Function Description

This function is used to configure an interface with an IP address, and net mask (either supernet or subnet). Can be used for Multiple IP Addresses on an Interface (Multihoming). It must be called before the interface can be used.

An example would be:

```
            errorCode = tfConfigInterface (
                        myInterfaceHandle,
                        inet_addr (“208.229.201.65”),
                        inet_addr (“255.255.255.0”),
                        0,
                        1,
                        (unsigned char)0);
```

Note: tfConfigInterface is deprecated for the first multihome. Please use tfOpenInterface for the first multihome configuration, instead of tfConfigInterface.

DHCP or BOOTP configuration.

tfConfigInterface with a TM_DEV_IP_DHCP (respectively TM_DEV_IP_BOOTP) flag will send a DHCP (respectively BOOTP) request to a DHCP/BOOTP server, and will return with a TM_EINPROGRESS errorCode.

Note that **tfUseDhcp** (respectively **tfUseBootp**) needs to have been called prior to calling **tfConfigInterface**, otherwise the call will fail with error code TM_EPERM.

An example of a configuration using the DHCP protocol would be:

```
errorCode = tfConfigInterface (
                myInterfaceHandle,
                (ttUserIpAddress)0,
                (ttUserIpAddress)0,
                TM_DEV_IP_DHCP,
                1,
                (unsigned char)0);
```

Checking on completion of DHCP or BOOTP configuration.

- **Synchronous check**: The user can make multiple calls to **tfConfigInterface** to determine when the configuration has completed.. Additional calls to **tfConfigInterface** will return TM_EALREADY as long as the BOOTP/DHCP server has not replied. **tfConfigInterface** will return TM_ENOERROR, if the BOOTP/DHCP server has replied and the configuration has completed.

- **Asynchronous check**: To avoid this synchronous poll, the user can provide a user call back function to **tfUseDhcp** (respectively tfUseBootp), that will be called upon completion of **tfConfigInterface**. See **tfUseDhcp** (respectively **tfUseBootp**) for details.

Parameters

Parameter	**Description**
interfaceId	The device entry. This is returned by **tfAddInterface**.
ipAddress	The IP Address for this Interface.
netMask	The net mask for this device (Subnet or Supernet)
flag	Special flags for this deviceOred together (see below).
buffersPerFrameCount	Number of scattered buffers allowed for each frame being sent

	out. If scattered buffers are not allowed by the driver, this number should be one, otherwise it should be bigger than one.
multiHomeIndex	The index for this IP address for multihoming. Zero must be the first multi home index used.

Flags

Value	**Meaning**
TM_DEV_SCATTER_SEND_ENB	This device supports sending data in multiple buffers per frame. If this flag is set, then the buffersPerFrameCount should be bigger than 1. This flag should always be set for SLIP or PPP serial devices.
TM_DEV_MCAST_ENB	This device supports multicast addresses
TM_DEV_IP_FORW_ENB	Allow IP Forwarding to and from this device
TM_DEV_IP_FORW_DBROAD_ENB	Allow forwarding of IP directed broadcasts to and from this device
TM_DEV_IP_FORW_MCAST_ENB	Allow forwarding of IP multicast messages to and from this device
TM_DEV_IP_BOOTP	Configure IP address using BOOTP client protocol. **tfUseBootp** need to have been called first.
TM_DEV_IP_DHCP	Configure IP address using DHCP client protocol. **tfUseDhcp** need to have been called first.
TM_DEV_IP_USER_BOOT	Allow the user to temporarily configure the interface with a zero IP address, to allow the user to use a proprietary protocol to retrieve an IP address from the network.
TM_DEV_IP_NO_CHECK	Allow the Turbo Treck stack to function in promiscuous mode, where all packets received by this interface whill be handed to the application without checking for an IP address match on the incoming interface.

Returns

Value	**Meaning**
0	Success
TM_EADDRNOTAVAIL	Attempt to configure the device with a broadcast address.
TM_EINPROGRESS	**tfConfigInterface** call has not completed. This error will be returned for a DHCP or BOOTP configuration for example.
TM_ENOBUFS	Not enough memory to complete operation
TM_EINVAL	Bad parameter, or first configuration should be for multihome index 0. Note that a zero IP address is allowed for Ethernet if the BOOTP, DHCP, or USER_BOOT flag is on, or for PPP, otherwise a TM_EINVAL errorCode is returned.
TM_EALREADY	A previous call to **tfConfigInterface** has not yet completed.
TM_EPERM	User attempted to configure an IP address via DHCP (respectively BOOTP) without having called **tfUseDhcp** (respectively **tfUseBootp)** successfully first.
TM_EMFILE	Not enough sockets to open the BOOTP client UDP socket (TM_IPDEV_BOOTP or TM_IP_DEV_DHCP configurations only.)
TM_ADDRINUSE	Another socket is already bound to the BOOTP client UDP port. (TM_IP_DEV_BOOTP or TM_IP_DEV_DHCP configurations only.)
TM_ETIMEDOUT	DHCP or BOOTP request timed out
TM_EAGAIN	A PPP session is currently closing. Call **tfConfigInterface** again after receving notification that the previous session has ended.
other	Error value as returned by the device

driver open function.

tfDeviceClearPointer

```
#include <trsocket.h>

void * tfDeviceClearPointer
(
ttUserInterface interfaceHandle
);
```

Function Description

tfDeviceClearPointer will clear the association created by **tfDeviceStorePointer** between the interface handle, and the device specific structure allocated in the device driver open function.

Call **tfDeviceClearPointer** from the driver close function, and then, if the pointer returned by **tfDeviceClearPointer** is non null, free it.

Parameters

Parameter	**Description**
interfaceHandle	Interface Id as returned by **tfAddInterface**.

Returns

Value	**Meaning**
Non zero pointer	Success. Pointer to the device driver pointer.
(ttVoidPtr)0	No associated device driver pointer on this interface

Note :See also tfDeviceGetPointer, tfDeviceStorePointer

tfDeviceGetPointer

```
#include <trsocket.h>

void *              tfDeviceGetPointer
(
ttUserInterface     interfaceHandle
);
```

Function Description

Call **tfDeviceGetPointer** in any device driver function, to retrieve the pointer to the device specific structure allocated, and stored (with the **tfDeviceStorePointer** API) in the device driver open function

tm_device_get_pointer()

This macro has the same functionality as **tfDeviceGetPointer**. It allows the user to avoid a function call in order to get the device driver pointer. In that case the following header files need to be included:

```
#include <trsocket.h>
#include <trmacro.h>
#include <trtype.h>
#include <trproto.h>
#include <trglobal.h>
```

Parameters

Parameter	**Description**
interfaceHandle	Interface Id as returned by **tfAddInterface**.

Returns

Value	**Meaning**
Non zero pointer	Success
(void *)0	No pointer was stored on the interface

*Note: See also tfDeviceClearPointer***,** *tfDeviceStorePointer*

tfDeviceStorePointer

```
#include <trsocket.h>

int                tfDeviceStorePointer
(
ttUserInterface    interfaceHandle,
void *             deviceDriverPtr
);
```

Function Description

In the device driver open function, after having allocated a device driver specific structure, call **tfDeviceStorePointer** to store a pointer to that structure on the interface.

Parameters

Parameter	**Description**
interfaceHandle	Interface Id as returned by **tfAddInterface**.
deviceDriverPtr	Device Driver Pointer to be stored on the interface

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EALREADY	A device driver pointer was already stored on that interface
TM_EINVAL	Invalid interface handle

*Note: See also tfDeviceClearPointer****,*** *tfDeviceGetPointer*

tfFinishOpenInterface

```
include <trsocket.h>

int                tfFinishOpenInterface
(
ttUserInterface    interfaceHandle,
ttUserIpAddress    ipAddress,
ttUserIpAddress    netMask,
);
```

Function description.

Finish opening an Interface that the user had started to open with the TM_DEV_IP_USER_BOOT flag. That flag caused the Turbo Treck stack not to store the IP address/netmask in the routing table, leaving the interface in a half configured state. **tfFinishOpenInterface** will attempt to insert the ipaddress/netmask into the routing table.

Parameters

Parameter	Description
interfaceId	The device entry. This is returned by **tfAddInterface**.
ipAddress	The IP Address for this Interface at multihome index 0.
netMask	The net mask for this device (Subnet or Supernet) at multihome index 0.

Returns

Value	Meaning
0	Success
TM_EADDRNOTAVAIL	Attempt to configure the device with a broadcast address.
TM_ENOBUFS	Not enough memory to complete operation
TM_EINVAL	Bad parameter, such as invalid interfaceId.
TM_EALREADY	ipAddress/netMask already in the routing table.
TM_EPERM	User did not set the TM_DEV_IP_USER_BOOT flag when calling **tfOpenInterface**.

tfFreeDriverBuffer

```
#include <trsocket.h>

int                 tfFreeDriverBuffer
(
ttUserBuffer        userBuffer,
);
```

Function Description

User frees a buffer that was allocated with either **tfGetEthernetBuffer()**, or **tfGetDriverBuffer()**, and that was not given to the Turbo Treck stack.

Note: If the buffer has been given to the Turbo Treck stack, the user no longer owns the buffer, and should not call tfFreeDriverBuffer.

Parameters

Parameter	**Description**
userBuffer	A user buffer handle as stored by **tfGetEthernetBuffer** or **tfGetDriverBuffer**.

Returns

Value	**Meaning**
0	Success
TM_DEV_ERROR	Null user buffer handle, or user does not own the buffer.

tfGetDriverBuffer

```
#include <trsocket.h>

char *            tfGetDriverBuffer
(
ttUserBufferPtr   userBufferPtr,
int               length,
int               alignment
);
```

Function Description

This function is used to get a buffer from the Turbo Treck buffer pool for the user to use for a device driver. It is not required for a user to use the Turbo Treck buffer pool (i.e. this function) because some devices may not support it.

Parameters

Parameter	**Description**
userBufferPtr	A pointer to a ttUserBuffer variable that the user buffer handle is stored into
length	Number of bytes to be allocated by the system
alignment	The returned buffer pointer will be aligned on a multiple of this value.

Returns

A char * to the beginning of the data area to store the received data into or a NULL pointer if there is no memory to complete the operation

tfGetBroadcastAddress

```
int                tfGetBroadcastAddress
(
ttUserInterface    interfaceHandle,
ttUserIpAddress*   ifBroadcastAddressPtr,
unsigned char      multiHomeIndex
);
```

Function Description

This function is used to retrieve the broadcast address for an interface. The multi home index is used for interfaces that have more than one IP address. If the interface has only one IP address, then the Multi home index should be set to zero. The broadcast address is automatically calculated from the IP address and Netmask combination

Parameters

Parameter	**Description**
interfaceHandle	The device driver entry that we wish to get the broadcast address on.
ifBroadcastAddressPtr	The pointer to the area that the function will store the broadcast address into.
multiHomeIndex	An index for multiple IPs.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	Bad parameter.
TM_ENETDOWN	Interface/multi home index is not configured.

tfGetIfMtu

```
int                tfGetIfMtu
(
ttUserInterface    interfaceHandle,
int *              ifMtuPtr
);
```

Function Description

This function is used to retrieve the Maximum Transmission Unit (MTU) for a device.

Parameters

Parameter	**Description**
interfaceHandle	The device driver entry that we wish to get the physical address on.
ifMtuPtr	Pointer to the integer that will contain the Maximum Transmission Unit for a device

Returns

Value	**Meaning**
0	Success
TM_EINVAL	One of the parameters is null or 0.

tfGetIpAddress

```
int                  tfGetIpAddress
(
ttUserInterface      interfaceHandle,
ttUserIpAddress *    ifIpAddressPtr,
unsigned char        mutiHomeIndex
);
```

Function Description

This function is used to get the IP address of an interface. The multi home index is used for interfaces that have more than one IP address. If the interface has only one IP address then the Multi home index should be set to zero.

Parameters

Parameter	**Description**
interfaceHandle	The device driver entry that we wish to get the physical address on.
ifIpAddressPtr	A pointer to the location where to store the IP address for the interface
mutiHomeIndex	An index for multiple IPs

Returns

Value	**Meaning**
0	Success
TM_EINVAL	Bad parameter.
TM_ ENETDOWN	Interface/multi home index is not configured.

tfGetNetMask

```
int                     tfGetNetMask
(
ttUserInterface         interfaceHandle,
ttUserIpAddress *       netMaskPtr,
unsigned char           mutiHomeIndex
);
```

Function Description

This function is used to get the Net Mask from a given interface. The multi home index is used for interfaces that have more than one IP address. If the interface has only one IP address then the Multi home index should be set to zero.

Parameters

Parameter	**Description**
interfaceHandle	The device driver entry that we wish to get the net mask for
netMaskPtr	A pointer to a location where to store the Net Mask upon function completion.
mutiHomeIndex	An index for multiple IP’s per device

Returns

Value	**Meaning**
0	Success
TM_EINVAL	Bad parameter
TM_ENETDOWN	Interface/multi home index is not configured.

tfInterfaceGetVirtualChannel

```
int               tfInterfaceGetVirtualChannel
(
ttUserInterface   interfaceHandle,
ttUser32Bit *     virtualChannelPtr
);
```

Function Description

tfInterfaceGetVirtualChannel is called by the user, in the send path, from the driver send routine, to get the ATM virtual channel associated with the interface and multi home the data is sent from.

Note: this function should only be called from the device driver send function.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle as returned by ***tfAddInterface,*** and passed to the device driver send function.
virtualChannelPtr	Pointer to a 32-bit variable where the virtual channel will be stored, if the call is successful.

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EINVAL	Invalid interface handle, or interface handle does not match the interface handle of the packet currently being sent.
TM_ENOENT	Empty device send queue

Note: Also see tfInterfaceSetVirtualChannel.

tfInterfaceSetOptions

```
#include <trsocket.h>

int                 tfInterfaceSetOptions
(
ttUserInterface     interfaceHandle,
int                 optionName,
void TM_FAR *       optionValuePtr,
int                 optionLength
);
```

Function Description
Configure interface options. *optionValuePtr* points to a variable of type as described below. OptionLenth contains the size of that variable.

Options

TM_DEV_OPTIONS_RECV_COPY Make the Turbo Treck stack (inside the tfRecvInterface function) copy received driver buffers whose sizes are smaller than the value pointed to by optionValuePtr into a newly allocated buffer. Option is disallowed on a point to point (i.e SLIP, or PPP) interface since the SLIP, and PPP link layer copy the data into a newly allocated buffer.
Data Type: unsigned short

TM_DEV_OPTIONS_SCAT_RECV_LENGTH Valid only if TM_USE_DRV_SCAT_RECV is defined. Set the minimum number of bytes at the head of a packet that have to be contiguous. If a scattered recv buffer is received from the driver with a first link length below that minimum, that minimum number of bytes (but no more than the total length of the buffer) is copied into a new stack allocated buffer. **Default value is** TM_DEV_DEF_RECV_CONT_HDR_LENGTH.

TM_DEV_OPTIONS_XMIT_TASK

Turn on/off usage of a transmit task. Default is off. Option can only be used when device/interface is not opened/configured. Cannot turn on this option, if the device/interface transmit queue is used.
Data Type unsigned short

`TM_DEV_OPTIONS_BOOT_TIMEOUT`

Base number of seconds for a BOOTP/DHCP request timeout. BOOTP/DHCP timeouts increase with each retransmission, so if this value is set to two seconds, the first timeout will be two seconds, the second will be four seconds, the third will be eight seconds, etc.
Default: 4 seconds
Data Type unsigned char

`TM_DEV_OPTIONS_BOOT_RETRIES`

Total number of BOOTP/DHCP requests to send without receiving a response from a BOOTP/DHCP server.
Default: 6.
Data Type unsigned char

Example on how to turn the usage of a transmit task on:

```
unsigned short optionValue;

optionValue = 1; /* turn option on */
errorCode = tfInterfaceSetOptions(myInterfaceHandle,
                                  TM_DEV_OPTIONS_XMIT_TASK,
                                  &optionValue,
                                  sizeof(unsigned short));
```

Parameters

Parameter	**Description**
interfaceHandle	Interface handle of the SLIP interface we want to set the option on.
optionName	The option to set. See above.
optionValuePtr	The pointer to a user variable into which the option value is set. User variable is of data type described above.
optionLength	size of the user variable, which is the size of the option data type.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	Invalid optionName, or invalid option length for option, or invalid option value for option.
TM_EPERM	TM_DEV_OPTIONS_RECV_COPY: Interface/device is a point to point interface.TM_DEV_OPTIONS_XMIT_TASK: Interface/device already configured/opened,or device/ interface transmit queue is used.

tfInterfaceSetVirtualChannel

```
#include <trsocket.h>

int                tfInterfaceSetVirtualChannel
(
ttUserInterface    interfaceHandle,
ttUser32Bit        virtualChannel,
int                mhomeIndex
);
```

Function Description

tfInterfaceSetVirtualChannel allows the user to set an ATM virtual channel for a given multi home on an interface.

Parameters

Parameter	Description
interfaceHandle	The interface handle as returned by ***tfAddInterface***
virtualChannel	ATM virtual channel set by the user.
mhomeIndex	Multi home index on the interface.

Returns

Value	Meaning
TM_ENOERROR	Success
TM_EINVAL	Invalid interface handle, or multi home index.

Note: also see tfInterfaceGetVirtualChannel

tfInterfaceSpinLock

```
#include <trsocket.h>

int                  tfInterfaceSpinLock
(
ttUserInterface      interfaceHandle
);
```

Function Description

tfInterfaceSpinLock is called from the device driver send routine to yield the CPU to allow spin lock while waiting for room to transmit the data. This function will yield the CPU, only if the user is using a transmit task to interface between the Turbo Treck stack and the device driver.

Parameters

Parameter	Description
interfaceHandle	The Interface Handle of the device we are waiting to be ready.

Returns

Value	Meaning
0	Success
TM_EPERM	Function did not yield the CPU, because the user is not using a transmit task (i.e. did not call tfInterfaceSetOptions to turn on the transmit task option.)

tfIoctlInterface

```
#include <trsocket.h>

void                 tfIoctlInterface
(
ttUserInterface      interfaceHandle,
int                  flag,
void *               optionPtr,
int                  optionLen
);
```

Function Description

The function allows the user to call the driver special functions. An example of a special function would be to perform driver maintenance or refill a receive pool. This function is provided to allow the user to call the driver and be guaranteed that no other tasks are using the driver at the same time. The flag parameter if below 0x100 is driver specific and has no meaning to the TCP/IP stack.

Empty the transmit ring of buffers:

This function should also be called if the user uses the Turbo Treck device/interface transmit queue. In this case, the user needs to periodically call **tfIoctlInterface** to empty the device driver transmit queue as follows:

```
errorCode =
tfIoctlInterface(interfaceHandle, TM_DEV_IOCTL_EMPTY_XMIT_FLAG,
                                (void *)0, 0);
```

In this case, the device driver ioctl function will not be called. The Turbo Treck stack will internally attempt to empty the device Turbo Treck device/interface transmit queue.

Refill the Turbo Treck ISR recv pool:

If the user need to get a Turbo Treck buffer from the receive ISR, and uses the Turbo Treck pre-allocated recv pool of buffers, then the user needs to periodically call **tfIoctlInterface** to replenish the Turbo Treck ISR recv pool as follows:

```
errorCode =
tfIoctlInterface(interfaceHandle, TM_DEV_IOCTL_REFILL_POOL _FLAG,
                                (void *)0, 0);
```

In this case, the device driver ioctl function will not be called. The pool refill will be done internally by the Turbo Treck stack

Parameters

Parameter	**Description**
interfaceHandle	The interface handle of the driver's ioctl routine to call
flag	See below.
optionPtr	Pointer to a **tfIoctlInterface** specific parameter.
optionLen	length of the option.

The flags parameter is formed by ORing one or more of the following:

[0x0 - 0x100]
Device specific flag

TM_DEV_IOCTL_EMPTY_XMIT_FLAG
Empty the device driver transmit queue:

TM_DEV_IOCTL_REFILL_POOL_FLAG
Refill the Treck recv pool:

TM_DEV_IOCTL_OFFLOAD_GET
Query the device driver for offload capabilities

TM_DEV_IOCTL_OFFLOAD_SET
Set the device driver for offload

Returns

TM_ENOERROR
(xmit) All buffers have been transmitted.
(refill) All buffers are allocated

TM_EINPROGRESS
(xmit) Some buffers (but not all) have been transmitted
(refill) Some buffers (but not all) have been allocated

TM_EPERM
(xmit) if transmit queue not enabled (i.e user did not
call tfUseInterfaceXmitQueue() successfully.
(refill) if the user did not create a recv pool
TM_ENOBUFS (refill) No buffer could be allocated.

Other As returned from the device driver send routine.
Buffers are still in the interface transmit queue.
None was sent.

tfNotifyInterfaceIsr

```
#include <trsocket.h>
void                tfNotifyInterfaceIsr
(
ttUserInterface     interfaceHandle,
int                 numberRecvPackets,
int                 numberSendCompletePackets,
unsigned long       numberBytesSent,
unsigned long       flag
);
```

Function Description

This function notifies the stack with both the number of packets received in an ISR and the number of packets transmitted by the chip since the last ISR. This function is to be called by the user inside an ISR, with the appropriate number of received and sent packets. This will in turn cause the stack to notify the user's application about what has occurred. In the case of received packets, **tfCheckReceiveInterface** will return 0 once for each received packets if the user is in polling mode.

In blocking mode, **tfWaitReceiveInterface** will return once for each received packet. For sent packets, **tfCheckSentInterface** will return 0 once for each sent packet in polling mode, or **tfWaitSentInterface** will return once for each sent packet in blocking mode. These functions only notify the user of a sent packet in the case that the accumulated **numberBytesSent** has reached TM_NOTIFY_SEND_LOW_WATER (2048 by default), since the previous notification.

Parameters

Parameter	Description
interfaceHandle	The interface handle to notify Turbo Treck of.
numberRecvPackets	The number of received packets to notify of.
numberSendCompletePackets	The number of packets actually sent on this interface.
numberBytesSent	The number bytes actually sent on this interface.
flag	Unused

Note: The values for* numberRecvPackets,numberSendCompletePackets, *and* numberBytesSent *may be 0.

Returns

None

Note: This function should be called only once per Isr. tfNotifyInterfaceIsr replaces both tfNotifyReceiveInterfaceIsr and tfNotifySentInterfaceIsr.

tfNotifyInterfaceTask

```
#include <trsocket.h>
void                tfNotifyInterfaceTask
(
ttUserInterface     interfaceHandle,
int                 numberRecvPackets,
int                 numberSendCompletePackets,
unsigned long       numberBytesSent,
unsigned long       flag
);
```

Function Description

This function notifies the stack with both the number of packets received and the number of packets transmitted by the chip. This function is to be called by the user inside a Task (rather than an ISR) with the appropriate number of received and sent packets. This will in turn cause the stack to notify the user's application about what has occurred. In the case of received packets, **tfCheckReceiveInterface** will return 0 once for each received packets if the user is in polling mode. In blocking mode, **tfWaitReceiveInterface** will return once for each received packet. For sent packets, **tfCheckSentInterface** will return 0 once for each sent packet in polling mode, or **tfWaitSentInterface** will return once for each sent packet in blocking mode. These functions only notify the user of a sent packet in the case that the accumulated **numberBytesSent** has reached TM_NOTIFY_SEND_LOW_WATER (2048 by default), since the previous notification.

Parameters

Parameter	Description
interfaceHandle	the interface handle to notify Turbo Treck of.
numberRecvPackets	The number of received packets to notify of.
numberSendCompletePackets	The number of packets actually sent on this interface.
numberBytesSent	The number bytes actually sent on this interface.
flag	Unused

Note*: *The values for* numberRecvPackets,numberSendCompletePackets, *and* numberBytesSent *may be 0.

Returns

None

tfNotifyReceiveInterfaceIsr

Note: tfNotifyReceiveInterfaceIsr has been deprecated.Please use tfNotifyInterfaceIsr. tfNotifyReceiveInterfaceIsr will still function in your code.

tfNotifySentInterfaceIsr

Note: tfNotifySentInterfaceIsr has been deprecated. Please use tfNotifyInterfaceIsr. tfNotifySentInterfaceISr will still function in your code.

tfOpenInterface

```
#include <trsocket.h>

int                 tfOpenInterface
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ipAddress,
ttUserIpAddress     netMask,
int                 flag,
int                 buffersPerFrameCount
);
```

Function Description

This function is used to configure an interface with an IP address, and net mask (either supernet or subnet). It must be called before the interface can be used.

An example would be:

```
            errorCode = tfOpenInterface (
                        myInterfaceHandle,
                        inet_addr (“208.229.201.65”),
                        inet_addr (“255.255.255.0”),
                        0,
                        1);
```

Note: tfConfigInterface is deprecated for the first multihome. Please use tfOpenInterface for the first configuration on an interface.

DHCP or BOOTP configuration.

tfOpenInterface with a TM_DEV_IP_DHCP (respectively TM_DEV_IP_BOOTP) flag will send a DHCP (respectively BOOTP) request to a DHCP/BOOTP server, and will return with a TM_EINPROGRESS errorCode.

Note that **tfUseDhcp** (respectively **tfUseBootp**) needs to have been called prior to calling **tfOpenInterface**, otherwise the call will fail with error code TM_EPERM.

An example of a configuration using the DHCP protocol would be:

```
errorCode = tfOpenInterface (
            myInterfaceHandle,
            (ttUserIpAddress)0,
            (ttUserIpAddress)0,
            TM_DEV_IP_DHCP,
            1);
```

Checking on completion of DHCP or BOOTP configuration.

- **Synchronous check**: The user can make multiple calls to **tfOpenInterface** to determine when the configuration has completed.. Additional calls to **tfOpenInterface** will return TM_EALREADY as long as the BOOTP/DHCP server has not replied. **tfOpenInterface** will return TM_ENOERROR, if the BOOTP/DHCP server has replied and the configuration has completed.

- **Asynchronous check**: To avoid this synchronous poll, the user can provide a user call back function to **tfUseDhcp** (respectively **tfUseBootp**), which will be called upon completion of **tfOpenInterface**. See **tfUseDhcp** (respectively **tfUseBootp**) for details.

Parameters

Parameter	**Description**
interfaceId	The device entry. This is returned by **tfAddInterface**.
ipAddress	The IP Address for this Interface.
netMask	The net mask for this device (Subnet or Supernet)
flag	Special flags for this deviceOred together (see below).
buffersPerFrameCount	Number of scattered buffers allowed for each frame being sent out. If scattered buffers are not allowed by the driver, this number should be one, otherwise it should be bigger than one.

Flags

Value	**Meaning**
TM_DEV_SCATTER_SEND_ENB	This device supports sending data in multiple buffers per frame. If this flag is set, then the buffersPerFrameCount should be bigger than 1. This flag should always be set for SLIP or PPP serial devices.
TM_DEV_MCAST_ENB	This device supports multicast addresses
TM_DEV_IP_FORW_ENB	Allow IP Forwarding to and from this device
TM_DEV_IP_FORW_DBROAD_ENB	Allow forwarding of IP directed broadcasts to and from this device
TM_DEV_IP_FORW_MCAST_ENB	Allow forwarding of IP multicast messages to and from this device
TM_DEV_IP_BOOTP	Configure IP address using BOOTP client protocol. **tfUseBootp** need to have been called first.
TM_DEV_IP_DHCP	Configure IP address using DHCP client protocol. **tfUseDhcp** need to have been

called first.

TM_DEV_IP_USER_BOOT	Allow the user to temporarily configure the interface with a zero IP address, to allow the user to use a proprietary protocol to retrieve an IP address from the network.
TM_DEV_IP_NO_CHECK	Allow the Turbo Treck stack to function in promiscuous mode, where all packets received by this interface will be handed to the application without checking for an IP address match on the incoming interface.

Returns

Value	**Meaning**
0	Success
TM_EADDRNOTAVAIL	Attempt to configure the device with a broadcast address.
TM_EINPROGRESS	**tfOpenInterface** call has not completed. This error will be returned for a DHCP or BOOTP configuration for example.
TM_ENOBUFS	Not enough memory to complete operation
TM_EINVAL	Bad parameter. Note that a zero IP address is allowed for Ethernet if the BOOTP or DHCP flag is on, or for PPP, otherwise a TM_EINVAL errorCode is returned. TM_EALREADY A previous call to **tfOpenInterface** has not yet completed.
TM_EPERM	User attempted to configure an IP address via DHCP (respectively BOOTP) without having called **tfUseDhcp** (respectively **tfUseBootp)** successfully first.
TM_EMFILE	Not enough sockets to open the BOOTP client UDP socket (TM_IPDEV_BOOTP or TM_IP_DEV_DHCP configurations only.)

TM_ADDRINUSE	Another socket is already bound to the BOOTP client UDP port. (TM_IP_DEV_BOOTP or TM_IP_DEV_DHCP configurations only.)
TM_ETIMEDOUT	DHCP or BOOTP request timed out
TM_EAGAIN	A PPP session is currently closing. Call **tfOpenInterface** again after receving notification that the previous session has ended.
Other	Error value as returned by the device driver open function.

tfPoolCreate

```
#include <trsocket.h>

int                 tfPoolCreate
(
ttUserInterface      interfaceHandle,
int                  numberFullSizeBuffers,
int                  numberSmallSizeBuffers,
int                  fullBufferSize,
int                  smallBufferSize,
int                  alignment,
int                  flag
);
```

Function Description

Allocate a ring of pre-allocated device driver receive buffers, so that the device driver can get a pre-allocated buffer from this recv pool during an ISR. The user can specify a number of maximum size packets, and a number of small size packets. For Ethernet device driver the maxim size packet (fullBufferSize parameter) should be TM_ETHER_MAX_PACKET_CRC. This function should be called from the device driver open function, and the device driver should only get a buffer from this pool during an ISR.

Note: The user must periodically replenish the Turbo Treck ISR recv pool as follows: errorCode =tfIoctlInterface(interfaceHandle, TM_DEV_IOCTL_REFILL_POOL _FLAG, (void *)0, 0);

Parameters

Parameter	**Description**
interfaceHandle	Interface handle as given in the first parameter of the driver open function, or as returned by **tfAddInterface**.
numberFullSizeBuffers	Number of full size buffers to pre-allocate in the ring.
numberSmallSizeBuffers	Number of small size buffers to pre-allocate in the ring.
fullBufferSize	Maximum buffer size (TM_ETHER_MAX_PACKET_CRC for Ethernet).
smallBufferSize	Small buffer size (for example 128 bytes).
alignment	Specify data pointer alignment for pre-allocated receive buffers.

flag	Indicates whether buffers should be re-allocated in line when the tfPoolReceive function is called (in the context of the recv task). 0 means no in-line reallocation. TM_POOL_REFILL_IN_LINE means in-line reallocation.

Returns

Value	**Meaning**
TM_EINVAL	numberFullSizeBuffers parameter is less or equal to zero, or fullBufferSize parameter is less or equal to zero, or fullBufferSize is less than TM_ETHER_MAX_PACKET_CRC on Ethernet, or numberSmallSizeBuffers is negative, or smallBufferSize is negative, or smallBufferSize is zero and numberSmallSizeBuffers is not zero, or alignment is negative, or alignment is bigger than 64, or flag is neither 0, nor TM_POOL_REFILL_IN_LINE
TM_EPERM	Function not called from driver device open
TM_EALREADY	Pool already created on that interface
TM_ENOBUFS	Not enough memory to allocate the recv pool

tfPoolDelete

```
#include <trsocket.h>

int                 tfPoolDelete
(
ttUserInterface     interfaceHandle
);
```

Function Description

Free the pool of pre-allocated recv buffers, which was allocated in fPoolCreate. ***Should be called by the user from the device driver close function.***

Parameters

Parameter	**Description**
interfaceHandle	Interface handle as given in the first parameter of the driver close function, or as returned by tfAddInterface.

Returns

Value	**Meaning**
TM_EALREADY	Pool already deleted, or not created on that interface

tfPoolIsrFreeBuffer

```
#include <trsocket.h>
void               tfPoolIsrFreeBuffer
(
ttUserInterface    interfaceHandle
int                dataSize
);
```

Function description

Called from recv ISR, if last packet returned by tfPoolIsrGetBuffer(), queued to be processed by the tfRecvInterface, needs to be freed, i.e. removed from the pool recv queue, and put back in the ring of free buffers to be used by the chip.

Parameters

Parameter	**Description**
ttUserInterface	Interface handle
size	size of buffer just allocated

Returns

None

tfPoolIsrGetBuffer

```
#include <trsocket.h>

char TM_FAR *       tfPoolIsrGetBuffer
(
ttUserInterface    interfaceHandle,
int                size
);
```

Function Description

Get a pre-allocated buffer from the recv pool, and queue that buffer in the pool receive queue. ***Can only be called from an ISR***.

Parameters

Parameter	Description
interfaceHandle	Interface handle as returned by **tfAddInterface**
size	Size of buffer to get from the pre-allocated pool

Returns

A char * to the beginning of the data area to store the received data into or a NULL pointer if there are no available receive pool buffers.

tfPoolReceive

#include <trsocket.h>

```
int                         tfPoolReceive
(
ttUserInterface             interfaceHandle,
char TM_FAR * TM_FAR *      dataPtrPtr,
int  TM_FAR *               dataLengthPtr,
ttUserBufferPtr             bufHandlePtr
);
```

Function Description

Remove first received buffer, from the pool receive queue. Store in dataPtrPtr, dataLengthPtr, bufHandlePtr, the buffer data pointer, data length, and buffer handle respectively. This function could be the drvRecvFuncPtr parameter to **tfAddInterface**. Otherwise the user calls this function from the device driver receive function. The parameters are identical to the device driver receive function.

Parameters

Parameter	**Description**
interfaceHandle	Interface handle as returned by tfAddInterface
dataPtrPtr	Pointer to area where to store the buffer data pointer
dataLengthPtr	Pointer to area where to store the buffer data length
bufHandlePtr	Pointer to area where to store the buffer handle

Returns

Value	**Meaning**
TM_DEV_OKAY	Success
TM_DEV_ERROR	No buffer in the pool receive queue

tfRecvInterface

#include <trsocket.h>

```
int                 tfRecvInterface
(
ttUserInterface     interfaceHandle
);
```

Function Description

This function is used to start processing of an incoming packet from the device driver. It first calls the driver's receive routine and then begins processing of the packet. The packet is processed in the context of the caller of this function.

Parameters

Parameter	**Description**
interfaceHandle	The Interface Handle of the device to receive data from

Returns

Value	**Meaning**
0	Success
TM_ENOBUFS	Insufficient memory to complete the operation

tfRecvScatInterface

```
#include <trsocket.h>


int tfRecvScatInterface
(
ttUserInterface interfaceHandle
);
```

Function Description

This function is used to start processing of an incoming packet from the device driver, and can handle scattered data within a single frame ("Gather Read"). When the user has been notified that an incoming packet has arrived (via either **tfCheckReceiveInterface**, or **tfWaitReceiveInterface**), it then calls **tfRecvScatInterface** so that the stack can call the driver scattered receive function, and retrieve the data. **tfRecvScatInterface** first calls the driver's scattered recv routine (as specified in the **tfUseInterfaceScatRecv** API), and then begins processing of the packet. The packet is processed in the context of the caller of this function.

The user should use tfRecvScatInterface instead of tfRecvInterface when the user wishes to support scattered data within a received frame ("Gather Read") in the device driver recv processing. The user needs to define TM_USE_SCAT_DRV_RECV, and needs to have called tfUseInterfaceScatRecv successfully on that interface. If the user has not called tfUseInterfaceScatRecv on that interface, then tfRecvInterface should be called instead.

Parameters

Parameter	**Description**
interfaceHandle	The interface Handle of the device to receive the scattered data from.

Returns

Value	**Meaning**
0	Success.
TM_ENOBUFS	Insufficient memory to complete the operation.
TM_EINVAL	One of the parameters returned by the scattered device driver recv function is invalid

tfSendCompleteInterface

```
#include <trsocket.h>

void                tfSendCompleteInterface
(
ttUserInterface     interfaceHandle,
int                 flag
);
```

Function Description

This function is used to notify the stack that send has been completed and allow the stack to free the outgoing buffers. If the device driver copies the packet before attempting the send, it is permissible to call this routine from the context of the send, otherwise, it must only be called after the device had actually sent the packet.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle for the device to process the send complete.
flag	Indicates from where **tfSendCompleteInterface** is called. See below. **It is very important to set the appropriate flag**.

Flag	**Description**
TM_DEV_SEND_COMPLETE_DRIVER	t**fSendCompleteInterface** is being called from the device driver.
TM_DEV_SEND_COMPLETE_APP	**tfSendCompleteInterface** is being called either from a send task or from the main loop.

Returns

Nothing

tfSendCompletePacketInterface

```
#include <trsocket.h>

void                   tfSendCompletePacketInterface
(
ttUserInterface        interfaceHandle,
ttUserPacketPtr        packetPtr,
int                    devDriverLockFlag
);
```

Function Description

This function is used to notify the stack that send has completed for a given frame, so that the stack can remove it from the device send queue, and free it. If the device driver copies the packet before attempting the send, it is permissible to call this routine from the context of the send, otherwise, it must only be alled after the device had actually sent the packet.

Use tfSendCompletePacketInterface, instead of tfSendCompleteInterface when the device driver transmits the frames out of order. tfSendCompletePacketInterface can only be used in conjunction with the single device driver send call per frame. (See tfUseInterfaceOneScatSend, and TM_USE_DRV_ONE_SCAT_SEND.)

Parameters

Parameter	**Description**
interfaceHandle	The interface handle for the device to process the send complete
packetPtr	Indicates which frame in the device send queue needs to be dequeued, and freed.
flag	Indicates from where **tfSendCompletePacketInterface** is called. See below. It is very important to set the appropriate flag.

Flag	**Description**
TM_DEV_SEND_COMPLETE_DRIVER	**tfSendCompletePacketInterface** is being called from the device driver.
TM_DEV_SEND_COMPLETE_APP	**tfSendCompletePacketInterface** is being called from either a send task, or the main loop

Returns

Nothing

tfSetIfMtu

```
int              tfSetIfMtu
(
ttUserInterface  interfaceHandle,
int              ifMtu
);
```

Function Description

This function is used to set the Maximum Transmission Unit (MTU) for a device. For Ethernet and PPP, this is typically set to 1500 bytes. The link MTU is always the size of the largest IP packet which can be sent unfragmented over the link, which is the maximum link-layer frame size minus any link-layer header and trailer overhead. For PPP and Ethernet, this value can be changed via Path MTU Discovery to allow larger frames and to prevent IP datagram fragmentation.

Parameters

Parameter	**Description**
interfaceHandle	The device driver entry that we wish to get the physical address on.
ifMtu	The Maximum Transmission Unit for a device

Returns

Value	**Meaning**
0	Success
TM_EINVAL	One of the parameter is null or 0.

By default, the path MTU discovery code is enabled. If you do not need path MTU in your *trsystem.h,* define:

```
#define TM_DISABLE_PMTU_DISC
```

This will prevent the compilation of the path MTU discovery code.

Turning off the path MTU discovery mechanism, setsockopt

If the path MTU discovery code has been compiled in, and there is a need to disable path MTU discovery on a given connection, the user can call the Berkeley socket API **setsockopt** on the connection or listening socket prior to the connection establishment, with the *IP_PROTOTCP* protocol level and *TCP_MAXSEG* option name. If the option value is less than 64 bytes, then the Turbo Treck stack will set the TCP MSS to the default value, which is the outgoing device IP MTU minus 40 bytes.

Turning off the path MTU discovery mechanism, tfDisablePathMtuDisc

Another way of turning off the Path MTU discovery mechanism for a given destination IP address is to use the new API **tfDisablePathMtuDisc**. This function will disable path MTU discovery for the route to the given destination IP address.

Path MTU estimates timeout values

When Path MTU discovery is enabled for an indirect TCP destination IP address, the Turbo Treck stack will try to increase the path MTU estimate (up to the device IP MTU) after the current path MTU estimate times out. Two new options have been added to **tfSetTreckOptions()** to allow the user to change the default timeout values for path MTU estimates.

tfUnConfigInterface

```
#include <trsocket.h>

int                 tfUnConfigInterface
(
ttUserInterface     interfaceId,
unsigned char       multiHomeIndex
);
```

Function Description

This function is used to remove an IP address from an interface for a particular multi home index. By un-configuring an interface, we can then re-configure it with a new IP address and netmask.

Parameters

Parameter	**Description**
interfaceId	The device entry
multiHomeIndex	The Index for this IP address.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	One of the parameters is invalid
TM_ENOENT	Interface was not configured

tfUseInterfaceOneScatSend

```
#include <trsocket.h>

int                     tfUseInterfaceOneScatSend
(
ttUserInterface         interfaceHandle,
ttDevOneScatSendFunc    devOneScatSendFunc
);
```

Function Description

Specify a driver driver scattered send function, that is called once per frame even when the data is scattered on a given interface. **tfUseInterfaceOneScatSend** must be called prior to calling **tfConfigInterface**, and after having called **tfAddInterface**, to replace the device driver send function specified in **tfAddInterface**.

Once **tfUseInterfaceOneScatSend** has been called, the device send logic in the Turbo Treck stack will call the specified single call device driver scattered send function, instead of the one provided in **tfAddInterface**.

Note: TM_USE_DRV_ONE_SCAT_SEND need to be defined in trsystem.h.

Note: This call is not supported on Point to Point interfaces, and is not supported in combination with an interface transmit queue.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle of the device to send data through.
devOneScatSendFunc	A function pointer to the device driver's send function that handles scattered data within a frame in a single call

Returns

Value	**Meaning**
0	Success
TM_EINVAL	devOneScatSendFunc is null
TM_EPERM	The interface has already been opened, or the interface is a point to point interface (SLIP, PPP), or a transmit queue has been configured on the interface.

devOneScatSendFunc type is defined as follows:

```
int devOneScatSendFunc
(
ttUserInterface interfaceHandle,
ttUserPacketPtr packetUPtr
);
```

devOneScatSendFunc Parameters:

Parameter	**Description**
interfaceHandle	The interface handle of the device to send data through.
packetUPtr	Pointer to a structure (ttUserPacket) that contains all the information on the frame to be sent.

ttUserPacket Fields:

ttUserPacket field	**Description**
ptkuLinkNextPtr	Points to the next ttUserPacket structure
pktuLinkDataPtr	Points to the data in the current link
pktuLinkDataLength	Contains the length of the data in the current link
pktuLinkChainDataLength	Contains the total length of the scattered data. Its value is ony valid in the first link
pktuLinkExtraCount	Contains the number of extra links besides the first one. Its value is only valid in the first link.

The single call per frame user device driver send function will loop through all the links of the scattered frame, in order to send a complete frame.

tfUseInterfaceScatRecv

```
#include <trsocket.h>

int tfUseInterfaceScatRecv
(
ttUserInterface        interfaceHandle,
ttDevScatRecvFunc      devScatRecvFunc
);
```

Function Description

By default the stack expects all data within a frame given by the user device driver recv function to be contiguous. Some device drivers support receiving data within a frame in scattered buffers ("Gather Read"), because it is more efficient.

tfUseInterfaceScatRecv replaces the default device driver recv function added in **tfAddInterface** with a driver driver scattered recv function that is called once per frame, even when the device driver received data is scattered on a given interface ("Gather Read").

tfUseInterfaceScatRecv must be called prior to calling **tfConfigInterface**, and after having called **tfAddInterface**, to replace the device driver recv function specified in **tfAddInterface**.

If **tfUseInterfaceScatRecv** has been called successfully, the user needs to call **tfRecvScatInterface** instead of **tfRecvInterface**. **tfRecvScatInterface** will call the modified device driver recv function specified in the **tfUseInterfaceScatRecv** call.

TM_USE_DRV_SCAT_RECV need to be defined in trsystem.h.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle of the device to recv data from.
devScatRecvFunc	A function pointer to the device driver's recv function that handles scattered data within a frame in a single call.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	devScatRecvFunc is null
TM_EPERM	The interface has already been opened

devScatRecvFunc type is defined as follows:

```
int devScatRecvFunc
(
ttUserInterface   interfaceHandle,
ttDruBlockPtrPtr  uDevBlockPtrPtr,
int               * uDevBlockCountPtr,
int               * flagPtr
);
```

devScatRecvFunc Parameters:

devScatRecvFunc Parameter	**Description**
interfaceHandle	The interface handle of the device to recv data from.
uDevBlockPtrPtr	Address of an area where to store an array of user blocks describing the scattered data (one block per scattered buffer).
uDevBlockCountPtr	Address of an area where to store the number of scattered buffers within a frame.
flagPtr	Address of an area where to store ownership of the scattered buffers that compose the received frame.

Upon return from the device driver scattered recv function, *uDevBlockPtrPtr points to an array of ttDruBlock.

There is one ttDruBlock per scattered buffer in the received frame.
Each ttDruBlock element contains a pointer to a user buffer, a pointer to the beginning of the user data in the user buffer, and the user data length in the user buffer.

ttDruBlock fields	Description
druDataPtr	points to beginning of data
druDataLength	indicates the data length
druBufferPtr	pointer to be passed to the device driver free function if user sets the flag to TM_DEV_SCAT_RECV_USER_BUFFER in the driver recv call function. (See *flagPtr below.)
druStackBufferPtr	pointer to stack pre-allocated user buffer if user sets the flag to TM_DEV_SCAT_RECV_STACK_BUFFER in the driver recv call function. (See *flagPtr below.)

If the device driver scattered recv function sets *flagPtr to TM_DEV_SCAT_RECV_STACK_BUFFER, then druStackBufferPtr should point to a buffer pointed to by the first parameter of either **tfGetEthernetBuffer**, or **tfGetDriverBuffer**, and the stack will be responsible for freeing that buffer, when the stack is done processing that buffer. If the device driver scattered recv function sets *flagPtr to TM_DEV_SCAT_RECV_USER_BUFFER, then druBufferPtr points to a user allocated buffer. When the stack is done processing that buffer, the stack will call the device driver free function (as set in **tfAddInterface**).

The user is responsible for managing the memory containing the array of ttDruBlock. It is guaranteed that when the stack calls the modified driver recv function on an interface, the array of ttDruBlock previously given to the stack by a previous call to the driver recv function on the same interface will not be accessed anymore. So it is safe for the user to re-use the array itself, then. On the other hand, it is only safe to re-use a user allocated buffer after the device driver free function has been called.

*flagPtr parameter	Description
TM_DEV_SCAT_RECV_STACK_BUFFER	The scattered buffers used in the frame are pre-allocated stack buffers. The stack will be responsible for freeing these buffers.
TM_DEV_SCAT_RECV_USER_BUFFER	The scattered buffers used in the frame belong to the user. When the stack is done processing a buffer in a received frame, the device driver free function as specified in tfAddInterface will be called for

that buffer.

tfUseInterfaceXmitQueue

```
#include <trsocket.h>

int                 tfUseInterfaceXmitQueue
(
ttUserInterface     InterfaceHandle
short               numberXmitDescriptors
);
```

Function Description

Enable/Disable the use of an interface transmit queue. If specified depth (numberXmitDescriptors parameter) is zero, this option is turned off. Otherwise this function allocate an empty transmit ring of numberXmitDescriptors elements. When an interface transmit queue is used, if the device driver send function returns an error, because the chip is not ready to transmit, a pointer to the buffer that could not be transmitted (along with its length, and flag) is stored in an empty slot, in the Turbo Treck device transmit queue. The device transmit queue should be big enough to hold pointers to all the buffers that will be sent by the application. The size of the data sent by an application is limited by the socket send queue size. So, an interface transmit queue should be big enough to hold pointers to buffers sent from all the application sockets through that particular interface. The space allocation overhead for a x entries device transmit queue is 12 + x * 8 bytes. So for a device transmit queue containing 1000 slots, the overhead is 8012 bytes.

Note: This function can only be called when the device is not configured. Cannot be called for a point to point interface. Cannot be called if a transmit task is used.

If the interface transmit queue is enabled, the user should call

```
errorCode =
tfIoctlInterface(interfaceHandle, TM_DEV_IOCTL_EMPTY_XMIT_FLAG,
                              (void *)0, 0);
```

periodically to try and empty the device transmit queue (which contains all the buffers that the device driver could not send right away).

Parameters

Parameter	**Description**
interfaceHandle	The Interface Handle of the device we are waiting to be ready.
numberXmitDescriptors	Length of the transmit queue. If positive, the interface transmit queue is enabled. If zero, it is disabled.

Returns

Value	**Meaning**
0	Success
TM_ENOBUFS	Not enough memory to allocate the empty transmit ring.
TM_EPERM	Device/Interface already configured/opened, or User already uses a transmit task, or device/ interface is a point to point interface (i.e PPP, or SLIP.)

tfUseIntfDriver

```
#include <trsocket.h>

ttUserInterface   tfUseIntfDriver
(
char *            namePtr,
ttUserLinkLayer   linkLayerHandle,
int *             *drvAddErrorPtr
);
```

Function Description

tfUseIntfDriver adds a loop back driver below the link layer, and returns an interface handle. An example would be as follows:

```
ethernetLinkLayerHandle = tfUseEthernet();
myInterfaceHandle=tfUseIntfDriver (
            "TEST1",
            etherLinkLayerHandle,
            &errorCode);
```

This function is very useful to debug a link layer. When using the loop back driver, some macro definitions, configurations are necessary to insure that the loop back driver is used and not bypassed for a given destination IP address (peer IP address).

Peer IP address	**Steps to ensure that the loop back driver is used**
configured in the stack	Define TM_LOOP_TO_DRIVER, or Define TM_SINGLE_INTERFACE_HOME
not configured in the stack	Define TM_DEV_IP_NO_CHECKIf the link layer is Ethernet, then a proxy arp entry should be added for the peer IP address.

Parameters

Parameter	**Description**
namePtr	The callers name for the device (each call to **tfUseIntfDriver** must use a unique name). The name length cannot exceed TM_MAX_DEVICE_NAME –1 (13 bytes).

linkLayerHandle The link layer (layer 2) protocol handle that this interface will use. This is returned by **tfUseEthernet**, **tfUseAsyncPpp**, or **tfUseSlip**.

drvAddErrorPtr A pointer to an int that will contain an error (if one occurred).

Returns

An Interface Handle or a NULL handle when an error occurs. If an error occurs, the error is stored in **drvAddErrorPtr.*

ErrorCode	**Description**
TM_EINVAL	One of the parameters is invalid
TM_EALREADY	Device with same name has already been added.
TM_ENOBUFS	Not enough buffers to allocate a device entry.

tfUseScatIntfDriver

```
#include <trsocket.h>

ttUserInterface   tfUseScatIntfDriver
(
char              * namePtr,
ttUserLinkLayer   linkLayerHandle,
int               * errorCode
);
```

Function Description

tfUseScatIntfDriver adds a loop back driver below link layer using one scattered device driver send call, and scattered device driver recv. **tfUseIntfDriver** and **tfUseScatIntfDriver** are very useful to debug a link layer. **tfUseScatIntfDriver** avoids copying the data in the loop back driver.

Example:

```
ethernetLinkLayerHandle = tfUseEthernet();
myInterfaceHandle = tfUseScatIntfDriver(
                             "TESTSCAT",
                             ethernetLinkLayerHandle
                             &errorCode);
```

Note: Both TM_USE_ONE_SCAT_DRV_SEND, and TM_USE_SCAT_DRV_RECV need to be defined. If either one of these macros is not defined, then tfUseScatIntfDriver will revert to tfUseIntfDriver.

When using the loop back driver (scattered or non scattered version), some macro definitions, configurations are necessary to insure that the loop back driver is used and not bypassed for a given destination IP address (peer IP address).

Peer IP address	**Steps to ensure the loop back driver is used**
configured in the stack	Define TM_LOOP_TO_DRIVER, or Define TM_SINGLE_INTERFACE_HOME
not configured in the stack	Define TM_DEV_IP_NO_CHECK If the link layer is Ethernet, then a proxy arp entry should be added for the peer IP address.

Parameters

Parameter	**Description**
namePtr	The callers name for the device (each call to **tfUseScatIntfDriver** must use a unique name). The name length cannot exceed TM_MAX_DEVICE_NAME –1 (13 bytes).
linkLayerHandle	The link layer (layer 2) protocol handle that this interface will use. This is returned by **tfUseEthernet**, **tfUseAsyncPpp**, or **tfUseSlip**.
drvAddErrorPtr	A pointer to an int that will contain an error (if one occurred).

Returns

An Interface Handle or a NULL handle when an error occurs. If an error occurs, the error is stored in **drvAddErrorPtr.*

ErrorCode	**Description**
TM_EINVAL	One of the parameters is invalid
TM_EALREADY	Device with same name has already been added.
TM_ENOBUFS	Not enough buffers to allocate a device entry.

tfWaitReceiveInterface

```
#include <trsocket.h>

int                 tfWaitReceiveInterface
(
ttUserInterface     interfaceHandle
);
```

Function Description

This function is used to wait for the **tfNotifyInterfaceIsr** routine to be called from the device driver's receive interrupt service routine. If there is no call to **tfNotifyInterfaceIsr** , then it waits. It returns each time that there is ONE new packet to be received (which is signaled by tfNotifyInterfaceIsr). The caller should then call **tfRecvInterface** when this call completes successfully.

Note: tfWaitReceiveInterface can only be used when there is a separate receive task.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle for the device we wish to wait for data for.

Returns

Value	**Meaning**
0	Success
TM_ENOBUFS	Insufficient memory to complete operation

tfWaitSentInterface

```
#include <trsocket.h>

int                 tfWaitSentInterface
(
ttUserInterface     interfaceHandle
);
```

Function Description

This function is used to wait for a tfNotifyInterfaceIsr function to be called from the transmit complete interrupt service routine, if there is none, then it waits. This function will wait until the accumulated numberBytesSent notified by tfNotifyInterfaceIsr has reached TM_NOTIFY_SEND_LOW_WATER (2048 by default).

Note: This function can only be used when there is a separate send complete task.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle of the device to wait for a transmit complete to occur

Returns

Value	**Meaning**
0	Success
TM_ENOBUFS	Insufficient memory to complete the operation

tfWaitXmitInterface

```
#include <trsocket.h>

int
(                  tfWaitXmitInterface
ttUserInterface    interfaceHandle
);
```

Function Description

This function is used to wait for data ready to be sent from the bottom of the Turbo Treck stack to the device driver (data queued to the interface send queue). If there is no data ready to be transmitted, then it waits. The caller should then call **tfXmitInterface,** when this call completes successfully.

Note: tfWaitXmitInterface can only be used when there is a separate transmit task.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle for the device to which to send data.

Returns

Value	**Meaning**
0	Success
TM_ENOBUFS	Insufficient memory to complete operation

tfXmitInterface

```
#include <trsocket.h>

int                tfXmitInterface
(
ttUserInterface    interfaceHandle
);
```

Function Description

tfXmitInterface is used to call the device driver send function with the next packet ready to be transmitted in the device/interface send queue. The packet is sent in the context of the caller of this function.

Parameters

Parameter	**Description**
interfaceHandle	The Interface Handle of the device to which to send data.

Returns

Value	**Meaning**
0	Success
TM_ENOENT	No packet was ready to be transmitted TM_ENXIO Device/interface was not opened (or configured)
TM_EIO	Device driver send function returned an error. The Turbo Treck stack freed the packet since the device driver send failed.

Ethernet Link Layer API

tfGetEthernetBuffer

```
#include <trsocket.h>

char *             tfGetEthernetBuffer
(
ttUserBufferPtr    userBufferPtr
);
```

Function Description

This function is used to get a buffer from the system for the user to use for an Ethernet device. The data pointer is offset by two bytes to allow the data portion to be long word aligned. It is not required for a user to use our buffer pool because some devices may not support it.

Parameters

Parameter	Description
userBufferPtr	A pointer to a ttUserBuffer variable that the user buffer handle is stored.

Returns

A char * to the beginning of the data area to store the received data into or a NULL pointer if there is no memory to complete the operation

tfUseEthernet

```
#include <trsocket.h>

ttUserLinkLayer   tfUseEthernet
(
void
);
```

Function Description
This function is used to initialize the Ethernet Link Layer

Parameters

None

Returns

The Ethernet link layer handle or NULL if there is an errorNull Link Layer API

Null Link Layer API

tfUseNullLinkLayer

```
#include <trsocket.h>

ttUserLinkLayer   tfUseNullLinkLayer
(
void
);
```

Function Description
This function is used to initialize the Null Link Layer

Parameters

None

Returns

The Null link layer handle or NULL if there is an error

SLIP Link Layer API

tfGetSlipPeerIpAddress

```
#include <trsocket.h>

int                tfGetSlipPeerIpAddress
(
ttUserInterface    interfaceHandle,
ttUserIpAddress *  ifIpAddress
);
```

Function Description

This function is used to get the SLIP address that the remote SLIP has used. This function should be called after **tfOpenInterface** has completed successfully.

If a default gateway needs to be added for that interface, then the retrieved IP address should be used to add a default gateway through the corresponding interface.

If a static route needs to be added for that interface, then the retrieved IP address should be used to add a static route through the corresponding interface.

Parameters

Parameter	**Description**
interfaceHandle	The interface Handle to get the Peer IP address from.
ifIpAddressPtr	The pointer to the buffer where the Peer slip IP address will be stored into.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	One of the parameters is null, or the device is a LAN device
TM_ENETDOWN	Interface is not configured

Note: Also see tfSetSlipPeerIpAddress

tfSetSlipPeerIpAddress

```
#include <trsocket.h>

int                 tfSetSlipPeerIpAddress
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ifIpAddress
);
```

Function Description

This function is used to set a default remote SLIP IPaddress. This IP address will be used as the default remote point to point IP address, in case no remote IP address is negotiated with SLIP. If no IP address is set with this function, then the local IP address + 1 will be used as the default IP address for the remote SLIP for routing purposes **tfSetSlipPeerIpAddress** can only be called between **tfAddInterface** (or **tfCloseInterface**) and **tfOpenInterface**.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle to update the Peer IP address in
ifIpAddress	The IP address to use for routing purposes for the remote SLIP system

Returns

Value	**Meaning**
0	Success
TM_EINVAL	The interface handle is null, or the device is a LAN device
TM_EISCONN	SLIP connection is already established

Note: Also see tfGetSlipPeerIpAddress

tfSlipSetOptions

```
#include <trsocket.h>

int                 tfSlipSetOptions
(
ttUserInterface     interfaceHandle,
int                 optionName,
void TM_FAR *       optionValuePtr,
int                 optionLength
);
```

Function Description

Configure SLIP options. OptionValuePtr points to a variable of type as described below. OptionLenth contains the sizeof that variable. The only option supported is TM_SLIP_OPT_SEND_BUF_SIZE.

optionName

TM_SLIP_OPT_SEND_BUF_SIZE

Description

Length of data buffered by the SLIP link layer (but not beyond the end of a packet) before the device driver send function is called.
Default: 1 byte

Data Type

unsigned short

Parameters

Parameter	**Description**
interfaceHandle	Interface handle of the SLIP interface we want to set the option on.
optionName	The option to set. See above.
optionValuePtr	The pointer to a user variable into which the option value is set. User variable is of data type described above.
optionLength	Size of the user variable, which is the size of the option data type.

Returns

Value	**Meaning**
0	Success
TM_ENOBUFS	Insufficient memory to complete operation
TM_EINVAL	Invalid option length for option, or invalid option value for option.
TM_EPERM	Device/interface is not a SLIP interface
TM_EPROTONOSUPPORT	Option name not supported.

tfUseSlip

#include <trsocket.h>

```
ttUserLinkLayer   tfUseSlip
(
void
);
```

Function Description

This function is used to initialize the SLIP link layer.

Parameters

None

Returns

The SLIP link layer handle or NULL if there is an error.

ARP/Routing Table API

tfAddArpEntry

#include <trsocket.h>

```
int                 tfAddArpEntry
(
ttUserIpAddress     arpIpAddress,
char *              physAddrPtr,
int                 physAddrLength
);
```

Function Description

This function is used add an entry to the ARP cache. This function will allow the user to manipulate the ARP cache beyond standard means. Normally the TCP/IP stack maintains the ARP cache.

Parameters

Parameter	Description
arpIpAddress	The IP address to add to the ARP cache
physAddrPtr	A pointer to the character array that contains the physical address
physAddrLength	The length of the physical address

Returns

Value	Meaning
0	Success
TM_EINVAL	Bad parameter.

tfAddDefaultGateway

#include <trsocket.h>

```
int                 tfAddDefaultGateway
(
ttUserInterface     interfaceId,
ttUserIpAddress     gatewayIpAddress
);
```

Function Description
This function is used to add the system default gateway for all interfaces.

Parameters

Parameter	Description
interfaceId	The device entry
gatewayIpAddress	The default gateway IP address in Network Byte Order

Returns

Value	Meaning
0	Success
TM_ENOBUFS	Not enough buffer to allocate a routing entry
TM_EALREADY	A default gateway is already in the routing table
TM_EHOSTUNREACH	The gateway is not directly accessible
TM_EINVAL	One of the parameters is bad: the gatewayIpAddress is zero, or interfaceId refers to the loopback interface

tfAddMcastRoute

#include <trsocket.h>

```
int                 tfAddMcastRoute
(
ttUserInterface     interfaceId,
ttUserIpAddress     mcastAddress,
ttUserIpAddress     netmask,
unsigned char       mhomeIndex
);
```

Function Description

This function is used to add a multicast route associating a specific multicast destination address with a specific outgoing interface. Normally the netmask parameter is set to all 1’s.

Parameters

Parameter	**Description**
interfaceId	The interface ID to use to add this routing entry. Identifies the outgoing interface to use for packets sent to the multicast destination address specified by mcastAddress.
mcastAddress	The multicast destination address to add the route for.
netmask	The netmask for the route. Normally, this will be all 1’s, i.e. **inet_addr**(“255.255.255.255”), which means that there must be an exact match of the packet destination address with mcastAddress for this route to be used to send the packet.
mhomeIndex	The multi-home index to use to add this routing entry. This is the multihome index of a valid IP address that is already configured on the interface identified by interfaceId. This IP address is used as the default source IP address in packets sent to mcastAddress.

Returns

Value	**Meaning**
TM_ENOERROR	Successful.
TM_EINVAL	interfaceId is an invalid interface ID.
TM_ENETDOWN	The interface specified by interfaceId is not configured.
TM_EADDRNOTAVAIL	Either the interface specified by interfaceId was not configured with the multicast enabled flag (i.e. TM_DEV_MCAST_ENB), or mcastAddress is not a valid multicast IP address.
TM_EALREADY	This multicast route already exists.

tfAddProxyArpEntry

#include <trsocket.h>

```
int                 tfAddProxyArpEntry
(
ttUserIpAddress     arpIpAddress
);
```

Function Description
Add an entry to the Proxy ARP table for the given IP address. arpIpAddress is expected to be in network byte order.

Parameters

Parameter	**Description**
arpIpAddrss	Ip address on behalf of which the system will reply to ARP requests

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EINVAL	Bad parameter (0 IP address parameter)
TM_EALREADY	Entry already in PROXY ARP table
TM_ENOBUFS	Couldn't allocate proxy ARP entry

tfAddStaticRoute

#include <trsocket.h>

```
int                 tfAddStaticRoute
(
ttUserInterface     interfaceId,
ttUserIpAddress     destIpAddress,
ttUserIpAddress     destNetMask,
ttUserIpAddress     gateway,
int                 hops
);
```

Function Description

This function is used to add a route for the interface. It allows packets for a different network to be routed to the interface.

Parameters

Parameter	Description
interfaceId	The interface ID to use to add this routing entry
destIpAddress	The IP address to add the route for
destNetMask	The net mask for the route
gateway	IP address of the gateway for this route.
hops	Number of routers between this host and the route.

Returns

Value	Meaning
0	Success
TM_ENOBUFS	Not enough buffer to allocate a routing entry
TM_EALREADY	The route is already in the routing table.
TM_EHOSTUNREACH	The gateway is not directly accessible.
TM_EINVAL	One of the first 4 parameters is null or 0.

tfDelArpEntryByIpAddr

#include <trsocket.h>

```
int               tfDelArpEntryByIpAddr
(
ttUserIpAddress   arpIpAddress
);
```

Function Description

This function is used delete an entry in the ARP cache. This function will allow the user to manipulate the ARP cache beyond standard means. Normally, the TCP/IP stack maintains the ARP cache.

Parameters

Parameter	**Description**
arpIpAddress	The IP address to delete in the ARP cache

Returns

Value	**Meaning**
0	Success
TM_EINVAL	Bad parameter.

tfDelArpEntryByPhysAddr

#include <trsocket.h>

```
int                 tfDelArpEntryByPhysAddr
(
char *              physAddrPtr,
int                 physAddrLength
);
```

Function Description

This function is used to delete an entry in the ARP cache by looking up the entry by the Physical Address. This function will allow the user to manipulate the ARP cache beyond standard means. Normally, the TCP/IP stack maintains the ARP cache.

Parameters

Parameter	Description
physAddrPtr	The Physical Address to delete in the ARP cache
physAddrLength	The length of the physical address

Returns

Value	Meaning
0	Success
TM_EINVAL	Bad parameter.

tfDelDefaultGateway

#include <trsocket.h>

```
int              tfDelDefaultGateway
(
ttUserIpAddress  gatewayIpAddress
);
```

Function Description

This function is used to delete the system default gateway for all interfaces.

Parameters

Parameter	**Description**
gatewayIpAddress	The default gateway IP address in Network Byte Order

Returns

Value	Meaning
0	Success
TM_EINVAL	Parameter is 0
TM_ENOENT	No default gateway was found

tfDelProxyArpEntry

#include <trsocket.h>

```
int                 tfDelProxyArpEntry
(
ttUserIpAddress     arpIpAddress
);
```

Function Descrpition:
This function deletes an entry from the Proxy ARP table for the given IP address. *arpIpAddress* is expected to be in network byte order.

Parameters

Parameter	**Description**
arpIpAddrss	IP address on behalf of which the system will stop replying to ARP requests.

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EINVAL	Bad parameter (0 IP address parameter)
TM_ENOENT	Entry was not in PROXY ARP table.

tfDelStaticRoute

#include <trsocket.h>

```
int               tfDelStaticRoute
(
ttUserIpAddress   destIpAddress,
ttUserIpAddress   destNetMask
);
```

Function Description

This function is used to delete a route from the interface.

Parameters

Parameter	**Description**
destIpAddress	The IP Address to add the route for
destNetMask	The net mask for the route

Returns

Value	**Meaning**
0	Success
TM_EINVAL	One of the parameter is 0
TM_ENOENT	No routing enry for this route in the routing table.
TM_EPERM	Cannot delete an ARP entry with **tfDelStaticRoute**

tfDisablePathMtuDisc

```
#include <trsocket.h>

int                  tfDisablePathMtuDisc
(
ttUserIpAddress      destIpAddress,
unsigned short       pathMtu
);
```

Function Description

This function is used to disable path MTU discovery for a given route. If pathMtu is zero, or bigger than the outgoing device IP MTU, then we will default the route IP MTU to the outgoing device IP MTU; otherwise we will set the route IP MTU with the passed parameter value.

Parameters

Parameter	**Description**
destIpAddress	The IP address destination on which route we want to disable path MTU discovery.
pathMtu	New fixed IP MTU. If zero, we default to the device IP MTU.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	DestIpAddress parameter is zero.
TM_EPERM	Route is direct. No path MTU discovery is ever going to take place.
TM_EHOSTUNREACH	No route to destination IP address.
TM_ENOBUFS	Not enough memory to allocate new routing entry.

tfGetArpEntryByIpAddr

```
#include <trsocket.h>
int                   tfGetArpEntryByIpAddr
(
ttUserIpAddress       arpIpAddress,
char  *               physAddrPtr,
int                   physAddrLength
);
```

Function Description

This function is used retrieve an entry from the ARP cache by looking up the entry by the IP address. This function will allow the user to manipulate the ARP cache beyond standard means. Normally the TCP/IP stack maintains the ARP cache.

Parameters

Parameter	Description
arpIpAddress	The IP address to use to lookup the entry by
physAddrPtr	The Pointer to the buffer where to store the physical address
physAddrLength	The length of the physical address buffer

Returns

Value	Meaning
0	Success
TM_EINVAL	Bad parameter
TM_ENOENT	No ARP entry found with this IP address

tfGetArpEntryByPhysAddr

```
#include <trsocket.h>

int                  tfGetArpEntryByPhysAddr
(
char *               physAddrPtr,
int                  physAddrLen,
ttUserIpAddress *    arpIpAddressPtr
);
```

Function Description

This function is used retrieve an entry from the ARP cache by looking up the entry by the Physical Address. This function will allow the user to manipulate the ARP cache beyond standard means. Normally, the TCP/IP stack maintains the ARP cache.

Parameters

Parameter	Description
physAddrPtr	The Physical Address to lookup the entry in the ARP cache.
physAddrLen	The length of the physical address
arpIpAddressPtr	The location where to store the IP address of the matching physical entry

Returns

Value	Meaning
0	Success
TM_EINVAL	Bad parameter
TM_ENOENT	No ARP entry found with this physical address

tfGetDefaultGateway

```
#include <trsocket.h>

int                 tfGetDefaultGateway
(
ttUserIpAddress     TM_FAR * gwayIpAddPtr
);
```

Function Description

This function is called from the socket interface to get the default gateway IP address. The default gateway IP address will be stored in network byte order.

Parameters

Parameter	Description
gwayIpAddrPtr	Pointer to store gateway IP address into.

Returns

Value	Meaning
TM_ENOERROR	Success
TM_EINVAL	Bad parameter
TM_ENOENT	No default gateway

tfRtDestExists

```
#include <trsocket.h>

int                 tfRtDestExists
(
ttUserIpAddress     destIpAddress,
ttUserIpAddress     destNetMask
);
```

Function Description

Find out whether a route to a destination, given by the pair destination IP address and destination IP network mask, exists.

Parameters

Parameter	Description
destIpAddress	Destination IP address
destNetMask	Destination IP Network Mask

Returns

Value	Meaning
0	Success
TM_EHOSTUNREACH	No route to destination.

tfRegisterIpForwCB

```
#include <trsocket.h>

int                           tfRegisterIpForwCB
(
ttUserIpForwCBFuncPtr         ipForwCBFuncPtr
);
```

Function Description

Used to register a function for the Treck stack to call when a packet cannot be forwarded. The function's parameters will indicate the source IP address, and destination IP address of the packet in network byte order. If the call back function returns an error code, then the stack will send a host unreachable ICMP error message as before. If the call back function returns TM_ENOERROR, then the stack will silently drop the packet. The prototype for the callback function is:

int ipForwardCallback(ttUserIpAddress srcIpAddress, ttUserIpAddress destIpAddress);

Parameters

Parameter	**Description**
ipForwCBFuncPtr	A pointer to the forwarding callback function.

Returns

Value	**Meaning**
0	Success

tfUseRip

```
#include <trsocket.h>

int                 tfUseRip
(
void
);
```

Function Description

This function is called to start the RIPv2 Listener. It will also turn on the TM_OPTION_RIP_ENABLE mentioned in **tfSetTreckOptions** above.

Parameters

None

Returns

Value	Meaning
0	Success
TM_EMFILE	Not enough sockets to open the RIP socket
TM_ADDRINUSE	Another socket is already bound to the RIP UDP port.
TM_EALREADY	**tfUseRip** has already been called successfully.

Timer Interface API

tfTimerExecute

```
#include <trsocket.h>

void                **tfTimerExecute**
(
void
);
```

Function Description

This function is used to execute timers that have expired.

Parameters

None

Returns

Nothing

tfTimerUpdate

```
#include <trsocket.h>

void                  tfTimerUpdate
(
void
);
```

Function Description

This function updates the internal Turbo Treck timers. It tells them that one clock tick has passed. This call cannot be made from an Interrupt Service Routine.

Note: The integrator must decide if the Turbo Treck timers should be updated from an ISR, main loop, or a timer task and choose this call or tfTimerUpdateIsr.

Parameters

None

Returns

Nothing

tfTimerUpdateIsr

```
#include <trsocket.h>

void                  tfTimerUpdateIsr
(
void
);
```

Function Description
This function updates the internal Turbo Treck timers. It tells them that one clock tick has passed. This call is designed to be called from an Interrupt Service Routine.

Note: The integrator must decide if the Turbo Treck timers should be updated from an ISR, main loop, or a timer task and choose this call or tfTimerUpdate.

Parameters
None

Returns
Nothing

Kernel/RTOS Interface

tfKernelCreateCountSem

```
#include <trsocket.h>

int                      tfKernelCreateCountSem
(
ttUserGenericUnionPtr    countingSemaphorePtr
);
```

Function Description
RTOS SPECIFIC

This function is used to create a counting semaphore that is used by Turbo Treck.

Parameters

Parameter	Description
countingSemaphorePtr	A RTOS Counting semaphore

Returns

Value	Meaning
0	Success
-1	An error occurred

tfKernelCreateEvent

```
#include <trsocket.h>

void                      tfKernelCreateEvent
(
ttUserGenericUnionPtr     eventPtr
);
```

Function Description
RTOS SPECIFIC

This function is used to create an event structure for pend/post from an ISR.

Parameters

Parameter	Description
eventPtr	A pointer to an ISR event structure that is returned from an RTOS

Returns
Nothing

tfKernelDeleteCountSem

```
#include <trsocket.h>

int                  tfKernelDeleteCountSem
(
ttUserGenericUnionPtr
countingSemaphorePtr
);
```

Function Description
RTOS SPECIFIC

This function is used to remove a counting semaphore.

Parameters

Parameter	Description
countingSemaphorePtr	The counting semaphore to delete

Returns

Value	Meaning
0	Operation completed successfully
-1	An error occurred

tfKernelError

```
#include <trsocket.h>

void                  tfKernelError
(
char *                functionName,
char *                errorMessage
);
```

Function Description
RTOS SPECIFIC

This function is used to report an error and stop the system (Debug Only).

Parameters

Parameter	Description
functionName	The Null terminated string containing the function name where the error occurred
errorMessage	The Null terminated string containing the error message

Returns

Nothing

tfKernelFree

```
#include <trsocket.h>

int                 tfKernelFree
(
void *              memoryBlockPtr
);
```

Function Description
RTOS SPECIFIC

This function is used to free a block of memory back to the RTOS.

Parameters

Parameter	Description
memoryBlockPtr	The pointer to the area of memory to free

Returns

Value	Meaning
0	Operation completed successfully
-1	An error occurred

tfKernelInitialize

```
#include <trsocket.h>
```

```
void                  tfKernelInitialize
(
void
);
```

Function Description
RTOS SPECIFIC

This function is used to initialize the Kernel interface.

Parameters

None

Returns

Nothing

tfKernelInstallsrHandler

```
#include <trsocket.h>

void                  tfKernelInstalIsrHandler
(
ttUserIsrHandlerPtr  handlerPtr,
unsigned long        offset
);
```

Function Description
RTOS SPECIFIC

This function is used to install an Interrupt Service Routine (ISR) handler.

Parameters

Parameter	Description
handlerPtr	The pointer to the function that the ISR should call
offset	Offset into the vector table, where this handler should be installed

Returns

Nothing

tfKernelIsrPostEvent

```
#include <trsocket.h>

void                     tfKernelIsrPostEvent
(
ttUserGenericUnionPtr    eventPtr
);
```

Function Description

RTOS SPECIFIC

This function is used to resume waiting tasks that were waiting on this event. It is called from an ISR.

Parameters

Parameter	Description
eventPtr	The event to post on

Returns

Nothing

tfKernelMalloc

```
#include <trsocket.h>

void *              tfKernelMalloc
(
unsigned            size
);
```

Function Description
RTOS SPECIFIC

This function is used to allocate a block of memory.

Parameters

Parameter	Description
size	The amount of memory to allocate

Returns

A pointer to the beginning of the memory block

tfKernelPendCountSem

```
#include <trsocket.h>

int                     tfKernelPendCountSem
(
ttUserGenericUnionPtr   countingSemaphore
);
```

Function Description
RTOS SPECIFIC

This function is used to wait on a counting semaphore.

Parameters

Parameter	Description
countingSemaphore	The counting semaphore to wait on

Returns

Value	Meaning
0	Operation completed successfully
-1	An error occurred

tfKernelPendEvent

#include <trsocket.h>

```
void                        tfKernelPendEvent
(
ttUserGenericUnionPtr   eventPtr
);
```

Function Description

RTOS SPECIFIC

This function is used to wait on an event from ISR.

Parameters

Parameter	**Description**
eventPtr	The event to wait on

Returns

Nothing

tfKernelPostCountSem

#include <trsocket.h>

```
int                        tfKernelPostCountSem
(
ttUserGenericUnionPtr      countingSemaphore
);
```

Function Description
RTOS SPECIFIC

This function is used for the signal processes waiting on a counting semaphore.

Parameters

Parameter	**Description**
countingSemaphore	The counting semaphore to post to

Returns

Value	**Meaning**
0	Operation completed successfully
-1	An error occurred

tfKernelReleaseCritical

#include <trsocket.h>

```
void          tfKerneReleaseCritical
(
void
);
```

Function Description
RTOS SPECIFIC

This function is used to end a critical section.

Parameters

None

Returns

Nothing

tfKernelSetCritical

#include <trsocket.h>

```
void        tfKernelSetCritical
(
void
);
```

Function Description
RTOS SPECIFIC

This function is used to begin a Critical section

Parameters
None

Returns
Nothing

tfKernelSheapCreate

#include <trsocket.h>

```
ttUser32Bit *     tfKernelSheapCreate
(
void
);
```

Function Description
RTOS SPECIFIC

Normally, the Treck simple heap is implemented as a static array. However, you can dynamically allocate it by defining the macro TM_DYNAMIC_CREATE_SHEAP in your trsystem.h file, in which case you implement **tfKernelSheapCreate** to perform the allocation. The size of the simple heap allocated by this function must be TM_SHEAP_SIZE, and it must start on a 32-bit address boundary.

Parameters
None

Returns
A pointer to the beginning of the Treck simple heap. This address must be 32-bit aligned, and must point to a block of memory which is TM_SHEAP_SIZE bytes in size.

tfKernelTaskPostEvent

#include <trsocket.h>

```
void                     tfKernelIsrPostEvent
(
ttUserGenericUnionPtr    eventPtr
);
```

Function Description
RTOS SPECIFIC

This function is used to resume waiting tasks that were waiting on this event. It is called from a task.

Parameters

Parameter	**Description**
eventPtr	The event to post on

Returns

Nothing

tfKernelTaskYield

#include <trsocket.h>

```
void                           tfKernelTaskYield
(
void
);
```

Function Description
RTOS SPECIFIC

This function is used to yield the CPU. It is called from a task.

Returns
Nothing

tfKernelWarning

#include <trsocket.h>

```
void        tfKernelWarning
(
char *      functionName,
char *      errorMessage
);
```

Function Description
RTOS SPECIFIC

This function is used to report a non fatal-warning. The system should continue normally.

Parameters

Parameter	Description
functionName	The Null terminated string containing the function name where the error occurred
errorMessage	The Null terminated string containing the error message

Returns

Nothing

Compiler Library Replacement Functions

tfMemCpy

```
#include <trsocket.h>

void                tfMemCpy
(
void *              destination,
const void *        source,
unsigned            length
);
```

Function Description

This function is used to copy data (i.e. memcpy)

Parameters

Parameter	Description
destination	Beginning of destination
source	Beginning of source
length	Length of area to copy

Returns

Number of bytes copied

tfMemSet

```
#include <trsocket.h>

void                tfMemSet
(
void *              buffer,
unsigned char       fillCharacter,
unsigned            length
);
```

Function Description

This function is used to fill an area of memory (i.e. set all of the bytes to 0x00). In versions 2.0 and earlier, the order of the *fillCharacter* and *length* parameters was reversed.

Parameters

Parameter	Description
buffer	Beginning of area to fill
fillCharacter	What to fill the area with
length	Length of area to fill

Returns

Number of bytes filled

tfQSort

```
#include <trsocket.h>

void                  tfQSort
(
void *                a,
unsigned int          n,
unsigned int          es,
ttCmpFuncPtr          cmpFuncPtr
);
```

Function Description

tfQsort sorts the array pointed to by *a*. There are *n* elements of size *es* in this array. These elements are sorted according to a comparison function, pointed to by *cmpFuncPtr*. This function is defined as:

```
typedef int (* ttCmpFuncPtr)(const void * parmPtr1,
                             const void * parmPtr2);
```

This function should take the two parameters, *parmPtr1* and *parmPtr2*, and should return 1 if the first element is greater than the second, 0 if they are equal or –1 if the first element is less than the second.

Parameters

Parameter	**Description**
a	Pointer to the array to be sorted
n	Number of elements in the above array
es	Size of each element in the array
cmpFuncPtr	Pointer to the function used to compare array elements

Returns

Nothing

tfSPrintF

```
int                 tfSPrintF
(
char *              buffer,
const char*         format,
...
);
```

Function Description

This function writes formatted data to an array using the following format specifiers:

Examples

call:	tfSPrintF(myBuffer, “This is example number %d”, 1);
result:	“This is example number 1”
call:	tfSPrintf(myBuffer, “This is %s number %d”, “example”, 2);
output:	“This is example number 2”

Code	**Description**
%c	Print a character
%d	Print a signed decimal integer
%e	Scientific notation: prints a lower case e
%E	Scientific notation: prints an upper case E
%f	Decimal floating point
%g	Choose between %e or %f depending on which is shorter
%G	Choose between %E or %F depending on which is shorter
%i	Print a signed decimal integer
%n	The value to which %n corresponds is a pointer to a variable representing the number of characters successfully output.
%o	Print an unsigned octal
%p	Show a pointer
%s	Print a string (up to the first null character encountered)
%u	Print an unsigned decimal integer
%x	Print an unsigned hex integer (lowercase)
%X	Print an unsigned hex integer (uppercase)
%%	Print a percent sign

Parameters

Parameter	**Description**
buffer	The string to be written
format	The string containing the format specifiers

Returns

A number representing the number of characters put into an array.

tfSScanF

```
int                   tfSScanF
(
const char*           buffer,
const char*           format,
...
);
```

Function Description

This function is used to parse a string; it reads data from *buffer* into locations designated by arguments provided by the user.

Code	Description
%c	Read a character. White space characters are not read unless %c is used
%d	Read a decimal number
%e	Read a floating-point number. It can have a + or – sign. It can also be a series of digits with a decimal point and an exponent (e or E) followed by a signed or unsigned integer
%f	Read a floating-point number. It can have a + or – sign. It can also be a series of digits with a decimal point and an exponent (e or E) followed by a signed or unsigned integer
%g	Read a floating-point number. It can have a + or – sign. It can also be a series of digits with a decimal point and an exponent (e or E) followed by a signed or unsigned integer
%i	Read a decimal integer
%n	Use an integer number to express the number of bytes successfully read
%o	Read an octal number
%p	Read a pointer (platform specific)
%s	Read a string
%u	Read an unsigned decimal integer
%x	Read a hexadecimal number
%[]	Read a set of characters

Parameters

Parameter	**Description**
buffer	The string to be parsed
format	The string containing the format specifiers dictating which type of data is to be read

Returns

The number of input fields that were both formatted and assigned a value. If an error occurs, **tfSScanF** will return End of File (EOF)

tfStrCat

```
int                 tfStrCat
(
char*               stringD,
const char*         stringS
);
```

Function Description

This function takes the content of one string and appends it to another. The user of this function must make certain the contents of both strings will fit in the size allotted for the fist. This function accomplishes this by overwriting the null character at the end of the first string with the first character in the second.

Parameters

Parameter	**Description**
stringD	The string to be appended
strings	The string appended to *stringD*

Returns

stringD

tfStrChr

```
int                 tfStrChr
(
const char*         stringPtr,
int                 character
);
```

Function Description

This function searches for the low-ordered byte of an integer in a string.

Parameters

Parameter	**Description**
stringPtr	Pointer to the string being searched
character	The integer containing the byte for which tfStrChar is searching

Returns

A pointer to the first match. If there are no matches this function returns a null pointer.

tfStrCmp

```
int                 tfStrCmp
(
const char*         stringPtr1,
const char*         stringPtr2
);
```

Function Description

This function alphabetically compares two strings.

Parameters

Parameter	Description
stringPtr1	A string
stringPtr2	A string

Returns

Value	Meaning
Less than Zero	stringPtr1 is less than stringPtr2
Zero	stringPtr1 is equal to stringPtr2
Greater than Zero	stringPtr1 is greater than stringPtr2

tfStrCpy

```
int                  tfStrCpy
(
char*                destinationPtr,
const char*          sourcePtr
);
```

Function Description

This function copies the contents of one array (pointed to by sourceptr) and places it in the array pointed to by destinationPtr. The array pointed to by destinationptr must be large enough to hold the contents of the array pointed to by sourcePtr. If the two arrays overlap, the behavior is undefined.

Parameters

Parameter	**Description**
destinationPtr	The array to which data is copied
sourcePtr	The array from which data is copied

Returns

destinationPtr

tfStrCSpn

```
int                 tfStrCSpn
(
char*               destinationPtr,
const char*         sourcePtr
);
```

Function Description

This function searches for any characters in sourcePtr that also appear in the destinationPtr and returns the offset of the first matched character in destinationPtr.

Parameters

Parameter	**Description**
destinationPtr	The string to be searched
sourcePtr	A list of the characters to be searched for in destinationPtr

Returns

Offset of the first matched character in destinationPtr

tfStrError

```
int                  tfStrError
(
int                  errorcode
);
```

Function Description

This function returns the error message correlated with *errnum*.

Parameters

Parameter	**Description**
errorcode	An error value

Returns

error message

tfStrLen

```
int                 tfStrLen
(
const char*         strPtr
);
```

Function Description

This function returns the length of string pointed to by strPtr.

Parameters

Parameter	Description
strPtr	Pointer to the string

Returns

The length of the string pointed to by strPtr

Application Reference

Application Reference Function List

PING API

tfPingClose
tfPingGetStatistics
tfPingOpenStart

DNS Resolver

tfDnsInit
tfDnsGetHostAddr
tfDnsGetHostByName
tfDnsGetMailHost
tfDnsGetNextMailHost
tfDnsSetOption
tfDnsSetServer

FTPD API

tfFtpdUserExecute
tfFtpdUserStart
tfFtpdUserStop

FTP

tfFtpAbor
tfFtpAppe
tfFtpCdup
tfFtpClose
tfFtpConnect
tfFtpCwd
tfFtpDele
tfFtpDirList
tfFtpFreeSession
tfFtpGetReplyText
tfFtpHelp
tfFtpLogin
tfFtpMkd
tfFtpNewSession
tfFtpNoop
tfFtpPort
tfFtpPwd
tfFtpQuit
tfFtpRein
tfFtpRename
tfFtpRetr
tfFtpRmd
tfFtpStor
tfFtpTurnPasv
tfFtpSyst
tfFtpType
tfFtpUserExecute

TFTP

tfTftpGet
tfTftpInit
tfTftpPut
tfTftpSetTimeout
tfTftpUserExecute

TFTPD

tfTftpdInit
tfTftpdUserExecute
tfTftpdUserStart
tfTftpdUserStop

File System Interface

tfFSChangeDir
tfFSChangeParentDir
tfFSCloseDir
tfFSCloseFile
tfFSDeleteFile
tfFSGetNextDirEntry
tfFSGetUniqueFileName
tfFSGetWorkingDir
tfFSMakeDir
tfFSOpenDir
tfFSOpenFile
tfFSReadFile
tfFSReadFileRecord
tfFSRemoveDir
tfFSRenameFile
tfFSStructureMount
tfFSSystem
tfFSUserAllowed
tfFSUserLogin
tfFSUserLogout
tfFSWriteFile
tfFSWriteFileRecord

Telnet Daemon

tfTeldClosed
tfTeldIncoming
tfTeldOpened
tfTeldUserClose
tfTeldUserExecute
tfTeldUserSend
tfTeldUserStart
tfTeldUserStop

Turbo Treck Test Suite

tfTestTreck
tfTestUserExecute

PING Application Program Interface

Description

Three calls are provided in the PING Application Program Interface. First, the user calls **tfPingOpenStart**. This opens an ICMP socket and sends periodic PING echo request packets. **tfPingOpenStart** is passed a pointer to a character array containing a dotted decimal IP address representation of the remote host, the interval in seconds between PING echo requests, the user data length of the PING echo requests, and a pointer to a call back function to be called upon reception of a PING echo reply, or a network ICMP error message. If successful, **tfPingOpenStart** returns a socket descriptor.

The system will keep sending PING echo requests from the timer until the user calls **tfPingClose**, passing the socket descriptor as returned by **tfPingOpenStart** as the parameter. Prior to calling **tfPingClose**, the user can call **tfPingGetStatistics** to retrieve results and statistics of the PING connection such as number of packets transmitted, number of packets received, round trip time of the last PING echo request, error code as given by a received network ICMP error message, etc.. The user passes the socket descriptor returned by **tfPingOpenStart** as the first parameter to **tfPingGetStatistics**, and a pointer to a *ttPingInfo* structure as the second parameter, where the system will copy the results and statistics of the PING connection if the **tfPingGetStatistics** returns with no error. The user can be notified of incoming PING echo replies or incoming ICMP error messages if he specifies a non-null call back function pointer as the last parameter to **tfPingOpenStart**. In that case, the call back function is called every time a PING echo reply or an ICMP error message is received. The call back function takes one parameter- the socket descriptor as returned by the **tfPingOpenStart**. The user can then retrieve the PING connection information by calling **tfPingGetStatistics**.

tfPingClose

```
#include <trsocket.h>

int                 tfPingClose
(
int                 socketDescriptor
);
```

Function Description

This function stops the sending of any PING echo requests and closes the ICMP socket that had been opened via **tfPingOpenStart**.

Parameters

Parameter	**Description**
socketDescriptor	An ICMP PING socket descriptor as returned by **tfPingOpenStart**

Returns

Value	**Meaning**
0	Success
-1	An error occurred

If **tfPingClose** fails, the associated error code can be retrieved using **tfGetSocketError** (*socketDescriptor*):

TM_EBADF	*socketDescriptor* is not a valid descriptor

tfPingGetStatistics

```
#include <trsocket.h>

int                 tfPingGetStatistics
(
int                 socketDescriptor,
ttPingInfoPtr       pingInfoPtr
);
```

Function Description

This function gets Ping statistics in the *ttPingInfo* structure that *pingInfoPtr* points to. *socketDescriptor* should match the socket descriptor returned by a call to **tfPingOpenStart**. *pingInfoPtr* must point to a *ttPingInfo* structure allocated by the user.

Parameters

Parameter	Description
socketDescriptor	The socket descriptor as returned by a previous call to **tfPingOpenStart**.
pingInfoPtr	The pointer to an empty structure where the results of the PING connection will be copied upon success of the call. (See below for details.)

ttPingInfo Data structure:

Field	data type	Description
pgiTransmitted	unsigned long	Number of transmitted PING echo request packets so far
pgiReceived	unsigned long	Number of received PING echo reply packets so far (not including duplicates)
pgiDuplicated	unsigned long	Number of duplicated received PING echo reply packets so far.
pgiLastRtt	*unsigned long*	Round trip time in milliseconds of the last PING request/reply.
pgiMaxRtt	*unsigned long*	Maximum round trip time in milliseconds of the PING request/reply packets.

pgiMinRtt	unsigned long	Minimum round trip time in milliseconds of the PING request/ reply packets.
pgiAvrRtt	unsigned long	Average round trip time in milliseconds of the PING request/ reply packets.
pgiSumRtt	unsigned long	Sum of all round trip times in milliseconds of the PING request/reply packets.
pgiSendErrorCode any.	int	PING send request error code if
pgiRecvErrorCode	int	PING recv error code if any (including ICMP error from the network).

Returns

Value	**Meaning**
0	Success
-1	An error occurred

If **tfPingGetStatistics** fails, the associated error code can be retrieved using **tfGetSocketError** (*socketDescriptor*):

TM_EBADF	*socketDescriptor* is not a valid descriptor.
TM_EINVAL	*pingInfoPtr* is a NULL pointer
TM_EINVAL	*socketDescriptor* was not opened with **tfPingOpenStart**

tfPingOpenStart

```
#include <trsocket.h>

int                 tfPingOpenStart
(
char *              remoteHostNamePtr,
int                 pingInterval,
int                 pingDataLength,
ttPingCBFuncPtr     pingUserCBFuncPtr
);
```

Function Description

This function opens an ICMP socket and starts sending PING echo requests to a remote host as specified by the *remoteHostName* parameter. PING echo requests are sent every *pingInterval* seconds. The PING length (not including IP and ICMP headers) is given by the *pingDataLength* parameter. If the *pingUserCBFuncPtr* parameter is non-null, the function it points to is called for each received PING echo reply or ICMP error message, with the socket descriptor returned by **tfPingOpenStart** passed as a parameter. To get the PING connection results and statistics, the user must call **tfPingGetStatistics**. To stop the system from sending PING echo requests and to close the ICMP socket, the user must call **tfPingClose**.

Parameters

Parameter	**Description**
remoteHostNamePtr	Pointer to character array containing a dotted decimal IP address.
pingInterval	Interval in seconds between PING echo requests. If set to zero, defaults to 1 second.
pingDataLength	User Data length of the PING echo request. If set to zero, defaults to 56 bytes. If set to a value between 1, and 3, defaults to 4 bytes.
pingUserCBFuncPtr	Pointer to a user function to be called upon receiving a network PING echo reply, or an ICMP error message, with the socket descriptor as returned by **tfPingOpenStart** passed as a parameter. Can be set to null function pointer if the user does not wish to be notified of incoming network traffic.

Returns

New ICMP Socket Descriptor or TM_SOCKET_ERROR (-1) on error

If **tfPingOpenStart** fails, the errorCode can be retrieved with **tfGetSocketError** (*TM_SOCKET_ERROR):*

TM_EINVAL	*remoteHostNamePtr* was a null pointer
TM_EINVAL	*pingInterva*l was negative
TM_EINVAL	*pingDataLength* was negative of bigger than 65595, maximum value allowed by the IP protocol.
TM_ENOBUFS	There was insufficient user memory available to complete the operation.
TM_EMSGSIZE	*pingDataLength* exceeds socket send queue limit, or *pingDataLength* exceeds the IP MTU, and fragmentation is not allowed.
TM_EHOSTUNREACH	No route to remote host

Example with a non null function pointer

```
#include <trsocket.h>


void tfPingUserCB (int socketDescriptor);

..
socketDescriptor =  tfPingOpenStart (“192.168.0.2”, 0, 0,
tfPingUserCB);
```

Example with a null function pointer

```
#include <trsocket.h>

..
socketDescriptor =
                 tfPingOpenStart (“127.0.0.1”,
                                 0,
                                 0,
ttPingCBFuncPtr)0);
```

DNS Resolver

Description

The DNS Resolver allows a user to translate a hostname to an IP address, an IP address to a hostname, and to retrieve information about a host's mail exchangers (which is necessary to send SMTP e-mail). The user API consists of three functions that are called *before* a DNS operation is performed and then three functions to perform these operations.

Initialization functions

tfDnsInit
Initializes the DNS service and should be called before any other API call is made.

tfDnsSetServer
Specifies the DNS server(s) to retrieve hostname information from.

tfDnsSetOption
Sets various options regarding the operation of DNS: timeout lengths, number of retries, etc.

User Interface

tfDnsGetHostByName
Translates the given hostname to its corresponding IP address (e.g., "elmic.com" translates to 208.229.201.1).

tfDnsGetHostByAddr
Translates the given IP address to its corresponding hostname (also called a reverse lookup).

tfDnsGetMailHost, tfDnsGetNextMailHost
Returns the first MX record in the list (**tfDnsGetMailHost**) and then retrieves any records that follow (**tfDnsGetNextMailHost**). Please see below for more information on using MX records.

Any of the user interface calls may return an error from theTurbo Treck stack or from the DNS server. The errors from the Turbo Treck stack are the normal socket errors (e.g., TM_ENOMEM, TM_EHOSTUNREACH). Errors returned from the DNS server are outlined below (derived from RFC-1035):

TM_DNS_EFORMAT

Format error – The name server was unable to interpret the query.

TM_DNS_ESERVER

Server Failure – The name server was unable to process this query due to a problem with the name server.

TM_DNS_ENAME_ERROR

Name Error – Meaningful only for responses from an authoritative name server, this code signifies that the domain name referenced in the query does not exist.

TM_DNS_ENOT_IMPLEM

Not Implemented – The name server does not support the requested kind of query.

TM_DNS_EREFUSED

Refused – The name server refuses to perform the specified operation for policy reasons. For example, a name server may not wish to provide the information to the particular requester, or a name server may not wish to perform a particular operation (e.g., zone transfer) for particular data.

TM_DNS_EANSWER

No answer received from name server (i.e., response packet received, but it did not contain the answer to our query).

Non-Blocking Mode

The call to perform a DNS function will simply block until the operation is complete (or an error is received). However, with non-blocking mode it is necessary to poll the DNS Resolver to determine if the operation has completed. If an operation is still in progress, the call will return `TM_EWOULDBLOCK`. If an error code other than `TM_EWOULDBLOCK` is received, the operation has completed.

Mail Exchanger (MX) Records

Each hostname contains a list of machines that are willing to accept mail destined for that hostname. This is necessary should one of the machines be unable to receive mail. Each one of the hosts in this list contains a *preference* value that indicates the order that these hosts should be used. An e-mail program should first attempt delivery to the hostname with the lowest preference value, and if unable to connect, to attempt delivery to the hostname with the next highest preference. For instance, “elmic.com” may have the following MX entries:

```
      mail1.elmic.com   10    208.229.201.1
      mail2.elmic.com   20    208.229.201.2
      mail3.elmic.com   30    208.229.201.3
```

Mail should first be sent to `mail1.treck.com`. If that fails it should be sent to `mail2.treck.com` and so on. This behavior can be emulated with the Treck resolver by first calling **tfDnsGetMailHost** and then repeatedly calling **tfDnsGetNextMailHost**, which will return `TM_EANSWER` when no more records are available. With each call to **tfDnsGetNextMailHost**, the IP address and preference value of the *previous* lookup must be included. For instance, the following code retrieves all three entries in the example above:

```
unsigned short  mxPref1, mxPref2, mxPref3;
ttUserIpAddress ipAddress1, ipAddress2, ipAddress3;

/* Get the first MX record (mail1.elmic.com) */
errorCode = tfDnsGetMailHost(“treck.com”,
                             &ipAddress1,
                             &mxPref1);
/* Get the second MX record (mail2.elmic.com) */
errorCode = tfDnsGetNextMailHost(“elmic.com”,
                                 ipAddress1,
                                 mxPref1,
                                 &ipAddress2,
                                 &mxPref2);
/*
 * Get the third MX record (mail3.elmic.com)
 * If tfDnsGetNextMailHost were called with ipAddress3
 * and mxPref33, it would return TM_EANSWER.
 */
errorCode = tfDnsGetNextMailHost(“elmic.com”,
                                 ipAddress2,
                                 mxPref3,
                                 &ipAddress3,
                                 &mxPref3);
```

tfDnsInit

```
int                 tfDnsInit
(
int                 blockingMode
);
```

Function Description

This function initializes the DNS resolver service. This should be called once and only once when the system is started.

Parameters

Parameter	**Description**
blockingMode	Specifies whether the resolver should operate in blocking or non-blocking mode (TM_BLOCKING_ON or TM_BLOCKING_OFF)

Returns

Value	**Meaning**
TM_EINVAL	*blockingMode* not set to either TM_BLOCKING_ON or TM_BLOCKING_OFF
TM_EALREADY	The DNS resolver has already been started.
TM_ENOERROR	Resolver started successfully.

tfDnsGetHostAddr

```
int                 tfDnsGetHostAddr
(
ttUserIpAddress     serverIpAddr,
char *              hostnameStr,
int                 hostnameStrLength
);
```

Function Description

This function retrieves the hostname associated with the given IP address (a "reverse DNS lookup").

Parameters

Parameter	**Description**
serverIpAddr	IP address to retrieve the hostname for.
hostnameStr	Buffer to place the retrieved hostname in.
hostnameStrLength	Size of the above buffer.

Returns

Value	**Meaning**
TM_EINVAL	Invalid host name string or IP address pointer.
TM_EWOULDBLOCK	DNS lookup in progress. The user should continue to call *tfDnsGetHostAddr* with the same parameters until it returns a value other than TM_EWOULDBLOCK. Only returned when operating in non-blocking mode.
TM_ENOERROR	DNS lookup successful, hostname stored in *hostnameStr*, length stored in **hostnameStrLength.*

tfDnsGetHostByName

```
int                 tfDnsGetHostName
(
const char *        hostnameStr,
ttUserIpAddressPtr  ipAddressPtr
);
```

Function Description

This function retrieves the IP address associated with the given hostname.

Parameters

Parameter	**Description**
hostnameStr	Hostname to resolve.
ipAddressPtr	Set to the IP address of the host.

Returns

Value	**Meaning**
TM_EINVAL	Invalid host name string or IP address pointer.
TM_EWOULDBLOCK	DNS lookup in progress. The user should continue to call *tfDnsGetHostName* with the same parameters until it returns a value other than **TM_EWOULDBLOCK**. Only returned in non-blocking mode.
TM_ENOERROR	DNS lookup successful, IP address stored in **ipAddressPtr.*

tfDnsGetMailHost

```
int                    tfDnsGetMailHost
(
const char *           hostnameStr,
ttUserIpAddressPtr     ipAddressPtr,
unsigned short *       mxPrefPtr
);
```

Function Description

This function retrieves the IP address of the first MX record for this hostname.

Parameters

Parameter	Description
hostnameStr	Hostname to resolve.
ipAddressPtr	Set to the IP address of the mail host.
mxPrefPtr	Set to the preference value of this mail host.

Returns

Value	Meaning
TM_EINVAL	Invalid host name string or IP address pointer.
TM_EWOULDBLOCK	DNS lookup in progress. The user should continue to call *tfDnsGetHostName* with the same parameters until it returns a value other than **TM_EWOULDBLOCK**.
TM_ENOERROR	DNS lookup successful, IP address stored in **ipAddressPtr.*

tfDnsGetNextMailHost

```
int                   tfDnsGetNextMailHost
(
const char *          hostnameStr,
ttUserIpAddress       lastIpAddress,
unsigned short        lastPreference,
ttUserIpAddressPtr    ipAddressPtr,
unsigned short *      mxPrefPtr
);
```

Function Description

This function returns the IP address for the next mail exchanger for this hostname. The record that immediately follows the record for *lastIpAddress/lastPreference* will be retrieved.

Parameters

Parameter	**Description**
hostnameStr	Hostname to resolve.
lastIpAddress	IP address of the last retrieved mail host for this host.
lastPreference	Preference of the last retrieved mail host for this host.
ipAddressPtr	Set to the IP address of the host.

Returns

Value	**Meaning**
TM_EINVAL	Invalid host name string or IP address pointer.
TM_EWOULDBLOCK	DNS lookup in progress. The user should continue to call *tfDnsGetHostName* with the same parameters until it returns a value other than **TM_EWOULDBLOCK**.
TM_ENOERROR	DNS lookup successful, IP address stored in **ipAddressPtr.*

tfDnsSetOption

```
int                 tfDnsSetOption
(
int                 optionType,
int                 optionValue
);
```

Function Description

This function sets various DNS options which are outlined below:

Option type	**Description**
TM_DNS_OPTION_RETRIES	Maximum number of times of retransmit a DNS request.
TM_DNS_OPTION_CACHE_SIZE	Maximum number of entries in the DNS cache. Must be greater than zero.
TM_DNS_OPTION_TIMEOUT	Amount of time (in seconds) to wait before retransmitting a DNS request.

Parameters

Parameter	**Description**
optionType	See above
optionValue	Value for above option.

Returns

Value	**Meaning**
TM_EINVAL	Invalid value for above option.
TM_ENOPROTOOPT	Option not supported (not in above list).
TM_ENOERROR	Option set successfully.

tfDnsSetServer

```
int                 tfDnsSetServer
(
ttUserIpAddress     serverIpAddr,
int                 serverNumber
);
```

Function Description

This function sets the address of the primary and secondary DNS server. To set the primary DNS server *serverNumber* should be set to **TM_DNS_PRI_SERVER**; for the secondary server it should be set to **TM_DNS_SEC_SERVER**. To remove a previously set entry, set *serverIpAddr* to zero.

Parameters

Parameter	**Description**
serverIpAddr	IP address of the DNS server
serverNumber	Primary or secondary server

Returns

Value	**Meaning**
TM_EINVAL	*serverNumber* is not **TM_DNS_PRI_SERVER** or **TM_DNS_SEC_SERVER**.
TM_ENOERROR	DNS server set successfully.

FTPD Application Program Interface

Description

The FTPD Application Program Interface allows the user to run an FTP server. It consists of two parts:

User interface
The user interface allows the user to start/stop the FTP server to allow/stop remote FTP clients to connect and exchange files with the host.

File system interface
The file system interface allows the FTP server to interact with the operating system's file system to store and retrieve files for example.

User Interface

Three calls are provided in the FTPD User Interface.

1. tfFtpdUserStart

The user calls **tfFtpdUserStart**, to open an FTP server socket and start listening for incoming connections. **tfFtpdUserStart** can be either blocking or non blocking, as specified by its last parameter.

Blocking Mode

In blocking mode, **tfFtpdUserStart** should be called from a task. **tfFtpdUserStart** will not return unless an error occurs, and will block and wait for incoming connections, and execute the FTP server code in the context of the calling task. Choose the blocking mode, if you are using an RTOS/ Kernel.

Non-Blocking Mode

In non-blocking mode, **tfFtpdUserStart** will return immediately after listening for incoming connections. It is the user's responsibility to then call **tfFtpdUserExecute** periodically to execute the FTP server code. Choose the non-blocking mode if you do not have an RTOS/Kernel.

2. tfFtpdUserExecute

If the user had called **tfFtpdUserStart** in non-blocking mode, then the user needs to call **tfFtpdUserExecute** periodically. If the user had called **tfFtpdUserStart** in blocking mode, then there is no need to call **tfFtpdUserExecute**.

3. tfFtpdUserStop

The user calls **tfFtpdUserStop** to close the FTP server socket and kill all existing FTP connections.

File System Interface from the FTP server

Entry points from the FTP server to the file system:

tfFSChangeDir	Change current working directory.
tfFSChangeParentDir	Change current working directory to parent Directory.
tfFSCloseDir	Close a directory that we had opened earlier.
tfFSCloseFile	Close a file.
tfFSDeleteFile	Delete a file.
tfFSGetNextDirEntry	Get the next directory entry in the directory open with **tfFSOpenDir**, either a long listing of the directory entry (including volumes, sub directories, and file names), or a short listing of the directory (file name only), depending on how the directory was open.
tfFSGetUniqueFileName	Given a file name, return a unique file name in the current directory (i.e, if the file name already exists, make up a new name that is unique in the current directory.)
tfFSGetWorkingDir	Get user working directory.
tfFSMakeDir	Create specified directory.
tfFSOpenDir	Open specified directory, or directory corresponding to a specified pattern to allow getting a long or short listing of the directory or of the directory entries matching the specified pattern.
tfFSOpenFile	Open a file (creating it if it does not exist), for read, write, or append, specifying type (ASCII, or binary), structure (stream, or record).
tfFSReadFile	Read n bytes from a file into a buffer.
tfFSReadFileRecord	Read a record from a file up to n bytes. Indicates whether EOR has been reached.

tfFSRemoveDir	Remove specified directory
tfFSRenameFile	Rename a file.
tfFSStructureMount	Mount the user to a new file system data structure.
tfFSSystem	Return the system name.
tfFSUserAllowed	Indicates whether a specified user is allowed on the system.
tfFSUserLogin	Login a user if password is valid.
tfFSUserLogout	Logout a user.
tfFSWriteFile	Write some bytes from a buffer to a file.
tfFSWriteFileRecord	Write a record from a buffer to a file.

Note: File system calls for the FTP server can be found in the File System Section of this manual.

tfFtpdUserExecute

#include <trsocket.h>

```
int    tfFtpdUserExecute
(
void
);
```

Function description

This function executes the Ftp server main loop. This call is valid only if **tfFtpdUserStart** has been called in non-blocking mode, and **tfFtpdUserExecute** is not currently executing.

Parameters

None

Returns

Value	**Meaning**
0	Success
TM_EPERM	Ftp server currently executing, or has not been started, or has been stopped.

tfFtpdUserStart

```
#include <trsocket.h>

int                  tfFtpdUserStart
(
int                  fileFlags,
int                  maxConnections,
int                  maxBackLog,
int                  idleTimeout,
int                  blockingState
);
```

Function Description

This function opens an FTP server socket and starts listening for incoming connections. **tfFtpdUserStart** can be either blocking or non-blocking as specified by its *blockingState* parameter.

Blocking Mode

In blocking mode, **tfFtpdUserStart** should be called from a task. **tfFtpdUserStart** will not return unless an error occurs. It will block and wait for incoming connections, and execute the FTP server code in the context of the calling task. Choose the blocking mode if you are using an RTOS/Kernel.

Non-Blocking Mode

In non-blocking mode, **tfFtpdUserStart** will return immediately after listening for incoming connections. It is the user's responsibility to then call **tfFtpdUserExecute** periodically to execute the FTP server code. Choose the non-blocking mode if you do not have an RTOS/Kernel.

Parameters

Parameter	**Description**
fileFlags	Indicates which FTP file commands are supported by the file system. It is the result of ORing together the flags (described below) corresponding to the FTP commands supported by the file system.
maxConnections	Maximum number of concurrent accepted incoming FTP connections allowed. If zero, then the FTP server will accept as many connections as there are available sockets.
maxBackLog	Maximum number of concurrent pending (before being accepted) incoming FTP connections allowed.
idleTimeout	Amount of time in seconds that a connection can sit idle before the server closes that connection. The idle Timeout value cannot be less than 300 seconds (5 minutes).
blockingState	TM_BLOCKING_ON for blocking mode, TM_BLOCKING_OFF for non-blocking mode.

File system flags:

File system flags	**Description**
TM_FS_CWD_FLAG	Supports change working directory.
TM_FS_SMNT_FLAG	Supports structure mount.
TM_FS_RETR_FLAG	Supports reading from a file.
TM_FS_STOR_FLAG	Supports writing to a file.
TM_FS_STORU_FLAG	Supports writing to a file, making up a new name, if the file name already exists.
TM_FS_APPEND_FLAG	Supports append to a file.
TM_FS_RENAME_FLAG	Supports renaming of file name.
TM_FS_DELETE_FLAG	Supports deletion of file.
TM_FS_RMD_FLAG	Supports removing directory.
TM_FS_MKD_FLAG	Supports making directory.
TM_FS_PWD_FLAG	Supports retrieving the current working directory.
TM_FS_LIST_FLAG	Supports long listing of directory (file

	names, volume, and directories)
TM_FS_NLST _FLAG	Supports short listing of directory (file names only)
TM_FS_CR_LF_FLAG	The file system end of line is CR, LF
TM_FS_RECORD_FLAG	The file system supports record structures. If this flag is set the FTP server will interpret the FTP record bytes if the FTP client transfers data with record structure.
TM_FS_ALLCMND_MASK	ORing of all above command flags.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	Incorrect file flag specified.
TM_EINVAL	maxConnections is either negative, or if non zero exceeds or equals the current number of available FTP connections.*
TM_EINVAL	maxBackLog is either negative or null, or exceeds or equals the current number of available FTP connections.*
TM_EINVAL	The idle timeout is less than 300 seconds.
TM_EINVAL	blockingState is neither TM_BLOCKING_ON, nor TM_BLOCKING_OFF.
TM_EALREADY	**tfFtpdUserStart** has already been called.
TM_EMFILE	No more socket available to open the FTPD listening socket.
TM_ENOBUFS	Insufficient user memory available to complete the operation.
TM_EADDRINUSE	The FTP server port is already in use.
TM_ENOMEM	Could not obtain a counting semaphore to be used for blocking the FTP server (blocking mode only).

*The number of available connections is computed by figuring out the number of unused sockets in the system, subtracting one for the FTP listening socket, one for a transient listening socket for a passive data connection, one to send a 421 reply error message when we reach the maximum number of available connections, and dividing by 2 to allow for 2 sockets (one control socket, and one data socket) per connection:
(numberUnusedSockets – 3) / 2

tfFtpdUserStop

```
#include <trsocket.h>

int                 tfFtpdUserStop
(
void
);
```

Function description

This function stops execution of the FTP server, by means of closing the listening socket and killing all existing connections.

Parameters

None

Returns

Value	Meaning
0	Success
TM_EALREADY	The FTP server has already been stopped.

FTP Client Application Program Interface

Description

The FTP Client Application Program Interface allows the user to interact with a remote FTP server: sending and receiving data and performing various file maintenance functions on the remote filesystem. It consists primarily of two parts:

User interface:	The user interface allows the user to send commands to the remote FTP server.
File system interface:	The file system interface allows the FTP client to interact with the operating system's file system to store and retrieve files for example.

User Interface

There are two types of calls provided in the FTP Client User Interface. The first of these is to create and end an FTP session. A session can be composed of multiple consecutive connects to various servers. These calls include **tfFtpNewSession** and **tfFtpFreeSession**. A session can be called in either blocking mode or non-blocking mode.

Blocking Mode

In this mode, each FTP command will return only once the operation has completed and will block until completed. Choose this mode if you are using an RTOS/Kernel.

Non-Blocking Mode

In this mode, each FTP command returns immediately after beginning the operation. After each initial command is made, it is necessary for the user to call **tfFtpUserExecute** until the command has completed, which is indicated by **tfFtpUserExecute** returning a value other than TM_EWOULDBLOCK.

The next type of call is to send various commands to the FTP server. These commands perform various file operations such as send or receive, delete and create directory. These calls are composed of routines such as **tfFtpConnect, tfFtpStor** and **tfFtpDirList**.

As an example of the FTP client user interface, suppose we have an embedded device that wishes to upload a log file to a main server, and also to retrieve the most recent configuration settings. See the example code that would perform these functions on the following page.

```
ttUserFtpHandle ftpHandle;
int             errorCode;

/*
  *  Create  a  new  FTP  client  session,  which  includes
 * logging into the local filesystem.
 */
ftpHandle = tfFtpNewSession(0,
                            TM_BLOCKING_ON,
                            “fsusername”,
                            “fspassword”);

/* Connect to the FTP server */
errorCode = tfFtpConnect(ftpHandle,”10.129.5.10”);

/* Login to the server */
errorCode = tfFtpLogin(ftpHandle,
                       “ftpuser”,
                       “ftppass”,
                       “ftpacct”);

/* Transmit daily log file to server */
errorCode = tfFtpStor(ftpHandle, 0,
                      “/home/treck/100999.log”,
                      “logfile”);

/*
 *  Retrieve  most  recent  configuration  file  from
 * server.
 */
errorCode = tfFtpRetr(ftpHandle, 0,
                      “/home/treck/device.cfg”,
                      “configfile”);

/* Close FTP connection */
errorCode = tfFtpQuit(ftpHandle);

/* End FTP session */
errorCode = tfFtpFreeSession(ftpHandle);
```

File System Interface

The user must provide this interface to the file system. These functions, outlined below, allow the FTP client to access the local file system. This interface is a subset of that required by the FTP server (FTPD) and the same routines may be used for both the FTP client and server.

Entry points from the FTP client to the file system:

tfFSCloseDir	Close a directory that we had opened earlier
tfFSCloseFile	Close a file
tfFSOpenDir	Open specified directory, or directory corresponding to a specified pattern to allow getting a long or short listing of the directory or of the directory entries matching the specified pattern
tfFSOpenFile	Open a file (creating it if it does not exist), for read, write or append, specifying type (ascii, or binary), structure (stream, or record)
tfFSReadFile	Read n bytes from a file into a buffer
tfFSUserLogin	Login a user if password is valid.
tfFSUserLogout	Logout a user
tfFSWriteFile	Write some bytes from a buffer to a file

Note: File system calls for the FTP client can be found in the File System Section of this manual.

FTP Client API Summary

tfFtpNewSession	Creates a new FTP session
tfFtpFreeSession	Frees a FTP session
tfFtpClose	Closes a FTP connection; used if **tfFtpQuit** cannot close the connection properly.
tfFtpConnect	Connects to a remote FTP server
tfFtpLogin	Log on and authenticate to a FTP server
tfFtpCwd	Changes current remote directory
tfFtpCdup	Changes to the current directory's parent directory.
tfFtpQuit	Ends current FTP connection.
tfFtpRein	Resets current FTP connection
tfFtpType	Sets the current transfer type (ASCII or binary)
tfFtpRetr	Retrieves a file from the remote file system.
tfFtpStor	Stores a file on the remote file system.
tfFtpAppe	Appends a file to a file on the remote file system.
tfFtpRename	Rename a remote file
tfFtpAbor	Aborts the transfer in progress.
tfFtpDele	Deletes a remote file
tfFtpRmd	Removes a remote directory
tfFtpMkd	Creates a directory
tfFtpPwd	Retrieves the present working directory
tfFtpDirList	Retrieves a directory listing
tfFtpSyst	Returns information about the system hosting the remote file system
tfFtpHelp	Retrieves help from the FTP server about a command
tfFtpMode	Sets the FTP transfer mode.
tfFtpNoop	Does nothing; used to keep an idle connection open
tfFtpPort	Sets the port used for incoming data connections
tfFtpGetReplyText	Returns the full text reply from the last executed command
tfFtpUserExecute	Called only in non-blocking mode. Should be called periodically to allow the FTP client to continue to process requests.

Return Codes

All of the functions in the FTP Client API return standard error codes. In cases where the call itself succeeded, but the FTP server returned an error, a different, but mutually exclusive, set of error codes are used. These constants are included with each of the user interface calls below and all begin with TM_FTP. For further information on the meaning of these codes, please see RFC 640.

tfFtpAbor

```
#include <trsocket.h>

int                 tfFtpAbor
(
ttUserFtpHandle     ftpSessionPtr
);
```

Function description

This function aborts the current file transfer in process.

Parameters

Parameter	Description
ftpSessionPtr	FTP session handle

Returns

Value	Meaning
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_ENOTLOGIN	User is not currently logged in.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.

tfFtpAppe

```
#include <trsocket.h>

int                 tfFtpRetr
(
ttUserFtpHandle     ftpUserHandleA,
ttUserFtpHandle     ftpUserHandleB,
char      TM_FAR *  fromFileNamePtr,
char      TM_FAR *  toFileNamePtr
);
```

Function description

This function is used to append data to a file on remote FTP server (corresponding to ftpUserHandleA). This function is implemented for both passive mode and normal mode, and also works with both one FTP server model and two FTP servers model.

- One server model:
  If the *ftpUserHandleB* parameter is NULL, then the one-server model is used, and the operation could be either in server active mode or server passive mode. By default the server is in active mode. The user needs to call **tfFtpTurnPasv** with the TM_FTP_PASSIVE_MODE_ON flag to turn the server in passive mode.
- Two-server model:
  If the *ftpUserHandleB* parameter is non null, then the two-server model is used, and the first session should be in passive mode, and the second session should be in active mode. The user needs to call **tfFtpTurnPasv,** passing the *ftpUserHandleA* parameter, and the TM_FTP_PASSIVE_MODE_ON.

Parameters

Parameter	**Description**
ftpUserHandleA	The user handle of main FTP session will be operated on.
ftpUserHandleB	The user handle of the 2nd FTP session for two FTP server model.
fromFileNamePtr	Pointer to source file name string
toFileNamePtr	Pointer to destination file name string.

Returns

Value	**Meaning**
TM_EWOULDBLOCK	The call is non blocking and did not complete.
TM_EINVAL	Invalid ftpSessionPtr or bad filename.
TM_EACCES	Previous command has not finished
TM_ENOTLOGIN	Command requires user to be loggedin, and user is not.
TM_ENOTCONN	Command requires connection, and user is not connected
TM_EOPNOTSUPP	Command not supported by the user
TM_ENOERROR	No error (Success.)
TM_FTP_NAVAIL	Requested action not taken: file unavailable.
TM_FTP_NOCMD	Command not implemented
TM_FTP_XFERSTART	Data connection already open; transfer started
TM_FTP_FILEOKAY	File status okay; about to open data connection.
TM_FTP_DATAOPEN	Can't open data connection
TM_FTP_XFERABOR	Connection trouble, closed; transfer aborted.
TM_FTP_LOCALERR	Requested action aborted: local error in processing
TM_FTP_EXSPACE	Requested file action aborted: exceed storage allocation
TM_FTP_NOSPACE	Request action not taken: insufficient storage space in system.
TM_FTP_FILENAVAIL	Requested file action not taken: file unavailable.
TM_FTP_FILENAME	Requested action not taken: file name not allowed.
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMDPARAM	Command not implemented for that parameter.
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_NOTLOGIN	Not logged in.

tfFtpCdup

```
#include <trsocket.h>

int                 tfFtpCdup
(
ttUserFtpHandle     ftpSessionPtr
);
```

Function description

Changes directory on the remote file system to the current directory's parent directory.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection
TM_FTP_NOTLOGIN	Not logged in
TM_FTP_NAVAIL	Requested action not taken: file unavailable
TM_FTP_NOCMD	Command not implemented

tfFtpClose

#include <trsocket.h>

```
int                 tfFtpClose
(
ttUserFtpHandle     ftpSessionPtr
);
```

Function description

This function closes an FTP client socket, without sending a QUIT command. Normally, an FTP connection should be closed with the **tfFtpQuit** call. However, if the connection or the call to **tfFtpQuit** fails, this call may be used.

Parameters

Parameter	Description
ftpSessionPtr	FTP session handle

Returns

Value	Meaning
TM_ENOERROR	Success
TM_EINVAL	Invalid FTP session pointer

tfFtpConnect

```
#include <trsocket.h>

int                 tfFtpConnect
(
ttUserFtpHandle     ftpSessionPtr,
char *              ipAddressPtr
);
```

Function description

This function attempts to connect to a remote FTP server.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
ipAddressPtr	String specifying the IP address of the remote FTP server.

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EALREADY	Command in progress (previous call did not yet finish)
TM_EACCES	Trying to connect to a different FTP server with disconnecting from current server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SERVREADY	Service ready in n minutes (for exact time, use **tfFtpGetReplyText** to retrieve full reply text)
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection

tfFtpCwd

```
#include <trsocket.h>

int                 tfFtpCwd
(
ttUserFtpHandle     ftpSessionPtr,
char *              pathNamePtr
);
```

Function description

Changes directory on the remote file system.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
pathNamePtr	New directory name

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection
TM_FTP_NOTLOGIN	Not logged in
TM_FTP_NAVAIL	Requested action not taken: file unavailable
TM_FTP_NOCMD	Command not implemented

tfFtpDele

#include <trsocket.h>

```
 int                  tfFtpDele
(
ttUserFtpHandle      ftpSessionPtr,
char *               pathNamePtr
);
```

Function description

This function deletes file on remote file system.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
pathNamePtr	Filename to delete

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_ENOTLOGIN	User is not currently logged in.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_FILENAVAIL	Requested file action not taken: file unavailable
TM_FTP_NAVAIL	Requested action not taken: file unavailable
TM_FTP_NOTLOGIN	Not logged in.

tfFtpDirList

#include <trsocket.h>

```
int                 tfFtpDirList
(
ttUserFtpHandle     ftpSessionPtr,
char *              pathNamePtr,
int                 directoryFlag,
ttFtpCBFuncPtr      ftpDirCBFuncPtr
);
```

Function description

This function retrieves a directory listing for a specified directory on remote file system. When this data is received, a user-supplied function is called. The prototype for the callback function is:

```
int                 ftpCBFunc
(
ttUserFtpHandle     ftpSessionPtr,
char *              bufferPtr,
int                 bufferSize
);
```

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
pathNamePtr	Directory name
directoryFlag	Indicates whether a short (TM_DIR_SHORT) or long (TM_DIR_LONG) directory listing is desired.
ftpDirCBFuncPtr	The function pointer of the user function to call when the directory listing is received. See above for function prototype.

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_ENOTLOGIN	User is not currently logged in.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_FILENAVAIL	Requested file action not taken: file unavailable
TM_FTP_XFERSTART	Data connection already open; transfer starting
TM_FTP_FILEOKAY	File status okay; about to open data communications.
TM_FTP_DATAOPEN	Can't open data connection
TM_FTP_XFERABOR	Connection trouble, closed; transfer aborted TM_FTP_LOCALERR Requested action aborted: local error in processing.
TM_FTP_NOTLOGIN	Not logged in.

tfFtpFreeSession

```
#include <trsocket.h>

int                 tfFtpFreeSession
(
ttUserFtpHandle     ftpSessionPtr
);
```

Function description

This function frees resources associated with specified FTP session. Should be called by a user when a session has completed.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EINVAL	Invalid FTP session pointer

tfFtpGetReplyText

```
#include <trsocket.h>

int                 tfFtpGetReplyText
(
ttUserFtpHandle     ftpSessionPtr,
char *              replyStrPtr,
int                 replyStrLen
);
```

Function description

This function retrieves the full text of the reply to the most recently executed FTP command.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
replyStrPtr	Pointer to string to place reply text
replyStrLen	Length of above string buffer

Returns

Value	**Meaning**
char *	Length of data copied into user string, zero if invalid parameter or no reply string available.

tfFtpHelp

```
#include <trsocket.h>

int                  tfFtpHelp
(
ttUserFtpHandle      ftpSessionPtr,
char *               commandPtr,
ttFtpCBFuncPtr       ftpDirCBFuncPtr
);
```

Function description

This function retrieves help on the specified command from the remote FTP server. When this data is retrieved, a user-supplied function is called. The prototype for the callback function is:

```
int                  ftpCBFunc
(
ttUserFtpHandle      ftpSessionPtr,
char *               bufferPtr,
int                  bufferSize
);
```

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
commandPtr	Command to receive help on
ftpDirCBFuncPtr	The function pointer of the user function to call when the directory listing is received. See above for function prototype.

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.

tfFtpLogin

```
#include <trsocket.h>

int                tfFtpLogin
(
ttUserFtpHandle    ftpSessionPtr,
char *             userNamePtr,
char *             passwordNamePtr,
char *             accountNamePtr
);
```

Function description

Attempts to log on and authenticate to the remote FTP server.

Parameters

Parameter	Description
ftpSessionPtr	FTP session handle
userNamePtr	Username string
passwordNamePtr	Password string
accountNamePtr	Account string (if needed)

Returns

Value	Meaning
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized.
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_NEEDPASS	User name okay, need password
TM_FTP_NEEDACCTLOGIN	Need account for login
TM_FTP_BADCMDSEQ	Bad sequence of commands.

tfFtpMkd

```
#include <trsocket.h>

int                 tfFtpMkd
(
ttUserFtpHandle     ftpSessionPtr,
char *              directoryPathNamePtr,
char *              bufferPtr,
int                 bufferSize
);
```

Function description

This Function creates directory on remote file system.

Parameters

Parameter	Description
ftpSessionPtr	FTP session handle
directoryPathNamePtr	Path of directory to create
bufferPtr	Pointer to buffer to receive result from MKD command
bufferSize	Size of above result buffer

Returns

Value	Meaning
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_ENOTLOGIN	User is not currently logged in.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_FILENAME	Request action not taken: file name not allowed.
TM_FTP_NOTLOGIN	Not logged in.

tfFtpNewSession

#include <trsocket.h>

```
 ttUserFtpHandle   tfFtpNewSession
(
int               fileSystemFlags,
int               blockingState,
char *            fsUsernamePtr,
char *            fsPasswordPtr
);
```

Function description

This function creates a new FTP client session. This session may be used for multiple consecutive connections to multiple servers. This call returns a handle that should be used for all future commands associated with this session. When the session is complete, the user should call **tfFtpFreeSession** to free the resources used by this session. The user can specify whether this session should be blocking or non-blocking by setting **blockingState** accordingly **(TM_BLOCKING_ON and TM_BLOCKING_OFF**, respectively).

Parameters

Parameter	**Description**
fileSystemFlags	Flags to be passed to the local file system when it is opened.
blockingState	Specifies the blocking mode. TM_BLOCKING_ON or TM_BLOCKING_OFF.
fsUsernamePtr	Username used to log on to **local** file system.
fsPasswordPtr	Password used to log on to **local** file system.

Returns

Value	**Meaning**
0	Error creating new FTP session
ttUserFtpHandle	Handle associated with the newly created FTP session.

tfFtpNoop

```
#include <trsocket.h>

int                 tfFtpNoop
(
ttUserFtpHandle     ftpSessionPtr
);
```

Function description

This function sends the NOOP command to the FTP server (which does nothing). This command is most frequently used to keep an idle FTP connection open.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized

tfFtpPort

#include <trsocket.h>

```
int                 tfFtpPort
(
ttUserFtpHandle     ftpSessionPtr,
ttUserIpPort        ftpPortNo
);
```

Function description

This function sets the local port number that incoming data connections should use.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
ftpPortNo	Port number to use for incoming data connections

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTLOGIN	The user is not currently logged in.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer

tfFtpPwd

```
#include <trsocket.h>

int                  tfFtpPwd
(
ttUserFtpHandle      ftpSessionPtr,
char *               bufferPtr,
int                  bufferSize
);
```

Function description

Returns present working directory on remote file system.

Parameters

Parameter	Description
ftpSessionPtr	FTP session handle
bufferPtr	Pointer to buffer to receive
bufferSize	Size of above result buffer

Returns

Value	Meaning
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_ENOTLOGIN	User is not currently logged in.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_NAVAIL	Requested action not taken: file unavailable

tfFtpQuit

```
#include <trsocket.h>

int                 tfFtpQuit
(
ttUserFtpHandle     ftpSessionPtr
);
```

Function description

This function quits the current FTP connection (informs the remote server that we are closing all current connections).

Parameters

Parameter	Description
ftpSessionPtr	FTP session handle

Returns

Value	Meaning
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized.

tfFtpRein

```
#include <trsocket.h>

int                tfFtpRein
(
ttUserFtpHandle    ftpSessionPtr
);
```

Function description

Resets current FTP connection to the state just following the initial connection. This requires that the user be authenticated again, using **tfFtpLogin**.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SERVREADY	Service ready in n minutes (for exact time, use **tfFtpGetReplyText** to retrieve full reply text).
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_NOCMD	Command not implemented.

tfFtpRename

```
#include <trsocket.h>

int                  tfFtpRename
(
ttUserFtpHandle      ftpSessionPtr,
char *               fromNamePtr,
char *               toNamePtr
);
```

Function description

This function renames file on remote file system.

Parameters

Parameter	Description
ftpSessionPtr	FTP session handle
fromNamePtr	Old filename
toNamePtr	New filename

Returns

Value	Meaning
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete.
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_ENOTLOGIN	User is not currently logged in.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_NOCMDPARAM	Command not implemented for that parameter.
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.

TM_FTP_FILENAVAIL	Requested file action not taken: file unavailable
TM_FTP_NAVAIL	Requested action not taken: file unavailable
TM_FTP_NEEDACCTFILE	Need account for storing files
TM_FTP_FILENAME	Request action not taken: file name not allowed.
TM_FTP_NOTLOGIN	Not logged in.

tfFtpRetr

```
#include <trsocket.h>

int                 tfFtpRetr
(
ttUserFtpHandle   ftpUserHandleA,
ttUserFtpHandle   ftpUserHandleB,
char     TM_FAR * fromFileNamePtr,
char     TM_FAR * toFileNamePtr
);
```

Function Description

This function is used to retrieve a file from remote FTP server (corresponding to ftpUserHandleA). This function is implemented for both passive mode and normal mode, and also works with both one FTP server model and two FTP servers model.

- One server model:
  If the *ftpUserHandleB* parameter is NULL, then the one-server model is used, and the operation could be either in server active mode or server passive mode. By default the server is in active mode. The user needs to call **tfFtpTurnPasv** with the TM_FTP_PASSIVE_MODE_ON flag to turn the server in passive mode.

- Two-server model:
  If the *ftpUserHandleB* parameter is non null, then the two-server model is used, and the first session should be in passive mode, and the second session should be in active mode. The user needs to call **tfFtpTurnPasv,** passing the *ftpUserHandleA* parameter, and the TM_FTP_PASSIVE_MODE_ON.

Parameters

Parameter	**Description**
ftpUserHandleA	The user handle of main FTP session will be operated on.
ftpUserHandleB	The user handle of the 2nd FTP session for two FTP server model.
fromFileNamePtr	Pointer to source file name string
toFileNamePtr	Pointer to destination file name string.

Returns

Value	**Meaning**
TM_EWOULDBLOCK	The call is non blocking and did not complete.
TM_EINVAL	Invalid ftpSessionPtr or bad filename.
TM_EACCES	Previous command has not finished
TM_ENOTLOGIN	Command requires user to be loggedin, and user is not.
TM_ENOTCONN	Command requires connection, and user is not connected
TM_EOPNOTSUPP	Command not supported by the user
TM_ENOERROR	No error (Success.)
TM_FTP_XFERSTART	Data connection already open; transfer starting.
TM_FTP_FILEOKAY	File status okay; about to open data connection.
TM_FTP_DATAOPEN	Can't open data connection.
TM_FTP_XFERABOR	Connection trouble, closed; transfer aborted.
TM_FTP_LOCALERR	Requested action aborted: local error in processing.
TM_FTP_FILENAVAIL	Requested file action not taken: file unavailable.
TM_FTP_NAVAIL	Requested action not taken: file unavailable
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_NOTLOGIN	Not logged in.

tfFtpRmd

#include <trsocket.h>

```
int                 tfFtpRmd
(
ttUserFtpHandle     ftpSessionPtr,
char *              directoryPathNamePtr
);
```

Function Description

Removes directory on the remote file system.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
directoryPathNamePtr	Path of directory to remove

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete
TM_EACCES	Previous command did not complete
TM_ENOTCONN	The user is not currently connected to a FTP server
TM_ENOTLOGIN	User is not currently logged in.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_NAVAIL	Requested action not taken: file unavailable
TM_FTP_NOTLOGIN	Not logged in

tfFtpStor

```
#include <trsocket.h>

int                 tfFtpRetr
(
ttUserFtpHandle     ftpUserHandleA,
ttUserFtpHandle     ftpUserHandleB,
char     TM_FAR *   fromFileNamePtr,
char     TM_FAR *   toFileNamePtr
);
```

Function Description

This function is used to transmit a file to remote FTP server (corresponding to *ftpUserHandleA*). This function is implemented for both passive mode and normal mode, and also works with both one FTP server model and two FTP servers model.

- One server model:
  If the *ftpUserHandleB* parameter is NULL, then the one-server model is used, and the operation could be either in server active mode or server passive mode. By default the server is in active mode. The user needs to call **tfFtpTurnPasv** with the TM_FTP_PASSIVE_MODE_ON flag to turn the server in passive mode.

- Two-server model:
  If the *ftpUserHandleB* parameter is non null, then the two-server model is used, and the first session should be in passive mode, and the second session should be in active mode. The user needs to call **tfFtpTurnPasv,** passing the *ftpUserHandleA* parameter, and the TM_FTP_PASSIVE_MODE_ON.

Parameters

Parameter	**Description**
ftpUserHandleA	The user handle of main FTP session will be operated on.
ftpUserHandleB	The user handle of the 2nd FTP session for two FTP server model.
fromFileNamePtr	Pointer to source file name string
toFileNamePtr	Pointer to destination file name string.

Returns

Value	**Meaning**
TM_EWOULDBLOCK	The call is non blocking and did not complete.
TM_EINVAL	Invalid ftpSessionPtr or bad filename.
TM_EACCES	Previous command has not finished
TM_ENOTLOGIN	Command requires user to be loggedin, and user is not.
TM_ENOTCONN	Command requires connection, and user is not connected
TM_EOPNOTSUPP	Command not supported by the user
TM_ENOERROR	No error (Success.)
TM_FTP_XFERSTART	Data connection already open; transfer started
TM_FTP_FILEOKAY	File status okay; about to open data connection.
TM_FTP_DATAOPEN	Can’t open data connection
TM_FTP_XFERABOR	Connection trouble, closed; transfer aborted
TM_FTP_LOCALERR	Requested action aborted: local error in processing
TM_FTP_EXSPACE	Requested file action aborted: exceed storage allocation
TM_FTP_NOSPACE	Request action not taken: insufficient storage space in system
TM_FTP_FILENAVAIL	Requested file action not taken: file unavailable
TM_FTP_FILENAME	Requested action not taken: file name not allowed.
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMDPARAM	Command not implemented for that parameter
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection.
TM_FTP_NOTLOGIN	Not logged in

tfFtpSyst

```
#include <trsocket.h>

int                tfFtpSyst
(
ttUserFtpHandle    ftpSessionPtr,
char *             bufferPtr,
int                bufferSize
);
```

Function Description

This function returns information about the system hosting the remote file system.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
bufferPtr	Pointer to buffer to receive result from SYST command
bufferSize	Size of above result buffer

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete
TM_EACCES	Previous command did not complete.
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMD	Command not implemented
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection

tfFtpTurnPasv

```
int                 tfFtpTurnPasv
(
ttUserFtpHandle     ftpUserHandle,
int                 onFlag
);
```

Function Description

This function is used to set the mode of the FTP data connection on the peer FTP server. Note that this API will not send the PASV command to the server. It will merely set the mode on the FTP client session.

- If the TM_FTP_PASSIVE_MODE_OFF flag is set, then the peer FTP server will initiate the data connection, i.e. connect on the data connection (default setting).
- If the TM_FTP_PASSIVE_MODE_ON flag is set, then the peer FTP server will not initiate the connection on the data connection, but rather wait for a data connection
- The User can turn on/off passive mode by using this function before setting up data connections.

Parameters

Parameter	**Description**
ftpUserHandle	The user handle of FTP session will be operated on.
onFlag	The FTP passive mode. This parameter can be TM_FTP_PASSIVE_MODE_ON or TM_FTP_PASSIVE_MODE_OFF.

Returns

Name	**Description**
TM_EINVAL	Invalid argument.
TM_ENOERROR	No error

tfFtpType

```
#include <trsocket.h>

int                tfFtpType
(
ttUserFtpHandle    ftpSessionPtr,
int                type
);
```

Function Description

This function sets the transfer type to either binary or ASCII.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle
type	Transfer type: TM_TYPE_ASCII or TM_TYPE_BINARY

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_WOULDBLOCK	This FTP session is non-blocking and the call did not complete
TM_EACCES	Previous command did not complete
TM_ENOTCONN	The user is not currently connected to a FTP server.
TM_EINVAL	Invalid FTP session pointer
TM_FTP_SYNTAXCMD	Syntax error, command unrecognized
TM_FTP_SYNTAXARG	Syntax error in parameters or arguments
TM_FTP_NOCMDPARAM	Command not implemented for that parameter
TM_FTP_SERVNAVAIL	Service not available, closing TELNET connection
TM_FTP_NOTLOGIN	Not logged in

tfFtpUserExecute

```
#include <trsocket.h>

int               tfFtpUserExecute
(
ttUserFtpHandle   ftpSessionPtr
);
```

Function Description

This function executes the FTP client main loop. Call valid only if **tfFtpNewSession** had been called in non-blocking mode, and **tfFtpUserExecute** is not currently executing.

Parameters

Parameter	**Description**
ftpSessionPtr	FTP session handle

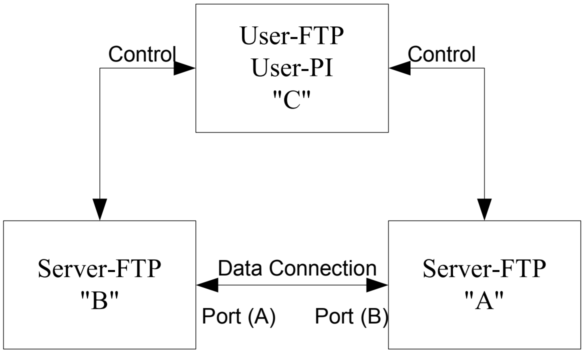
Returns

Value	**Meaning**
TM_ENOERROR	The current command has successfully completed.
TM_EWOULDBLOCK	The current command has not yet completed, and **tfFtpUserExecute** should be called again.
TM_EINVAL	Invalid FTP handle.
TM_EACCES	Blocking is currently on; This function should only be called when blocking is disabled.

FTP Passive Mode

Description

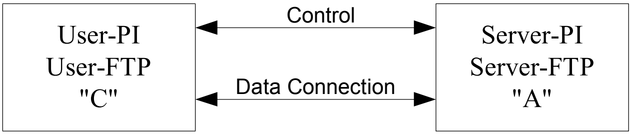
FTP passive mode is fully supported in the new release. The user may use it in both one-server and two-server passive models. Operation in two-server passive model is illustrated here:



User-PI - Server A	User-PI - Server B
C->A : Connect	C->B : Connect
C->A : PASV	
A->C : 227 Entering Passive Mode. A1,A2,A3,A4,a1,a2	
	C->B : PORT A1,A2,A3,A4,a1,a2
	B->C : 200 Okay
C->A : STOR	C->B : RETR

B->A : Connect to HOST-A, PORT-a

Operation in one-server mode is illustrated here:



C->A : Connect
C->A : PASV
A->C : 227 Entering Passive Mode. A1,A2,A3,A4,a1,a2
C->A : Connect to HOST-A, PORT-a
C->A : RETR

Example

Example for two server passive mode operation (STOR):

```
#include        <trsocket.h>
#include        <stdio.h> /* for printf() */

ttUserFtpHandle setupFtpCtrl(char * ipaddr)
{
    int                 retCode;
    ttUserFtpHandle     ftpHandle;

/* Create New Session for FTP server */
    ftpHandle = tfFtpNewSession(0, TM_BLOCKING_ON,
“fsusername”, “fspassword”);
    if (ftpHandle == (ttUserFtpHandle) 0 )
    {
        printf(“\n Failed to create FTP session!”);
    }
    else
    {
/* Connect to FTP server */
        retCode = tfFtpConnect(ftpHandle, ipaddr);
        if (retCode != TM_ENOERROR )
        {
            printf(“\n Failed to connect FTP server
!”);
            (void)tfFtpFreeSession(ftpHandle);
            ftpHandle = (ttUserFtpHandle)0;
        }
        else
        {

/* Login to FTP server */
            retCode = tfFtpLogin(ftpHandle,
“ftpusername”, “ftppassword”,””);
            if (retCode != TM_ENOERROR )
            {
                printf(“\n Failed to login on FTP
server !”);
                (void)tfFtpClose(ftpHandle);
                (void)tfFtpFreeSession(ftpHandle);
                ftpHandle = (ttUserFtpHandle)0;
            }
            else
            {
                printf(“\n Successfully setup CTRL
conn to FTP ! “);
            }
        }
    }

    return ftpHandle;
}

void main(void)
{
    ttUserFtpHandle     ftpHandleA;
    ttUserFtpHandle     ftpHandleB;

…. ….

/* Start the Treck stack */

…. ….

/* Setup Ctrl connection to FTP server A */
    ftpHandleA = setupFtpCtrl(“192.168.1.100”);

/* Setup Ctrl connection to FTP server B */
    ftpHandleB = setupFtpCtrl(“192.168.1.200”);

/* Set FTP session A operating in passive mode */
   (void)tfFtpTurnPasv(ftpHandleA,TM_FTP_PASSIVE_MODE_ON);

/* STOR: file will be transferred from server B to
 *server A
 */
    (void)tfFtpStor(ftpHandleA, ftpHandleB,
                       “testFtpFile”, “testFtpFile” );
…. ….
    return;
}
```

Example for one server passive mode operation (STOR):

```
#include        <trsocket.h>
#include        <stdio.h> /* for printf() */

void main(void)
{
    ttUserFtpHandle     ftpHandle;
```

```
 …. ….

/* Start the Treck stack */

 …. ….

/* Setup Ctrl connection to FTP server */
    ftpHandle = setupFtpCtrl(“192.168.1.100”);

/* Set FTP session operating in passive mode */
   (void)tfFtpTurnPasv(ftpHandle,TM_FTP_PASSIVE_MODE_ON);

/* STOR: file will be transferred from client to
 * server
 */
    (void)tfFtpStor(ftpHandle, 0,
                       “testFtpFile”, “testFtpFile” );
 …. ….
    return;
}
```

* For function prototype **setupFtpCtrl**, please refer to example of Two-FTP-Server Passive Mode.

TFTP Client Application Program Interface

Description

The TFTP Client Application Program Interface allows the user to retrieve files from and store files to a remote TFTP server.

User interface
The user interface allows the user to get and put files from a remote TFTP server.

User Interface

Five calls are provided in the TFTP Client User Interface.

tfTftpGet	Called by the user to get a file from a TFTP server.
tfTftpInit	This function must be called before any other TFTP API calls are made. It initializes various data associated with the TFTP client.
tfTftpPut	Called by the user to store a file to a TFTP server.
tfTftpSetTimeout	Called by the user to set timeout and retry values.
tfTftpUserExecute	If the stack is running in non-blocking mode, this function must be called periodically in order to get the client to execute. In blocking mode, this function should not be called

Blocking Mode

In blocking mode, **tfTftpGet** and **tfTftpPut** will both block until the transfer is completed or an error is returned. The TFTP client code is executed in the context of the calling task. Choose blocking mode if you are using an RTOS/Kernel.

Non-Blocking Mode

In non-blocking mode, calls to **tfTftpGet** and **tfTftpPut** will return immediately. **tfTftpUserExecute** must be called periodically to cause them to execute. Choose non-blocking mode if you are not using an RTOS/Kernel.

tfTftpGet

```
#include <trsocket.h>

int                     tfTftpGet
(
char *                  filename,
struct sockaddr *       remote_addr,
char *                  tftpbuf,
unsigned long int       bufsize,
unsigned short int      mode,
int                     blocking
);
```

Function Description

This function retrieves a file from a TFTP server.

Blocking Mode

In blocking mode, **tfTftpGet** should be called from within a task. It will block until the file transfer is completed, or an error is returned. The TFTP client code is executed in the context of the calling task. Choose blocking mode if you are using an RTOS/Kernel.

Non-Blocking Mode

In non-blocking mode, **tfTftpGet** will return immediately. It is then the user's responsibility to then call **tfTftpUserExecute** periodically to execute the TFTP client code. Choose non-blocking mode if you do not have an RTOS/Kernel.

Parameters

Parameter	**Description**
filename	Null-terminated string containing the file name to retrieve from the server.
remote_addr	Structure representing the address of the server. The type (AF_INET), address, and port (for TFTP, usually 69) must be filled in.
tftpbuf	A pointer to a buffer to store the file into.
bufsize	The size, in bytes, of the buffer.
mode	TM_TYPE_ASCII for ASCII transfers, TM_TYPE_BINARY for binary transfers.
blocking	TM_BLOCKING_ON for blocking mode, TM_BLOCKING_OFF for non-blocking mode.

Returns

Value	**Meaning**
0	Success
TM_TFTP_EXECUT	A session is already in progress. Only one session may be in progress at a time.
TM_TFTP_EINVAL	mode is neither TM_TYPE_ASCII nor TM_TYPE_BINARY
TM_TFTP_EINVAL	blockingState is neither TM_BLOCKING_ON nor TM_BLOCKING_OFF.
TM_TFTP_EINVAL	blockingState is TM_BLOCKING_ON and blocking-mode is not enabled for the stack.
TM_TFTP_ESOCK	An error occurred with one or more of the socket calls within the function.
TM_TFTP_EBUF	the buffer is not large enough to hold the received file

tfTftpInit

```
#include <trsocket.h>

void                 tfTftpInit
(
void
);
```

Function description
This function initializes various data associated with the TFTP client. It must be called before any other TFTP client API call.

Parameters

None

Returns

None

tfTftpPut

#include <trsocket.h>

```
int                     tfTftpPut
(
char *                  filename,
struct sockaddr *       remote_addr,
char *                  tftpbuf,
unsigned long int       bufsize,
unsigned short int      mode,
int                     blocking
);
```

Function Description

This function sends a file to a TFTP server.

Blocking Mode

In blocking mode, **tfTftpPut** should be called from within a task. It will block until the file transfer is completed, or an error is returned. The TFTP client code is executed in the context of the calling task. Choose blocking mode if you are using an RTOS/Kernel.

Non-Blocking Mode

In non-blocking mode, **tfTftpPut** will return immediately. It is then the user’s responsibility to then call **tfTftpUserExecute** periodically to execute the TFTP client code. Choose non-blocking mode if you do not have an RTOS/Kernel.

Parameters

Parameter	**Description**
filename	Null-terminated string containing the file name to send to the server
remote_addr	Structure representing the address of the server. The type (AF_INET), address, and port (for TFTP, usually 69) must be filled in
tftpbuf	A pointer to a buffer to store the file into
bufsize	The number of bytes to send from the buffer
mode	TM_TYPE_ASCII for ASCII transfers, TM_TYPE_BINARY for binary transfers
blocking	TM_BLOCKING_ON for blocking mode, TM_BLOCKING_OFF for non-blocking mode

Returns

Value	**Meaning**
0	Success
TM_TFTP_EXECUT	A session is already in progress. Only one session may be in progress at a time
TM_TFTP_EINVAL	mode is neither TM_TYPE_ASCII nor TM_TYPE_BINARY
TM_TFTP_EINVAL	blockingState is neither TM_BLOCKING_ON nor TM_BLOCKING_OFF
TM_TFTP_EINVAL	blockingState is TM_BLOCKING_ON and blocking-mode is not enabled for the stack
TM_TFTP_ESOCK	An error occurred with one or more of the socket calls within the function

tfTftpSetTimeout

#include <trsocket.h>

```
void                tfTftpSetTimeout
(
int                 timeout,
int                 retry
);
```

Function description

This function allows the user to set the retry and timeout values for the TFTP client.

Parameters

Parameter	Description
timeout	The length of time (in seconds) a connection can remain silent before it times out.
retry	The number of consecutive times a connection can time out before it is closed.

Returns

None

tfTftpUserExecute

#include <trsocket.h>

```
ttUser32Bit tfTftpUserExecute
(
void
);
```

Function description

This function executes the TFTP client's main loop. This call is valid only if **tfTftpGet** or **tfTftpPut** has been called in non-blocking mode and **tfTftpUserExecute** is not currently executing.

Parameters

None

Returns

Value	**Meaning**
>0	tfTftpGet transfer was successful. This value is the size of the file received.
TM_TFTP_SUCCESS	tfTftpPut transfer was successful.
TM_TFTP_EXECUT	This value is returned while the transfer is in progress.
TM_TFTP_TIMEOUT	A TFTP operation timed out.
TM_TFTP_EBUF	Insufficient memory.
TM_TFTP_ESOCK	A socket error occurred.
TM_TFTP_ERROR	General TFTP error.

TFTPD Application Program Interface

Description

The TFTPD Application Program Interface allows the user to run a TFTP server. It consists of two parts:

User interface
The user interface allows the user to start/stop the TFTP server to allow/stop remote TFTP clients to connect and exchange files with the host.

File system interface
The file system interface allows the TFTP server to interact with the operating system's file system to do such things as store and retrieve files.

User Interface

Four calls are provided in the TFTPD User Interface.

1. tfTftpdInit

This function must be called before any other TFTP API calls are made. It initializes various data associated with the TFTP server.

2. tfTftpdUserStart

The user calls **tfTftpdUserStart** to open a TFTP server socket and to begin listening for incoming connections. **tfTftpdUserStart** can be either blocking or non blocking, as specified by its last parameter.

Blocking Mode

In blocking mode, **tfTftpdUserStart** should be called from a task. It will block and wait for incoming connections, and will not return unless an error occurs. The TFTP server code is executed in the context of the calling task. Choose blocking mode if you are using an RTOS/Kernel.

Non-Blocking Mode

In non-blocking mode, **tfTftpdUserStart** will return immediately after checking for incoming connections. It is the user's responsibility to then call **tfTftpdUserExecute** periodically to execute the TFTP server code. Choose non-blocking mode if you do not have an RTOS/Kernel.

3. tfTftpdUserExecute

If **tfTftpdUserStart** was called in non-blocking mode, **tfTftpdUserExecute** must be called periodically. If **tfTftpdUserStart** was called in blocking mode, there is no need to call **tfTftpdUserExecute**.

4. tfTftpdUserStop

The user calls **tfTftpdUserStop** to close the TFTP server socket and kill all existing TFTP connections.

File System Interface

Note: File system calls for TFTP can be found in the File System Section of this manual.

tfTftpdInit

```
#include <trsocket.h>

void                  tfTftpdInit
(
void
);
```

Function description
This function initializes various data associated with the TFTP server. It must be called before any other TFTP API call.

Parameters
None

Returns
Nothing

tfTftpdUserExecute

```
#include <trsocket.h>

int                 tfTftpdUserExecute
(
void
);
```

Function description

This function executes the TFTP server's main loop. This call is valid only if **tfTftpdUserStart** has been called in non-blocking mode and **tfTftpdUserExecute** is not currently executing.

Parameters

None

Returns

Value	**Meaning**
0	Success
TM_EPERM	TFTP server currently executing, not started, or stopped

tfTftpdUserStart

#include <trsocket.h>

```
int             tfTftpdUserStart
(
int             maxConnections,
int             sendTimeout,
int             timeoutTime,
int             blockingState
);
```

Function Description

This function opens a TFTP server socket and starts listening for incoming connections. **tfTftpdUserStart** can be either blocking or non-blocking, as specified by the *blockingState* parameter.

Blocking Mode

In blocking mode, **tfTftpdUserStart** should be called from a task. It will block and wait for incoming connections, and will not return unless an error occurs. The TFTP server code is executed in the context of the calling task. Choose blocking mode if you are using an RTOS/Kernel.

Non-Blocking Mode

In non-blocking mode, **tfTftpdUserStart** will return immediately after checking for incoming connections. It is the user's responsibility to then call **tfTftpdUserExecute** periodically to execute the TFTP server code. Choose non-blocking mode if you do not have an RTOS/Kernel.

Parameters

Parameter	**Description**
maxConnections	Maximum number of concurrent incoming TFTP connections allowed. Must be at least one
sendTimeout	The amount of time, in seconds, a connection can be idle before it times out
timeoutTime	The number of consecutive retries a connection will make before it gives up. Each time a connection times out, it counts as one retry
blockingState	TM_BLOCKING_ON for blocking mode, TM_BLOCKING_OFF for non-blocking mode

Returns

Value	**Meaning**
0	Success
TM_EINVAL	maxConnections is less than one
TM_EINVAL	blockingState is neither TM_BLOCKING_ON nor TM_BLOCKING_OFF.
TM_EINVAL	blockingState is TM_BLOCKING_ON and blocking-mode is not enabled for the stack.
TM_SOCKET_ERROR	The function was unable to create a socket.
All other error codes	This function uses various sockets calls. Any other error codes returned will be from said sockets calls. The error codes are defined in **trsystem.h**.

tfTftpdUserStop

```
#include <trsocket.h>

int                 tfTftpdUserStop
(
void
);
```

Function description

This function stops execution of the TFTP server. It closes the listening socket and killing all existing connections.

Parameters

None

Returns

Value	Meaning
0	Success
TM_EALREADY	The TFTP server has already been stopped.

File system interface

Description

The file system interface is used by the Turbo Treck FTP server, Turbo Treck TFTP server, and Turbo Treck FTP Client.

Entry points from the FTP server to the file system:

tfFSChangeDir	Change current working directory
tfFSChangeParentDir	Change current working directory to parent directory
tfFSCloseDir	Close a directory that we had opened earlier.
tfFSCloseFile	Close a file
tfFSDeleteFile	Delete a file
tfFSGetNextDirEntry	Get the next directory entry in the directory open with **tfFSOpenDir**, either a long listing of the directory entry (including volumes, sub directories, and file names), or a short listing of the directory (file name only), depending on how the directory was open
tfFSGetUniqueFileName	Given a file name, return a unique file name in the current directory (i.e, if the file name already exists, make up a new name that is unique in the current directory.)
tfFSGetWorkingDir	Get user working directory
tfFSMakeDir	Create specified directory
tfFSOpenDir	Open specified directory, or directory corresponding to a specified pattern to allow getting a long or short listing of the directory or of the directory entries matching the specified pattern
tfFSOpenFile	Open a file (creating it if it does not exist), for read, write, or append, specifying type (ASCII, or binary), structure (stream, or record)
tfFSReadFile	Read n bytes from a file into a buffer

tfFSReadFileRecord	Read a record from a file up to n bytes. Indicates whether EOR has been reached
tfFSRemoveDir	Remove specified directory
tfFSRenameFile	Rename a file
tfFSStructureMount	Mount the user to a new file system data structure
tfFSSystem	Return the system name
tfFSUserAllowed	Indicates whether a specified user is allowed on the system
tfFSUserLogin	Login a user if password is valid
tfFSUserLogout	Logout a user.
tfFSWriteFile	Write some bytes from a buffer to a file
tfFSWriteFileRecord	Write a record from a buffer to a file

Entry points from the FTP client to the file system:

Note: The Turbo Treck FTP client only uses a subset of the functions described above, namely the eight functions in the list below.

tfFSCloseDir
tfFSCloseFile
tfFSOpenDir
tfFSOpenFile
tfFSReadFile
tfFSUserLogin
tfFSUserLogout
tfFSWriteFile

Entry points from the TFTP server to the file system:

Note: The Turbo Treck TFTP server only uses a subset of the functions described above, namely the six functions in the list below:

tfFSCloseDir
tfFSCloseFile
tfFSOpenDir
tfFSOpenFile
tfFSReadFile
tfFSWriteFile

tfFSChangeDir

```
#include <trsocket.h>

int                 tfFSChangeDir
 (
void *              userDataPtr,
char *              pathNamePtr
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin** and a directory name, this function changes the user's working directory to the new directory.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
pathNamePtr	Pointer to a null terminated string containing directory path name

Returns

Value	**Meaning**
0	Success
-1	Failure

tfFSChangeParentDir

```
#include <trsocket.h>

int                  tfFSChangeParentDir
(
void                 userDataPtr
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin,** this function changes the user's working directory to its parent directory.

Parameters

Parameter	Description
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**

Returns

Value	Meaning
0	Success
-1	Failure

tfFSCloseDir

```
#include <trsocket.h>

void                  tfFSCloseDir
(
void *                userDataPtr,
void *                dirDataPtr
);
```

Function description

Given a unique user data pointer as returned by **tfFSUserLogin** and a directory data pointer as returned by **tfFSOpenDir**, this function closes the directory and frees the directory data structure pointed to by dirDataPtr.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
dirDataPtr	Pointer to newly allocated directory data structure

Returns

Nothing

tfFSCloseFile

```
#include <trsocket.h>

int                  tfFSCloseFile
(
void *               userDataPtr,
void *               fileDataPtr
);
```

Function description

Given a unique user data pointer as returned by **tfFSUserLogin** and a file data pointer as returned by **tfFSOpenFile**, this function closes the file and frees the file data structure pointed to by fileDataPtr.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
fileDataPtr	Pointer to file data structure as returned by **tfFSOpenFile**

Returns

Value	**Meaning**
0	Success
-1	Failure

tfFSDeleteFile

```
#include <trsocket.h>

int                 tfFSDeleteFile
 (
void *              userDataPtr,
char *              pathNamePtr
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin** and a file name, this function deletes the file.

Parameters

Parameter	Description
userDataPtr	Pointer to user data structure as returned by **tfFsUserLogin**
pathNamePtr	Pointer to a null terminated string containing file name.

Returns

Value	Meaning
0	Success
-1	Failure

tfFSFlushFile

```
#include <trsocket.h>
int                  tfFSFlushFile
(
void *               userDataPtr,
void *               fileDataPtr
);
```

Function description

Given a unique user data pointer as returned by **tfFSUserLogin** and a file data pointer as returned by **tfFSOpenFile**, this function flushes the file**.**

Parameters

Parameter Description	
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
fileDataPtr	Pointer to file data structure as returned by **tfFSOpenFile**

Returns

Value	**Meaning**
0	Success
-1	Failure

tfFSGetNextDirEntry

```
#include <trsocket.h>

int                  tfFSGetNextDirEntry
(
void *               userDataPtr,
void *               dirDataPtr,
char *               bufferPtr,
int                  bufferSize
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin** and a directory data pointer as returned by **tfFSOpenDir**, this function retrieves the next entry in the directory that matches the path given as an argument to **tfFSOpenDir**. The next entry should be either a long listing of the next directory entry (either volume name, sub-directory, or file name with its attribute), or a short listing of the directory entry (file name only without any attribute), and should be stored in the buffer pointed to by bufferPtr (up to bufferSize bytes). The FTP server will continue calling **tfFSGetNextDirEntry** until it gets a return value of 0 bytes, indicating that all matching directory entries have been retrieved. It is up to the developer of **tfFSGetNextDirEntry** to keep track of how many entries of the listing have been read so far, whether the directory was open for long or short listing, and the matching pattern (using dirDataPtr).

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
dirDataPtr	Pointer to newly allocated directory data structure
bufferPtr	Pointer to a buffer where to copy the next directory entry
bufferSize	Size in bytes of the buffer

Returns

Value	**Meaning**
-1	Failure
> 0	Number of bytes copied into the buffer pointed to by bufferPtr
0	End of directory

tfFSGetUniqueFileName

#include <trsocket.h>

```
int         tfFSGetUniqueFileName
void *      userDataPtr,
char *      bufferPtr,
int         bufferSize
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin**, this function finds and copies a unique file name that does not conflict with any existing file names in the user's working directory in the buffer pointed to by bufferPtr (up to bufferSize bytes).

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFsUserLogin**
bufferPtr	Pointer to buffer where to store the unique file name
bufferSize	Size in bytes of the buffer

Returns

Value	**Meaning**
-1	Failure
> 0	Number of copied bytes

tfFSGetWorkingDir

#include <trsocket.h>

```
int         tfFSGetWorkingDir
(
void        userDataPtr,
char        bufferPtr,
int         bufferSize
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin**, copy the user's working directory in the buffer pointed to by bufferPtr (up to bufferSize).

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
bufferPtr	Pointer to a buffer where to copy the pathname of the user working directory
bufferSize	Size in bytes of the buffer.

Returns

Value	**Meaning**
-1	Failure
> 0	Number of bytes copied into the buffer pointed to by bufferPtr

tfFSMakeDir

```
#include <trsocket.h>

int                  tfFSMakeDir
 (
void *               userDataPtr,
char *               pathNamePtr,
char *               bufferPtr,
int                  bufferSize
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin**, and a directory name, this function creates the directory. If successful, copy the directory path name in bufferPtr (up to bufferSize). The directory path name could be either absolute or relative to the user working directory path, but should be such that a subsequent **tfFSChangeDir** with that pathname as an argument should not fail.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
pathNamePtr	Pointer to a null terminated string containing directory path
bufferPtr	Pointer to a buffer where to copy the pathname of the newly created directory
bufferSize	Size in bytes of the buffer

Returns

Value	**Meaning**
-1	Failure
> 0	Number of bytes copied into the buffer pointed to by bufferPtr

tfFSOpenDir

```
#include <trsocket.h>

void *              tfFSOpenDir
 (
void *              userDataPtr,
char *              pathNamePtr,
int                 flag
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin**, a pointer to a path name, and a flag, this function opens the directory corresponding to the path for reading either a long directory (flag == TM_DIR_LONG) or short directory (TM_DIR_SHORT). Subsequent calls to **tfFSGetNextDirEntry** will fetch each entry in the directory matching the pattern as pointed to by pathNamePtr. The **tfFSOpenDir** implementer should allocate a directory data structure to keep track of the path name matching pattern, the reading position in the directory, and the directory read flag (TM_DIR_LONG, or TM_DIR_SHORT). Note that if pathNamePtr points to a directory name, then the matching pattern is "*.*". If pathNamePtr points to "*.*", then the user working directory should be open, and the matching pattern is "*.*".

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
pathNamePtr	Pointer to a null terminated string containing pathname.
flag	Either TM_DIR_LONG, or TM_DIR_SHORT

Returns

Value	**Meaning**
(void *)0	Failure
dirDataPtr	Pointer to newly allocated directory data structure

tfFSOpenFile

#include <trsocket.h>

```
void *       tfFSOpenFile
 (
void *       userDataPtr,
char *       pathNamePtr,
int          flag,
int          type,
int          structure
);
```

Function description:
Given the unique user data pointer as returned by **tfFSUserLogin** and a file name, this function opens the file for either read (if flag is TM_FS_READ), write (if flag is TM_FS_WRITE), or append (if flag is TM_FS_APPEND). The parameter type specifies if file type is ASCII (TM_TYPE_ASCII) or binary (TM_TYPE_BINARY). Parameter structure specifies if the file structure is stream (TM_STRU_STREAM), or record (TM_STRU_RECORD). This function allocates a file data structure to store the file pointer, file type, file structure, etc.

Note: This call should fail if the file name is a directory.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFsUserLogin**
pathNamePtr	Pointer to a null terminated string containing file name
flag	Open flag: TM_FS_READ, TM_FS_WRITE, TM_FS_APPEND
type	File type: TM_TYPE_ASCII or TM_TYPE_BINARY.
structure	File structure: TM_STRU_RECORD, or TM_STRU_STREAM

Returns

Value	**Meaning**
(void *)0	Failure
fileDataPtr	Pointer to newly allocated file data structure.

tfFSReadFile

#include <trsocket.h>

```
int          tfFSReadFile
(
void *       userDataPtr,
void *       fileDataPtr,
char *       bufferPtr,
int          bufferSize
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin** and a file data pointer as returned by **tfFSOpenFile**, this function reads up to bufferSize bytes into the buffer. It returns the number of bytes actually read, 0 if end of file has been reached, -1 on error.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
fileDataPtr	Pointer to file data structure as returned by **tfFSOpenFile**
bufferPtr	Pointer to buffer where to copy the data from the file
bufferSize	Size in bytes of the buffer

Returns

Value	**Meaning**
> 0	Number of copied bytes
0	End of file
-1	Failure

tfFSReadFileRecord

```
#include <trsocket.h>

int                   tfFSReadFileRecord
(
void *                userDataPtr,
void *                fileDataPtr,
char *                bufferPtr,
int                   bufferSize,
int  *                eorPtr
);
```

Function description

Given the unique user data pointer as returned by t**fFSUserLogin** and a file data pointer as returned by **tfFSOpenFile**, this function reads until it reaches bufferSize bytes or end of record (whichever comes first). If end of record has been reached it stores 1 in the integer pointed to by eorPtr, otherwise it stores 0. If the file system does not support records, then the system end of line (i.e. <CR><LF> for DOS, <LF> for Unix) is used for the end of record. This routine should convert every end of line (i.e. <CR><LF> for DOS, <LF> for Unix) to an end of record, i.e. store 1 in eorPtr when the end of line character(s) have been read; the end of line character(s) themselves should not be copied into the buffer pointed to by bufferPtr.

To indicate that end of file was reached, this routine stores 0 in eorPtr, and returns 0 to indicate that no characters were read. Note that if 1 is stored in eorPtr, this does not indicate end of file, since when a blank line consisting of nothing but the end of line characters (i.e. <CR><LF>)occurs in the middle of an ASCII text file, this routine returns 0 to indicate that no characters were read (since the end of line characters are stripped out and are not copied into the buffer pointed to by bufferPtr) and stores 1 in eorPtr to indicate end of record.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
fileDataPtr	Pointer to file data structure as returned by **tfFSOpenFile**.
bufferPtr	Pointer to buffer where to copy the data from the file.
bufferSize	Size in bytes of the buffer.
eorPtr	Pointer to end of record

Returns

Value	**Meaning**
> 0	Number of copied bytes (not including end of record)
0	End of file has been reached
-1	Failure

tfFSRemoveDir

#include <trsocket.h>

```
int          tfFSRemoveDir
(
void *       userDataPtr,
char *       pathNamePtr
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin**, and a directory name, this function removes the directory.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
pathNamePtr	Pointer to a null terminated string containing directory path name.

Returns

Value	**Meaning**
0	Success
-1	Failure

tfFSRenameFile

```
#include <trsocket.h>

int                 tfFSRenameFile
(
void *              userDataPtr,
char *              fromPathNamePtr,
char *              toPathNamePtr
);
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin,** a current file name, and a new file name, this function renames the file to the new file name.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFsUserLogin**
fromPathNamePtr	Pointer to null terminated string containing current file name
toPathNamePtr	Pointer to null terminated string containing new file name

Returns

Value	**Meaning**
0	Success
-1	Failure

tfFSStructureMount

```
#include <trsocket.h>

int                 tfFSStructureMount
 (
void *              userDataPtr,
char *              pathNamePtr
)
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin** and a file system, this function mounts the user to the new file system.

Parameters

Parameter	Description
userDataPtr	Pointer to user data structure as returned by **tfFsUserLogin**.
pathNamePtr	Pointer to a null terminated string containing a file system name

Returns

Value	Meaning
0	0
-1	Failure

tfFSSystem

```
#include <trsocket.h>

int                 tfFSSystem
(
char *              bufferPtr,
int                 bufferSize
);
```

Function description

This function copies the official system name, as assigned in the list of OPERATING SYSTEM NAMES" in the "Assigned Numbers" RFC (RFC 1700) into bufferPtr (up to bufferSize bytes). For example, the DOS operating system has been assigned DOS as system name. If the file system is a DOS file system, then this function should copy "DOS" into bufferPtr. If the system has not been assigned a system name in the RFC, then this function should return -1.

Parameters

Parameter	**Description**
bufferPtr	Pointer to a buffer where to copy the system name
bufferSize	Size in bytes of the buffer

Returns

Value	**Meaning**
-1	Failure
> 0	Number of bytes copied

tfFSUserAllowed

```
#include <trsocket.h>

int                 tfFSUserAllowed
(
char *              userNamePtr
);
```

Function description

This function verifies whether a user is allowed on the system.

Parameters

Parameter	**Description**
userNamePtr	Pointer to a null terminated string containing the user name

Returns

Value	**Meaning**
0	Success
-1	User is not allowed on the system

tfFSUserLogin

#include <trsocket.h>

```
void *        tfFSUserLogin
(
char *        userNamePtr,
char *        passwordPtr
);
```

Function description

When given a User name string and Password string, this function returns a unique user data pointer (if password is correct, it returns a Null pointer otherwise). The just allocated user data pointer points to a data structure containing information unique to the just logged in current user, such as its current working directory.

Parameters

Parameter	Description
userNamePtr	Pointer to a null terminated string containing the user name
passwordPtr	Pointer to a null terminated string containing the password

Returns

Value	Meaning
(void *)0	Failure
userDataPtr	Pointer to a unique user data structure containing information about the given user (such as its working directory.)

tfFSUserLogout

```
#include <trsocket.h>

void                  tfFSUserLogout
(
void *                userDataPtr
);
```

Function Description

Given the unique user pointer as returned by **tfFSUserLogin**, this function logs the user out and frees the structure pointed to by userDataPtr.

Parameters

Parameter	Description
userDataPtr	Pointer to the user data structure as returned by **tfFSUserLogin**

Returns

Nothing

tfFSWriteFile

```
#include <trsocket.h>

int           tfFSWriteFile
(
void *        userDataPtr,
void *        fileDataPtr,
char *        bufferPtr,
int           bytes
)
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin** and a file data pointer as returned by **tfFSOpenFile**, this function writes bytes from the buffer pointed to by bufferPtr to the file.

Parameters

Parameter	**Description**
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
fileDataPtr	Pointer to file data structure as returned by **tfFSOpenFile**
bufferPtr	Pointer to buffer data to copy into the file
bytes	Size in bytes of the data in the buffer

Returns

Value	**Meaning**
0	Success
-1	Failure

tfFSWriteFileRecord

```
#include <trsocket.h>

int                  tfFSWriteFileRecord
(
void *               userDataPtr,
void *               fileDataPtr
char *               bufferPtr,
int                  bytes,
int                  eor
)
```

Function description

Given the unique user data pointer as returned by **tfFSUserLogin** and a file data pointer as returned by **tfFSOpenFile**, this function writes up to bytes from the buffer, in addition to an end of record if EOR is set to 1. If the file system does not support records, then the system end of line (i.e. <CR><LF> for DOS, <LF> for Unix) should be used instead of EOR.

Parameters

Parameter	Description
userDataPtr	Pointer to user data structure as returned by **tfFSUserLogin**
fileDataPtr	Pointer to file data structure as returned by **tfFSOpenFile**.
bufferPtr	Pointer to buffer data to copy into the file
bytes	Size in bytes of the data in the buffer
eor	End of record indicator (1 if end of record need to be written)

Returns

Value	Meaning
0	Success
-1	Failure

Telnet Daemon

Description

The TELNETD Application Program Interface allows the user to run a TELNET server. It consist of 2 parts:

User to Telnet server interface:
The user to Telnet interface allows the user to start/stop the Telnet server to allow/stop remote telnet clients to connect and login on the server. It also allows the user to send data to each connected telnet client.

Telnet server to user interface:
The Telnet server to user interface allows the telnet server to notify the user of incoming connections. It also allows the telnet server to hand data received from each connected telnet client to the user, after the NVT conversion has been performed by the telnet server.

User to Telnet server interface

Five calls are provided in the user to telnet server interface:

tfTeldUserStart
The user calls **tfTeldUserStart**, to open a TELNET server socket and start listening for incoming connections. **tfTeldUserStart** can be either blocking or non blocking, as specified by its last parameter.

Blocking Mode
In blocking mode, **tfTeldUserStart** should be called from a task. **tfTeldUserStart** will not return unless an error occurs, will block, wait for incoming connections, and execute the Telnet server code in the context of the calling task. Choose the blocking mode, if you are using an RTOS/Kernel.

Non-Blocking Mode
In non-blocking mode, **tfTeldUserStart** will return immediately after listening for incoming connections. It is the user responsibility to then call **tfTeldUserExecute** periodically to execute the Telnet server code. Choose the non-blocking mode, if you do not have an RTOS/Kernel.

tfTeldUserExecute
If the user calls *tfTeldUserStart* in non-blocking mode, then the user must call *tfTeldUserExecute* periodically. If the user calls *tfTeldUserStart* in blocking mode, then there is no need to call **tfTeldUserExecute**

tfTeldUserStop
The user calls *tfTeldUserStop* to close the telnet server socket and kill all existing Telnet connections.

tfTeldUserSend
The user calls tfTeldUserSend to send data to a specified telnet client. The telnet server will perform the NVT conversion before sending the data on the network.

tfTeldUserClose
The user calls *tfTeldUserClose* to force the server to close a specified telnet connection.

Telnet server to user interface

tfTeldOpened
Call provided by the user, and called by the telnet server.
The telnet server calls *tfTeldOpened* to inform the user that a specified telnet client has established a new connection to the telnet server.

tfTeldIncoming
Call provided by the user, and called by the telnet server. The telnet server calls *tfTeldIncoming* to give data received from a specified telnet client, to the user. The telnet server performs the NVT conversion before handing the data to the user.

tfTeldClosed
Call provided by the user, and called by the telnet server. The telnet server calls *tfTeldClosed* to inform the user that a specified telnet client closed the connection.

tfTeldClosed

```
#include <trsocket.h>

void                tfTeldClosed
(
ttUserTeldHandle    teldHandle
);
```

Function description

The user provides this function. It is called from the telnet server to let the user know that the telnet client has closed the telnet connection.

Parameters

Parameter	**Description**
teldHandle	Unique identifier for a telnet connection

Returns

Nothing

tfTeldIncoming

```
#include <trsocket.h>

int                tfTeldIncoming
(
ttUserTeldHandle  teldHandle,
char *            teldRecvBufPtr,
int               teldBytes,
int               eolFlag
);
```

Function description

The user provides this function. It is called from the telnet server to pass incoming data from the specified telnet client to the user. If eolFlag is non-zero, then the sequence <CR, LF> or <CR, NUL> has been received in ASCII mode, or <IAC, EOR> in binary mode. The user should copy all the data to their own buffers. If there is not enough room, the user should return TM_EWOULDBLOCK, and the telnet server will try to call this routine again later.

Parameters

Parameter	**Description**
teldHandle	Unique identifier for a telnet connection
teldRecvBufPtr	Pointer to buffer containing received data
teldBytes	Number of bytes in the receive buffer
eolFlag	End of line flag, 1 to indicate EOL, 0 otherwise

Returns

Value	**Meaning**
TM_ENOERROR	User had room for the data and copied it in its buffer
TM_EINVAL	Invalid Telnet Handle
TM_EWOULDBLOCK	User did not have enough room for all the data. No data has been copied

tfTeldOpened

#include <trsocket.h>

```
void                     tfTeldOpened
(
ttUserTeldHandle         teldHandle,
struct sockaddr_in *     sockAddrPtr
);
```

Function description

The user provides this function. It is called from the telnet server to let the user know that a telnet client has established a new connection to the telnet server.

Parameters

Parameter	Description
teldHandle	Telnet handle, Unique identifier for a telnet connection
sockAddrPtr	Pointer to a sockaddr_in structure containing the IP address of the telnet client

Returns

Nothing

tfTeldSendQueueBytes

#include <trsocket.h>

```
int                tfTeldSendQueueBytes
(
ttUserTeldHandle   teldHandle
);
```

Function Description

This function returns the number of bytes currently in the TCP socket send queue for the Telnet session identified by *teldHandle*.

Parameters

Parameter	Description
teldHandle	Telnet handle, unique identifier for a telnet connection.

Returns

Value	Meaning
>= 0	The number of bytes currently in the TCP socket send queue for the specified Telnet session.
-1	An error occurred. Invalid telnet handle specified.

tfTeldSendQueueSize

```
#include <trsocket.h>

int                tfTeldSendQueueSize
(
ttUserTeldHandle   teldHandle
);
```

Function Description

This function returns the total size of the TCP socket send queue for the Telnet session identified by *teldHandle*.

Parameters

Parameter	**Description**
teldHandle	Telnet handle, unique identifier for a telnet connection.

Returns

Value	**Meaning**
>= 0	The total size of the TCP socket send queue for the specified Telnet session.
-1	An error occurred. Invalid telnet handle specified.

tfTeldUserClose

```
#include <trsocket.h>

int                 tfTeldUserClose
(
ttUserTeldHandle    teldHandle
);
```

Function description

This function allows the user to force the telnet server to close the specified telnet connection

Parameters

Parameter	Description
teldHandle	Telnet handle, Unique identifier for a telnet connection

Returns

Value	Meaning
TM_ENOERROR	Success
TM_EINVAL	Invalid Telnet Handle

tfTeldUserExecute

```
#include <trsocket.h>

int                 tfTeldUserExecute
(
void
);
```

Function description

The function executes the TELNET server (non blocking mode only) main loop. This is to be used only if **tfTeldUserStart** had been called in non-blocking mode.

Parameters

None

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EPERM	Telnet server currently executing, or had not been started, or has been stopped

tfTeldUserSend

```
#include <trsocket.h>

int                 tfTeldUserSend
(
ttUserTeldHandle  teldHandle,
char *              teldSendBufPtr,
int                 teldBytes,
int                 flag
);
```

Function description

This function is called to give data to the telnet server to send it to the telnet client.

Flag values:

If the *flag* value is TM_TELD_SEND_EOL, it indicates that end of command has been reached and means that the telnet server needs to append <CR, LF> in ASCII mode, or <IAC, EOR> in binary mode to the user data. If the flag value is TM_TELD_SEND_COMMAND, it indicates that the user is sending a TELNET IAC command, and that the Turbo Treck Telnet server should treat the data as a command, and not map the IAC character.

Flow control:

If the Turbo Treck server does not have enough room for the data, it will not copy the data and return TM_EWOULBLOCK. The caller should try and re-send the data at a later time.

Parameters

Parameter	**Description**
teldHandle	Telnet handle, Unique identifier for a telnet connection
teldSendBufPtr	Pointer to user send buffer containing the data.
teldBytes	Number of bytes in *teldSendBuf* to be sent
flag	Flag: TM_TELD_SEND_EOL to indicate that an EOL should be sent TM_TELD_SEND_COMMAND to indicate that the user is sending a Telnet IAC command, and not data

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_ENOBUFS	Failed to allocate a buffer to copy the send data into
TM_EINVAL	Invalid Telnet Handle
TM_EWOULDBLOCK	User did not have enough room for all the data

tfTeldUserStart

```
#include <trsocket.h>

int                  tfTeldUserStart
(
int                  telnetOptionsAllowed,
int                  maxConnections,
int                  maxBackLog,
int                  blockingState
);
```

Function description

This function opens a TELNET server socket and start listening for incoming connections.

tfTeldUserStart can be either blocking or non-blocking, as specified by its blockingState parameter.

Blocking Mode

In blocking mode, **tfTeldUserStart** is to be called from a task. **tfTeldUserStart** will not return unless an error occurs. It will block and wait for incoming connections, and execute the telnet server code in the context of the calling task. Choose the blocking mode, if you are using an RTOS/Kernel.

Non-Blocking Mode

In non-blocking mode**, tfTeldUserStart** will return immediately after listening for incoming connections. It is the user's responsibility to then call **tfTeldUserExecute** periodically to execute the telnet server code. Choose the non-blocking mode, if you do not have an RTOS/Kernel.

Parameters

Parameter	**Description**
telnetOptionsAllowed	Indicates which options we allow the client to negotiate. The user can OR together TM_TELD_BINARY_ON and TM_TELD_ECHO_ON to allow binary transfer, and echo by the server respectively.
maxConnections	Maximum number of concurrent accepted incoming telnet connections allowed. If zero, then the telnet server will accept as many connections as there are available sockets.
maxBackLog	Maximum number of concurrent pending (before being accepted) incoming telnet connections allowed.
blockingState	TM_BLOCKING_ON for blocking mode, TM_BLOCKING_OFF for non-blocking mode.

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EINVAL	Incorrect telnet options flag, maxConnections is either negative (if non-zero exceeds or equals the current number of available telnet connections), or blockingState is neither TM_BLOCKING_ON, nor TM_BLOCKING_OFF
TM_EALREADY	**tfTeldUserStart** has already been called
TM_EMFILE	There are no more socket available to open the telnet server listening socket
TM_ENOBUFS	Insufficient user memory available to complete the operation
TM_EADDRINUSE	The telnet server port is already in use.
TM_ENOMEM	Could not obtain a counting semaphore to be used for blocking the telnet server (blocking mode only).

tfTeldUserStop

```
#include <trsocket.h>

int                  tfTeldUserStop
(
void
);
```

Function description

This function stops execution of the telnet server by closing the listening socket and killing all existing connections.

Parameters

None

Returns

Value	Meaning
TM_ENOERROR	Success
TM_EALREADY	The telnet server has already been stopped

Turbo Treck Test Suite

Description

The Turbo Treck stack includes a module that allows you to perform a variety of tests on your system. These tests are outlined below:

UDP Tests

TM_TEST_UDP_SEND	Sends UDP data to a remote server.
TM_TEST_UDP_RECV	Receives data from a remote client.
TM_TEST_UDP_ECHO_CLIENT	Sends UDP data to a remote echo server, and waits for the server to echo it back.
TM_TEST_UDP_ECHO_SERVER	Receives UDP data from a remote client, and echoes this data back to the client

TCP Tests

TM_TEST_TCP_SEND	Sends TCP data to a remote server.
TM_TEST_TCP_RECV	Receives TCP data from a remote client.
TM_TEST_TCP_ECHO_CLIENT	Connects to a remote server, and sends data to it, waiting for the server to echo it back.
TM_TEST_TCP_ECHO_SERVER	Accepts connections from remote peers. Receives TCP data and echoes it back to the client
TM_TEST_TCP_CONNECT	Repeatedly connects, and then disconnects from a TCP server.

Miscellaneous Tests

TM_TEST_LOCK	Verify that the Turbo Treck locking mechanism are set up and are functioning properly.

The behavior of these tests is modified by various flags:

TM_TEST_FLAG_ZEROCOPY	Send and receive data using the Turbo Treck zero copy extensions rather than the standard sockets calls (eg, **tfZeroCopySend** rather than **send**)
TM_TEST_FLAG_NONBLOCKING	Causes the test suite to run in non-blocking mode. This should be used if no kernel is available on your system.
TM_TEST_FLAG_UDP_CONNECT	Rather than using the standard UDP (**sendto**, **recvfrom**) API, call 'connect' and use the TCP calls (eg, **send**, **recv**).

TM_TEST_FLAG_UDP_CS_OFF
Disable UDP checksums. Checksums are enabled by default.

TM_TEST_FLAG_TCP_NODELAY
Sets the TCP_NO_DELAY option for this test. This causes TCP to send data immediately rather than delaying until more data is sent.

TM_TEST_FLAG_FILL_DATA
This flag causes all outgoing data to be set to a repeating incremental pattern (0x00, 0x01, etc) before being sent.

TM_TEST_FLAG_VALIDATE
Verify that all incoming data is as expected (ie, as TM_TEST_FLAG_FILL_DATA sets it – see above).

TM_TEST_FLAG_RANDOM
Choose random values for various parameters.

Blocking Mode

The Turbo Treck test suite may be run in either blocking or non-blocking mode. In blocking mode, the **tfTestTreck** function will return either when the test has successfully completed, or when an error has occurred. In non-blocking mode, **tfTestTreck** simply initiates the test. This function returns immediately, returns a session handle to the user, and returns the result code TM_EWOULDBLOCK.

After the test is started, the user should call **tfTestUserExecute** in their main loop to execute an iteration of the test. This function will return TM_EWOULDBLOCK if the test is still in progress; any other return code indicates that the test is complete and the user should no longer call **tfTestUserExecute**.

For instance, a non-blocking test’s main loop may look something like:

```
ttUserTestHandle testHandle;
int                 testStarted;

testStarted = 0;
while (1)
{

    if (testStarted == 0)
    {

/*
 * Attempt to send 25 UDP buffers, of 500 bytes each, to
 * remote host 10.0.1.5 at port 9.  This should be done in
 * non-blocking mode, * using the Turbo Treck Zero
 * Copy socket extensions.
 */
        errorCode = tfTestTreck(TM_TEST_UDP_SEND,
                 “10.0.1.5”,
                                      htons(9),
                                      500,
                                      25,
                                      TM_TEST_FLAG_NONBLOCKING |
                                             TM_TEST_FLAG_ZEROCOPY,
                                       &testHandle,
                                       0);

        testStarted = 1;
    }
    else
    {
        errorCode = tfTestUserExecute(testHandle);
        if (errorCode != TM_EWOULDBLOCK)
        {
/* TEST COMPLETE! */
          }
    }
    tfTimerExecute();

 if (tfCheckInterface(interfaceHandle) == TM_ENOERROR)
    {
        tfRecvInterface(interfaceHandle);
    }
}
```

Data validation

There are two flags that can be used to validate that incoming and outgoing data is correct. TM_TEST_FLAG_FILL_DATA will fill any outgoing data with any incrementing byte pattern of 0x00, 0x01 … 0xFF. The remote host receiving data can then validate that the incoming data matches this pattern.

The flag TM_TEST_FLAG_VALIDATE performs the inverse operation: it verifies that any received data matches this 0x00, 0x01 … 0xFF pattern. If any validation fails, the test will return with a TM_EIO error code.

These two flags can be used in a variety of ways to validate data. For instance, if an echo test is being performed, setting both of these flags will verify that both incoming and outgoing data is correct. Another example application of these flags is when the Turbo Treck test suite is being run between two devices, one device could run the TM_TEST_TCP_SEND test with the TM_TEST_FLAG_FILL_DATA option set, and the other device could run the TM_TEST_TCP_RECV test with the TM_TEST_FLAG_VALIDATE option set, and vice versa.

Random testing mode

When the TM_TEST_FLAG_RANDOM option is set, **tfTestTreck** will execute a random client test, with random parameters. The test type is chosen from the set of send send tests (TM_TEST_UDP_SEND, TM_TEST_TCP_SEND), echo tests (TM_UDP_ECHO_CLIENT, TM_TCP_ECHO_CLIENT) and the TCP connect test (TM_TEST_TCP_CONNECT). If a send or TCP connect test is chosen, it will send data to port 9 of the remote host; if an echo test is chosen it

will echo data with port 7. The test will be executed the number of times specified in the parameter *testCount*.
A random data length is also chosen and is no larger than the data length specified in parameter *dataSize*. Unless IP fragmentation is enabled, the data length should be no larger than the MTU of the device, less the space for the UDP & IP headers (1472 for Ethernet, as well as most other link layers) in case a UDP test is chosen.

Random options (flags) are chosen appropriately for the selected test. The exception to this is the TM_TEST_FLAG_NONBLOCKING flag which remains set to the value passed in by the user.

Locking test

This test operates differently than the rest of the test suite, due to the way that locks work. This test is only run once. If locks are working correctly, the user will receive a message via **tfKernelWarning** and if locks fail, **tfKernelError** will be called. For this test to operate, the macro TM_ERROR_CHECKING must be defined in your **trsystem.h** file.

tfTestTreck

```
#include <trsocket.h>

int                     tfTestTreck
(
int                     testType,
char                    ipAddrStr,
int                     portNumber,
int                     dataSize,
unsigned long           testCount,
unsigned long           flags,
ttUserTestHandlePtr     sessionHandlePtr,
void                    testParam
);
```

Function description
Performs a network test according to the specified parameters. Not all parameters are meaningful for all tests. For instance, dataSize has no meaning when the test type is TM_TEST_TCP_CONNECT since no data is being sent and testType is not used when the TM_TEST_FLAG_RANDOM option is chosen since the test type is selected at random.

Parameters

Parameter	**Description**
testType	The type of test to perform (see above).

ipAddrStr	String containing the IP address of the remote host (eg, “10.0.1.5”)
portNumber	Port on the remote host to perform the test against.
dataSize	Data buffer size for this test (eg, the size of the UDP datagram to send, or the amount of data to receive)
testCount	The number of times to execute this test
flags	Options to apply to this test (see above)
sessionHandlePtr	Pointer to a test session handle. Used in non-blocking mode to return a session handle to be used with tfTestUserExecute. Not used in blocking mode.
testParam	Not currently used.

Returns

Value	**Meaning**
TM_EINVAL	Invalid test type specified
TM_EINVAL	No session handle pointer specified in non-blocking mode.
TM_ENOERROR	Test completed successfully
TM_EWOULDBLOCK	Returned when the test is run in non-blocking mode. The user should call **tfTestUserExecute** with the value specified in **sessionHandlePtr* to continue the test
TM_EIO	Incoming data failed validation

tfTestUserExecute

#include <trsocket.h>

```
int                 tfTestUserExecute
(
ttUserTestHandle    testSessionHandle
);
```

Function description

Called in non-blocking mode to execute an iteration of the current test using the session handle returned from **tfTestTreck**. If **tfTestUserExecute** returns TM_EWOULDBLOCK, the test has not yet completed; any other return value indicates that the test is complete and **tfTestUserExecute** should not be called with this session handle again. This function should never be called in blocking mode.

Parameters

Parameter	**Description**
testSessionHandle	Session handle returned from **tfTestTreck**

Returns

Value	**Meaning**
TM_ENOERROR	The current test has completed successfully.
TM_EWOULDBLOCK	The current test has not yet completed. The user should call **tfTestUserExecute** again.
TM_EINVAL	Invalid session handle.
TM_EIO	Incoming data failed validation

Netstat

Introduction

The netstat tool outputs useful information retrieved from the stack, such as the routing table, the ARP table, the UDP socket table, and the TCP vector table. A user can pick the information he is interested in and output it in the prefered way through the call back functions.

The netstat API consists of a set of data structures, a function **tfNetStat()** that enumerates the table entries**,** a callback interfaces **ttNtEntryCBFuncPtr()** that handles each entry, and a set of helper functions. The helper functions facilitate the writing of user call back functions. They provide the table header string and convert a netstat entry into a string.

tfNtGetRteHeaderStr()
tfNtGetArpHeaderStr()
tfNtGetTcpHeaderStr()
tfNtGetUdpHeaderStr
tfNtRteEntryToStr()
tfNtArpEntryToStr()
tfNtTcpEntryToStr()
tfNtUdpEntryToStr()

These data structures are defined for netstat.

ttNtRteEntry
ttNtArpEntry
ttNtTcpEntry
ttNtUdpEntry
ttNtEntryU

Note: In order to use the netstat tool, TM_USE_NETSTAT must be #defined in trsystem.h

Public API

tfNetStat

```
int                     tfNetStat
(
ttNtTableId             tableId,
ttNtEntryCBFunctPtr     ntEntryCBFuncPtr
ttUserGenericUnion      genParam1
ttUserGenericUnion      genParam2
)
```

Function description

This function walks through the Treck internal data**.** For each entry found, a *ttNtEntryU* structure is prepared and passed to the provided *ntEntryCBFuncPtr* together with the *genParam1* and *genParam2.* The function walk continues until *ntEntryCBFuncPtr* returns a value other than TM_ENOERROR, or when there is no more entry.

Parameters

Parameter	**Description**
tableId	specifies the table that the user would like to walk/output. Valid values are : TM_NT_TABLE_RTE TM_NT_TABLE_ARP TM_NT_TABLE_UDP TM_NT_TABLE_TCP
ntEntryCBFuncPtr,	Call back function for handling each of the entry. The user must implement this call back function.
genParam1	Generic parameter, for passing information on to the callback functions.
genParam2	Generic parameter, for passing information on to the callback functions.

Returns

Value	**Meaning**
TM_ENOERROR	Competed successfully
TM_ENOENT	Call back function failed or told tfNetStat to stop.
TM_EINVAL	One of the passed in parameter is invalid

ttNtEntryCBFuncPtr

```
int                     (*ttNtEntryCBFuncPtr)
(
ttNtEntryUPtr           NtEntryPtr
ttUserGenericUnion      genParam1
ttUserGenericUnion      genParam2
)
```

Description

This is a callback interface definition, it's passed as the second parameter to **tfNetStat(), tfNetStat** calls this function back for handling each of the entries found.

Parameters

Parameter	**Description**
NtEntryPtr	Points to a union holding the information of an entry. The function needs to access the appropriate union member to get the entry.
genParam1	Generic parameter passed into tfNetStat
genParam2	Generic parameter passed into tfNetStat

Returns

Value	**Meaning**
TM_ENOERROR	tell tfNetStat to continue
TM_ENOENT	tell tfNetStat to stop

tfNtGetRteHeaderStr

```
char  *          tfNtGetRteHeaderStr
(
char  *          buffer
int  *           sizePtr
)
```

Description

Get a string of the routing table header with major fields

Parameters

Parameter	**Description**
Buffer	buffer to hold the result string. Its size should be TM_NT_ENTRY_STR_LEN. bytes
SizePtr	In: pointer to the size of the buffer (TM_NT_ENTRY_STR_LEN) Out: pointer to the actual size of the result string

Returns

Value	**Meaning**
Valid pointer	Pointer to the result string
NULL	One of the parameters is bad

tfNtGetArpHeaderStr

```
char  *          tfNtGetArpHeaderStr
(
char  *          buffer
int  *           sizePtr
)
```

Description

Get a string as the ARP table header with major fields

Parameters

Parameter	**Description**
Buffer	buffer to hold the result string. Its size should be TM_NT_ENTRY_STR_LEN.
SizePtr	In: pointer to the size of the buffer (TM_NT_ENTRY_STR_LEN) Out: pointer to the actual size of the result string

Returns

Value	**Meaning**
Valid pointer	Pointer to the result string
NULL	One of the parameters is bad

tfNtGetTcpHeaderStr

```
char  *          tfNtGetTcpHeaderStr
(
char  *          Buffer
int  *           SizePtr
)
```

Description

Get a string as the TCP socket table header with major fields

Parameters

Parameter	**Description**
buffer	buffer to hold the result string. Its size should be TM_NT_ENTRY_STR_LEN.
sizePtr	In: pointer to the size of the buffer (TM_NT_ENTRY_STR_LEN) Out: pointer to the actual size of the result string

Returns

Value	**Meaning**
Valid pointer	Pointer to the result string
NULL	One of the parameters is bad

tfNtGetUdpHeaderStr

```
char  *          tfNtGetUdHeaderStr
(
char  *          Buffer
int  *           SizePtr
)
```

Description

Get a string as the UDP socket table header with major fields

Parameters

Parameter	**Description**
buffer	buffer to hold the result string. Its size should be TM_NT_ENTRY_STR_LEN.
sizePtr	In: pointer to the size of the buffer (TM_NT_ENTRY_STR_LEN) Out: pointer to the actual size of the result string

Returns

Value	**Description**
Valid pointer	Pointer to the result string
NULL	One of the parameters is bad

tfNtRteEntryToStr

```
char  *           tfNtRteEntryToStr
(
ttNtRteEntryPtr   ntRteEntryPtr
char  *           Buffer
int  *            SizePtr
)
```

Description

Converts a routing entry into a string that has the major fields in.

Parameters

Parameter	**Description**
ntRteEntryPtr	Routing entry to be converted
buffer	buffer to hold the result string. Its size should be TM_NT_ENTRY_STR_LEN.
sizePtr	In: pointer to the size of the buffer (TM_NT_ENTRY_STR_LEN) Out: pointer to the actual size of the result string

Returns

Value	**Meaning**
Valid pointer	Pointer to the result string
NULL	One of the parameters is bad

tfNtArpEntryToStr

```
char  *          tfNtArpEntryToStr
(
ttNtArpEntryPtr  NtArpEntryPtr
char  *          Buffer
int  *           SizePtr
)
```

Description

Converts a ARP entry into a string that has the major fields in.

Parameters

Parameter	**Description**
ntArpEntryPtr	ARP entry to be converted
buffer	buffer to hold the result string. Its size should be TM_NT_ENTRY_STR_LEN.
sizePtr	In: pointer to the size of the buffer (TM_NT_ENTRY_STR_LEN) Out: pointer to the actual size of the result string

Returns

Value	**Description**
Valid pointer	Pointer to the result string
NULL	One of the parameters is bad

tfNtTcpEntryToStr

```
char  *          tfNtTcpEntryToStr
(
ttNtTcpEntryPtr  NtTcpEntryPtr
char  *          buffer
int  *           sizePtr
)
```

Description

Converts a TCP entry into a string that has the major fields in.

Parameters

Parameter	**Description**
ntTcpEntryPtr	TCP entry to be converted
buffer	buffer to hold the result string. Its size should be TM_NT_ENTRY_STR_LEN.
sizePtr	In: pointer to the size of the buffer (TM_NT_ENTRY_STR_LEN) Out: pointer to the actual size of the result string

Returns

Value	**Meaning**
Valid pointer	Pointer to the result string
NULL	One of the parameters is bad

tfNtUdpEntryToStr

```
char  *            tfNtUdpEntryToStr
(
ttNtUdpEntryPtr    ntUdpEntryPtr
char  *            buffer
int  *             sizePtr
)
```

Description

Converts a UDP entry into a string that has the major fields in

Parameters

Parameter	description
ntUdpEntryPtr	UDP entry to be converted
buffer	buffer to hold the result string. Its size should be TM_NT_ENTRY_STR_LEN.
sizePtr	In: pointer to the size of the buffer (TM_NT_ENTRY_STR_LEN)Out: pointer to the actual size of the result string

Returns

Return value	**Description**
Valid pointer	Pointer to the result string
NULL	One of the parameters is bad

Data structures

These data structures holds a mapped copy of the internal stack data, they are only for the output purpose, modifying these data has no effect on the internal data.

ttNtRteEntry

To hold the information of a routing entry.

```
typedef struct tsNtRteEntry
{
    struct sockaddr_storage   ntRteDestSockAddr;
    ttUser8Bit    ntRtePrefixLength;
    ttUser32Bit   ntRteOwnerCount;
    ttUser32Bit   ntRteMhomeIndex;
    char          ntRteDeviceName
                  [(((TM_MAX_DEVICE_NAME + 3) / 4) * 4)];
    struct sockaddr_storage  ntRteGwSockAddr;
    ttUser8Bit    ntRteHwAddress
                  [(((TM_MAX_PHYS_ADDR + 3)/4)*4)];
    int           ntRteHwLength;
    ttUser8Bit    ntRteClonePrefixLength;
    ttUser32Bit   ntRteMtu;
    ttUser32Bit   ntRteHops;
    ttUser32Bit   ntRteTtl;
    ttUser16Bit   ntRteFlags;
} ttNtRteEntry;
typedef ttNtRteEntry  * ttNtRteEntryPtr;
```

Member	Description
ntRteDestSockAddr	ntRteDestSockAddr.ss_family indicates the family of the address.ntRteDestSockAddr.addr.ipv6 or ntRteDestSockAddr.addr.ipv4 has the actual key IP Address.
ntRtePrefixLength	Matching prefix length associated with the routing entry. [1-128] for IPV6 and [1-32] for IPV4 entries.
ntRteOwnerCount	Owner count for the routing entry.
ntRteMhomeIndex	Multihome index in device entry
ntRteDeviceName	Name of the device associated with the routing entry.

ntRteGwSockAddr For gateway route, this is the address of the gateway . For direct routes, this address is set to the IP address of the corresponding interface just to indicate that it is a direct route. This follows the UNIX convention. ntRteGwSockAddr.ss_family indicates the family of the address.ntRteGwSockAddr.addr.ipv6 or ntRteGwSockAddr.addr.ipv4 has the actual IP Address of the gateway.

ntRteHwAddress This field is valid if the routing entry is a local route, in which case it is the hardware address of the target host.

ntRteHwLength This field is valid if the routing entry is a local route, in which case it is thelength of the hardware address of the target host

ntRteClonePrefixLength Prefix length for cloning a local host routing entry.

ntRteMtu Mtu for the routing entry

ntRteHops Maximum hops for the routing entry.

ntRteTtl Lifetime of the routing entry, in milli-seconds. an special value is reserved for “infinite”, TM_RTE_INF (as defined in trsocket.h)

ntRteFlags Routing flags, see the table for routing flags

Routing flags

The ntRteFlags can be an ored value of one or more of the following value:

Possible values of ntRteFlags	**symbol**	**Description**
TM_NT_RTE_UP	U	route is up, if not set, route is down
TM_NT_RTE_HOST	H	target is a host, if not set, target is a network
TM_NT_RTE_GW	G	Use gate way, if not set, target is directly connected.
TM_NT_RTE_DYNAMIC	D	dynamically installed by daemon or redirect, if not set, the routing entry is created staticly
TM_NT_RTE_REJECT	J	output through this route rejected
TM_NT_RTE_CLONABLE	C	Clonable entry
TM_NT_RTE_CLONED	L	Cloned entry
TM_NT_RTE_STATIC	S	Static entry configured by user

ttNtArpEntry

Description

Holds the information of an ARP entry for printing.

```
typedef struct tsNtArpEntry
{
    struct sockaddr_storage    ntArpSockAddr;
    ttUser8Bit    ntArpHwAddress
                  [(((TM_MAX_PHYS_ADDR + 3)/4)*4)];
    int           ntArpHwType;
    int           ntArpHwLength;
    ttUser8Bit    ntArpDeviceName
                  [(((TM_MAX_DEVICE_NAME + 3)/4)*4)];
} ttNtArpEntry;
typedef ttNtArpEntry  * ttNtArpEntryPtr;
```

Member	**Description**
NtArpSockAddr	Destination IP address for the ARP entry, ntArpSockAddr.ss_family indicates the family of the address.ntArpSockAddr.addr.ipv6 or ntArpSockAddr.addr.ipv4 has the actual key IP Address.
NtArpHwAddress	Array of unsigned char as the physical address
NtArpHwLength	Length of the physical address, in bytes
NtArpHwType	Can be one of the below: TM_NT_HWTYPE_ETHERNET TM_NT_HWTYPE_TOKENRING TM_NT_HWTYPE_TOKENBUS
NtArpDeviceName	Name of the device associated with the ARP entry.

ttNtTcpEntry

Holds the information of a TCP entry for printing.

```
typedef struct tsNtTcpEntry
{
   ttUser16Bit                   ntTcpSockDesc;
   ttUser32Bit                   ntTcpBytesInRecvQ;
   ttUser32Bit                   ntTcpBytesInSendQ;
   ttUser32Bit                   ntTcpRecvQSize;
   ttUser32Bit                   ntTcpSendQSize;
   struct sockaddr_storage       ntTcpLocalSockAddr;
   ttUser16Bit                   ntTcpOwnerCount;
   struct sockaddr_storage       ntTcpPeerSockAddr;
   ttUser16Bit                   ntTcpFlags;
   ttUser32Bit                   ntTcpRto;
   ttUser32Bit                   ntTcpReXmitCnt;
   ttUser32Bit                   ntTcpSndUna;
   ttUser32Bit                   ntTcpIss;
   ttUser32Bit                   ntTcpSndNxt;
   ttUser32Bit                   ntTcpMaxSndNxt;
   ttUser32Bit                   ntTcpIrs;
   ttUser32Bit                   ntTcpSndWL1;
   ttUser32Bit                   ntTcpSndWL2;
   ttUser32Bit                   ntTcpMaxSndWnd;
   ttUser32Bit                   ntTcpRcvNxt;
   ttUser32Bit                   ntTcpRcvWnd;
   ttUser32Bit                   ntTcpRcvAdv;
   ttUser32Bit                   ntTcpCwnd;
   ttUser32Bit                   ntTcpSsthresh;
   ttUser16Bit                   ntTcpBackLog;
   ttUser8Bit                    ntTcpDupAck;
   ttUser8Bit                    ntTcpAcksAfterRexmit;
   ttUser8Bit                    ntTcpState;
   ttUser8Bit                    ntTcpFiller[3];
} ttNtTcpEntry;
typedef ttNtTcpEntry  * ttNtTcpEntryPtr;
```

Many of this structure's members follow RFC 793 section 3.2, "**Terminology".** Where applicable, the original name used in the RFC is indicated. Users are referred to RFC 793 for more information on these variables.

Member	**Description**
ntTcpSockDesc;	Socket descriptor, integer
ntTcpBytesInRecvQ;	Data in the receive queue, in bytes
ntTcpBytesInSendQ;	Data in the send queue, in bytes
ntTcpRecvQSize;	Size of the receive queue, in bytes
ntTcpSendQSize;	Size of the send queue, in bytes
ntTcpLocalSockAddr;	Local address, port and protocol family information of the socket.
ntTcpOwnerCount;	Owner count of the socket.
ntTcpPeerSockAddr;	Peer's address, port and protocol family information of the socket.
ntTcpFlags;	TCP flags
ntTcpRto;	Retransmission timeout
ntTcpReXmitCnt;	number of retransmitted segments
ntTcpSndUna;	Send Sequence Variables: SND.UNA - send unacknowledged
ntTcpIss;	Send Sequence Variables: ISS - initial send sequence number
ntTcpSndNxt;	Send Sequence Variables: SND.NXT - send next
ntTcpMaxSndNxt;	Highest sequence number sent (used because of retransmit)
ntTcpIrs;	Receive Sequence Variables:IRS - initial receive sequence number
ntTcpSndWL1;	Send Sequence Variables: SND.WL1 - segment sequence number used for last window update SND.
ntTcpSndWL2;	Send Sequence Variables: SND.WL2 - segment acknowledgment number used for last window update
ntTcpMaxSndWnd;	Maximum send window seen so far
ntTcpRcvNxt;	Receive Sequence VariablesRCV.NXT - receive next
ntTcpRcvWnd;	Receive Sequence Variables:RCV.WND - receive window
ntTcpRcvAdv;	Sequence number of right edge of advertized window. (Could be smaller than Rcv.Nxt+Rcv.Wnd to avoid silly window syndrome. Also used to prevent shrinkage of our receive window.)
ntTcpCwnd;	Send congestion window

ntTcpSsthresh;	Send slow start threshold size
ntTcpBackLog;	Back logs
ntTcpDupAck;	Duplicate ACKs
ntTcpAcksAfterRexmit;	number of non duplicate acks after duplicate ACK(s)
ntTcpState	One of the following: TM_NT_TCPS_CLOSED TM_NT_TCPS_LISTEN TM_NT_TCPS_SYN_SENT TM_NT_TCPS_SYN_RECEIVED TM_NT_TCPS_ESTABLISHED TM_NT_TCPS_CLOSE_WAIT TM_NT_TCPS_FIN_WAIT_1 TM_NT_TCPS_CLOSING TM_NT_TCPS_LAST_ACK TM_NT_TCPS_FIN_WAIT_2 TM_NT_TCPS_TIME_WAIT TM_NT_TCPS_INVALID

ttNtUdpEntry

Description

Holds the information of a UDP entry for printing.

```
typedef struct tsNtUdpEntry
{
    ttUser16Bit                  ntUdpSockDesc;
    ttUser32Bit                  ntUdpBytesInRecvQ;
    ttUser32Bit                  ntUdpBytesInSendQ;
    ttUser32Bit                  ntUdpRecvQSize;
    ttUser32Bit                  ntUdpSendQSize;
    struct sockaddr_storage      ntUdpLocalSockAddr;
    ttUser16Bit                  ntUdpOwnerCount;
} ttNtUdpEntry;
typedef ttNtUdpEntry  * ttNtUdpEntryPtr;
```

Member	**Description**
ntUdpSockDesc	Socket descriptor, integer
ntUdpBytesInRecvQ	Data in the receive queue, in bytes
ntUdpBytesInSendQ	Data in the send queue, in bytes
ntUdpRecvQSize	Size of the receive queue, in bytes
ntUdpSendQSize	Size of the send queue, in bytes
ntUdpLocalSockAddr	Address, port and protocol family information of the socket.
ntUdpOwnerCount	Owner count of the socket.

ttNtEntryU

Description

A union to hold any of the netstat entries.

```
typedef union tuNtEntryU
{
    ttNtRteEntry  ntRteEntry;
    ttNtArpEntry  ntArpEntry;
    ttNtUdpEntry  ntUdpEntry;
    ttNtTcpEntry  ntTcpEntry;
} ttNtEntryU;
typedef ttNtEntryU  * ttNtEntryUPtr;
```

Member	**Description**
ntRteEntry	route entry information
ntArpEntry	ARP entry informaiton
ntTcpEntry	TCP socket information
ntUdpEntry	UDP socket information

Examples

As examples, we provide a set of call back functions in txntstat.c in the examples directory of the distribution. One of them uses printf to output the tables to the standard console, since printf is non-reentrant and therefore should never be used in a preemptive enviroment.

tfxNtArpEntryCBPrintf()
tfxNtRteEntryCBPrintf()
tfxNtTcpEntryCBPrintf()
tfxNtUdpEntryCBPrintf()

For example, **tfxNtTcpEntryCBPrintf** is defined as below:

```
int tfxNtTcpEntryCBPrintf(
    ttNtEntryUPtr    ntEntryUPtr,
    ttUserGenericUnion  genParam1,
    ttUserGenericUnion  genParam2)
{

    char    buffer[TM_NT_ENTRY_STR_LEN];
    int     entryStrLen;

    genParam1 = genParam1;
    genParam2 = genParam2;
    entryStrLen = TM_NT_ENTRY_STR_LEN;

    if(tfNtTcpEntryToStr(&ntEntryUPtr->ntTcpEntry,
buffer,  &entryStrLen) != NULL)
    {
        printf(“%s\n”, buffer);
    }

/*
 * return TM_ENOERROR to ask tfNetStat to call back
with more entries
 * return TM_ENOENT to ask tfNetStat to stop calling
back
 */
    return TM_ENOERROR;
}
```

Below is a example of how to use tfNetStat() and the call back function defined above to print out the TCP socket table

```
#include <trsocket.h>
main()
{
    char                    buffer[TM_NT_ENTRY_STR_LEN];
    int                     headerStrLen;
    ttUserGenericUnion      genUnion;


…
/* print the TCP socket table using netstat tools */
  printf(“\n——————\nTCP sockets\n——————\n”);
  headerStrLen = TM_NT_ENTRY_STR_LEN;
  if(tfNtGetTcpHeaderStr(buffer,&headerStrLen)!= NULL)
   {
       printf(“%s\n”, buffer);
   }
   errorCode = tfNetStat(
                  TM_NT_TABLE_TCP,
                  tfxNtTcpEntryCBPrintf,
                  genUnion,
                  genUnion);
}
```

Optional Protocols

Optional Protocols

Function List

AutoIP Configuration

tfAutoIPPickIpAddress
tfCancelCollisionDetection
tfConfigAutoIp
tfUseCollisionDetection
tfUserStartArpSend

BOOTP Automatic Configuration

tfConfGetBootEntry
tfUseBootp

BOOTP relay agent

tfStartBootRelayAgent
tfStopBootRelayAgent

DHCP Automatic Configuration

tfConfGetBootEntry

tfDhcpConfSet

tfUseDhcp

DHCP User configured

tfDhcpUserGetBootEntry
tfDhcpUserRelease
tfDhcpUserSet
tfDhcpUserStart

Dialer

tfDialerAddExpectSend
tfDialerAddSendExpect
tfUseDialer

IGMP API

drvIoctlFunc
tfSetMcastInterface

NAT

tfNatConfig
tfNatUnConfig
tfNatConfigNapt
tfNatConfigInnerTcpServer
tfNatConfigInnerUdpServer
tfNatConfigInnerFtpServer
tfNatConfigStatic
tfNatConfigDynamic
tfNatConfigMaxEntries
tfNatDump

PPP Interface

tfChapRegisterAuthenticate
tfGetPppDnsIpAddress
tfGetPppPeerIpAddress
tfPapRegisterAuthenticate
tfPppSetOption
tfSetPppPeerIpAddress
tfUseAsyncPpp
tfUseAsyncServerPpp
tfUsePppLqm
tfFreePppLqm
tfLqmRegisterMonitor
tfLqmSendLinkQualityReport
tfPppSendEchoRequest
tfLqmSetLqrTimerPeriod
tfLqmGetLocalLqrTimerPeriod
tfLqmGetPeerLqrTimerPeriod
tfMsChapRegisterAuthenticate
tfMsChapRegisterNewPassword
tfPppSetAuthPriority

AUTO IP Configuration

Description

The AUTO IP configuration APIs allow the user to configure the interface with an AUTO IP address, without having to pick a specific IP address. The IP address will automatically be selected in the AUTO IP v4 address range, i.e. IP addresses between 169.254.1.0, and 169.254.254.255.
To configure an interface with an AUTO IP address:

1. ***Starting the configuration:*** The user need to call **tfOpenInterface**, using a zero IP address, zero IP netmask, and setting the TM_DEV_IP_USER_BOOT flag in the flags parameter, as shown in the example below. Note that if the interface had already been opened on multi home index 0, and not closed, this step can be omitted.
2. ***Selecting an IP address, and starting the collision detection:*** Next the user need to call **tfAutoIpPickIpAddress** to pick an AUTO IP address, then call **tfUseCollisionDetection**, passing that IP address, and a call back function, information that will be stored in the Turbo Treck stack. Next the user need to call **tfUserStartArpSend** so that the Turbo Treck stack can start sending ARP probes, to check for collisions on the IP address. **tfConfigAutoIp**, provided as an example in *examples\txautoip.c*, and described below does all this.
3. ***Finishing the configuration, or trying another IP address:*** The call back function will be called if either a collision has been detected (non zero errorCode), or when the timeout for sending the probes has expired with no collision (zero errorCode). If no collision has been detected, then the user should call **tfFinishOpenInterface** with the selected AUTO IP address, if **tfOpenInterface** with the TM_DEV_IP_USER_BOOT flag had been called earlier (mhome index 0 configuration, as shown in step 1), otherwise **tfConfigInterface** should be called instead. If a collision has been detected, then the user should cancel the collision detection (by calling **tfCancelCollisionDetection** on the current IP address), and try another IP address, by calling **tfConfigAutoIp** or similar function again. The **tfAutoIpFinish** call back function provided as an example in *examples\txautoip.c*, and shown below does all this. It also cancels the collision detection if no collision occurred.
4. ***Monitoring the network for collision after finishing the configuration:*** After the interface has been successfully configured, the user should still monitor the network for collision. This could be done by not canceling the collision detection in the call back function, and modifying the call back function to handle a collision detection, when the interface has been configured.

Enabling AUTO IP

The following macro must be uncommented out in trsystem.h so that the AUTO IP code can be enabled.

#define TM_USE_AUTO_IP

Example

The following code sample can also be found in *examples\txautoip.c*

```
...
  errorCode = tfOpenInterface(interfaceHandle, 0, 0,
                              TM_DEV_IP_USER_BOOT, 1, 0);

  if (errorCode == TM_ENOERROR)
  {

    errorCode = tfConfigAutoIp (interfaceHandle, 0);
  }
...

int tfConfigAutoIp( ttUserInterface interfaceHandle,
                    int             mhomeIndex )
{
    ttUserGenericUnion autoIpParam;
    ttUserIpAddress    ipAddress;
    int                errorCode;

    do
    {
/* Pick a random AUTO IP address */
        ipAddress = tfAutoIPPickIpAddress();
        if (ipAddress != (ttUserIpAddress)0)
        {
            autoIpParam.genIntParm = mhomeIndex;
/* Register the call back function for that IP address with
 * the stack
 */
            errorCode = tfUseCollisionDetection( ipAddress,
                                                  tfAutoIpFinish,
                                               autoIpParam );

        }
        else
        {
            errorCode = TM_ENOENT;
        }
    } while (errorCode == TM_EADDRINUSE);
    if (errorCode == TM_ENOERROR)
    {
/* Selected AUTO IP address is not in the ARP cache */
/* Start sending ARP probes on the interface */
/* We use the default probe interval (2s), and number of
 * probes (4)
 */
```

```
        errorCode = tfUserStartArpSend (interfaceHandle,
                                  ipAddress, 0, 0, 0);
    }
    return errorCode;
}

int tfAutoIpFinish ( ttUserInterface    interfaceHandle,
                     ttUserIpAddress    ipAddress,
                     int                errorCode,
                     ttUserGenericUnion autoIpParam )

{
    int mhomeIndex;

/* Cancel the collision detection check on that IP address */
    (void)tfCancelCollisionDetection(ipAddress);
    mhomeIndex = autoIpParam.genIntParm;
    if (errorCode == TM_ENOERROR)
    {
/* No collision occurred. Finish configuring the interface */
        tlConfigured = 1;
        if (mhomeIndex == 0)
        {
/* User used tfOpenInterface with TM_DEV_IP_USER_BOOT on
 * mhome index 0
 */
         errorCode = tfFinishOpenInterface( interfaceHandle,
                                            ipAddress,
                                           TM_IP_LOCAL_NETMASK);
            if (errorCode == TM_ENOERROR)
            {
                printf(“tfFinishOpenInterface with 0x%x\n”,
ipAddress);
            }
            else
            {
                printf(“tfFinishOpenInterface with 0x%x failed
‘%s’\n”,
                        ipAddress, tfStrError(errorCode));
            }
        }
        else
        {
/* User had already opened the interface on another mhome */
            errorCode = tfConfigInterface( interfaceHandle,
                                           ipAddress,
                                           TM_IP_LOCAL_NETMASK,
                                          0, /* not used. */
                                          1, /* has to be one */
                                          (unsigned
                                       char)mhomeIndex );

            if (errorCode == TM_ENOERROR)
            {
                printf(“tfConfigInterface with 0x%x\n”,
                   ipAddress);
            }
            else
            {
               printf("tfConfigInterface with 0x%x failed '%s'\n",
                      ipAddress, tfStrError(errorCode));
            }
        }
    }
    else
    {
/* A collision occurred on the IP address, try another IP
 * address
 */
       tfConfigAutoIp(interfaceHandle, mhomeIndex);
    }
    return 0;
}
```

tfAutoIPPickIpAddress

```
#include <trsocket.h>

ttUserIpAddress   tfAutoIPPickIpAddress
(
void
);
```

Description

Pick a valid AUTO IP v4 address in the AUTO IP IPv4 address pool, i.e. pick and IP address between 169.254.1.0, and 169.254.254.255 that has not been picked in the previous attempt to pick an IP address.

Parameters

None

Returns

Value	**Meaning**
Non zero IP address	Valid AUTO IP v4 address.
0	No valid AUTO IP v4 address was found.

tfCancelCollisionDetection

```
#include <trsocket.h>

int                tfCancelCollisionDetection
(
ttUserIpAddress    ipAddress
);
```

Function Description

Cancel a collision detection that had been registered with **tfUseCollisionDetection**.

The user will pass the IP address that is being checked for collision. **tfCancelCollisionDetection** will search the list of collision detection entries, and remove the matching entry and timer if any. Collision detection for that IP address will stop.

Parameters

Parameters	**Description**
ipAddress	IP address that is currently being checked for collision.

Returns

Value	**Meaning**
TM_ENOENT	No collision detection entry matching the IP address parameter has been found.
TM_ENOERROR	No error.

tfConfigAutoIp

```
#include <trsocket.h>

int                 tfConfigAutoIp
(
ttUserInterface     interfaceHandle
int                 mhomeIndex
);
```

Function Description

This function is provided as an example.
tfConfigAutoIp picks an AUTO IP v4 address, and calls **tfUseCollisionDetection** passing the selected IP address, a call back function (namely **tfAutoIpFinish**), with a call back function parameter, specifying the selected mhome index. If **tfUseCollisionDetection** is successful, that information will be stored in the Turbo Treck stack. Next it calls **tfUserStartArpSend** using the default values for the number of ARP probes (4), and ARP probe time interval (2 seconds), so that the Turbo Treck stack can start sending ARP probes, to check for collisions on the IP address.

Parameters

Parameters	**Description**
interfaceHandle	Interface on which to send the ARP probe(s)/request(s).
mhomeIndex	Multi home index of the IP address to be configured on the interface

Returns

Value	**Meaning**
TM_ENOENT	Could not pick a valid IP v4 IP address
TM_EALREADY	**tfUseCollisionDetection** has already been called for that IP address.
TM_ENOBUFS	Not enough memory to allocate a collision detection entry or a timer.
TM_EPERM	Interface is not a LAN interface, i.e. ARP not permitted on that interface.
TM_ENXIO	Interface has not been opened.
TM_ENOERROR	No error.

tfUseCollisionDetection

```
#include <trsocket.h>

int                    tfUseCollisionDetection
(
ttUserIpAddress        ipAddress,
ttArpChkCBFunc         userCbFunc,
ttUserGenericUnion     userCbParam
);
```

Function Description

Allow the Turbo Treck stack to check that no other host is using a given IP address. This command instructs the Turbo Treck stack to store the given IP address, so that it can later on check for ARP requests, or replies coming from any interface, not originated by this host, and whose ARP sender addresses are the same as the given IP address. The collision detection will only start after the user calls **tfUserStartArpSend**. The collision detection will continue until the user calls **tfCancelCollisionDetection**. If the interface given by **tfUserStartArpSend** is not configured yet, the Turbo Treck stack will also check for ARP probes sent by other hosts. The user gives a call back function to be called when a match on an ARP request, or ARP reply, or ARP probe, occurs. The interface handle, passed to the call back function, is the interface handle passed to the **tfUserStartArpSend** function. The call back function can also be called if the timeout given to the **tfUserStartArpSend** expires before an ARP request, or reply occurs.

The user will pass the IP address to check, a call back function, and a parameter to be passed as is to the call back function.

- **tfUseCollisionDetection** will first check the ARP cache for a match on the IP address to see whether another host is already using that IP address. If a match is found, it will return TM_EADDRINUSE, to indicate that another host is using the IP address. In that case the collision detection will stop right away.
- **tfUseCollistionDetection** will add an entry for that IP address in the global collision detection list. When incoming ARP requests, and replies come, the ARP sender IP address will be checked for a match with this entry in that list.
- If an ARP reply, or request, or probe (only if interface is not configured) is received, while the IP address is being checked for collision, then the Turbo Treck stack will call the call back function in the context of the recv task, with TM_EADDRINUSE error code.

- If the **tfUserStartArpSend** timeout expires before the Turbo Treck stack receives any ARP reply/request, then the Turbo Treck stack will call the call back function in the context of the timer task, with the TM_ENOERROR error code.
- Once started, the collision detection will stop only when the user calls **tfCancelCollisionDetection**.
- The user can call **tfUseCollisionDetection** at any time, after **tfStartTreck** has been called.

Parameters

Parameters	**Description**
ipAddress	IP address to check.
userCbFunc	Call back function. Called by the Turbo Treck stack when either an ARP reply or ARP request has been received whose source IP address matches the above IP address. Also called when a matching ARP probe is received while the interface has not been configured yet. Also called when the **tfUserStartArpSend** timeout expires. See userCbFunc section below.
userCbParam	Parameter to be passed as is to the user call back function. See below.

Returns

Value	**Meaning**
TM_EINVAL	userCbFunc is NULL.
TM_EADDRINUSE	Another host on the network has been configured with the IP address. (We found the ARP mapping in the ARP cache.)
TM_EALREADY	**tfUseCollisionDetection** has already been called for that IP address.
TM_ENOBUFS	Not enough memory to allocate a collision detection entry.
TM_ENOERROR	No error.

userCbFunc call back function

The call back function will be called as follows:

```
int                 userCbFunc
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ipAddress,
int                 errorCode,
ttUserGenericUnion  userCbParam
);
```

userCbFunc Parameters

userCbFunc Parameters	**Description**
InterfaceHandle	Interface as given to **tfUserStartArpSend.**
ipAddress	IP address to check for collision.
errorCode	Status of the ARP IP address collision detection. See below.
userCbParam	Parameter as passed by the user to **tfUseCollisionDetection**

errorCode Parameter Value	**userCbFunc is called**
TM_EADDRINUSE	An ARP reply, or ARP request, or ARP probe (interface not configured yet) has been received. Collision detection will still continue, until the user calls **tfCancelCollisionDetection.**
TM_ENOERROR	The timeout period for sending ARP requests/Arp probes expired. See **tfUserStartArpSend**. Collision detection will still continue, until the user calls **tfCancelCollisionDetection**.

tfUserStartArpSend

```
#include <trsocket.h>

int                 tfUserStartArpSend
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ipAddress,
int                 numberArpProbes,
ttUser32Bit         arpProbeInterval,
ttUser32Bit         timeout
);
```

Function Description

Start sending one or more ARP probes or ARP requests on a given interface.
If the interface is not configured yet with that IP address on any multi home, then this command instructs the Turbo Treck stack to send one or more ARP probes. An ARP probe is an ARP request with the sender net address set to zero.
If the interface has been configured with that IP address on one of its multi-home, then this command instructs the Turbo Treck stack to send one or more ARP requests with the configured IP address as the source address.
The user will pass the interface handle, the IP address to check for collision, a maximum number of ARP probes/requests, the interval of time between ARP probes/requests, and a timeout parameter. We send an ARP probe/request on the interface(s) specified by the interface handle, create a timer, and return. If numberArpProbes is bigger than one, then the Turbo Treck stack, in the context of the timer task, will send (numberArpProbes – 1) additional ARP probes/requests every time the arpProbeInterval expires.

- **tfUseCollisionDetectio**n has to be called before **tfUserStartArpSend** is called.
- If a "matching" ARP reply/request/probe is received before all ARP probes have been sent, the Turbo Treck stack will stop sending any more ARP probes, cancel the timer, and call the user call back function with a TM_EADDRINUSE error code. If a "matching" ARP reply/request is received before all ARP requests have been sent, the Turbo Treck stack will stop sending any more ARP probes, cancel the timer, and call the user call back function with a TM_EADDRINUSE error code.
- When the timeout parameter expires, the Turbo Treck stack will stop sending any more ARP probe(s) request(s), cancel the timer, and will call the user call back function passed by the user in **tfUseCollisionDetection**, with a TM_ENOERROR error code.

Interface configuration

- **tfUserStartArpSend** can be called before the interface has been configured with that IP address. In that case, the user has to have at least opened the interface with another IP address first, or with a zero IP address and the TM_DEV_IP_USER_BOOT flag. If the TM_DEV_IP_USER_BOOT flag has been used to open the interface, then the user need to use **tfFinishOpenInterface** to finish the configuration with the selected IP address, when it is determined that no other host uses that IP address.
- **tfUserStartArpSend** can be called, after the interface has been configured with the IP address parameter. In that case ARP requests are being sent, instead of ARP probes.

Parameters

Parameters	**Description**
interfaceHandle	Interface on which to send the ARP probe(s)/request(s).
ipAddress	Target IP address of the ARP probe(s)/request(s). Sender IP address of the ARP request(s).
arpProbeInterval	Interval in milliseconds between ARP probes/requests. If set to zero, the default **TM_PROBE_INTERVAL** (2000 milliseconds) is used for the probe interval.
numberArpProbes	Maximum number of ARP probes/ requests to send. Interval between ARP requests is ***arpProbeInterval*** in milliseconds. If set to zero, **TM_MAX_PROBE** (4) is used instead.
timeout	Number of milliseconds to wait after sending the first ARP probe/ request, and call the user call back function set by the user with **tfUseCollisionDetection**. If set to zero, it will be set to the default value of numberArpProbes * arpProbeInterval.

Returns

Value	Meaning
TM_EINVAL	Invalid interface, or NULL interface
	timeout is less than arpProbeInterval * numberArpProbes
TM_ENOENT	The user is not checking the IP address for collision detection, i.e. the user has not called **tfUseCollisionDetection** before calling **tfUserStartArpSend**.
TM_EALREADY	**tfUserStartArpSend** has already been called for that IP address, and has not timed out yet.
	tfUserStartArpSend has already been called for that IP address, for a different interface.
TM_ENOBUFS	No memory to allocate a timer.
TM_EPERM	Interface is not a LAN interface, i.e. ARP not permitted on that interface.
TM_ENXIO	Interface has not been opened.
TM_ENOERROR	No error.

BOOTP Automatic Configuration API

Description

The BOOTP user interface allows the user to query directly a BOOTP server, for IP addresses that the user can use. The Turbo Treck stack allows the user to retrieve the BOOTP addresses automatically. When the user configures an interface/ multihome, the Turbo Treck stack automatically queries a BOOTP server for an IP address and parameters, and finishes configuring the interface/multihome with the IP address, and default router given by the BOOTP server. Up to TM_MAX_IPS_PER_IF multihomes per interface can be configured that way.

In that case, the following steps are needed (after having called linkLayerHandle = tfUseEthernet ()):

```
interfaceHandle = tfAddInterface (namePtr,   linkLayerHandle,   ...);

errorCode = tfUseBootp (interfaceHandle, myNotifyFunction);

errorCode = tfOpenInterface (interfaceHandle,
                             0UL,
                             0UL,
                             TM_DEV_IP_BOOTP,
                             1);
```

Note that we have called **tfOpenInterface** with the TM_DEV_IP_BOOTP flag set. (Note other flags such as TM_DEV_IP_FORW_ENB could be ORed to TM_DEV_IP_BOOTP as needed.)

Note: The user must wait for the BOOTP configuration to complete. During the wait, ensure tfTimerUpdate (or tfTimerUpdateIsr), and tfTimerExecute are called.

ARP probes sent by the BOOTP client

If the TM_USE_AUTO_IP macro is defined in trsystem.h, and the AUTO IP option has been purchased, then the BOOTP client will send several ARP Probes (i.e. ARP request with a sender net address set to 0) to make sure that no other node has been configured with the same IP address.
If a node responds, then the BOOTP would notify the user with a TM_EADDRINUSE error code.

Checking on completion of a BOOTP configuration

- **Synchronous check**: The user can make multiple calls to **tfOpenInterface** to determine when the configuration has completed. Additional calls to **tfOpenInterface** will return TM_EALREADY as long as the BOOTP server has not replied. **tfOpenInterface** will return TM_ENOERROR if the BOOTP server has replied and the configuration has completed.

- **Asynchronous check**: To avoid this synchronous poll, the user can provide a user call back function to **tfUseBootp** as shown above. This function will be called upon completion of **tfOpenInterface**. The **notifyFunc** is called when the interface/multi home index is configured or if the BOOTP request timed out as follows: `myNotifyFunction(interfaceHandle, errorCode);` where errorCode is 0 if the configuration was successful, or TM_ETIMEDOUT if the BOOTP request timed out.

BOOTP Configuration parameters

When the BOOTP configuration has completed successfully as shown above, the user can retrieve the BOOTP configuration parameters by calling:

```
userBtEntryPtr = tfConfGetBootEntry(
                     ethernetInterfaceHandle,
                     multiHomeIndex);
```

If the BOOTP configuration has been successful, this function will return a pointer to a boot structure (null otherwise).

The **userBtEntryPtr** is a pointer to a structure defined in trsocket.h.
In particular, the **userBtEntryPtr** will contain:

The allocated IP address in the field *userBtEntryPtr->btuYiaddr*.

The subnet mask in the field *userBtEntryPtr->btuNetMask*.

Default router entry in the field *userBtEntryPtr->btuDefRouter*.

Configuring additional BOOTP IP addresses on the same interface

To configure additional BOOTP IP addresses on the same interface (multi homing), use **tfConfigInterface** instead of **tfOpenInterface**.

tfConfGetBootEntry

```
#include <trsocket.h>

ttUserBtEntryPtr  tfConfGetBootEntry
(
 ttUserInterface  interfaceHandle,
 unsigned char    multiHomeIndex
);
```

Function Description

Get a pointer to DHCP/BOOTP Conf BOOT entry (obtained while doing a **tfOpenInterface** with either TM_DEV_IP_BOOTP, or TM_DEV_IP_DHCP).

If successfull, the function will return a pointer to a ttUserBtEntry structure as defined in trsocket.h. In particular this structure will contain:

- The allocated IP address in the field btuYiaddr
- Default router entry in the field btuDefRouter
- Primary Domain name server btuDns1ServerIpAddress
- Secondary Domain name server btuDns2ServerIpAddress

Parameters

Parameter	Description
interfaceHandle	Ethernet interface handle
multiHomeIndex	Multi home index of the ethernet device

Returns

Value	Meaning
userBtEntryPtr	Pointer to a user boot entry as defined in `trsocket.h`
(ttUserBtEntryPtr) 0	Failure. No DHCP address bound.

tfUseBootp

#include <trsocket.h>

```
int                     tfUseBootp
(
ttUserInterface         interfaceHandle,
ttDevNotifyFuncPtr      devNotifyFuncPtr
);
```

Function Description
This function is used to initialize the BOOTP client interface, and should be called between **tfAddInterface**, and **tfOpenInterface**, to allow configuration using the BOOTP client protocol. If the second parameter is non null, then upon completion of **tfOpenInterface**, the function that it points to will be called. Function prototype for the user supplied notify function:

```
void                    devNotifyFunc
(
ttUserInterface         interfaceHandle,
int                     errorCode
);
```

This function will be called with the interfaceHandle passed to **tfOpenInterface**, and with an error code value (which is zero on success).

Example of a call with a non-null second parameter:

Given *interfaceHandle* (as returned by **tfAddInterface**), and the user defined interface notify function dev*NotifyFunc*:

```
void devNotifyFunc(ttUserInterface interfaceHandle,
                   int  5.          errorCode);

tfUseBootp (interfaceHandle, devNotifyFunc);
```

Example of a call with a null second parameter:

If the user does not wish to be notified of **tfOpenInterface** completion, then he can use the predefined NULL function pointer:

```
tfUseBootp (interfaceHandle,
           TM_DEV_NOTIFY_FUNC_NULL_PTR);
```

Parameters

Parameter	**Description**
interfaceHandle	The interface handle as returned by **tfAddInterface**
devNotifyFuncPtr	A pointer to a function that will be called upon completion of **tfOpenInterface** for a BOOTP configuration.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	The interface handle parameter is invalid.
TM_EMFILE	Not enough sockets to open the BOOTP client UDP socket
TM_ADDRINUSE	Another socket is already bound to the BOOTP client UDP port.

BOOTP relay agent

tfStartBootRelayAgent

```
#include <trsocket.h>

int                tfStartBootRelayAgent
(
ttUserIpAddress   ipAddress,
ttUserInterface   interfaceHandle,
unsigned char     multiHomeIndex
);
```

General description of a BOOTP relay agent

BOOTP and DHCP clients send out limited broadcasts messages to UDP port 67 (BOOTP server port) in order to get boot configuration from a BOOTP/DHCP server. If the server is not on the same subnet as the client, the broadcast message will not reach the server, since routers do not forward limited broadcasts. To avoid having a BOOTP/DHCP server on every subnet, BOOTP relay agents have been designed so that they can receive the limited broadcast message from the BOOTP or DHCP client, and send it (with some modifications) to a BOOTP/DHCP server. The BOOTP relay agent will also relay the replies from the BOOTP/DHCP server back to the client.

Function Description

This function is called to start the BOOTP relay agent to relay both BOOTP and DHCP requests to the specified remote server IP address from the specified interface handle, multihome index.

Parameters

Parameter	Description
ipAddress	IP address of the remove server. A limited broadcast address can also be specified (0xFFFFFFFF).
interfaceHandle	Interface through which to relay BOOTP and DHCP client requests.
multiHomeIndex	Multi home index of the interface through which to relay BOOTP and DHCP client requests.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	Invalid interface handle, or multi home index, or invalid IP address.
TM_EMFILE	Not enough sockets to open the BOOTP relay agent UDP socket
TM_ADDRINUSE	Another socket is already bound to the BOOTP relay agent UDP port.
TM_EALREADY	**tfStartBootRelayAgent** has already been called successfully.

tfStopBootRelayAgent

```
#include <trsocket.h>

int                  tfStopBootRelayAgent
(
void
);
```

Function Description

This function is called to stop the BOOTP relay agent. This will close the UDP BOOTP relay agent UDP socket.

Parameters

None

Returns

Value	Meaning
0	Success
TM_EALREADY	**tfStopBootRelayAgent** has already been called successfully.

DHCP Automatic Configuration API

Description

The DHCP user interface allows the user to query directly a DHCP server, for IP addresses that the user can use. The Turbo Treck stack allows the user to retrieve the DHCP addresses automatically.

When the user configures an interface/multihome, the Treck stack automatically queries a DHCP server for an IP address and parameters, and finishes configuring the interface/multihome with the IP address and default router given by the DHCP server. Up to TM_MAX_IPS_PER_IF multihomes per interface can be configured that way.

In that case, the following steps are needed (after having called linkLayerHandle = tfUseEthernet ()):

```
interfaceHandle = tfAddInterface (namePtr,
   linkLayerHandle,
   ...);

errorCode = tfUseDhcp (interfaceHandle, myNotifyFunction);

errorCode = tfOpenInterface (interfaceHandle,
                             0UL,
                             0UL,
                             TM_DEV_IP_DHCP,
                             1);
```

Note that we have called **tfOpenInterface** with the TM_DEV_IP_DHCP flag set. (Note other flags such as TM_DEV_IP_FORW_ENB could be ORed to TM_DEV_IP_DHCP as needed.).

Note: The user needs to wait for the DHCP configuration to complete. During the wait, ensure tfTimerUpdate (or tfTimerUpdateIsr), and tfTimerExecute get called.

User controlled configuration parameters

A user API ***tfDhcpConfSet*** is provided to allow the user

- to set the DHCP initial state to INIT_REBOOT (instead of the default INIT state) with a user specified requested IP address, and client ID,
- or to set a user specified requested IP address, and/or Client ID while DHCP is in the INIT state.

Or to allow the user to suppress the client ID option

ARP probes sent by the DHCP client

If the TM_USE_AUTO_IP macro is defined in trsystem.h, and the AUTO IP option has been purchased, then the DHCP client will send several ARP Probes (i.e. ARP request with a sender net address set to 0) to make sure that no other node has been configured with the same IP address. If a node responds, then the DHCP client would decline the IP address, and restart the discovery process in the INIT state after 10 seconds have elapsed

Checking on completion of an automatic DHCP configuration

- **Synchronous check**: The user can make multiple calls to **tfOpenInterface** to determine when the configuration has completed.. Additional calls to **tfOpenInterface** will return TM_EALREADY as long as the DHCP server has not replied. **tfOpenInterface** will return TM_ENOERROR, if the DHCP server has replied and the configuration has completed.

- **Asynchronous check**: To avoid this synchronous poll, the user can provide a user call back function to **tfUseDhcp** as shown above. This function will be called upon completion of **tfOpenInterface**. The notifyFunc is called when the interface/multi home index is configured or if the DHCP request timed out as follows: myNotifyFunction(interfaceHandle, errorCode); where errorCode is 0 if the configuration was successful, or TM_ETIMEDOUT if the DHCP request timed out.

Automatic DHCP Configuration parameters

When the DHCP configuration has completed successfully as shown above, the user can retrieve the DHCP configuration parameters by calling:

```
userBtEntryPtr = tfConfGetBootEntry(
                     ethernetInterfaceHandle,
                     multiHomeIndex);
```

If the DHCP configuration has been successful, this function will return a pointer to a boot structure (null otherwise).

The **userBtEntryPtr** is a pointer to a structure defined in trsocket.h.
In particular, the **userBtEntryPtr** will contain:

The allocated IP address in the field *userBtEntryPtr->btuYiaddr*.
The subnet mask in the field *userBtEntryPtr->btuNetMask*.
Default router entry in the field *userBtEntryPtr->btuDefRouter*.
Primary Domain name server *userBtEntryPtr->btuDns1ServerIpAddress*
Secondary Domain name server *userBtEntryPtr->btuDns2ServerIpAddress*

Configuring additional automatic DHCP IP addresses on the same interface
To configure additional automatic DHCP IP addresses on the same interface (multi homing), use **tfConfigInterface** instead of **tfOpenInterface**.

tfConfGetBootEntry

#include <trsocket.h>

```
ttUserBtEntryPtr  tfConfGetBootEntry
(
 ttUserInterface  interfaceHandle,
 unsigned char    multiHomeIndex
);
```

Function Description

Get a pointer to DHCP/BOOTP Conf BOOT entry (obtained while doing a **tfOpenInterface** with either TM_DEV_IP_BOOTP, or TM_DEV_IP_DHCP).

If successfull, the function will return a pointer to a ttUserBtEntry structure as defined in trsocket.h. In particular this structure will contain:

The allocated IP address in the field btuYiaddr
Default router entry in the field btuDefRouter
Primary Domain name server btuDns1ServerIpAddress
Secondary Domain name server btuDns2ServerIpAddress

Parameters

Parameter	**Description**
interfaceHandle	Ethernet interface handle
multiHomeIndex	Multi home index of the ethernet device

Returns

Value	**Meaning**
userBtEntryPtr	Pointer to a user boot entry as defined in trsocket.h
(ttUserBtEntryPtr) 0	Failure. No DHCP address bound.

tfDhcpConfSet

```
#include <trsocket.h>
int                tfDhcpConfSet
(,
ttUserInterface    interfaceHandle,
int                mHomeIndex,
int                flags,
ttUserIpAddress    requestedIpAddress,
unsigned char *    clientIdPtr,
int                clientIdLength
);
```

Function Description

Allows the user to set the DHCP initial state (INIT, or INIT_REBOOT) prior to the user calling

tfOpenInterface/***tfConfigInterface***.Called by the user when the user wants to specify his/her own Client ID, or suppress the Client ID option, or if the user wants to start in INIT_REBOOT state, or if the user wants to specify an IP address in the DISCOVERY phase.

- If user specifies INIT_REBOOT state, the Requested IP address needs to be specified as well.
- If user specifies INIT state (i.e. TM_DHCPF_INIT_REBOOT not set), optionally the user can specify the IP address, and or the CLIENT ID option.
- If CLIENT ID is not specified, and not suppressed, the stack will pick a unique CLIENT ID that will be the same across reboots provided that the user uses the same type of configuration and same index.

Parameters

Parameter	Description
interfaceHandle	Ethernet interface handle
mHomeIndex	multihome Index.
flags	0, or a combination of TM_DHCPF_INIT_REBOOT, TM_DHCPF_REQUESTED_IP_ADDRESS, TM_DHCPF_SUPPRESS_CLIENT_ID
requestedIpAddress	User requested IP address in network byte order
clientIdPtr	Pointer to client ID
clientIdLength	Length of client ID

Returns

Value	Meaning
TM_ENOERROR	success

tfUseDhcp

```
#include <trsocket.h>

int                    tfUseDhcp
(
ttUserInterface        interfaceHandle,
ttDevNotifyFuncPtr     devNotifyFuncPtr
);
```

Function Description

This function is used to initialize the DHCP client interface, and should be called between **tfAddInterface**, and **tfOpenInterface**, to allow configuration using the DHCP client protocol. If the second parameter is non null, then upon completion of **tfOpenInterface**, the function that it points to will be called. Function prototype for the user supplied notify function:

```
void                devNotifyFunc
(
ttUserInterface     interfaceHandle,
int                 errorCode
);
```

This function will be called with the interfaceHandle passed to **tfOpenInterface**, and with an error code value (which is zero on success).

Example of a call with a non-null second parameter:

Given *interfaceHandle* (as returned by **tfAddInterface**), and the user defined interface notify function **devNotifyFunc**:

```
void devNotifyFunc(ttUserInterface interfaceHandle,
                   int              errorCode);

tfUseDhcp (interfaceHandle, devNotifyFunc);
```

Example of a call with a null second parameter:

If the user does not wish to be notified of **tfOpenInterface** completion, then he can use the predefined NULL function pointer:

```
tfUseDhcp (interfaceHandle,
          TM_DEV_NOTIFY_FUNC_NULL_PTR);
```

Parameters

Parameter	**Description**
interfaceHandle	The interface handle as returned by **tfAddInterface**
devNotifyFuncPtr	A pointer to a function that will be called upon completion of **tfOpenInterface** for a DHCP configuration.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	The interface handle parameter is invalid.
TM_EMFILE	Not enough sockets to open the BOOTP client UDP socket
TM_ADDRINUSE	Another socket is already bound to the BOOTP client UDP port.

DHCP User Controlled Configuration API

Description

With the user-controlled configuration, the user can query a DHCP server directly without automatically configuring an interface. This is useful for example on a box that has an Ethernet interface, serial lines, and acts as a PPP server for the serial lines. The user could get IP addresses from a DHCP server via the Ethernet interface and then assign those IP addresses to PPP clients on its serial lines.

Those IP addresses should also be added to the box proxy ARP table, so that the box can respond to ARP requests on behalf of its PPP clients.

The following steps are needed:

tfInitTreckOptions must be called before **tfStartTreck** is called in order to reserve *numberSerialLines* DHCP user entries.

```
errorCode = tfInitTreckOptions (
       TM_OPTION_DHCP_MAX_ENTRIES,
       (unsigned long)numberSerialLines
       );
```

If no error:

```
errorCode = tfStartTreck();
```

Add all your link layers, and interfaces. For Ethernet, call:

```
ethernetLinkLayerHandle = tfUseEthernet ();

ethernetInterfaceHandle = tfAddInterface(
                       namePtr,
                       ethernetLinkLayerhandle,
                       ...);
```

For PPP, call **tfUseAsyncServerPpp** once, and **tfAddInterface** for each serial line. Save each PPP serial lines interface handle returned by each **tfAddInterface** in a table.

Configure the Ethernet interface with either a static IP address and IP netmask, or using DHCP as shown in the Automatic Configuration paragraph above. If using DHCP, wait for the **tfOpenInterface** to complete.

Ask for a DHCP address on the Ethernet interface for one PPP interface client.

```
errorCode = tfDhcpUserStart(
                        ethernetInterfaceHandle,
                        userIndex,
                        dhcpNotifyFunc);
```

userIndex must be initialized to zero for the first query (first PPP interface), 1 for a second query, etc..

A zero return value indicates that the DHCP request has completed successfully. A TM_EINPROGRESS return value indicates that a DHCP configure or request has been sent, and this is OK.

A TM_EALREADY return value indicates that a DHCP discover/request has already been sent because of a previous call to **tfDhcpUserStart.**

If *errorCode* is TM_EINPROGRESS, or TM_EALREADY, wait for the DHCP configuration to complete (i.e. for **dhcpNotifyFunc** to be called). During the wait, ensure **tfTimerUpdate** (or **tfTimerUpdateIsr**), and **tfTimerExecute** are called. When the DHCP request has completed or timed out, the **dhcpNotifyFunc** will be called as follows:

```
dhcpNotifyFunc(ethernetInterfaceHandle, userIndex, errorCode);
```

The *userIndex* corresponds to the second parameter of **tfDhcpUserStart,** and *errorCode* indicates whether the DHCP request has been successful (i.e *errorCode* == 0), or timed out (*errorCode* == TM_ETIMEDOUT).

User controlled configuration parameters
A user API ***tfDhcpUserSet*** is provided to allow the user

- to set the DHCP initial state to INIT_REBOOT (instead of the default INIT state) with a user specified requested IP address, and client ID,
- or to set a user specified requested IP address, and/or Client ID while DHCP is in the INIT state.
- Or to allow the user to suppress the client ID option

ARP probes sent by the DHCP client
If the TM_USE_AUTO_IP macro is defined in trsystem.h, and the AUTO IP option has been purchased, then the DHCP client will send several ARP Probes (i.e. ARP request with a sender net address set to 0) to make sure that no other node has been configured with the same IP address. If a node responds, then the DHCP client would decline the IP address, and restart the discovery process in the INIT state after 10 seconds have elapsed.

When the DHCP request is successful, the user can retrieve the IP address and parameters given by the DHCP server:

```
userBtEntryPtr = tfDhcpUserGetBootEntry( ethernetInterfaceHandle,
                                         userIndex );
```

If the DHCP request has been successful, this function will return a pointer to a boot structure (null otherwise). The *userBtEntryPtr* is a pointer to a structure defined in *trsocket.h*.

In particular, the *userBtEntryPtr* will contain:

The allocated IP address in the field *userBtEntryPtr->btuYiaddr*.
Network subnet mask in the field *userBtEntryPtr->btuNetMask*.
Default router entry in the field *userBtEntryPtr->btuDefRouter*.
Primary Domain name server *userBtEntryPtr->btuDns1ServerIpAddress*
Secondary Domain name server *userBtEntryPtr->btuDns2ServerIpAddress*

The user should save the *userBtEntryPtr* in its PPP table (where it previously stored the corresponding PPP interface handle, *pppInterfaceId*).

The user can now assign that IP address to its future serial line
PPP client, by calling:

```
errorCode = tfPppSetOption (
     pppInterfaceId, TM_PPP_IPCP_PROTOCOL,
     TM_PPP_OPT_ALLOW, TM_IPCP_IP_ADDRESS,
     (const char TM_FAR *)&userBtEntryPtr
->btuYiaddr,4);
```

The user can then configure that PPP interface calling **tfOpenInterface** with this *pppInterfaceId* parameter, using a static IP address for its interface. In the link layer call back function (set in **tfUseAsyncServerPpp**), if the flag is TM_LL_OPEN_COMPLETE, then the user can add the PROXY ARP entry: **tfAddProxyArpEntry**(userBtEntryPtr->btuYiaddr); In the link layer call back function, if the flag is TM_LL_CLOSE_STARTED then the user can delete the PROXY ARP entry: **tfDelProxyArpEntry**(userBtEntryPtr->btuYiaddr);

The user can request for additional DHCP IP addresses for additional PPP serial lines, repeating this process starting at **tfDhcpUserStart** above with different *userIndex*, and *pppInterfaceId* values.

If the user were to close the Ethernet interface, then all the DHCP addresses that were allocated with the **tfDhcpUserStart** calls should be released prior to closing the Ethernet interface, by calling:

```
tfDhcpUserRelease
(
ethernetInterfaceHandle,
userIndex
);
```

This must be done for each userIndex that was used in a successful **tfDhcpUserStart**.

tfDhcpUserGetBootEntry

#include <trsocket.h>

```
ttUserBtEntryPtr  tfDhcpUserGetBootEntry
(
ttUserInterface   interfaceHandle,
int               index
 );
```

Function Description:
Get a pointer to a DHCP user BOOT entry. This function can only be used after a successful call to **tfDhcpUserStart**. If successful, the function will return a pointer to a **ttUserBtEntry** structure as defined in **trsocket.h.** In particular this structure will contain:

The allocated IP address in the field btuYiaddr.
Default router entry in the field btuDefRouter.
Primary Domain name server btuDns1ServerIpAddress
Secondary Domain name server btuDns2ServerIpAddress

Parameters

Parameter	**Description**
interfaceHandle	Ethernet interface handle
userIndex	User Index (between 0, and tvMaxUserDhcpEntries - 1)

Returns

Value	**Meaning**
userBtEntryPtr	Pointer to a user boot entry as defined in `trsocket.h`
(ttUserBtEntryPtr) 0	Failure. No DHCP address bound.

tfDhcpUserRelease

```
#include <trsocket.h>

int                 tfDhcpUserRelease
(
ttUserInterface     interfaceHandle,
int                 userIndex
 );
```

Function Description

Cancel a DHCP request and/or a DHCP lease that had been previously obtained using **tfDhcpUserStart()**.

Parameters

Parameter	**Description**
interfaceHandle	Ethernet interface handle
userIndex	User Index (between 0, and tvMaxUserDhcpEntries - 1)

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EINVAL	Bad parameter

tfDhcpUserSet

```
#include <trsocket.h>
int                tfDhcpUserSet
(,
ttUserInterface    interfaceHandle,
int                userIndex
int                flags,
ttUserIpAddress    requestedIpAddress,
unsigned char *    clientIdPtr,
int                clientIdLength
);
```

Function Description

Allows the user to set the DHCP initial state (INIT, or INIT_REBOOT) prior to the user calling

tfDhcpUserStart. Called by the user when the user wants to specify his/her own Client ID, or suppress the Client ID option, or if the user wants to start in INIT_REBOOT state, or if the user wants to specify an IP address in the DISCOVERY phase.

- If user specifies INIT_REBOOT state, the Requested IP address needs to be specified as well.
- If user specifies INIT state (i.e. TM_DHCPF_INIT_REBOOT not set), optionally the user can specify the IP address, and or the CLIENT ID option.
- If CLIENT ID is not specified, and not suppressed, the stack will pick a unique CLIENT ID that will be the same across reboots provided that the user uses the same type of configuration and same index.

Parameters

Parameter	**Description**
interfaceHandle	Ethernet interface handle
userIndex	User Index (between 0, and tvMaxUserDhcpEntries - 1)
flags	0, or a combination of TM_DHCPF_INIT_REBOOT, TM_DHCPF_REQUESTED_IP_ADDRESS, TM_DHCPF_SUPPRESS_CLIENT_ID
requestedIpAddress	User requested IP address in network byte order
clientIdPtr	Pointer to client ID
clientIdLength	Length of client ID

Returns

Value	**Meaning**
TM_ENOERROR	Success
TM_EINVAL	Bad parameter
TM_EPERM	Call can only be made in INIT state prior to calling ***tfDhcpUserStart***
TM_ENOBUFS	Not enough memory

tfDhcpUserStart

```
#include <trsocket.h>

int                           tfDhcpUserStart
(
ttUserInterface               interfaceHandle,
int                           userIndex,
ttUserDhcpNotifyFuncPtr       dhcpNotifyFuncPtr
);
```

Function Description

This function allows the user to reserve a DHCP address on the Ethernet interface for another interface (i.e PPP for example). Prior to this call, the ethernet interface has to have been configured (possibly using DHCP as well). See **tfOpenInterface** for details. This function starts sending a DHCP request on the Ethernet interface handle parameter. The index corresponds to a unique user DHCP request. It has to be between 0 and **tvMaxUserDhcpEntries** - 1. (Note that **tvMaxUserDhcpEntries** default value is 0, and has to be changed with a tfInitSetTreckOptions, with option name TM_OPTION_DHCP_MAX_ENTRIES to a value equal to the number of DHCP IP addresses that we want to reserve for our other interface(s), prior to this call.). **dhcpNotifyFunc** is a user supplied call back function which will be called when either the DHCP request has been successful, or has timed out, or when a previously leased DHCP address has been cancelled by the server.

If **tfDhcpUserStart** returns TM_ENOERROR, the DHCP request has completed successfully. A TM_EINPROGRESS error return indicates that a DHCP configure or request has been sent (this is alright).

A TM_EALREADY error return indicates that a DHCP discover/request has been previously sent as a result of a previous call to **tfDhcpUserStart()**. If the function returns TM_EINPROGRESS, or TM_EALREADY, wait for the DHCP configuration to complete (i.e for **dhcpNotifyFunc** to be called). During the wait, make sure **tfTimerUpdate ()** (or **tfTimerUpdateIsr ()**), and **tfTimerExecute ()** get called.

If the function returns TM_EINPROGRESS, or TM_EALREADY, then, when the DHCP request has completed, or timed out the **dhcpNotifyFunc** will be called as follows:

(*dhcpNotifyFunc)(ethernetInterfaceHandle, userIndex, errorCode);
where the userIndex corresponds to the second parameter of tfDhcpUserStart(), and errorCode indicates whether the DHCP request has been successful (errorCode == 0), or timed out (errorCode ==TM_ETIMEDOUT).

Parameters

Parameter	**Description**
InterfaceHandle	Ethernet interface handle
UserIndex	User Index (between 0, and tvMaxUserDhcpEntries - 1)
dhcpNotifyFuncPtr Pointer	To a function that will be called when the DHCP request has completed, or timed out.

Returns

Value	**Meaning**
TM_ENOERROR	Success. The DHCP notify function will not be called.
TM_EINPROGRESS	DHCP request/discover sent with no error.
TM_EALREADY	DHCP request/discover previously sent.
TM_EINVAL	Bad parameter
TM_ENOBUFS	Not enough memory
Other	As returned by device driver send function.

Dialer

Description

Turbo Treck PPP, or SLIP includes a module to facilitate dialing a remote host (most likely, an ISP). Dialing occurs prior to the establishment of a PPP connection and is triggered by calling **tfOpenInterface**. Prior to this call, the user would first enable the dialer, and then set up how the dialer will behave. This is done with a series of 'send-expect' pairs, where the local client sends a string to the remote host and waits for its response. 'Expect-send' pairs behave in the opposite manner. This is similar to the method used in many UNIX scripted dialers.

Each 'send-expect' or 'expect-send' pairs have an error string associated with them. When this string is received, it triggers an error inside the dialer. The default behavior is simply to move to the next state (i.e., the next 'send-expect' pair). If TM_DIALER_FAIL_ON_ERROR is set and this error string is received, the entire dialing session will be aborted with an error.

Every 'send-expect' pair has a time-out value and a retry value associated with it. This allows, for instance, a dialer to attempt to dial a phone number a finite number of times, and if it still fails, to abort the session.

When the dialer has finished, either through completion or error, the user is notified through a notification function which is passed in when the dialer is enabled.

Example

Here is an example of a device dialing and logging in to and ISP:

Local:	**Remote:**
ATDT5551212	
	CONNECT14400
	Login:
username	
	Password:
password	
	Welcome!

This session could be accomplished with the following code:

```
/* Initialize the dialer */
tfUseDialer( interfaceHandle, dialerNotify );

/*
 * client: send ‘ATDT5551212’ -> server: send ‘CONNECT14400’
 * 30 second time-out, 5 retries, fail if ‘ERROR’ returned.
 */
tfDialerAddSendExpect( interfaceHandle,
                           “ATDT5551212”,
                           “CONNECT14400”,
                           “ERROR”,
                           5,
                           30,
                           TM_DIALER_FAIL_ON_ERROR );

/* server: send ‘Login:’ -> client: send ‘username’ */

 ( interfaceHandle,
                           “username”,
                           “Login:”,
                           (char *) 0,
                           0,
                           60,
                           0 );

/* server: send ‘Password:’ -> client: send ‘password’ */
tfDialerAddExpectSend ( interfaceHandle,
                           “password”,
                          “Password:”,
                           (char *) 0,
                           0,
                           60,
                           0 );


/*
 * server: send ‘Welcome’ -> client: send nothing (successful)
 * server: send ‘Invalid Login’ -> client: send nothing (failure)
 */
tfDialerAddExpectSend ( interfaceHandle,
                           “”,
                          “Welcome”,
                           “Invalid Login”,
                           0,
                           60,
                           TM_DIALER_FAIL_ON_ERROR );

/* Dialing starts now*/
tfOpenInterface (interface handle,...);
```

tfDialerAddExpectSend

```
#include <trsocket.h>

int                 tfDialerAddExpectSend
(
ttUserInterface     interfaceHandle,
char
sendString,
char
expectString,
char
errorString,
int                 numRetries,
int                 timeout,
unsigned char       flags
);
```

Function description

This function adds an expect-send pair to the dialer (client waits for the server to send a string, and then sends a string in response). When the dialer reaches this phase, it waits for the remote host to send a string. If this string matches *expectString*, *sendString* is sent back in response and the dialer moves to the next state. If the received string matches *errorString*, this phase is aborted, and if TM_DIALER_FAIL_ON_ERROR is set in *flags*, the entire dialing session is terminated with an error.

The dialer will wait for *expectString* or *errorString* for a time equal to *timeout* in seconds. If the number of retries specified by *numRetries* has not been exhausted when this time elapses, the dialer will begin waiting again. When the retry limit has been reached, the dialer will abort the session and return with an error.

tfDialerAddSendExpect

#include <trsocket.h>

```
int                 tfDialerSendExpect
(
ttUserInterface     interfaceHandle,
char
sendString,
char
expectString,
char
errorString,
int                 numRetries,
int                 timeout,
unsigned char       flags
);
```

Function description
This function adds a send-expect pair to the dialer (local peer sends a string and waits for the remote peer to respond). When the dialer reaches this phase, it will send *sendString* to the remote peer and waits for its response. If this response matches *expectString*, the dialer successfully moves to the next state. However, if the received string matches *errorString* the dialer will abort this phase, and if TM_DIALER_FAIL_ON_ERROR is set in *flags* the entire dialing session will be aborted with an error.

After sending *sendString*, the dialer will wait for a time equal to *timeout* in seconds. If the number of retries specified by *numRetries* has not been exhausted when this time elapses, the dialer will resend *sendString* and repeat this process. When the retry limit has been reached, the dialer will abort the session and return with an error.

Parameters

Parameter	**Description**
interfaceHandle	Interface handle used in previous call to **tfUseDialer**; this specifies the device to dial upon
sendString	String to send when the 'expectString' is received from the peer
expectString	String to send to peer immediately
errorString	String that, if received from the peer, indicates an error
numRetries	Number of times to retry sending 'sendString' before failing
timeout	Amount of times (in seconds) between time-outs
flags	Only one flag is currently defined: TM_DIALER_FAIL_ON_ERROR. This flag causes the dialer to immediately return if the error string is received. Normally, the dialer would simply attempt to resend the previous string

Returns

Value	**Meaning**
TM_ENOERROR	Expect/send string added successfully
TM_EINVAL	Bad interface handle
TM_EINVAL	Timeout value is zero
TM_EINVAL	Bad send/expect strings
TM_ENOMEM	Insufficient memory to add expect/send strings

tfUseDialer

#include <trsocket.h>

```
int                         tfUseDialer

(

ttUserInterface             interfaceHandle,

ttUserLnkNotifyFuncPtr      notifyFuncPtr

);
```

Function description
Enables and initializes the dialer. This does not *start* the dialer, but simply initializes it – the dialing will occur when **tfOpenInterface** is called. This should only be enabled on a serial interface running PPP/SLIP, and should be called before any calls to **tfDialerAddSendExpect** or **tfDialerAddExpectSend** are made.

The notification function is used to monitor the status of the dialer. The prototype of the notification function is

myDialerNotify(ttUserInterface interfaceHandle, int flags);

The notification function will only be called when the dialer has finished (either successfully or unsuccessfully). There are two possible values for the flags parameter:

TM_DIALER_OPEN_COMPLETE — The dialer has completed the dialing session successfully.

TM_DIALER_OPEN_FAILED — The dialer encountered an error and aborted the session.

Parameters

Parameter	**Description**
interfaceHandle	Interface to enable the dialer on
notifyFuncPtr	User's function to be called to notify of status of dialer

Returns

Value	**Meaning**
TM_ENOERROR	Dialer successfully initialized
TM_EINVAL	Bad interface handle
TM_EALREADY	Dialer already initialized
TM_ENOMEM	Insufficient memory to initialize dialer

IGMP Protocol

Introduction

IP multicasting is the transmission of an IP datagram to a “host group”, a set of zero or more hosts identified by a single IP destination address. A host group IP address is a multicast IP address, or class D IP address, where the first 4 bits of the address are 1110. The range of host group IP addresses (class D) are from 224.0.0.1 to 239.255.255.255. Class D addresses starting with 224.0.0.x and 224.0.1.x have been reserved for various permanent assignments (see RFC 1700). The range of addresses between 224.0.0.1 and 224.0.0.255 are local multicast addresses (i.e. they cannot be forwarded beyond the local subnet), and have been reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership recording. The membership of a host group is dynamic: hosts may join and leave groups at any time. IGMP (Internet Group Management Protocol) is the protocol used by IP hosts to report their multicast group memberships to multicast routers. Local host groups (224.0.0.0 through 224.0.0.255) memberships do not need to be reported, since as indicated above they cannot be forwarded beyond the local subnet. Without the IGMP protocol, a host can send multicast IP datagrams, but cannot receive any.

Description

The following RFC’s are implemented in the Turbo Treck IGMP implementation.

RFC 1112	Host Extensions for IP Multicasting
RFC 2236	Internet Group Management Protocol
RFC 2113	IP router alert option

We will describe the design of the IP multicasting, and the host IP IGMP protocol implementation in the Turbo Treck stack in detail.

Enabling the IGMP Code

To compile in the IGMP code, then the following macro should be added to trsystem.h:

```
#define TM_IGMP
```

Sending Multicast Packets

Send API

Sending multicast packets does not require new API functions. The user uses the **send** and **sendto** functions on a UDP socket with a multicast IP address destination.

IP Outgoing Interface for Multicast Packets

The user can choose a per socket default interface to send multicast packets by using a new **setsockopt** option (IPO_MULTICAST_IF) at the IPPROTO_IP level with the IP address of the outgoing interface as the option value. If the user does not specify a destination interface that way, then a default outgoing interface will be used, if, after having configured the interface, the user specified a system wide one, using the new **tfSetMulticastInterface** API. Otherwise, the multicast **sendto** will fail.

IP TTL for Multicast Packets

By default, the IP layer will send a multicast packet with a TTL value of 1, unless the user changes the default multicast IP ttl value for the socket with **setsockopt**. A TTL value of 1, will not allow the multicast IP datagram to be forwarded beyond the local subnet. The option IPO_MULTICAST_TTL at the IPPROTO_IP level has been added to **setsockopt**, and **getsockopt**.

Mapping Multicast Addresses to Layer 2 Hardware Addresses

To enable multicast mapping from the IP multicast address to an Ethernet multicast address, the TM_DEV_MCAST_ENB flag must be set in the **tfOpenInterface** flags parameter.

Receiving UDP Multicast Packets

When a UDP datagram whose destination IP address is a multicast address, arrives on an interface from the network, we check that the host is a member of the multicast group on the interface it came in. A host is a member of a group address for a given interface, if it joined that multicast group.

Note: In order to receive multicast packets, the user must join a host group.

Joining a Host Group

To join a host group, a user socket application calls **setsockopt** with the IPO_ADD_MEMBERSHIP option at the IPPROTO_IP level. The option pointer points to a structure containing the IP host group address and the interface (defined by its IP address). It will return TM_EINVAL if the IP host group address is not a class D address or if there is no interface configured with the IP address stored in the structure. It will return TM_EADDRINUSE if there was a previous successful IPO_ADD_MEMBERSHIP call for that host group on the same interface. Otherwise, this is the first call to IPO_ADD_MEMBERSHIP for the pair. The IP host group address will be added to the interface. In order to enable reception of that multicast address, the Turbo Treck stack will call the interface driver specific ioctl function. The driver specific ioctl function is called with the TM_DEV_SET_MCAST_LIST flag value, optionPtr pointing to the list of Ethernet multicast addresses corresponding to all the IP host group addresses added so far, and optionLen indicating the number of those Ethernet multicast addresses. We keep track of the list of Ethernet multicast addresses, so that the driver does not have to.

Note: In order to receive multicast packets, the user must implement a section in drvIoctl to create a multicast address from a list of Ethernet addresses, and store that multicast address in the Ethernet chip. (See drvIoctl for details.)

Leaving a Host Group

To stop receiving multicast packets for a given host group destination on an interface, the user need to leave that host group on that interface by calling **setsockopt** with the IPO_DROP_MEMBERSHIP option at the IPPROTO_IP level. The option pointer points to a structure containing the IP host group address, and the interface (defined by its IP address). It will return TM_EINVAL if the IP host group address is not a class D address or if there is no interface configured with the IP address stored in the structure. It will return TM_ENOENT if there is no such host group on the interface. Otherwise, the IP Host group address will be deleted from the interface, and the Turbo Treck stack will call the interface driver specific ioctl to update the list of Ethernet multicast addresses corresponding to the IP host group addresses that have been added and not dropped. The driver specific ioctl function is called with the TM_DEV_SET_MCAST_LIST flag value, and with optionPtr pointing to the list of Ethernet multicast addresses corresponding to all the IP host group addresses added (and not dropped), and with optionLen indicating the number of those Ethernet multicast addresses. We keep track of the list of Ethernet multicast addresses so that the driver does not have to.

Turbo Treck Stack Initialization of the IGMP Protocol

When a new interface is configured, the Turbo Treck stack will automatically join the 224.0.0.1 host group on that interface. If RIP is enabled, the Turbo Treck stack will also automatically join the 224.0.0.2 host group on that interface.

Limitations

This implementation is for IP hosts only. It does not include multicast routing. A host group can only be joined once on a given interface. This will prevent multiple sockets on the IP host from sharing the same multicast address, i.e. only one process on the IP host could receive multicast packets for a given multicast destination IP address. No loop back of multicast packet is allowed.

drvIoctlFunc

```
#include <trsocket.h>

void                drvIoctlFunc
(
ttUserInterface     interfaceHandle,
int                 flag,
void *              optionPtr,
int                 optionLen
);
```

Function Description

drvIoctlFunc is a pointer to a function provided by the user and given to the Turbo Treck stack as the 8th parameter of the tfAddInterface() function. The Turbo Treck TCP/IP stack will call the driver ioctl function with either TM_DEVICE_SET_MCAST_LIST, or TM_DEVICE_SET_ALL_MCAST as described below. The driver does not have to maintain a list of currently enabled multicast addresses (since the stack will do that) and passes the complete list every time a new multicast address needs to be added or deleted.

Parameters

Parameter	**Description**
interfaceHandle	The interface handle of the driver's ioctl routine to call
flag	See below
optionPtr	Pointer to a flag specific parameter. (See below.) *optionLen* Length of the option

Flag Parameter

TM_DEV _SET_MCAST_LIST — Enable the reception of the Ethernet multicast addresses pointed to by optionPtr. optionPtr points to a list of 48-bit multicast Ethernet addresses, and optionLen contains the number of multicast Ethernet addresses optionPtr points to. If optionLen is non-zero, The driver will enable reception of the optionLen multicast addresses in the list pointed to by optionPtr. If optionLen is zero, the driver will disable all multicast reception.

TM_DEV_SET_ALL _MCAST — The driver will enable reception of all multicast addresses and should ignore the optionPtr and optionLen arguments.

Returns

Value	**Meaning**
0	Success
NONZERO	Driver specific error code

tfSetMcastInterface

```
#include <trsocket.h>

int                tfSetMcastInterface
(
ttUserInterface    interfaceHandle,
unsigned char      mhomeIndex
);
```

Function Description

This function allows can specify a default interface to be used to send multicast destination IP packets. This default interface will be used to send outgoing multicast packets, when the user program does not specify an interface (i.e. the user did not use the IPO_MULTICAST_IF option on the socket).

Parameters

Parameter	**Description**
interfaceHandle	The interface handle of the default interface to be used as output interface for outgoing multicast packets

Returns

Value	**Meaning**
0	Success
TM_ENETDOWN	Interface not configured yet
TM_EADDRNOTAVAIL	Interface does not have the multicast enabled flag set
TM_EALREADY	Default multicast interface already set
TM_EINVAL	Invalid interface handle

NAT

The point of NAT is to free a network from certain limitations of IP such as the requirement for each machine to have a fixed unique address. It is primarily used when there is only one IP address to be shared by several machines, or there are multiple IP addresses that need to be flexibly assigned to multiple machines (either the machines change often or the set of IP addresses may change).

This module enables the Turbo Treck TCP/IP stack to become a NAT Router, which acts as a link between public and private networks. Resources inside the private network (clients and servers) are identified differently on the two networks. A common example is a home or office LAN with a private addressing scheme, and the public Internet with publicly assigned IP addresses.

Note: NAT is not a security feature and is not a substiture for a firewall. A NAT router should only be used between networks of equal trust levels.

All initialization of a NAT router implementation of the Turbo Treck stack is first done just as if the there were no NAT feature and the stack were acting as a transparent router. This includes functions like **tfUseEthernet()**, **tfAddInterface()** and **tfOpenInterface()**. Then the NAT configuration functions are called. First, **tfNatConfig()** is called on each public interface. Then, different functions are called depending on the type of NAT implementation.

One IP address

The two types of resources when there is only one public IP address to work with are NAPT and Inner Servers.

Several machines can access a public network through a NAT router using a single public IP address. This is technically NAPT, the "P" refers to the tricks of port number translation that are used to distinguish several processes on several machines inside the private network.

Normally, TCP and UDP port numbers distingish several processes but only on *one* machine. **tfNatConfigNapt()** assigns a range of synthetic port numbers to a public address on a public NAT interface. NAT draws from those ports to support every TCP and UDP connection that crosses the stack between private and public networks (actually between a public network and any other).

NAT connections are timed out based on traffic.
(See NAT_NTRTTL_...constants in trmacro.h.)

Default timeouts are:

Incomplete TCP connections	1 minute
Completed TCP connections	24 hours
Terminated TCP connections	1 minute
unmade FTP PORT connections	5 minutes
UDP DNS queries	1 minute
non-DNS UDP queries	5 minutes
ICMP packets with identifier	5 minutes

NAPT only supports connections initiated outbound (private client, public server). In order for a server on the private network to be accessible to a client on the public network in a NAPT environment, an Inner Server must be configured. There are three types of Inner Servers, TCP, FTP and UDP. In each case, an inner private IP address and port number are associated with an outer public IP and port. Clients on the public network call the public values. NAT translates this traffic to the private server's values.

NAPT and Inner Server support only TCP and UDP traffic.

Multiple IP addresses

The two types of resources in this environment are Static and Dynamic public IP address assignments. A Static assignment associates one public IP address at a time with one private address, permanently.

A Dynamic assignment does so automatically, as needed. New dynamic address associations are initiated only by outbound TCP SYN packets or by any outbound packets of another protocol (for example UDP DNS or ICMP Ping).

A Dynamic assignment expires when no packets have crossed for a certain amount of time (default 15 minutes, TM_NTRTTL_DYNAMIC in trmacro.h). Expired Dynamic associations become available for use by other private parties. As individual TCP and UDP sessions are not tracked on a Dynamic assignment, the timeout does not discriminate connection status or traffic type.

Static assignments would be useful for servers on the inside network or for users who require consistent full-featured access to the public network.

Dynamic and Static assignments support almost all IP traffic including TCP, UDP, and ICMP.

Mixing

Some mixing of the two schemes is possible. For example, you can configure static associations after a NAPT interface is configured if you have multiple IP's (though of course you are using only one for NAPT). Be sure to call tfNatConfigStatic() only after tfNatConfigNapt().

Ping

Ping, using ICMP echo, is supported via Static and Dynamic IP address associations. Outbound Ping will also work on NAPT configurations, that is a private client pinging a public server. Inner servers cannot be pinged however. If one were to try one would only ping the NAT router via its public interface. So this would not detect whether the inner server was up or not, just that the NAT router were up.

TraceRoute

Unix traceroute (or Windows tracert) uses unsolicited UDP datagrams and ICMP error messages to detect routing pathways. It will work over static and dynamic connections in either direction and NAPT in the outbound direction. One cannot thoroughly TraceRoute to an inner server however, just to the NAT router, as with Ping.

FTP Servers

This NAPT implementation goes to athletic lengths to accommodate FTP servers either outside or inside (as much as any other server is supported inside). Special handling is made of every outbound FTP PORT command and PASV reply 227. NAT translates the private IP addresses and port numbers in those messages into public values.

Proxy or 3rd party FTP transfers are not supported. Non-PORT, non-PASV transfers are not supported. (Unix "sendport" turned off.)

FTP transfers involving server port numbers other than the standard ports 21 and 20 are not supported (e.g. by entering the client command "open <host> <port>").

Private IP Addressing

The following IP addresses should be used for private networks when connected by a NAT router to the public Internet. They have been reserved specifically for this purpose by RFC 1918 "Address Allocation for Private Internets".

10.0.0.0 - 10.255.255.255	16 million class A addresses
172.16.0.0 - 172.31.255.255	1 million class B addresses
192.168.0.0 - 192.168.255.255	64 thousand class C addresses

Using other addresses may result in conflicts the NAT router can't resolve.

Triggers

The basis of this implementation of NAT is the "trigger" object. Each ttNatTrigger instance represents some possibility of interception of an incoming or outgoing packet. Every packet that comes in or out of a public NAT interface is scrutinized by all current triggers. There are several types of triggers (see TM_NTRTYPE_... in trmacro.h), each looking at different aspects of packets. Also, triggers are either permanent or temporary, the duration depending on type, traffic, and timing.

Some triggers recognize new TCP connections and "spawn" other triggers that live for the duration of those connections. Those session triggers look at addresses and ports. Some translate packets based on IP address alone. All permanent triggers are configured by API calls.

As the trigger list is a LIFO linked list, triggers configured or spawned *later* detect packets *earlier*, and so take precedence.

Public vs Private

Each "public" NAT interface maintains a linked list of triggers (ttDevice.devNatTriggerHead). A "private" interface is simply one that is not public. No transformations are needed on private interfaces. The task of the NAT software is to make a public NAT interface appear private to the rest of the TCP/IP stack. Internally, NAT is associated only with the interface to the public network.

The business of each public NAT interface involves public and private addresses and ports. The system on which the stack is running is known by the public information on that exernal public network, and by the private information by the TCP/IP stack and other internal private networks.

Reference Implementation

TSTNAT.C implements a NAT Router with a private Ethernet and a public dialup PPP connection to the public Internet

NAT API (Application Programming Interface) Functions

tfNatConfig

#include <trsocket.h>

```
int                tfNatConfig
(
ttUserInterface    interfaceHandle
);
```

Function Description

Identify a stack device as a public NAT interface. Call exactly once per public NAT interface. Call before any other NAT functions on this interface. If the stack ever shuts down orderly, call tfNatUnConfig for each interface where tfNatConfig() returned TM_ENOERROR

Parameters

Parameter	**Description**
interfaceHandle	interface handle of the public NAT device.

Return

Value	**Meaning**
TM_ENOERROR	success
TM_EINVAL	not a valid interface handle

tfNatUnConfig

#include <trsocket.h>

```
int                 tfNatUnConfig
(
ttUserInterface     interfaceHandle
);
```

Function Description
Dismantle a stack device's functioning as a public NAT interface. Call exactly once per public NAT interface, that is, for each interface for which tfNatConfig() returned TM_ENOERROR Call after all other NAT functions on this interface.

Parameters

Parameter	**Description**
interfaceHandle	interface handle of the NAT device to tfNatUnConfig.

Return

Value	**Meaning**
TM_ENOERROR	success
TM_EINVAL	not a valid interface handle or not an interface that was fNatConfig()'d

tfNatConfigNapt

#include <trsocket.h>

```
int                tfNatConfigNapt
(
ttUserInterface    interfaceHandle,
ttUserIpAddress    ipPublic,
ttUserIpPort       portPublicMin,
ttUserIpPort       portPublicMax
);
```

Function Description

This function defines a range of port numbers that are used to route traffic from multiple private machines through the NAT router to the public network.

Call tfConfigNatp only once on a public NAT interface. Do not use the ipPublic value in any other NAT configuration function but the one call to tfNatConfigNapt() except the tfNatConfigInner...Server() functions.

tfNatConfigNapt should be called before any tfNatConfigStatic() or tfNatConfigDynamic() calls on the same interface. In that case, NAPT would only be used for non-statically assigned machines after all dynamic associations were already in use.

Though UDP ports never conflict with TCP ports, there is only one "next port" maintained, so ports are allocated as if they did conflict.

Parameters

Parameter	**Description**
interfaceHandle	Interface handle of the public NAT device on which to configure routing.
ipPublic	Public IP address, network byte order
portPublicMin	Inclusive range of port numbers, host byte order.
portPublicMax	Inclusive range of port numbers, host byte order.

Return

Value	**Meaning**
TM_ENOMEM	Insufficient memory or too many triggers already
TM_ENOERROR	Newly allocated trigger, ip's and ports initialized

tfNatConfigInnerTcpServer

#include <trsocket.h>

```
int                 tfNatConfigInnerTcpServer
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ipPublic,
ttUserIpAddress     ipPrivate,
ttUserIpPort        portPublic,
ttUserIpPort        portPrivate
);
```

Function Description

Configure a server on the private network for access via a specific TCP port on the public network. The public world will know the server as ipPublic:portPublic. On the private network, the server will be implemented at ipPrivate:portPrivate.

Call this function as many times as you like, being sure not to assign conflicting public or private ip/port/protocol triads.

Parameters

Parameter	**Description**
interfaceHandle	Interface handle of the public NAT device to configure public a TCP port on.
ipPublic	The server's public identity, network byte order
ipPrivate	The server’s private identity, network byte order.
portPublic	The server’s pubic implementation, host byte order.
portprivate	The server's private implementation, host byte order

Returns

Value	**Meaning**
TM_ENOMEM	Insufficient memory or too many triggers already
TM_ENOERROR	Newly allocated trigger, ip's and ports initialized
TM_EINVAL	Erroneous use, for details enable

tfNatConfigInnerUdpServer

#include <trsocket.h>

```
int               tfNatConfigInnerUdpServer
(
ttUserInterface   interfaceHandle,
ttUserIpAddress   ipPublic,
ttUserIpAddress   ipPrivate,
ttUserIpPort      portPublic,
ttUserIpPort      portPrivate
);
```

Function Description

Configure a server on the private network for access via a specific UDP port on the public network. The public world will know the server as ipPublic:portPublic.

On the private network, the server will be implemented at ipPrivate:portPrivate.

Call this function as many times as you like, being sure not to assign conflicting public or private ip/port/protocol triads.

Parameters

Parameter	**Description**
interfaceHandle	Interface handle of the public NAT device to cofigure public UDP port on
ipPublic	The server's public identity, host byte order
ipPrivate	The server’s private identity, host byte order
portPublic	The server’s public implementation, host byte order
portprivate	The server's private implementation, host byte order

Returns

Value	**Meaning**
TM_ENOMEM	Insufficient memory or too many triggers already
TM_ENOERROR	Newly allocated trigger, ip's and ports initialized
TM_EINVAL	Erroneous use, for details enable

TM_NAT_TRACE

tfNatConfigInnerFtpServer

#include <trsocket.h>

```
int                 tfNatConfigInnerFtpServer
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ipPublic,
ttUserIpAddress     ipPrivate,
ttUserIpPort        portPublicMin,
ttUserIpPort        portPublicMax
);
```

Function Description

This function configures an FTP server on the private network for access as an FTP server via the public IP address.
The public world will know the FTP server as ipPublic:21.
On the private network, the server will be implemented at ipPrivate:21.
Only one FTP server can be configured for a public NAT interface.

In case the FTP server ever gets a PASV command from the public client, the inbound connection must be accommodated on a unique public port. Those ports are taken from the inclusive range For PORT-mode transfers to work, you should have NAPT configured for the same public IP address. The data connection initiated by the private server will have a private port of 20 and a synthetic public port. Though unconventional, this does not appear to thwart the FTP client on SunOS (at least). Non-PORT, non-PASV transfers are not supported (i.e. with the sendport option in some Unix FTP clients turned off)

Parameters

Parameter	**Description**
interfaceHandle	Interface handle of the public NAT device on which to configure an FTP server
ipPublic	The FTP server's public IP address - network byte order
ipPrivate	The FTP server's private IP address - network byte order
portPublicMin	Inclusive range of synthetic ports TM_SINGLE_INTERFACE_HOME to be used for PASV
portPublicMax	Connections to the inner FTP server, similar to NAPT synthetic ports (ports are in host byte order)

Returns

Value	**Meaning**
TM_ENOMEM	Insufficient memory or too many triggers already
TM_ENOERROR	Newly allocated trigger, ip's and ports initialized
TM_EINVAL	Erroneous use, for details enable TM_NAT_TRACE

tfNatConfigStatic

#include <trsocket.h>

```
int                 tfNatConfigStatic
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ipPublic,
ttUserIpAddress     ipPrivate
);
```

Function Description

Permanently associate a public IP with a private IP. Useful to support the maximum Configure as many static associations as you like on a public NAT interface, being sure to avoid conflicts of course (different public IP's) with other static IP configurations. Inner server configurations on the same public IP should come later or the static configuration would hide them dynamic IP configurations should never use the same IP address

Parameters

Parameter	**Description**
interface handle	Interface handle of the public NAT device on which to create a static association
ipPublic	Public IP address, network byte order
ipPrivate	Private IP address, network byte order

Return

Value	**Meaning**
TM_ENOMEM	Insufficient memory or too many triggers already
TM_ENOERROR	Newly allocated trigger, ip's and ports initialized
TM_EINVAL	Erroneous use, for details enable TM_NAT_TRACE

tfNatConfigDynamic

#include <trsocket.h>

```
int               tfNatConfigDynamic
(
ttUserInterface   interfaceHandle,
ttUserIpAddress   ipPublic
);
```

Function Description

Allow automatic temporary association of public with private IP addresses associations will expire after TM_NTRTTL_DYNAMIC seconds of no traffic may be called multiple times on a public NAT interface, with different public IP addresses. Later calls will specify the most used public IP addresses. Should be called after tfNatConfigStatic() if called. Otherwise the statically assigned private system may commandeer a dynamic IP address. This is not harmful, just unlikely to be useful.

Parameters

Parameter	**Description**
interface handle	Interface handle of the public NAT device on which to create a dynamic association
ipPublic	Public IP address, network byte order

Return

Value	**Meaning**
TM_ENOMEM	Insufficient memory or too many triggers already
TM_ENOERROR	Newly allocated trigger, ip's and ports initialized
TM_EINVAL	Erroneous use, for details enable TM_NAT_TRACE

tfNatConfigMaxEntries

#include <trsocket.h>

```
void                tfNatConfigMaxEntries
(
int                 maxentries
);
```

Function Description

Set the maximum number of triggers for NAT to track on a public interface. This maximum will apply to all public interfaces until changed again The default is 64 (TM_DEF_NAT_MAX_ENTRIES in trmacro.h)

Parameters

Parameter	**Description**
maxentries	Maximum simultaneous triggers

Returns

Return	**Meaning**
None	

tfNatDump

#include <trsocket.h>

```
void                tfNatDump
(
ttUserInterface     interfaceHandle
);
```

Function Description
Displays details of the current set of NAT triggers to stderr.
#define TM_NAT_DUMP in trsystem.h to enable, or comment out to save code.

Parameters

Parameter	Description
interfaceHandle	Interface handle of the NAT device from which to retrieve information

Returns

Return	Meaning
None	

PPP Interface

Introduction to PPP

The Point-to-Point Protocol (usually referred to as PPP) is a low level protocol that performs two basic functions. It establishes a link between two peers, and it maintains that link. This protocol communicates through the hardware layer to the PPP layer on a peer system. Because communication takes place between two machines that have the same authority in the negotiation process, neither system can be considered a client or a server.

Let's examine how PPP creates a link and communicates between two machines. Communication is established in essentially three phases:

Link Control Protocol (LCP)
Negotiates options and establishes the link between two systems.

Authentication
This takes place after LCP option negotiation. It is a request for a peer machine to identify itself with some form of password scheme. We will discuss some of these methods later in this document. While authentication is optional, it is widely used with ISP's and other dial-ins that require secure logins.

Network Control Protocol (NCP)
Configures upper level protocols to operate over PPP. For example, our implementation of PPP uses IPCP (Internet Protocol Control Protocol).

PPP Negotiation

The first phase, LCP, is responsible for negotiating various options that will be enabled, disabled, or given specific values during communication. LCP negotiation makes PPP more effective than SLIP, Kermit, Z-Modem, and many of the other early peer-to-peer protocols. Unlike PPP, these early protocols did not have robust schemes for negotiating options, and they were not extensible. Newer versions of PPP may contain more options than previous versions, but can still effectively communicate with them.

Each system knows how it would like to receive data (the most efficient way, or options required by a system administrator), and tells its peer by sending a Configure-Request (Conf-Req). A Conf-Req specifies options such as authentication, control field compression, address field compression, Maximum Receive Unit (MRU), Magic Number, and many others. If a system receives a Conf-Req and all of the requested options are acceptable it replies with a Configure-Acknowledge (Conf-Ack).

The Configure-Acknowledge tells the sender of the Conf-Req that all the requested options are acceptable, have been enabled, and it is ready to start communicating.

Because some systems may have certain options disabled, or have version of PPP that does not support all the options, they may not be able to comply with a set of options a peer might request. For example, if a machine requests the MRU option set to 3000 bytes, and the peer system could not send data in blocks that large, it would return a Configure-Negative-Acknowledge (Conf-Nak). The Conf-Nak includes the options that the receiver of the Conf-Req found unacceptable, and a suggestion for an appropriate value. When a system receives a Conf-Req that contains options it is unfamiliar with, it responds with a Configure-Reject (Conf-Rej). It is similar to the Conf-Nak, in that it contains only those options that are unacceptable but it does not offer an alternative. If a system receives a Conf-Rej reply to its Conf-Req, it must send a new Conf-Req without the unfamiliar options, and exclude them from any future Conf-Req messages.

Sample LCP Negotiation

Here is a sample negotiation between two machines:

System X sends a Conf-Req to System Y that asks for several options:

- I would like you to send my PPP datagrams in blocks of 3000 bytes (MRU option set to 3000)
- I can handle compressed protocol field compression; enable this option
- I would also like you to identify yourself, let's use CHAP.

System Y looks at what System X asked for, and formulates a response

- I don't know what protocol field compression is.

Because System Y doesn't understand protocol field compression, it must send a Conf-Rej containing that option.

After receiving the Conf-Rej, System X must now send another Conf-Req, but it may not ask for protocol field compression again.

- I would like you to send my PPP datagrams in blocks of 3000 bytes (MRU option set to 3000)
- I would like you to authenticate using CHAP

System Y looks at the new set of options from System X, and formulates another reply:

- I don't know how to use CHAP, let's use PAP instead

This last reply was a Conf-Nak. As you can see, this message does not necessarily instruct System X to discard these options. It merely tells the peer that certain attributes are unacceptable and offers values or attributes that are more agreeable.

System X must send another Conf-Req:

- I would like my PPP datagrams in 3000 byte blocks
- I would like you to identify yourself using PAP

Because System X accepted the hints offered by System Y and included them in this Conf-Req, System Y will send a Conf-Ack that would look like this:

- I am going to send you data in 3000 byte blocks
- I will use PAP to identify myself

This is only one half of the LCP process. Both systems will send Conf-Reqs nearly simultaneously. In terms of our example negotiation, System Y would also send a Conf-Req to describe how it would like to receive its data as well.

PPP and Authentication

On some systems, a secure login is desirable. These systems could be an ISP, a business network with some form of remote access, or any system that must be mindful of what other systems should or should not have access. LCP negotiates what authentication method if any, will be used. Authentication actually taking place is a different phase. There are a variety of authentication methods, and we will briefly examine the most common three:

Password Authentication Protocol (PAP)

This is one of the earliest and simplest of the authentication methods available. With this method, both peers know a secret and one will send a user name and password when it is requested. Of course, if authentication is intended to be bi-directional, there should be two sets of secrets. PAP's weakness is that the user name and password are sent across the wire in clear-text and leaves the possibility that these secrets can be intercepted.

Systems that use PAP typically do not use re-authentication, though it is not impossible. If re-authentication is necessary, LCP must renegotiate and utilize PAP again. For more information regarding PAP, please refer to RFC 1334.

Challenge-Handshake Authentication Protocol (CHAP)
This authentication method is more secure than PAP because it does not send a clear-text user name and password. Instead, when a peer wishes to validate another, it sends a name and a randomly generated number. The peer attempting to gain access uses the name given by the authenticator to obtain a plain text secret (it may prompt the user, or look it up in a database). The secret is then hashed with the random number and the result is sent to the authenticator. The original sender then uses the same algorithm and compares the result with the value it received from its peer. Based on the comparison, a success or failure message is sent to the system requesting access. Unlike PAP, CHAP has the ability to send periodic re-authentication requests without renegotiating. More information on CHAP can be found in RFC 1994.

Microsoft-Challenge-Handshake Authentication Protocol (MS-CHAP)
This authentication method is not currently supported in the Turbo Treck stack. This authentication method is very similar to regular CHAP except for a few details: Standard CHAP uses 05 for the hashing algorithm while MS-CHAP uses 08. It does not require the authenticator to store a clear-text or reversibly encrypted password. The authenticator has the ability to choose the number of retries as well as the ability to change passwords. Unlike CHAP, MS-CHAP will give a reason for failure (for example: incorrect password). More information on MS-CHAP is available in RFC 2433.

Internet Protocol Control Protocol (IPCP)

The Internet Protocol Control Protocol enables, disables, and configures various IP modules on each end of the peer-to-peer link. This protocol has its own list of options such as various types of compression, MTU discovery, specifying IP addresses, and several more. It is important to remember no IP packets can be sent before PPP is in the NCP phase. In terms of the current Turbo Treck stack, the NCP phase consists solely of IPCP. This means a PPP link is first negotiated with LCP, authentication takes place (if it has been requested in LCP), and then IPCP establishes communication with the IP layer on the peer machine. For detailed information about the options within IPCP, please refer to RFC 1332.

IPCP in the Turbo Treck stack also includes options to obtain the addresses of Domain Name Servers (DNS servers). Domain name servers are remote servers that match domain names (such as elmic.com) to an appropriate IP address. Detailed information on the DNS options within IPCP and implementation can be found in RFC 1877.

tfChapRegisterAuthenticate

```
#include <trsocket.h>

int                 tfChapRegisterAuthenticate
(
ttUserInterface                interfaceHandle,
ttChapAuthenticateFunctPtr     authFunctionPtr
);
```

Function Description

When acting as a PPP server, a function must be called to validate an incoming authentication request. This function is passed the username, and will return the secret (password) for this particular username. Chap module will further compute the challenge using this secret, and compare with the challenge given by the peer. If both challenge match, the peer is positively authenticated, otherwise, the peer is denied.

The prototype for the authentication function is defined to be:

```
char *myChapAuthenticate(char *username);
```

which returns the secret corresponding to this user.

If the user name passed to myChapAuthenticate is invalid, this function should return TM_CHAP_INVALID_USER. Should the password for this user be empty, this function should return TM_CHAP_NULL_PASSWORD.

Parameters

Parameter	Description
interfaceHandle	The interface for which to use this authentication routine
authFunctionPtr	The pointer to the function to authenticate the remote user.

Returns

Value	Meaning
0	Success
TM_EINVAL	The interface handle is invalid

tfMsChapRegisterAuthenticate

```
#include <trsocket.h>

int tfMsChapRegisterAuthenticate
(
ttUserInterface                interfaceHandle,
ttMsChapAuthenticateFunctPtr   authFunctionPtr
);
```

Function Description
This function works with MS-CHAP only. (You need to define TM_USE_PPP_MSCHAP in trsystem.h to use MS-CHAP) When acting as a PPP server, a function must be called to validate an incoming authentication request. This function is passed the username, and will return the secret (password) and the secret length for this particular username. Chap module will further compute the challenge using this secret, and compare with the challenge given by the peer. If both challenge match, the peer is positively authenticated, otherwise, the peer is denied.

The prototype for the authentication function is defined to be:

```
char *myMsChapAuthenticate(char *username, int *
secretLenPtr);
```

which returns the secret corresponding to this username, and the length of secret will be returned in secretLenPtr.
If the user name passed to myChapAuthenticate is invalid, this function should return TM_CHAP_INVALID_USER. Should the password for this user be empty,
this function should return TM_CHAP_NULL_PASSWORD.

Parameters

Parameter	**Description**
interfaceHandle	The interface for which to use this authentication routine
authFunctionPtr	The pointer to the function to authenticate the remote user.

Returns

Value	Meaning
0 Success	
TM_EINVAL	The interface handle is invalid

tfMsChapRegisterNewPassword

```
#include <trsocket.h>

int tfMsChapRegisterNewPassword
(
ttUserInterface                interfaceHandle,
ttMsChapAuthenticateFunctPtr   authFunctionPtr
);
```

Function Description

This function works with MS-CHAP only. It is used by the peer when it receives message from the authenticator that an expired password was used. The peer will call the authFunctionPtr registered here to get a new password, and then send Change_Password packet to the authenticator in order to finish the authentication processing.

The prototype for the authentication function is defined in previous page

Parameters

Parameter	**Description**
interfaceHandle	The interface for which to use this authentication routine
authFunctionPtr	The pointer to the function to authenticate the remote user.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	The interface handle is invalid

tfGetPppDnsIpAddress

```
#include <trsocket.h>

int                 tfGetPppPeerIpAddress
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     dnsIpAddressPtr,
int                 flag
);
```

Function description

This function is used to return the DNS Addresses as negotiated by the remote PPP server. This function can only be used with PPP devices. If no DNS address is negotiated, the IP address returned will be 0.0.0.0

Parameters

Parameter	Description
interface handle	The interface handle to get the DNS IP address from.
dnsIpAddressPtr	The pointer to the buffer where the DNS IP address will be stored
flag	One of the following: TM_DNS_PRIMARY or TM_DNS_SECONDARY

Returns

Value	Meaning
0	Success
TM_EINVAL	One of the parameters is null or the device is a LAN device
TM_ENETDOWN	Interface is not configured

tfGetPppPeerIpAddress

```
#include <trsocket.h>

int                 tfGetPppPeerIpAddress
(
ttUserInterface     interfaceHandle,
ttUserIpAddress *   ifIpAddress
);
```

Function Description

This function is used to get the PPP address that the remote PPP has used (respectively SLIP address of the remote SLIP). This function should be called after **tfOpenInterface** has completed successfully.

If a default gateway needs to be added for that interface, then the retrieved IP address should be used to add a default gateway through the corresponding interface.

If a static route needs to be added for that interface, then the retrieved IP address should be used to add a static route through the corresponding interface.

Parameters

Parameter	**Description**
interfaceHandle	The interface Handle to get the Peer IP address from.
ifIpAddressPtr	The pointer to the buffer where the Peer PPP IP address will be stored into.

Returns

Value	**Meaning**
0	Success
TM_EINVAL	One of the parameters is null, or the device is a LAN device
TM_ENETDOWN	Interface is not configured

tfPapRegisterAuthenticate

```
#include <trsocket.h>

int                          tfPapRegisterAuthenticate

(
ttUserInterface              interfaceHandle,
ttPapAuthenticateFunctPtr    authFunctionPtr
);
```

Function Description

When acting as a PPP server, a function must be called to validate an incoming authentication request. This function is passed the username and password from the peer by the PPP layer. The function must return 1 for a valid username/password or 0 for an invalid username/password.

The prototype for the authentication function is defined to be:

```
int myPapAuthenticate( char *username, char password);
```

Parameters

Parameter	**Description**
interfaceHandle	The interface to use this authentication routine for
authFunctionPtr	The function to call to authenticate the remote user

Returns

Value	**Meaning**
0	Success
TM_EINVAL	The interface handle is invalid

tfPppSetOption

```
#include <trsocket.h>

int                tfPppSetOption
(
ttUserInterface    interfaceHandle,
int                protocolLevel,
int                remoteLocalFlag,
int                optionName,
const char *       optionValuePtr,
int                optionLength
);
```

Function Description

This function is used to set the PPP options that we wish to negotiate as well as those options that we will allow. This allows us to change the link away from the default parameters described in RFC1661.

Parameters

Parameter	Description
interfaceHandle	The Interface handle to set these options for
protocolLevel	The protocol which this option should be applied. Current supported protocols are: TM_PPP_LCP_PROTOCOL TM_PPP_IPCP_PROTOCOL TM_PPP_PAP_PROTOCOL TM_PPP_CHAP_PROTOCOL
remoteLocalFlag	This flag describes whether the option is for what we want to use for our side of the link (TM_PPP_OPT_WANT) or if it what we will allow the remote side to use (TM_PPP_OPT_ALLOW)
optionName	The name of the option (see below)
optionValuePtr	The value of the option (see below)
optionLength	The length of the option (see below)

LCP (TM_PPP_LCP_PROTOCOL):

Option Name	**Len**	**Meaning**
TM_LCP_ADDRCONTROL_COMP	1	A boolean value specifiying whether address field compression should be used. **Default: OFF**
TM_LCP_PROTOCOL_COMP	1	A boolean value specifying whether protocol field compression should be used. **Default: OFF**
TM_LCP_MAGIC_NUMBER	1	A boolean value indicating whether to specify a magic number. **Default: OFF**
TM_LCP_MAX_FAILURES	1	Sets the maximum number of LCP configuration failures. This determines the maximum number of configuration NAKs that will be sent before we reject an option. **Default: 5**
TM_LCP_MAX_RECV_UNIT	2	Specifies the largest MRU that we will allow and the default MRU that we want to use. **Default: 1500**
TM_LCP_ACCM	4	Specifies the async control character map that we want to use, and if we want to allow the remote side to be able to set his ACCM.**Default: 0xffffffff**
TM_LCP_AUTH_PROTOCOL	2	The protocol to use when authenticating to our peer, and vice versa (e.g. PAP or CHAP). Possible value are TM_PPP_PAP_PROTOCOL and

		TM_PPP_CHAP_PROTOCOL. **Default: No authentication**
TM_LCP_TERM_RETRY	1	Sets the maximum number of Terminate requests that the local peer will send (without receiving a Terminate Ack) before terminating the connection. **Default: 3**
TM_LCP_CONFIG_RETRY	1	Sets the maximum number of LCP config requests that will be sent without receiving a LCP ack/nak/reject. remoteLocalFlag has no effect.**Default: 10**
TM_LCP_TIMEOUT	1	Sets the LCP retransmission timeout in seconds. **Default: 3 seconds**
TM_LCP_QUALITY_PROTOCOL	4	Setting this option enables link quality monitoring. The option value is specified in hundredths of a second, and it configures the maximum time to delay (i.e. LQR timer period) between sending Link-Quality-Report messages (refer to RFC-1989, Reporting-Period field of the Quality-Protocol Configuration Option). This option can be set for either the local or the remote end of the link, however the direction in which it applies is the opposite of what one would expect: when *remoteLocalFlag* is set to TM_PPP_OPT_WANT, this specifies an option value that we want the remote end of the link to use, and when *remoteLocalFlag* is set to TM_PPP_OPT_ALLOW,

		this specifies an option value that we will allow the remote end to configure us to use. If a non-zero option value is specified, the LQR timer is started with the specified timeout period to pace the sending of Link-Quality-Report messages, and this timer is restarted whenever a Link-Quality-Report message is sent. If the specified option value is 0, no LQR timer is used, but instead a Link-Quality-Report message is sent as a response every time one is received from the peer. At least one side of the link must use the LQR timer to pace the sending of Link-Quality-Report messages when link quality monitoring is enabled, therefore this option value should not be set to 0 for both ends of the link.

LQM (TM_LCP_QUALITY_PROTOCOL)

Option Name	**Len**	**Meaning**
TM_LCP_QUALITY_PROTOCOL	4	Setting this option enables link quality monitoring. The option value is specified in hundredths of a second, and it configures the maximum time to delay (i.e. LQR timer period) between sending Link-Quality-Report messages (refer to RFC-1989, Reporting-Period field of the Quality-Protocol Configuration Option). This option can be set for either the local or the remote end of the link,

		however the direction in which it applies is the opposite of what one would expect: when *remoteLocalFlag* is set to TM_PPP_OPT_WANT, this specifies an option value that we want the remote end of the link to use, and when *remoteLocalFlag* is set to TM_PPP_OPT_ALLOW, this specifies an option value that we will allow the remote end to configure us to use. If a non-zero option value is specified, the LQR timer is started with the specified timeout period to pace the sending of Link-Quality-Report messages, and this timer is restarted whenever a Link-Quality-Report message is sent. If the specified option value is 0, no LQR timer is used, but instead a Link-Quality-Report message is sent as a response every time one is received from the peer. At least one side of the link must use the LQR timer to pace the sending of Link-Quality-Report messages when link quality monitoring is enabled, therefore this option value should not be set to 0 for both ends of the link.

IPCP (TM_PPP_IPCP_PROTOCOL)

Option Name	**Len**	**Meaning**
TM_IPCP_COMP_PROTOCOL	2	Specifies the type of compression to use over the link. Currently, only VJ TCP/IP header compression is available which is defined

		as TM_PPP_COMP_TCP_PROTOCOL **Default: No Compression**
TM_IPCP_MAX_FAILURES	1	Sets the maximum number of IPCP configuration failures. This determines the maximum number of configuration NAKsthat will be sent before we reject an option. **Default: 5**
TM_IPCP_VJ_SLOTS	1	The number of slots used to store state information for each side of a VJ compressed link. This value is determined by the maximum number of concurrent TCP sessions that you will have. **Default: 1 slot.**
TM_IPCP_IP_ADDRESS	4	Specifies if we want to allow the remote to set their IP address. Please see “setting a peer PPP IP address” below for explanations. **Default: Don’t Allow**
TM_IPCP_DNS_PRI	4	Specifies the IP addresses of the Primary DNS Server we will allow the remote to use or the Primary DNS server that we ant to us. Se the section setting a PPP IP Address **Default: Don’t Allow**
TM_IPCP_DNS_SEC	4	Specifies the IP Address of the Secondary DNS server we will allow the remote to use or the Secondary DNS server that we want to use. Se the section setting a PPP IP Address. **Default: Don’t Allow**

PPP (PPP_PROTOCOL)

Option Name	Len	Meaning
TM_PPP_SEND_BUFFER_SIZE	2	Length of data buffered by the PPP link layer (but not beyond the end of a packet) before the device driver send function is called. **Default: 1 byte**
TM_IPCP_RETRY	1	Sets the maximum number of IPCP config requests that will be sent without receiving a IPCP nak/ack/reject. remoteLocalFlag has no effect. **Default: 10**
TM_IPCP_TIMEOUT	1	Sets the IPCP retransmission timeout value (in seconds). remoteLocalFlag has no effect. **Default: 1 Second**

TM_PPP_PROTOCOL

Option Name	Len	Meaning
TM_PPP_SEND_BUFFER_SIZE	2	Length of data buffered by the PPP link layer (but not beyond the end of a packet) before the device driver send function is called.**Default: 1 byte**

Setting a PPP IP address:

The following applies to all of the IP Address optionNames TM_IPCP_IP_ADDRESS, TM_IPCP_DNS_PRI and TM_IPCP_DNS_SEC.

- If **tfPppSetOption** is not used with the IP Address optionName (default) then the remote will not be allowed to request its IP and/or DNS IP addresses.

- If **tfPppSetOption** is called with the IP Address optionNames, remoteLocalFlag TM_PPP_OPT_ALLOW, and optionValuePtr points to an IP address whose value is 0.0.0.0, then the remote will be allowed to request that its IP/DNS IP address be set to anything except 0.0.0.0.

  For example:

```
ttUserIpAddress remoteIpAddress;
remoteIpAddress = inet_addr (“0.0.0.0”);
      tfPppSetOption (interfaceHandle,
      TM_PPP_IPCP_PROTOCOL,
      TM_PPP_OPT_ALLOW,
      TM_IPCP_IP_ADDRESS,
      (const char *)&remoteIpAddress, 4);
```

- If **tfPppSetOption** is called with an IP Address optionName, remoteLocalFlag TM_PPP_OPT_ALLOW, and optionValuePtr points to an IP address whose value is not 0.0.0.0, two situations may occur. The remote will be allowed to set its IP/DNS IP address to this value, or will be returned this value, if it requests 0.0.0.0.

PAP (TM_PPP_PAP_PROTOCOL):

Option Name	**Len**	**Meaning**
TM_PAP_USERNAME	Any	Sets the username to use with PAP Client **Default: NONE**
TM_PAP_PASSWORD	Any	Sets the password to use with PAP **Default: NONE**
TM_PAP_RETRY	1	Sets the maximum number of PAP authentication requests that will be sent without receiving an ACK/NAK remoteLocalFlag has no effect. **Default: 10**
TM_PAP_TIMEOUT	1	Sets the PAP retransmission timeout value in seconds. remoteLocalFlag has no effect. **Default: 3 Seconds**

CHAP (TM_PPP_CHAP_PROTOCOL):

Option Name	Len	Meaning
TM_CHAP_USERNAME	Any	Sets the username to use with CHAP Client **Default: NONE**
TM_CHAP_SECRET	Any	Sets the secret to use with CHAP **Default: NONE**
TM_CHAP_RETRY	1	Sets the maximum number of CHAP challenges that will be sent without receiving a CHAP response remoteLocalFlag has no effect. **Default: 10**
TM_CHAP_TIMEOUT	1	Sets the CHAP retransmission timeout value in seconds. remoteLocalFlag has no effect. **Default: 3 Seconds**
TM_CHAP_ALG_ADD	1	Add a CHAP algorithm. You don't need to add TM_CHAP_MD5, because it is automatically added when TM_PPP_CHAP_PROTOCOL is used. You may add TM_CHAP_MSV1 in order to support MS_CHAP version 1.
TM_CHAP_ALG_DEL	1	Delete a CHAP algorithm. You may delete TM_CHAP_MD5 (standard CHAP) or TM_CHAP_MSV1 (MS-CHAP version 1)

Returns

Value	Meaning
0	Success
TM_ENOPROTOPT	*protocolLevel* or *optionName* is invalid
TM_EINVAL	The option value or length is invalid

optionLength

optionLength should be the size of the data type, *optionValuePtr* is pointing to, except for the TM_PAP_USERNAME, TM_PAP_PASSWORD, TM_CHAP_USERNAME, and TM_CHAP_SECRET options, where *optionValuePtr* points to the first byte of an array, and where *optionLength* is the size of the array *optionValuePtr* is pointing to.

The data types are as follows:

ProtocolLevel	**OptionName**	**data type**
TM_PPP_LCP_PROTOCOL	TM_LCP_ADDRCONTROL_COMP	unsigned char
	TM_LCP_PROTOCOL_COMP	unsigned char
	TM_LCP_MAGIC_NUMBER	unsigned char
	TM_LCP_MAX_RECV_UNIT	unsigned short
	TM_LCP_ACCM	unsigned long
	TM_LCP_AUTH_PROTOCOL	unsigned short
	TM_LCP_TERM_RETRY	unsigned char
	TM_LCP_CONFIG_RETRY	unsigned char
	TM_LCP_TIMEOUT	unsigned char
TM_PPP_IPCP_PROTOCOL	TM_IPCP_COMP_PROTOCOL	unsigned short
	TM_IPCP_VJ_SLOTS	unsigned char
	TM_IPCP_IP_ADDRESS	unsigned long
	TM_IPCP_RETRY	unsigned char
	TM_IPCP_TIMEOUT	unsigned char
TM_PPP_PAP_PROTOCOL	TM_PAP_USERNAME	char
	TM_PAP_PASSWORD	char
	TM_PAP_RETRY	unsigned char
	TM_PAP_TIMEOUT	unsigned char
TM_PPP_CHAP_PROTOCOL	TM_CHAP_USERNAME	char
	TM_CHAP_SECRET	char
	TM_CHAP_RETRY	unsigned char
	TM_CHAP_TIMEOUT	unsigned char
TM_PPP_PROTOCOL	TM_PPPSEND_BUFFER_SIZE	unsigned short

tfSetPppPeerIpAddress

```
#include <trsocket.h>

int                 tfSetPppPeerIpAddress
(
ttUserInterface     interfaceHandle,
ttUserIpAddress     ifIpAddress
);
```

Function Description

This function is used to set a default remote PPP/SLIP IPaddress. This IP address will be used as the default remote point to point IP address, in case no remote IP address is negotiated with PPP (see **tfPppSetOption**), or for SLIP. If no IP address is set with this function, (and no IP address is negotiated with the remote PPP for PPP), then the local IP address + 1 will be used as the default IP address for the remote PPP (or SLIP) for routing purposes (see **tfGetPppPeerIpAddress**). **tfSetPppPeerIpAddress** can only be called between **tfAddInterface** (or **tfCloseInterface**) and **tfOpenInterface**.

Parameters

Parameter	Description
interfaceHandle	The interface handle to update the Peer IP address in
ifIpAddress	The IP address to use for routing purposes for the remote PPP system

Returns

Value	Meaning
0	Success
TM_EINVAL	The interface handle is null, or the device is a LAN device
TM_EISCONN	PPP connection is already established

tfUseAsyncPpp

#include <trsocket.h>

```
ttUserLinkLayer          tfUseAsyncPpp
(
ttUserLnkNotifyFuncPtr   linkNotifyFuncPtr
);
```

Function Description

This function is used to initialize the asynchronous PPP client link layer. When link up or link down events occur, the stack will call the function passed in. If you do not need notification of events then the parameter should be set to TM_LNK_NOTIFY_FUNC_NULL_PTR.

Your prototype for your notification function should look like this:

```
void myPppNotifyFunction(ttUserInterface interfaceHandle,
                         int             flags);
```

This function is called with the interface handle and a flag. The flag is set to one of the following:

TM_LL_OPEN_COMPLETE: PPP Device is ready to accept data from the user.
TM_LL_CLOSE_STARTED: PPP Device has started to close this link.
TM_LL_CLOSE_COMPLETE: PPP Device has closed
TM_LL_LCP_UP: LCP negotiation has completed.
TM_LL_PAP_UP: PAP authentication has completed.
TM_LL_CHAP_UP: CHAP authentication has completed.
TM_LL_LQM_UP: LQM is enabled on the link.
TM_LL_LQM_DISABLED: LQM is disabled on the link.
TM_LL_LQM_LINK_BAD: Link quality is bad, user recovery should be attempted.

These events may be used to monitor the status of the PPP connection. The PPP connection should not be used before a TM_LL_OPEN_COMPLETE event is received, and the device should not be restarted (after a **tfCloseInterface**) before a TM_LL_CLOSE_COMPLETE event is received.

From these events, it is also possible to determine why PPP negotiation failed. For instance, if authentication fails, a TM_LL_LCP_UP event is first received indicating that the physical link has been negotiated. However, if authentication fails, the next event will be TM_LL_CLOSE_STARTED and then TM_LL_CLOSE_COMPLETE since the link must be closed if negotiation fails. If authentication is successful the events that are received are TM_LL_LCP_UP, TM_LL_PAP_UP or TM_LL_CHAP_UP and then TM_LL_OPEN_COMPLETE.

Parameters

Parameter	**Description**
linkNotifyFuncPtr	The function to call to notify PPP events or TM_LNK_NOTIFY_FUNC_NULL_PTR if notification is not needed The events are: TM_LL_OPEN_COMPLETE TM_LL_CLOSE_STARTED TM_LL_CLOSE_COMPLETE TM_LL_LCP_UP TM_LL_PAP_UP TM_LL_CHAP_UP

Returns

The PPP Client link layer handle or NULL if there is an error

tfUseAsyncServerPpp

#include <trsocket.h>

```
ttUserLinkLayer          tfUseAsyncServerPpp
(
ttUserLnkNotifyFuncPtr   linkNotifyFuncPtr
);
```

Function Description

This function is used to initialize the asynchronous PPP server link layer. When link up or link down events occur, the stack will call the function passed in. If you do not need notification of events then the parameter should be set to TM_LNK_NOTIFY_FUNC_NULL_PTR.

Your prototype for your notification function should look like this:

```
void myPppNotifyFunction(ttUserInterface interfaceHandle,
                         int             flags);
```

This function is called with the interface handle and a flag. The flag is set to one of the following:

TM_LL_OPEN_COMPLETE: PPP Device is ready to accept data from the user.
TM_LL_CLOSE_STARTED: PPP Device has started to close this link.
TM_LL_CLOSE_COMPLETE: PPP Device has closed
TM_LL_LCP_UP: LCP negotiation has completed.
TM_LL_PAP_UP: PAP authentication has completed.
TM_LL_CHAP_UP: CHAP authentication has completed.
TM_LL_LQM_UP: LQM is enabled on the link.
TM_LL_LQM_DISABLED: LQM is disabled on the link.
TM_LL_LQM_LINK_BAD: Link quality is bad, user recovery should be attempted.

These events may be used to monitor the status of the PPP connection. The PPP connection should not be used before a TM_LL_OPEN_COMPLETE event is received, and the device should not be restarted (after a **tfCloseInterface**) before a TM_LL_CLOSE_COMPLETE event is received.

From these events, it is also possible to determine why PPP negotiation failed. For instance, if authentication fails, a TM_LL_LCP_UP event is first received indicating that the physical link has been negotiated. However, if authentication fails, the next event will be TM_LL_CLOSE_STARTED and then TM_LL_CLOSE_COMPLETE since the link must be closed if negotiation fails. If authentication is successful the events that are received are TM_LL_LCP_UP, TM_LL_PAP_UP or TM_LL_CHAP_UP and then

TM_LL_OPEN_COMPLETE.

Parameters

Parameter	**Description**
linkNotifyFuncPtr	The function to call to notify PPP events or TM_LNK_NOTIFY_FUNC_NULL_PTR if notification is not needed The events are: TM_LL_OPEN_COMPLETE TM_LL_CLOSE_STARTED TM_LL_CLOSE_COMPLETE TM_LL_LCP_UP TM_LL_PAP_UP TM_LL_CHAP_UP

Returns

The PPP Server link layer handle or NULL if there is an error

tfPppSetAuthPriority

```
int   tfPppSetAuthPriority
(
ttUser8Bit    authMethod,
ttUser8Bit    priorityValue
);
```

Function Description
User calls this routine to set priority value for PPP authentication methods. The priority value can be any integar between 1 (with highest priority) and 15 (lowest priority), inclusive. The authenticator will try to negotiate authentication method with the highest priority. If that authentication method is NAK-ed by the peer, it will choose the second one according to the priority value. The less the priority value, the higher priority to use.

The default priority value for the following authentication method is :

TM_PPP_AUTHMETHOD_EAP	1
TM_PPP_AUTHMETHOD_MSCHAP_V2	2 (not supported yet)
TM_PPP_AUTHMETHOD_CHAP	3
TM_PPP_AUTHMETHOD_PAP	4
TM_PPP_AUTHMETHOD_MSCHAP_V1	5

Note that, whenever user calls tfPppSetAuthPriority, the default priority value is gone. User must call tfPppSetAuthPriority for all authentication methods if they want to change the default priority sequence.

EXAMPLE:
For example, if user prefers the following priority sequence:MSCHAPv1———CHAP——PAP, then user may call

```
tfPppSetAuthPriority(TM_PPP_AUTHMETHOD_MSCHAP_V1, 1);
tfPppSetAuthPriority(TM_PPP_AUTHMETHOD_CHAP, 2);
tfPppSetAuthPriority(TM_PPP_AUTHMETHOD_PAP, 3);
```

RETURN:

TM_ENOERROR	successful return
TM_EINVAL	invalid parameter

PPP EXAMPLE

Authenticator Policy: Authenticator wants the peer to authenticate itself by using PAP, CHAP or MS-CHAPv1, with PAP to be the most preferred one, and MS-CHAPv1 the least preferred.

Peer Policy: Supports MS-CHAPv1 only, needs Authenticator to set our IP address.

Expected behavior: (for authentication protocol)

1. Authenticator sends LCP config request with AUTH= PAP
2. Peer NAKs it with MS-CHAPv1
3. Authenticator processes the NAK packet, and sends LCP config request with AUTH = CHAP. (not MS-CHAPv1, because server prefers CHAP more)
4. Peer NAKs it with MS-CHAPv1
5. Authenticator processes the NAK packet, and sends LCP config request with AUTH=MS-CHAPv1
6. Peer ACKs it

Authenticator Part:

```
{
        linkLayerHandle = tfUseAsyncServerPpp
                  ((ttLnkNotifyFuncPtr)tfPppLinkNotify);
        interfaceHandle = tfUseXxxDriver("TEST",
                  linkLayerHandle, &errorCode);

/* 1. Authenticator wants IP to be 192.168.0.2. Au-
thenticator already  knows its IP address  */

        errorCode=tfPppSetOption(interfaceHandle,
                            (int)TM_PPP_IPCP_PROTOCOL,
                             TM_PPP_OPT_WANT,
                             TM_IPCP_IP_ADDRESS,
                             (char *) &ifIPAddr,
/* 192.168.0.2 */
                             4);
        assert (errorCode == TM_ENOERROR);

/* 2. Authenticator wants PAP, CHAP, MSCHAPv1 and in
                  this order */

        papValue=(int)TM_PPP_PAP_PROTOCOL;
```

```
/* net work  order*/
        chapValue=(int)TM_PPP_CHAP_PROTOCOL;
/* net work order */

/* 2.1 want PAP protocol */
        errorCode=tfPppSetOption(interfaceHandle,
                     (int)TM_PPP_LCP_PROTOCOL,
                      TM_PPP_OPT_WANT,
                      TM_LCP_AUTH_PROTOCOL,
                      (char *) &papValue,
                      2);
        assert (errorCode == TM_ENOERROR);
/* 2.2 want CHAP protocol */
        errorCode=tfPppSetOption(interfaceHandle,
                     (int)TM_PPP_LCP_PROTOCOL,
                      TM_PPP_OPT_WANT,
                      TM_LCP_AUTH_PROTOCOL,
                      (char *) &chapValue,
                      2);
        assert (errorCode == TM_ENOERROR);

/* 2.3 want MS-CHAP v1 (you must define
TM_USE_PPP_MSCHAP). Since we already add
TM_PPP_CHAP_PROTOCOL in TM_LCP_AUTH_PROTOCOL, here we
just need to add another algorithm for CHAP. */

#ifdef TM_USE_PPP_MSCHAP
        chapAlgorithm = TM_CHAP_MSV1; /* 0x80 */
        errorCode=tfPppSetOption(interfaceHandle,
                    (int)TM_PPP_CHAP_PROTOCOL,
                      TM_PPP_OPT_WANT,
                      TM_CHAP_ALG_ADD,
                      (char *) &chapAlgorithm,
                      1);
        assert (errorCode == TM_ENOERROR);
#endif /* TM_USE_PPP_MSCHAP */

/* 2.4 allows the peer to use this username for CHAP. */

        errorCode=tfPppSetOption(interfaceHandle,
                    (int)TM_PPP_CHAP_PROTOCOL,
                      TM_PPP_OPT_ALLOW,
                      TM_CHAP_USERNAME,
                      username,
                      strlen(username));
        assert (errorCode == TM_ENOERROR);

/* 2.5 set the priority order to be PAP – CHAP – MS-
                  CHAP */

        errorCode =
                  tfPppSetAuthPriority(TM_PPP_AUTHMETHOD_PAP,
                  1);
        assert (errorCode == TM_ENOERROR);
        errorCode =
                  tfPppSetAuthPriority(TM_PPP_AUTHMETHOD_CHAP,
                  2);
        assert (errorCode == TM_ENOERROR);
#ifdef TM_USE_PPP_MSCHAP
        errorCode =
                  tfPppSetAuthPriority(TM_PPP_AUTHMETHOD_MSCHAP_V1,
                  3);
        assert (errorCode == TM_ENOERROR);
#endif /* TM_USE_PPP_MSCHAP */

/* 3  Authenticator registers PAP, CHAP, MS-CHAP
authentication functions. */
/* 3.1 Register PAP authenticate function */

     errorCode=tfPapRegisterAuthenticate(interfaceHandle,
                              myPapAuthenticate);
        assert (errorCode == TM_ENOERROR);

/* 3.2 Register CHAP authenticate function */

     errorCode=tfChapRegisterAuthenticate(interfaceHandle,
                              myChapAuthenticate);
        assert (errorCode == TM_ENOERROR);

/* 3.2 Register MS-CHAP authenticate function*/
#ifdef TM_USE_PPP_MSCHAP
     errorCode=tfMsChapRegisterAuthenticate(interfaceHandle,
                              myMsChapAuthenticate);
        assert (errorCode == TM_ENOERROR);
#endif /* TM_USE_PPP_MSCHAP */

/* Open the interface */
        errorCode = tfOpenInterface( interfaceHandle,
                                     ifIPAddr,
                                     mask,
                                              configFlags,
                                          scatteredBufferCount
                          );
...
}

int myPapAuthenticate(char *username, char *password)
{
    search authenticator’s database to get the password of this username
    if the password matches return 1
    else password doesn’t match return 0
}

char* myChapAuthenticate(char *username)
{
    search authenticator’s database to get the password of this username
    return the password pointer
}
#ifdef TM_USE_PPP_MSCHAP
char* myMsChapAuthenticate(char *username, int * lenPtr)
{
    search authenticator’s database to get the password of this username
    calculate the UNICODE password length (in octets), and put the value in
lenPtr
    return the password pointer
}
#endif  /* TM_USE_PPP_MSCHAP */
```

Peer Part:

```
{
        linkLayerHandle = tfUseAsyncPpp
                  ((ttLnkNotifyFuncPtr)tfPppLinkNotify);
        interfaceHandle = tfUseXxxDriver(“TEST”,
                  linkLayerHandle, &errorCode);

/* 1. Peer allows the authenticator to set
authenticator’s IP address to the known value  */

        errorCode=tfPppSetOption(interfaceHandle,
                            (int)TM_PPP_IPCP_PROTOCOL,
                             TM_PPP_OPT_ALLOW,
                             TM_IPCP_IP_ADDRESS,
                             (char *) &serverIPAddr,
                  /* 192.168.0.2 */
```

```
                             4);
        assert (errorCode == TM_ENOERROR);

/* 2. Peer allows the authenticator to set peer’s IP
address to any non-zero address the authenticator
wants  */

        errorCode=tfPppSetOption(interfaceHandle,
                            (int)TM_PPP_IPCP_PROTOCOL,
                             TM_PPP_OPT_WANT,
                             TM_IPCP_IP_ADDRESS,
                             (char *) &zeroIPAddr,
/*0.0.0.0 */
                             4);
        assert (errorCode == TM_ENOERROR);

/* 3. Peer allows MSCHAPv1 only */

        chapValue=(int)TM_PPP_CHAP_PROTOCOL;
/* net work order */

/* 3.1 Allow CHAP protocol */

        errorCode=tfPppSetOption(interfaceHandle,
                             (int)TM_PPP_LCP_PROTOCOL,
                             TM_PPP_OPT_ALLOW,
                             TM_LCP_AUTH_PROTOCOL,
                             (char *) &chapValue,
                             2);
        assert (errorCode == TM_ENOERROR);

/* 3.2 Delete standard CHAP algorithm TM_CHAP_MD5 */

        char chapAlgorithm = TM_CHAP_MD5;
        errorCode=tfPppSetOption(interfaceHandle,
                            (int)TM_PPP_CHAP_PROTOCOL,
                             TM_PPP_OPT_ALLOW,
                             TM_CHAP_ALG_DEL,
                             (char *) &chapAlgorithm,
                             1);
        assert (errorCode == TM_ENOERROR);

/* 3.3 Adds MS-CHAP v1 algorithm TM_CHAP_MSV1 */
#ifdef TM_USE_PPP_MSCHAP
```

```
        chapAlgorithm = TM_CHAP_MSV1;
        errorCode=tfPppSetOption(interfaceHandle,
                            (int)TM_PPP_CHAP_PROTOCOL,
                             TM_PPP_OPT_ALLOW,
                             TM_CHAP_ALG_ADD,
                             (char *) &chapAlgorithm,
                             1);
        assert (errorCode == TM_ENOERROR);
#endif /* TM_USE_PPP_MSCHAP */


/* 3.4 Peer wants to use this username */
        errorCode=tfPppSetOption(interfaceHandle,
                            (int)TM_PPP_CHAP_PROTOCOL,
                             TM_PPP_OPT_WANT,
                             TM_CHAP_USERNAME,
                             username,
                             strlen(username));
        assert (errorCode == TM_ENOERROR);

/* 3.5 Peer wants to use this password */
        errorCode=tfPppSetOption(interfaceHandle,
                            (int)TM_PPP_CHAP_PROTOCOL,
                             TM_PPP_OPT_WANT,
                             TM_CHAP_SECRET,
                             password,
                             8);
/* Note: You generally can't use strlen here for MS-
CHAP unicode password length, count the octets in-
stead */

        assert (errorCode == TM_ENOERROR);

/* 3.6 Peer register new password function to get new
password if the old one expires */

#ifdef TM_USE_PPP_MSCHAP
        errorCode=tfMsChapRegisterNewPassword(
                              interfaceHandle,
                              myMsChapNewPassword);
        assert (errorCode == TM_ENOERROR);
#endif /* TM_USE_PPP_MSCHAP */
```

```
/* Open the interface using 0.0.0.0, we will get one
later from the authenticator */

        errorCode = tfOpenInterface( interfaceHandle,
                                     zeroIPAddr,
/*0.0.0.0*/
                                     mask,
                                     configFlags,
                                  scatteredBufferCount
                  );
...
}



#ifdef TM_USE_PPP_MSCHAP
char* myMsChapNewPassword (char *username, int *
                  lenPtr)
{
      search peer’s database to get the NEW password
                  of this username
      calculate the UNICODE password length (in oc-
                  tets), and put the value in lenPtr
      return the password pointer
}
#endif /* TM_USE_PPP_MSCHAP */
```

PPP Link Quality Monitoring (LQM)

PPP LQM enables the application to determine when PPP link quality has degraded to the point where recovery is necessary. PPP LQM does this via the exchange of Link-Quality-Report protocol messages at a negotiated interval (referred to as the Reporting-Period of the LQR timer). A Link-Quality-Report message contains the sender's state information w/ regards to how many packets and bytes have been successfully sent and received on the link. When the peer receives the Link-Quality-Report message, it can compare these counts against it's own similar state information to determine link quality, and then if it judges link quality bad, can recover the link (i.e. by bringing the link down and then up again, or some other application-specific recovery algorithm). The link can have a transient failure, in which case some amount of hysteresis is needed to average link quality over some multiple of the Reporting-Period so that transient link failures do not cause the link to go up and down too frequently.

PPP LQM does not specify the specific policy to be used to judge link quality, nor does it specify how to recover the link when link quality is judged bad. Since we believe that any user planning to use PPP LQM is sophisticated enough to implement their own link quality determination policy and recovery algorithm, also since the specific implementation will likely vary significantly depending on the application, we have decided not to implement any default link quality determination policy or recovery algorithm, but instead we provide a flexible API which enables the user to implement their own.

Description

First, before doing anything else, if link quality monitoring is desired, the user must #define TM_PPP_LQM in trsystem.h, and then rebuild all of the Treck TCP/IP code. **tfUsePppLqm** is called to allocate the LQM state vector. Before the link is opened, the user calls **tfPppSetOption** to set the TM_LCP_QUALITY_PROTOCOL configuration option, and if the user wants to register a link quality monitoring function, then they also negotiate a non-zero value for the LQR timer. After PPP opens the link, PPP calls **tfLqmEnable** which starts the LQR timer. Either before or after PPP opens the link, but after the call to **tfUsePppLqm,** the user calls **tfLqmRegisterMonitor** to register their link quality monitoring function. Whenever a LQR is received, the user's link quality monitoring function is called, and is it passed incoming and outgoing packet and byte counts (derived from information contained in this LQR and also from the previously received LQR) and the elapsed time since the this function was last called. The user's link quality monitoring function then determines the link quality based on these counts and the delta time, and returns 0 if link quality is good, and 1 if it is bad (or greater than one to short-circuit the hysteresis and cause link recovery to occur sooner). These link quality counts are accumulated over multiple calls to the user's link quality monitoring function, and if they surpass a specified max failure count within a specified number of times that this function was called, then the user is notified that the link is bad via a PPP callback flag, after which they should attempt recovery of the link.

Code Example

```
#include <trsocket.h>

/* Parameters controlling determination of link quality */
#define LQM_HYSTERESIS_MAX_FAILURES   3
#define LQM_HYSTERESIS_SAMPLES        4

/* Function prototype for link quality monitoring function
*/
ttUser8Bit myLinkQualityMonitor(
    ttUserInterface interfaceHandle,
    int reasonCode,
    unsigned long timeElapsedMsec,
    ttLqrCountDeltasPtr countDeltasPtr,
    ttConstLqrCountsPtr countsPtr,
    ttUser32Bit outLqrs,
    ttUser32Bit outPackets,
    ttUser32Bit outOctets );

/* Function prototype for PPP link notification function */
void myPppLinkNotify(ttUserInterface interfaceHandle, int
flag);

ttUserInterface interfaceHandle;
char errorMsgBuf[256];

/* Status flags */
int lqmEnabledStatus;
int linkBadStatus;
int linkQualityMonitorRegisteredStatus;

int main(void)
{
    int errorCode;

/* initialization */
    lqmEnabledStatus = 0;
    linkBadStatus = 0;
    linkQualityMonitorRegisteredStatus = 0;

/* start Treck */
    errorCode = tfStartTreck();
    …

/* initialize PPP */
```

```
    linkLayerHandle = tfUseAsyncPpp(
        (ttUserLnkNotifyFuncPtr) myPppLinkNotify);

/* add the PPP interface */
    interfaceHandle = tfAddInterface(
/* name of the device */
        “MYDEVICE.001”,
/* Link Layer to use */
        linkLayerHandle,
        …

/* initialize LQM with a 3 second retransmission timer */
    errorCode = tfUsePppLqm(
        interfaceHandle,
        (ttUser32Bit) 3000);

/* set PPP options for negotiation */
    errorCode = tfPppSetOption(
        interfaceHandle,
        …

/* open the PPP interface */
    errorCode=tfOpenInterface(
        interfaceHandle,
        …

/* main polling loop */
    while(1)
    {
        if (lqmEnabledStatus == 1)
        {
            if (linkQualityMonitorRegisteredStatus == 0)
            {
                linkQualityMonitorRegisteredStatus = 1;

/* after LQM is enabled, register the link quality monitor
*/
                errorCode = tfLqmRegisterMonitor(
                    interfaceHandle,
                    (ttLqmMonitorFuncPtr)
myLinkQualityMonitor,
                    LQM_HYSTERESIS_MAX_FAILURES,
                    LQM_HYSTERESIS_SAMPLES);
                if (errorCode != TM_ENOERROR)
                {
                    tfSPrintF(
                        errorMsgBuf,
                        “tfLqmRegisterMonitor failed
‘%s’\n”,
                        tfStrError(errorCode));
                    tfKernelWarning(
                        “main”,
                        errorMsgBuf);
                }
            }
        }

        if (linkBadStatus == 1)
        {
            linkBadStatus = 0;

/* perform custom link recovery processing */
            …
        }

/* other stuff */
        …
    }
}

void myPppLinkNotify(ttUserInterface interfaceHandle, int
flag)
{
    switch (flag)
    {
    case TM_LL_LQM_LINK_BAD:
        linkBadStatus = 1;
        break;

    case TM_LL_LQM_UP:
        lqmEnabledStatus = 1;
        break;

    case TM_LL_LQM_DISABLED:
        lqmEnabledStatus = 0;
        break;

/* other stuff */
    …
    }
}

ttUser8Bit myLinkQualityMonitor(
    ttUserInterface interfaceHandle,
    int reasonCode,
    unsigned long timeElapsedMsec,
    ttLqrCountDeltasPtr countDeltasPtr,
    ttConstLqrCountsPtr countsPtr,
    ttUser32Bit outLqrs,
    ttUser32Bit outPackets,
    ttUser32Bit outOctets )
{
    int linkQuality;
    int outboundLqrsInPipeline;

    linkQuality = 0;
    switch(reasonCode)
    {
    case TM_LQM_MONITOR_LQR:
/* calculate link quality of the outgoing link */
        if (countDeltasPtr->deltaLastOutPackets >
            (countDeltasPtr->deltaPeerInPackets
             + countDeltasPtr->deltaPeerInDiscards))
        {
           linkQuality += countDeltasPtr-
>deltaLastOutPackets
                - (countDeltasPtr->deltaPeerInPackets
                   + countDeltasPtr->deltaPeerInDiscards);
        }

/* calculate link quality of the incoming link */
        if (countDeltasPtr->deltaPeerOutPackets
            > countDeltasPtr->deltaSaveInPackets)
        {
           linkQuality += countDeltasPtr-
>deltaPeerOutPackets
                - countDeltasPtr->deltaSaveInPackets;
        }


/* calculate number of outbound LQRs still in the pipeline
*/
        outboundLqrsInPipeline = (int)
            ((ttUser32Bit) 0xffffffff
                & (outLqrs - countsPtr->lastOutLqrs));
        if (outboundLqrsInPipeline > 1)
        {
            linkQuality += outboundLqrsInPipeline - 1;
        }
        break;

    case TM_LQM_MONITOR_TIMEOUT:
        linkQuality = 1;
        break;

    default:
/* this should not happen */
        tfKernelWarning(
            “myLinkQualityMonitor”,
              “unrecognized value for reasonCode”);
         break;
    }

/* require at least 2 sample periods to declare a link bad
*/
    if (linkQuality > LQM_HYSTERESIS_MAX_FAILURES)

{
        linkQuality = LQM_HYSTERESIS_MAX_FAILURES;
    }

    return (ttUser8Bit) linkQuality;
}
```

Limitations

We do not implement a default policy for judging the quality of the link, and we do not attempt any recovery when the link quality is judged bad. Instead, the user must register a link quality monitoring function if they want to implement a link quality monitoring policy, and we will give them a PPP callback (after a user-specified amount of hysteresis has been applied) when the link quality is judged bad so that they can attempt recovery.

Public API
tfUsePppLqm

```
int                  tfUsePppLqm
(
ttUserInterface      interfaceHandle,
ttUser32Bit          lqrReTxPeriodMsec
);
```

Function Description

tfUsePppLqm is used to initialize PPP Link Quality Monitoring (LQM), and must be called before **tfOpenInterface** for each distinct PPP interface that the user wants to monitor link quality on. After calling **tfUsePppLqm** but before calling **tfOpenInterface**, the user should call **tfPppSetOption** to set the TM_LCP_QUALITY_PROTOCOL configuration option; otherwise, LQM won't be used on the link (unless the peer negotiates it).

After this, it is still possible that LQM is not being used on the link because the peer doesn't support it. To determine if LQM is being used on the link, install a PPP notification function (refer to **tfUseAsyncPpp**). When your PPP notification function is called with the flag TM_LL_LQM_UP, this indicates that LQM has been enabled on the link. When your PPP notification function is called with the flag TM_LL_LQM_DISABLED, this indicates that LQM has been disabled on the link.

Parameters

Parameter	Description
interfaceHandle	The PPP interface handle as returned by **tfAddInterface**.
lqrReTxPeriodMsec	Configures how long (in milliseconds) we wait for the LQR response to a LQR we sent (initiated LQR timer) before timing out and retransmitting the LQR. This should be chosen to be at least twice the smooth round trip time on the link. Setting this parameter to 0 disables the use of a retransmission timer (not recommended).

Returns

Value	Meaning
0	Success.
TM_EINVAL	The interface handle is invalid.
TM_EOPNOTSUPP	Failed initializing PPP LQM.

tfFreePppLqm

```
int              tfFreePppLqm
(
ttUserInterface  interfaceHandle
);
```

Function Description

tfFreePppLqm does the reverse of **tfUsePppLqm**, i.e. it disables LQM on the link, deallocates any memory allocated by LQM and removes any associated timers.

Parameters

Parameter	**Description**
interfaceHandle	The PPP interface handle as returned by **tfAddInterface**.

Returns

Value	Meaning
0	Success.
TM_EINVAL	The interface handle is invalid.

tfLqmRegisterMonitor

```
int                    tfLqmRegisterMonitor
(
ttUserInterface        interfaceHandle,
ttLqmMonitorFuncPtr monitorFuncPtr,
ttUser16Bit            hysteresisMaxFailures,
ttUser16Bit            hysteresisSamples
);
```

Function Description

tfLqmRegisterMonitor enables the user to specify a policy for determining link quality, specifically by registering a user-defined link quality monitoring function. **tfLqmRegisterMonitor** should be called after the call to **tfUsePppLqm** for each distinct PPP interface that the user wants to monitor link quality on. It is highly recommended that you use a LQR timer to pace the sending of LQRs, otherwise if the link is very bad incoming, your link quality monitoring function won't be called frequently enough to allow you recover the link in a timely fashion, since without a LQR timer it is only called when a LQR is received.

The user-defined link quality monitoring function (specified by *monitorFuncPtr*) must return 0 if link quality is good, and a weighted non-zero value if link quality is bad (i.e. a value of 2 is twice as bad as a value of 1, etc.). When the accumulated value of bad counts returned by your link quality monitoring function exceeds *hysteresisMaxFailures* within the last *hysteresisSamples* times of calling your function, you will be notified via the PPP callback flag TM_LL_LQM_LINK_BAD (if you registered a link notification function, refer to **tfUseAsyncPpp**) that the link is bad so that you can attempt recovery. The link quality monitoring function will be called when one of the following events happens:

1. A timeout occurs while waiting to receive a solicited Link-Quality-Report message from the peer. The peer negotiated for us to use a non-zero LQM reporting period, which means that we are using the LQR timer to pace our sending of Link-Quality-Report messages. This timer expired, and we haven't yet received a Link-Quality-Report message from the peer for a Link-Quality-Report message we sent earlier (i.e. *reasonCode* set to TM_LQM_MONITOR_TIMEOUT).

2. A Link-Quality-Report message is received from the peer (i.e. *reasonCode* set to TM_LQM_MONITOR_LQR).

The peer may not support LQM, in which case the link quality monitoring function will never be called indicating that LQM is not being used on the link.

The function prototype for the link quality monitoring function (specified by *monitorFuncPtr*) is defined as follows:

```
ttUser8Bit myLinkQualityMonitor(
                              ttUserInterface interfaceHandle,
                              int reasonCode,
 unsigned long timeElapsedMsec,
    ttLqrCountDeltasPtr countDeltasPtr,
    ttConstLqrCountsPtr countsPtr,
    ttUser32Bit outLqrs,
    ttUser32Bit outPackets,
    ttUser32Bit outOctets );
```

reasonCode can take on the values TM_LQM_MONITOR_LQR or TM_LQM_MONITOR_TIMEOUT, depending on whether the reason for the callback is that a Link-Quality-Report message was received (TM_LQM_MONITOR_LQR), or that the timeout occurred before the solicited/expected Link-Quality-Report message was received (TM_LQM_MONITOR_TIMEOUT). *timeElapsedMsec* is the time elapsed (in milliseconds) since the last time the link quality monitoring function was called, *outLqrs* is the count of LQRs sent, *outPackets* is the count of packets sent, and *outOctets* is the count of bytes sent

NOTE: since these are unsigned 32-bit counters, they may wrap around to 0

When *reasonCode* is set to TM_LQM_MONITOR_LQR:

1. *countDeltasPtr* points to the absolute counts of packets and bytes sent and received (as reported by the peer in the received Link-Quality-Report message) since the last time the link quality monitoring function was called.
2. *countsPtr* points to the relative counts of packets and bytes sent and received (as reported by the peer in the received Link-Quality-Report message). You must not change any of these counts, since they are used internally.
3. *countsPtr*->lastOutLQRs may be compared with *countsPtr*->peerInLQRs to determine how many outbound LQRs have been lost.
4. *countsPtr*->lastOutLQRs may be compared with *outLQRs* to determine how many outbound LQRs are still in the pipeline.
5. *countDeltasPtr*->deltaPeerInPackets may be compared with *countDeltasPtr*->deltaLastOutPackets to determine the number of lost packets over the outgoing link.
6. *countDeltasPtr*->deltaPeerInOctets may be compared with

countDeltasPtr->deltaLastOutOctets to determine the number of lost octets over the outgoing link.

7. *countDeltasPtr*->deltaSaveInPackets may be compared with *countDeltasPtr*->deltaPeerOutPackets to determine the number of lost packets over the incoming link.
8. *countDeltasPtr*->deltaSaveInOctets may be compared with *countDeltasPtr*->deltaPeerOutOctets to determine the number of lost octets over the incoming link.
9. *countDeltasPtr*->deltaPeerInDiscards and *countDeltasPtr*->deltaPeerInErrors may be used to determine whether packet loss is due to congestion in the peer rather than physical link failure.

When *reasonCode* is set to TM_LQM_MONITOR_TIMEOUT, *countDeltasPtr* and *countsPtr* are NULL.

If link quality is good, your link quality monitoring function should return 0, otherwise it should return 1, unless you want to reduce the hysteresis and speed up the process of link failure in which case it should return a value greater than 1.

NOTE: *Your link quality monitoring function is called with the device locked. You cannot call any LQM public API functions from the monitoring function (doing so will result in deadlock, if you have #define'd the macro TM_LOCK_NEEDED to enable locking). For example, if your monitoring function determines that the link is good outgoing but very bad incoming, and you want to send LQRs at a faster rate in this case (per RFC-1989), you cannot call tfLqmSendLinkQualityReport() or tfLqmSetLqrTimerPeriod() directly from your monitoring function since doing so will result in deadlock. Instead, have your monitoring function set a flag, which you can then poll in another task (or in your main polling loop) that can then call the appropriate LQM public API functions.*

Parameters

Parameter	**Description**
interfaceHandle	The PPP interface to use this link quality monitoring routine with.
monitorFuncPtr	The function to call to monitor link quality.
hysteresisMaxFailures	Set to 0 if we aren't using any hysteresis, otherwise this is the maximum number of bad link quality counts we are allowed to get back from calls to the user's

link quality monitoring function within the specified sampling period (i.e. *hysteresisSamples)* before we will notify the user that the link is bad.

hysteresisSamples Set to 0 if we aren’t using any hysteresis, otherwise this is the sampling period (specified as the number of calls to the user’s link quality monitoring function) used to determine if the link is bad.

Returns

Value	**Meaning**
0	Success.
TM_EINVAL	The interface handle is invalid, or *monitorFuncPtr* was NULL.
TM_EOPNOTSUPP	You must call tfUsePppLqm first to initialize PPP LQM.

tfLqmSendLinkQualityReport

```
int                 tfLqmSendLinkQualityReport
(
ttUserInterface     interfaceHandle
);
```

Function Description

The PPP LQM specification (RFC-1989) allows the Link-Quality-Report message to be sent more frequently than the negotiated reporting period, and **tfLqmSendLinkQualityReport** gives you direct control over when Link-Quality-Report messages are sent. You may want to use **tfLqmSendLinkQualityReport** if your link quality monitoring function is not being called frequently enough, and you want to get feedback on what the current link quality is.

Parameters

Parameter	**Description**
interfaceHandle	The PPP interface to send the Link-Quality-Report message on.

Returns

Value	**Meaning**
0	Success.
TM_EINVAL	The interface handle is invalid.
TM_ENETDOWN	The PPP interface is not open/ connected.
TM_EOPNOTSUPP	Could not send Link-Quality-Report message because LQM is disabled on the link (i.e. the peer does not support LQM).
TM_ENOBUFS	Could not allocate a buffer for the Link-Quality-Report message.

tfPppSendEchoRequest

```
int                    tfPppSendEchoRequest
(
ttUserInterface        interfaceHandle,
ttUser8Bit             echoRequestId,
const char *           dataPtr,
int                    dataLen,
ttEchoReplyFuncPtr     echoReplyFuncPtr
);
```

Function Description

ttPppSendEchoRequest sends a LCP Echo-Request message, and then later will call the user-defined function (specified by *echoReplyFuncPtr*) to process received LCP Echo-Reply messages. For each Echo-Request message sent, there should be one Echo-Reply message received containing the same data that was in the Echo-Request (possibly truncated, depending on what was negotiated for TM_LCP_MAX_RECV_UNIT). If you do not want to process received LCP Echo-Reply messages, set *echoReplyFuncPtr* to TM_PPP_ECHO_REPLY_FUNC_NULL_PTR.

It is important to note that the user is responsible for matching Echo-Request messages with their associated Echo-Reply messages. No attempt is made in this function to keep track of Echo-Request messages that have been sent for later matching them up with received Echo-Reply messages. However, the *echoRequestId* parameter is provided to assist the user in performing this matching. The correct usage of this parameter is described in RFC-1661 as follows:

> *"On transmission, the Identifier field MUST be changed whenever the content of the Data field changes, and whenever a valid reply has been received for a previous request. For retransmissions, the Identifier MAY remain unchanged. On reception, the Identifier field of the Echo-Request is copied into the Identifier field of the Echo-Reply packet."*

The user is responsible for incrementing *echoRequestId* as appropriate to ensure that the above RFC requirements on the use of the Identifier field are met.

Since poor link quality can result in no Echo-Reply message being received after an Echo-Request message was sent, **ttPppSendEchoRequest** can be used to implement some level of link quality monitoring. This is especially useful when PPP LQM is not supported by the peer on the link.

The function prototype for the function called to process received LCP Echo-Reply messages (specified by *echoReplyFuncPtr*) is defined as follows:

```
int myHandleEchoReply
(
ttUserInterface interfaceHandle,
ttUser8Bit echoRequestId,
const char* dataPtr,
int dataLen );
```

Parameters

Parameter	**Description**
interfaceHandle	The PPP interface to send the Echo-Request message on.
echoRequestId	A unique ID for the Echo-Request message. This can be used to match Echo-Request messages with their associated Echo-Reply messages.
dataPtr	Pointer to the data to send in the Echo-Request message.
	dataLen Length of the data (in bytes) to send in the Echo-Request message.
echoReplyFuncPtr	Function pointer pointing to a user-defined routine that handles a received LCP Echo-Reply message.

Returns

Value	**Meaning**
0	Success.
TM_EINVAL	The interface handle is invalid.
TM_ENETDOWN	The PPP interface is not open/ connected.
TM_EMSGSIZE	The length of the Echo-Request message exceeds the MRU used on the link.
TM_ENOBUFS	Could not allocate a buffer for the Echo-Request message.

tfLqmSetLqrTimerPeriod

```
int                 tfLqmSetLqrTimerPeriod
(
ttUserInterface     interfaceHandle,
ttUser32Bit         lqrTimerPeriodMsec
);
```

Function Description

tfLqmSetLqrTimerPeriod can be called to increase the frequency of sending LQRs (i.e. decrease the LQR timer period) when using a LQR timer. This function should be called by the user to send LQRs faster when the link is good outgoing, but very bad incoming, since in that case incoming LQRs will be frequently lost. The LQR timer period cannot be increased to be longer than the negotiated period.

Parameters

Parameter	**Description**
interfaceHandle	The PPP interface handle as returned by **tfAddInterface**.
lqrTimerPeriodMsec	Configures how long (in milliseconds) we wait before sending a LQR. This period cannot be longer than the negotiated period for the LQR timer, and cannot be 0.

Returns

Value	**Meaning**
0	Success.

TM_EINVAL	The interface handle is invalid, or *lqrTimerPeriod* is invalid (i.e. was 0, or was longer than the negotiated period).
TM_EOPNOTSUPP	No LQR timer is being used locally, or LQM is disabled on the link.

tfLqmGetLocalLqrTimerPeriod

```
ttUser32Bit       tfLqmGetLocalLqrTimerPeriod
(
ttUserInterface   interfaceHandle
);
```

Function Description

tfLqmGetLocalLqrTimerPeriod returns the negotiated period of our local LQR timer (in milliseconds), or 0 if we aren't using a LQR timer.

Parameters

Parameter	**Description**
interfaceHandle	The PPP interface handle as returned by **tfAddInterface**.

Returns

Value	**Meaning**
0	Either LQM is disabled on the link, or we do not use a LQR timer.
!= 0	The negotiated period (in millisecond) of our local LQR timer. This will not be the same as the current LQR timer period if you have called **tfLqmSetLqrTimerPeriod**.to change the LQR timer period.

tfLqmGetPeerLqrTimerPeriod

```
ttUser32Bit       tfLqmGetPeerLqrTimerPeriod
(
ttUserInterface   interfaceHandle
);
```

Function Description

tfLqmGetPeerLqrTimerPeriod returns the negotiated period of the peer’s LQR timer (in milliseconds), or 0 if the peer isn’t using a LQR timer.

Parameters

Parameter	**Description**
interfaceHandle	The PPP interface handle as returned by **tfAddInterface**.

Returns

Value	**Meaning**
0	Either LQM is disabled on the link, or the peer does not use a LQR timer.
!= 0	The negotiated period (in milliseconds) of the peer’s LQR timer.

Appendix A
Configuration Notes

Configuring IP Forwarding and IP Fragmentation

IP Forwarding

By default, IP Forwarding is not enabled in the Turbo Treck stack. To enable IP Forwarding, the user must turn on the IP forwarding option after the **tfStartTreck** call:

```
errorCode= tfSetTreckOptions (TM_OPTION_IP_FORWARDING, 1UL);
```

In addition, each device that the user wants to forward packets from and to, should be configured with the TM_DEV_IP_FORW_ENB bit set in the **tfOpenInterface** flag parameter. See the **tfOpenInterface** description for a list of the **tfOpenInterface** parameters.

Removing IP Fragmentation and IP Reassembly Code

To save code space, the user can optionally not compile the IP fragmentation and IP reassembly code in the Turbo Treck stack.

Warning: in this case, no IP fragmentation or IP reassembly will ever take place in the Turbo Treck stack.

To do that, ***delete*** the following macro from *trsystem.h*:

```
#define TM_IP_FRAGMENT
```

IP Fragmentation

By default, if the TM_IP_FRAGMENT macro is defined in *trsystem.h*, then IP Fragmentation is turned on in the Turbo Treck stack. To disable IP Fragmentation, the user must turn off the IP Fragmentation option after the **tfStartTreck** call as follows:

```
errorCode= tfSetTreckOptions(TM_OPTION_IP_FRAGMENT, 0UL);
```

Counting Semaphores in the Turbo Treck Stack

Description

The Turbo Treck stack uses an operating system's existing counting semaphores. If your operating system does not support counting semaphores, Turbo Treck can use your OS's event flag mechanism to create its own.
Using an OS's existing counting semaphores is typically a more efficient method. We have two separate counting semaphore versions. The first method will grant counting semaphore requests by task priority, and is implemented in the *kernel\trcousem.c* module. The second method (lighter version) will grant counting semaphore requests in FIFO (*First In, First Out*) order, and is implemented in the *kernel\trctsem2.c* module.

Counting Semaphore Implementation with task priority order:

The user must call the following functions.
This may require some data type modifications:

1. The user must call **tfTaskInit** prior to calling **tfStartTreck**, and prior to launching any task that utilizes the Turbo Treck task.
2. Before launching a task that utilizes the Turbo Treck stack, the user must call **tfTaskRegister** for that task, passing the *taskIdPtr* filled out by **tfKernelGetCurrentTaskId** and set the priority of the task. The task index of the particular task will be returned. An index bigger than or equal to TM_NUMTSK indicates an overflow or a failure.
3. The data type for the task ID may also need modification. It is assumed that the task ID can be stored as an integer. If this is not the case: Replace `taskIdPtr->genInParm` and `tskIdUnion.genIntParm` (located in **tfKernelGetCurrentTaskId** and **tfTaskIdToIndex**), with the appropriate data type (as defined in *ttUserGenericUnion*).

The user must modify the following macros:

TM_NUMTSK
This is the number of tasks making calls to the Turbo Treck stack.
Default 12

tm_task_lower_same
This macro is defined as ((taskAPri) <= (taskBPri)). If priorities are arranged so that lower values have higher priority, modify the macro to ((taskAPri) >= (taskBPri).

The user must define the following functions:

tfKernelGetCurrentTaskId

```
int                       tfKernelGetCurrentTaskId
(
ttUserGenericUnionPtr     taskIdPtr
)
```

Function Description

The user must write this function to make use of their specific operating system. This function stores the task ID of the currently running task in taskIdPtr.

Parameters

Parameter	Description
taskIdPtr	Pointer to a ttUserGenericUnionPtr into which the task ID is stored.

Returns

TM_KERN_OK

tfKernelTaskPendEvent

```
void                    tfKernelTaskPendEvent
(
ttUserGenericUnionPtr  eventPtr
)
```

Function Description

The user must write this function to make use of their specific operating system. This function pends on an event, pointed to by eventPtr, which is posted from another task. This should be similar if not identical to **tfKernelPendEvent**.

Parameters

Parameter	**Description**
eventPtr	Pointer to the event flag upon which this function must pend.

Returns

Nothing

tfKernelTaskPostEvent

```
void                    tfKernelTaskPostEvent
(
ttUserGenericUnionPtr  eventPtr
)
```

Function Description

The user must write this function to make use of their specific operating system. This function resumes tasks that were waiting on the event pointed to by *eventPtr*. This should be similar to **tfKernelIsrPostEvent**() with one major difference: **tfKernelIsrPostEvent**() is called from within an ISR, whereas **tfKernelTaskPostEvent** is called from a task. Most operating systems have different calls when posting occurs from an ISR or from a task.

Parameters

Parameter	**Description**
eventPtr	Pointer to the event flag to which this function must post.

Returns

Nothing

Counting Semaphore Implementation with FIFO order:

The user must call the following function:

The user must call **tfTaskInit**, prior to calling **tfStartTreck**, and prior to launching any task that utilizes the Turbo Treck task.

The user must define the following macro:

TM_NUMEVT

This is the number of events needed by the Turbo Treck stack. This number should match the number of tasks making calls to the Turbo Treck stack. **Default 12**

The user must define the following functions:

tfKernelTaskPendEvent

```
void                      tfKernelTaskPendEvent
(
ttUserGenericUnionPtr  eventPtr
)
```

Function Description

The user must write this function to make use of their specific operating system. This function pends on an event pointed to by eventPtr, which is posted from another task. This should be similar if not identical to **tfKernelPendEvent**.

Parameters

Parameter	**Description**
eventPtr	Pointer to the event flag upon which this function must pend.

Returns

Nothing

tfKernelTaskPostEvent

```
void                       tfKernelTaskPostEvent
(
ttUserGenericUnionPtr  eventPtr
)
```

Function Description

The user must write this function to make use of their specific operating system. This function resumes tasks that were waiting on the event pointed to by *eventPtr*. This should be similar to **tfKernelIsrPostEvent** with one major difference: **tfKernelIsrPostEvent**() is called from within an ISR, whereas **tfKernelTaskPostEvent** is called from a task. Most operating systems have different calls when posting occurs from an ISR or from a task.

Parameters

Parameter	**Description**
eventPtr	Pointer to the event flag to which this function must post.

Returns

Nothing

Running multiple instances of Turbo Treck

A set of new APIs described in this section have been added to allow users to run multiple instances of the Turbo Treck TCP/IP stack. Each instance of the Turbo Treck stack needs to be run in a given context. If the user wants to run multiple instances of the Tubo Treck TCP/IP stack, any Turbo Treck TCP/IP stack function should be called after a context has been set, except for the context insensitive functions described below.

Context insensitive functions

These Turbo Treck TCP/IP stack functions can be called from any context, or prior to setting any context

Called from *any* or *no* context	**Comments**
tfInitTreckMultipleContext()	This function initializes the Turbo Treck global variables, prior to any context creation.
tfTimerUpdate()	Called from a Timer task.
tfTimerUpdateIsr()	Called from Timer ISR.

Initialization Sequence

First initialization (called before any other Turbo Treck TCP/IP Stack call)

```
tfInitTreckMultipleContext(void);
```

For each context:

```
contextHandle = tfCreateTreckContext();
if (contextHandle != (ttUserContext)0)
{
  tfSetCurrentContext(contextHandle);
  errorCode = tfStartTreck();
}
```

Summary of new context API’s

Context API	Comments
int tfInitTreckMultipleContext(void);	Initializes multiple context global variables (valid across all contexts)
ttUserContext tfCreateTreckContext(void);	Create a Treck context, i.e. allocate a structure containing all Treck variables for an instance of the Treck stack, and returns a pointer to the newly allocated structure.
void tfSetCurrentContext (ttUserContext contextHandle);	Set the current context handle to the one passed in (i.e. set the Treck global variable tvCurrentContext to the passed parameter).
ttUserContext tfGetCurrentContext(void)	Get the current context handle (i.e. the context handle stored in tvCurrentContext).

Enabling the Multiple Instances code in Turbo Treck

To enable the multiple instances code in the Turbo Treck stack, uncomment the following macro in your trsystem.h:
#define TM_MULTIPLE_CONTEXT

Blocking mode/non blocking mode

The following indicates the modifications needed for each type of embedded kernel:

No Kernel

All applications have to run in non-blocking mode. All calls to the Treck stack are made from a main loop. Example with 2 contexts, and 2 interfaces per context:

```
errorCode = tfInitTreckMultipleContext();
contextHandle1 = tfCreateTreckContext();
contextHandle2 = tfCreateTreckContext();

tfSetCurrentContext(contextHandle1);
errorCode = tfStartTreck();
context1InterfaceHandle1= tfAddInterafce(…);
errorCode = tfOpenInterface(context1InterfaceHandle1, ..);
context1InterfaceHandle2 = tfAddInterface(…);
errorCode = tfOpenInterface(context1InterfaceHandle2, ..);

tfSetCurrentContext(contextHandle2);
errorCode = tfStartTreck();
context2InterfaceHandle1= tfAddInterafce(…);
errorCode = tfOpenInterface(context1InterfaceHandle1, ..);
context2InterfaceHandle2 = tfAddInterface(…);
errorCode = tfOpenInterface(context2InterfaceHandle2, ..);
for (;;)
{
     tfSetCurrentContext(contextHandle1);
     tfTimerExecute()
     if (tfCheckReceiveInterface(context1InterfaceHandle1) ==
TM_ENOERROR)
     {
         tfRecvInterface(context1InterfaceHandle1);
     }
     if (tfCheckReceiveInterface(context1InterfaceHandle2) ==
TM_ENOERROR)
     {
         tfRecvInterface(context1InterfaceHandle2);
     }
     <…Non-blocking application code for context 1 ..>

     tfSetCurrentContext(contextHandle2);
     tfTimerExecute()
     if (tfCheckReceiveInterface(context2InterfaceHandle1) ==
TM_ENOERROR)
     {
         tfRecvInterface(context2InterfaceHandle1);
     }
     if (tfCheckReceiveInterface(context2InterfaceHandle2) ==
TM_ENOERROR)
     {
         tfRecvInterface(context2InterfaceHandle2);
     }
     <…Non-blocking application code for context 2 ..>
}
```

Note 1: that this sample code does not check for error for ease of reading.

Note 2: For more than 2 contexts, it would make sense to save the context variables, and interface handles in a 2 dimensional array, and loop on the array indices.

Non preemptive kernel

In this case, the applications can run in blocking mode.

1. Each application task will have to set its Treck context prior to making the first Treck call.

2. Inside each tfKernel..() call the user will have to save the current Treck context, before calling the OS, and then restore the Treck context upon return from the OS call, since the current task could be pre-empted by a higher priority task during the OS call.

```
int tfKernel…(…)
{
      ttUserContext contextHandle;
      contextHandle = tfGetCurrentContext();
      <Make the OS call>
      tfSetCurrentContext(contextHandle);
}
```

Preemptive Kernel

In this case, the OS could switch out a task at any time.

1. If the user can modify the OS, and can save the current Treck context on a task stack, prior to a task being switched out, and restore the task Treck context when the task runs, then the applications can also run in blocking mode as described in the previous section.

    a. Each application task will have to set its Treck context prior to making the first Treck call.
    b. The user will need to modify the OS, and save the current Treck context on a task stack, prior to a task being switched out, and restore the task Treck context when the task is scheduled to run.

If the user cannot modify the OS, then all the applications will have to run in non-blocking mode from a single Treck task. All the calls to the Treck stack will be made from a main loop from within that single task as described in the No OS section above.

Device Driver Modifications

Each device driver should be modified as described in the "**Further Device Driver Modifications to allow a device driver to be shared by several Ethernet Interfaces**" section of chapter 4 of this manual. The modification to the Device driver ISR is a little bit different, and is described below.

Device driver ISR

The user should keep a global mapping between an interrupt vector, and a pair interface handle, context, instead of just a global mapping between an interrupt vector, and an interface handle. In the device driver ISR, the user should save the current context, then the user should set the Turbo Treck context as found in the global mapping described here, and call **tfDeviceGetPointer** in order to access the device driver specific data. Before returning from the ISR, the user should set the context to the saved value at the beginning of the ISR function.

tfInitTreckMultipleContext

#include <trsocket.h>

```
void                tfInitTreckMultipleContext
(                   void
);
```

Function Description

This function is used to initialize multiple context global variables (valid across all contexts). This function should be called prior to any other Tubo Treck TCP/IP stack functions when using multiple instances of the stack.

Parameters

None

Returns

None

tfCreateTreckContext

```
#include <trsocket.h>

ttUserContext     tfCreateTreckContext
(                 void
);
```

Function Description

This function creates a Turbo Treck context for an instance of the Turbo Treck TCP/IP stack. It will allocate a structure containing all Turbo Treck variables for a context, and returns a pointer to the newly allocated structure. **tfCreateTreckContext** should be called once for each instance of the Turbo Treck TCP/IP stack. It should be called after **tfInitTreckMultipleContext**, and prior to any other Turbo Treck functions, when using multiple instances of the Turbo Treck stack.

Parameters

None

Returns

Value	**Meaning**
Non-null pointer	Pointer to newly created context.
(ttUserContext)0	An error occurred

tfSetCurrentContext

```
#include <trsocket.h>

void                tfSetCurrentContext
(
ttUserContext       contextHandle
);
```

Function Description

This function is used to set the current Turbo Treck stack context. It is usually called after a context switch, or after a kernel call.

Parameters

Parameter	**Description**
contextHandle	Turbo Treck stack context to be set.

Returns

None

tfGetCurrentContext

```
#include <trsocket.h>

ttUserContext     tfGetCurrentContext
(
                  void
);
```

Function Description

tfGetCurrentContext returns the current Turbo Treck stack context, as known by the Turbo Treck stack. It is usually called before a context switch, or kernel call.

Parameters

None

Returns

Value	**Meaning**
Non zero pointer	Current context handle
Null pointer.	Invalid context handle.

Appendix B
RTOS Notes

RTOS Application Note for uC/OS

uC/OS is a simple preemptive kernel. It does not have any way to perform dynamic memory allocation. It does contain counting semaphores, message queues and mail boxes as well a preemptive scheduling. With your CD-ROM you will have received various ports of uC/OS for different processors. If your processor is not included in the ports of uC/OS that Elmic Systems provides, then you might want to visit the uC/OS web site at http://www.ucos-ii.com/

Predefined Settings in TRSYSTEM.H

There are processor specific settings for uC/OS in trsystem.h. They will setup uC/OS the same way that it was used in the Elmic test facilities for various processors.

TM_KERNEL_UCOS_X86 is used for the Intel 80x86 family in real mode
TM_KERNEL_UCOS_PPC is used for the PowerPC (specifically the MPC860 for Motorola)
TM_KERNEL_UCOS_CPU32 is used for the 32 Bit 68K family of processors from Motorola.

If you define one of the above in trsystem.h, the processor, inline assembly, task, and memory allocation definitions will be made for you. You can modify the uC/OS processor specific area of trsystem.h to suite your applications needs.

Initialization

uC/OS does not require any special initialization therefore the *tfKernelIniticalize* should be a stub function
Memory Allocation and Free
Because uC/OS does not contain any method for performing dynamic memory allocation, the Turbo Treck Simple Heap must be used.

In *trsystem.h* be sure to define the following

#define TM_USE_SHEAP
#define TM_SHEAP_SIZE *size*

where *size* is the size in bytes of the simple heap area. Most simple applications will need about 32K bytes of RAM. To defining 32K you simply #define TM_SHEAP_SIZE 32L*1024L. Note that *size* is defined as a long constant.

Because the simple heap is used, there is no need to create the functions *tfKernelMalloc* and *tfKernelFree*

Critical Section Handling

uC/OS does have critical section handling via two macros:
OS_ENTER_CRITICAL and OS_EXIT_CRITICAL

For *tfKernelSetCritical* you simply call OS_ENTER_CRITICAL and for *tfKernelReleaseCritical* you simply call OS_EXIT_CRITICAL.

It is always best to use inline assembly for critical section handling by setting the macros *tm_kernel_set_critical* and *tm_kernel_release_critical* for your processor in trsystem.h. You will find preexisting critical section macros for specific processors and compiler combinations in trmacro.h.

Error Logging and Warning Information Logging

uC/OS does not have any error reporting mechanism. Since this is the case, *tfKernelError* and *tfKernelWarning* should be written to write out to a display, serial port or a predefined area of memory that can be examined via a debugger. In the case of *tfKernelError*, it should cause a CPU reset after displaying the error message.

Task Suspend and Resume

uC/OS does implement counting semaphores, so all we need to do here is create a mapping between Turbo Treck semaphores and uC/OS semaphores. uC/OS semaphores are pointers to an OS_EVENT type and should be stored in the *genVoidParmPtr* portion of the *ttUserGenericUnion* data type.

To create a counting semaphore, we simply call *OSSemCreate*(0). This creates a counting semaphore that is initialized to zero. All counting semaphores that Turbo Treck uses must be initialized to zero.

To pend on a counting semaphore, we simply call *OSSemPend* with a semaphore that was created.

To post on a counting semaphore, we simply call *OSSemPost* with a semaphore that was created.

tfKernelTaskYield does not need to do anything with uC/OS because uC/OS is fully preemptive and every task has a unique priority. This function should just be a stub function

ISR Interface

uC/OS does not have anyway to install Interrupt Service Routines. We have put a routine in the uC/OS ports that we supply called *OSInstallISR*. This function is NOT supplied with uC/OS. This function is very processor specific. If you are not using a Turbo Treck port of uC/OS, but are using a Turbo Treck device driver, then you should add this function for your processor.

Note that all ISR routines that call Turbo Treck or uC/OS must call *OSIntEnter* upon entry and *OSIntExit* upon leaving.

Since uC/OS does not have event flags and it is okay to post on a counting semaphore from an ISR, the functions *tfKernelCreateEvent*, *tfKernelPendEvent*, and *tfKernelPostEvent* use uC/OS counting semaphores. You can use the same code as you used in *tfKernelCreateCountSem*, *tfKernelPendCountSem*, and *tfKernelPostCountSem*.

Hooking up the Timer

The best way to hook a timer to Turbo Treck from uC/OS is to create a separate timer task. This task simply calls *tfTimerUpdate*, *tfTimerExecute* then *OSTimeDelay*(1). The *OSTimeDelay*(1) causes us to delay for one RTOS clock tick. This task should be the highest priority of any task calling the Turbo Treck stack.

Task Usage

All uC/OS tasks are different priorities. With uC/OS you should have one timer task, one receive task per interface and your application tasks. For best performance, you should create one receive task per device that calls *tfWaitReceiveInterface* and *tfRecvInterface*. The receive tasks should have a higher priority than the application tasks but lower than the timer task.

RTOS Application Note for AMX-86

AMX-86 is a Commercial Off The Shelf (COTS) real-time operating system from Kadak Products Ltd. It contains counting semaphores and memory allocation. AMX is broken up into two variants. These are commonly referred to as the "AJ Kernel" and "CJ Kernel". AMX-86 is the AJ kernel (thus all the calls to the operating system are prefixed with "aj".

Initialization

AMX-86 does not require any special initialization therefore the tfKernelIniticalize should be a stub function

Memory Allocation and Free

AMX-86 includes dynamic memory allocation via the calls ajmget and ajmfre. tfKernelMalloc should call ajmget to allocate a block of memory and tfKernelFree should call ajmfre to free a block of memory.

Critical Section Handling

AMX-86 is used with the Intel x86 processor family. For both Microsoft and Borland compilers, we have included inline assembly in trmacro.h to set and release critical sections. If you are using another compiler, you can define the macros tm_kernel_set_critical and tm_kernel_release_critical in trsystem.h to be the appropriate inline assembly for your compiler for setting and releasing critical sections.

Error Logging and Warning Information Logging

AMX-86 does not have any error reporting mechanism. Since this is the case, tfKernelError and tfKernelWarning should be written to write out to a display, serial port or a predefined area of memory that can be examined via a debugger. In the case of tfKernelError, it should cause a CPU reset after displaying the error message.

Task Suspend and Resume

AMX-86 does implement counting semaphores, so all we need to do here is create a mapping between Turbo Treck semaphores and AMX-86 semaphores. AMX-86 semaphores are pointers to an AMX_ID type and should be stored in the genVoidParmPtr portion of the ttUserGenericUnion data type.

To create a counting semaphore, we simply call ajsmcre. This creates a counting semaphore. Be sure to initialize to zero. All counting semaphores that Turbo Treck uses must be initialized to zero.

To pend on a counting semaphore, we simply call ajsmwat with a semaphore that was created.

To post on a counting semaphore, we simply call ajsmsig with a semaphore that was created.

tfKernelTaskYield does not need to do anything with AMX-86 because AMX-86 is fully preemptive and every task has a unique priority. This function should just be a stub function.

ISR Interface

AMX-86 has a method to install ISRs. in AMX-86 these are called Interrupt Service Procedures (ISP). Since we normally use “C” code for the ISR code, we need to setup our tfKernelInstallIsrHandler to call ajispm to create a wrapper ISP that will call our “C” function. Then we call ajivtw to write the address of the wrapper that calls our “C” code to the interrupt vector table.

With AMX-86 it is okay to post on a counting semaphore from an ISR, the functions tfKernelCreateEvent, tfKernelPendEvent, and tfKernelPostEvent use uC/OS counting semaphores. You can use the same code as you used in tfKernelCreateCountSem, tfKernelPendCountSem, and tfKernelPostCountSem.

Hooking up the Timer

The best way to hook a timer to Turbo Treck from AMX-86 is to create a separate timer task. This task simply calls tfTimerUpdate, tfTimerExecute then ajwatm. The ajwatm takes as it parameter the number of milliseconds to wait. This task should be the highest priority of any task calling the Turbo Treck stack.

Task Usage

All AMX-86 tasks are different priorities. With AMX-86 you should have one timer task, one receive task per interface and your application tasks. For best performance, you should create one receive task per device that calls tfWaitReceiveInterface and tfRecvInterface. The receive tasks should have a higher priority than the application tasks but lower than the timer task.

AMX-86 System Configuration Module

With the system configuration tool for AMX-86 you will need to define the maximum number of semaphores and tasks. The maximum number of tasks is calculated from the following formula

MaxTasks=ReceiveTasks+SendCompleteTasks+XmitTasks+TimerTask+ApplicationTasks

The absolute maximum number of semaphores needed by the is calculated via the following formula:
MaxSemaphores=(NumberInterfaces*NumberTasksPerInterface)+MaxTasks.

NumberTasksPerInterface is the number of tasks that are dedicated to interface processing. The maximum this can be is 3 (RecvTask+SendCompleteTask+XmitTask). This is controlled via #defines in trsystem.h.

Maximum Stack Size should not exceed 2048. Selecting too small of a stack can lead to unpredictable behavior.

Turbo Treck does not require any AMX timers since we are using a separate Timer task.

Appendix C
Debugging

Debugging a Device Driver

How to debug a device driver

1. **Use a network analyzer.**
   Network analyzers can be obtained from KLOS (www.klos.com), and Shomiti (www.shomiti.com). Network analyzers are invaluable when trying to debug a protocol stack. They allow you to see all of the traffic on the network, and thus inspect what was going on in the stack when problems were encountered. For technical support to be effective in helping you with any problems you may encounter, some form of packet capturing device is of the utmost importance.

2. **Disable all Compiler Optimizations**
   While debugging your device driver, you should disable all compiler optimizations. Certain compiler optimizations may cause problems with your network device driver and should not be enabled until the driver is completely debugged.

3. **Use InetD95 for windows 95/98 included on distribution CD.**
   This tool emulates some of the standard services available on Unix machines. Most interesting to us are the echo port and the discard port (ports 7 and 9, respectively). They allow you to send data from another source, which they will either echo back to you, or discard silently. One of the easiest initial tests to do with the stack is to have it send data to another host, which will then silently discard it. If a Unix machine is unavailable to you, this program makes it possible to run a similar test using a Windows platform.

4. **Send pre-initialized data.**
   Frequently, errors with the driver can cause memory to be corrupted. By sending pre-initialized data at the application level, it is easy to verify that the correct data are being sent.

5. **Use Turbo Treck test suite to test the locks and the driver.**
   The Turbo Treck stack comes with a test suite that allows you to verify if the locking mechanism is working correctly. Once that has been verified, the suite is also capable of testing various other aspects of TCP and UDP communications. Please see the Turbo Treck Test Suite section in this manual.

6. **Make sure the user interface handle is valid when calling tfRecvInterface(). It has to be the interface handle as returned by tfAddInterface().**

This is frequently interpreted as an error at the driver level, but is actually a bug in application code. If the interface handle passed to tfRecvInterface is not valid, it could cause the call to fail, memory corruption, or an application crash.

7. **Temporarily disable TM_DEV_SCATTER_SEND_ENB in your call to tfOpenInterface().**
   The TM_DEV_SCATTER_SEND_ENB flag informs the stack that scatter-send is available in the driver. This means that the stack can deliver a packet to the driver in parts – that is, a packet may be comprised of more than one buffer. This is desirable, especially for TCP, because it removes the necessity of a copy and speeds up performance. However, supporting scatter-send in the driver can be a little more complicated. If the flag is being passed in to tfOpenInterface() and you are seeing odd behavior, try disabling it. If you had a memory corruption problem before that now goes away, it is a sign that tfSendComplete() was being called too frequently. It should be called once per packet, as opposed to once per buffer.

8. **Look for memory leaks. (Break point on memory allocation function after several minutes).**
   The Turbo Treck stack uses an optimized memory allocation scheme in which it maintains its own memory pool. After the system becomes balanced, which is usually within 5 minutes of startup, the stack should not be requesting any more memory from the system. If this is occurring, it is a possible sign that memory is being lost within the driver. Putting breakpoints on the system memory allocation functions (tfKernelMalloc if you are using your operating system's memory allocation procedures, tfSheapMalloc if you are using Turbo Treck's simple heap) after the stack has been running for about five minutes will allow you to identify any suspicious memory allocations.

9. **Look at send and receive buffer rings.**
   Most Ethernet drivers will require the user to set up send and/or receive buffer rings. Even if not required, frequently the use of these buffer rings will speed up performance, and is thus very desirable. However, send and receive rings are generally the most complex part of a driver's logic. Check your ring logic very carefully – by hand, if necessary. We also provide an API for use at the driver level that should alleviate this problem. Please see the 'Programmers Reference Section of this manual.

10. **Use counters to compare the number of times the device driver send function is called with the TM_USER_BUFFER_LAST flag set with the number of tfSendCompleteInterface() is called.**
    tfSendCompleteInterface() should be called once, and only once, for each time that the driver's send function is called with the TM_USER_BUFFER_LAST flag set. Calling tfSendCompleteInterface() either too frequently or too infrequently can lead to memory loss and/or corruption.

11. **Use counters to compare the number of receive ISR events with the number of calls made to driver receive function.**
    For each packet the chip receives, the stack should be notified. Similarly, for each packet the stack is notified of, it should call the driver receive function once. This test will verify that this is happening.

12. **No printf or anything that takes too much CPU time inside the ISR.**
    Putting any extra code inside an ISR can be dangerous, but printf statements and their relatives can be doubly so. On various platforms, printf involves a call to the operating system that may result in interrupts being re-enabled before the ISR has completed. Also, extra code inside the ISR can change the system in other ways. For example, printf calls are frequently mapped to a serial port. Serial ports are generally slow enough that making use of one inside an ISR can change a system's timing significantly, thus hiding potential timing bugs, or even cause interrupts to be missed.

13. **Make sure that any data that are either modified or checked during an ISR are protected by a critical section when accessed or changed anywhere else in the code.**
    Frequently, hard-to-track bugs will occur when variables that are used inside an ISR are not protected elsewhere in the code. For example, one section of the driver increments a variable by one and then does an 'if' statement based on that variable on the next line of code. However, the driver's ISR function also modifies that variable. If an interrupt occurs between the variable being incremented and the variable being checked, it could lead to unexpected results. This is also true in the reverse. If you are using a long integer on a 16-bit system, it takes more than one processor instruction to modify that variable. If this variable is checked in the ISR, it must be protected elsewhere in the code to ensure that an interrupt does not occur while the variable is only partially modified.

14. **Temporarily put a critical section around the send and receive functions in the driver to isolate the problem area.**
    If there is a bug such as mentioned immediately above, temporarily putting critical sections around your entire driver send or receive function may indicate which function the error is in. By moving the critical sections

around, you may be able to narrow down the area in which the error is occurring. If you decide to do this, be wary of printf() statements or the like which may unexpectedly re-enable interrupts.

15. **Double-check every item in the list of common symptoms. Don’t assume anything; don’t trust your memory.**
    Everybody has made this mistake. Be very careful to check everything in the list, and don’t make any assumptions about how you have coded something. Check it!

Common Symptoms and Their Causes

Running Out of Memory

- Not calling tfSendCompleteInterface every time TM_USER_BUFFER_LAST is passed into the driver send routine.
- A problem with the receive ring logic. For example, a bug could cause a receive ring to be refilled even though the ring is already has empty buffers. In this case, the empty buffers would all be lost.
- Notifying the stack of an incorrect number of received packets. Though this problem will most frequently just cause the receive ring to fill up and never be emptied (making the chip unable to receive any further data), there are cases where it could lead to loss of memory.
- A memory leak could also be caused by an error in the application. For example, failing to free zero copy receive buffers when the application calls tfZeroCopyRecv() or tfZeroCopyRecvFrom() will cause a loss of memory. Another common problem occurs when the user calls tfZeroCopySend() or tfZeroCopySendTo() in non-blocking mode. If the error code is TM_EWOULDBLOCK is returned, the user still owns the buffer. The buffer is freed in every other case.
- Not having enough memory available for your system. Frequently, what appears to be a memory leak is simply a case of not giving the Turbo Treck stack enough memory to work with. Try allocating more memory for the stack if available, and see if allocations top out at a certain level.

Corrupted Memory

- Calling tfSendCompleteInterface() too soon or too frequently. This could cause packets that have not been sent to be freed, or packets to be freed more than once.
- Endian mismatch between CPU and Ethernet chip. For example, an Ethernet chip with DMA capabilities write data to a buffer in main memory, the address of which is written into one of its registers by the driver. The driver writes the address in little-endian mode, but the chip interprets it as big-endian and writes the data to the wrong part of memory.
- Invalid physical memory address passed to the Ethernet chip. For example, passing a logical address (data segment, offset) to a chip that is expecting a physical address could cause this problem.
- Invalid alignment when receiving data. Many Ethernet chips (if they support DMA) require that buffers they write into begin on a certain byte-boundary. These can be as small as two bytes (that is, buffers must begin on an even-numbered memory address), but are frequently as large as 16 bytes.
- Not checking for failed memory allocation (null pointer returned when allocating memory).
- The ISR is changing or checking data that is changed or checked anywhere else in the driver. If the data are not protected in the rest of the driver with critical sections, an interrupt could occur at an inopportune time and lead to memory corruption or a host of other problems.
- Locks are not working properly (that is, counting semaphores are not working properly). This can be tested quickly with the tfTestTreck() API call .
- Make sure that counting semaphores you create have an initial value of zero.

Receiving Corrupted Data

- Many chips do not tell you their pre-allocated receive buffer alignment requirement. A safe value to use, if you do not know, is 16 bytes.
- Passing a receive buffer up the stack that has not been yet filled by the chip. This is most frequently caused by an error in the receive-ring logic.

Kernel Error: Send Too Much Scattered Data

- This is a sign of corrupted memory. See the "Corrupted Memory" section above.

Kernel Error: Attempt to free more than alloc a buffer

- This can be a sign of corrupted memory. See the “Corrupted Memory” section above.
- This can also be an error in the application. For example, attempting to free a zero copy send buffer that has already been given to the stack will cause this error.

Ping and UDP work but TCP does not

- tfOpenInterface has enabled scattered send, but the driver either does not support scatter send, or device driver scatter send does not work properly. Try disabling scatter-send in tfOpenInterface and testing again.
- If you have added an assembly checksum routine (see the Tech Note included on the distribution CD), it might not be working correctly. Temporarily use the standard ‘C’ version of the checksum routine to test this. This will only be the case if checksums have been disabled for UDP.

Ping works but UDP and TCP do not

- This is frequently actually a routing problem. If there was no call made in the application to add a default gateway or a static route for the final destination of the packet, this problem will occur.
- If you have added an assembly checksum routine (see the Tech Note included on the distribution CD), it might not be working correctly. Temporarily use the standard ‘C’ version of the checksum routine to test this.

Driver Will Not Send and/or Receive Data

- The ISR is not installed correctly. The interrupts are occurring correctly, but the ISR is not being executed.
- Interrupts are not firing.
- The interrupt fires once, but then is not dismissed correctly, and will not fire again.
- Edge vs level triggered interrupts. If an interrupt is set to be edge triggered, it is possible that an incoming interrupt will be missed. Some chips will not interrupt again until the previous interrupt has been cleared. This will cause the chip to not send or receive data, or to suddenly stop sending and receiving data.
- Some chips require you to copy while inside the ISR before they will dismiss the ISR. For example the 3C509 and Crystal (CS8900) Ethernet chips have this requirement. If this is the case, you should make use of the Turbo Treck Driver Pool API.

- Ensure that you are communicating correctly with the chip. If it is not being set up correctly, it will not function.
- Make sure that the chip is set up correctly. Even if you are communicating with it, verify that the registers are being initialized correctly.
- tfRecvInterface() is not being called in the application.

Driver stops sending and/or receiving data

- Make sure that interrupts are being triggered correctly. That is, it is possible for an edge-triggered interrupt to be missed. Some chips will not continue operation until you have cleared the interrupt, causing the chip to cease working.
- The receive ring is not being refilled correctly. This can cause a chip to stop receiving data, as it believes there are no available buffers for it to write into.
- Receive-ring / send-ring miscoding can cause deadlock with the Ethernet chip. There are various cases where bugs in the ring logic within a driver can cause the chip to stop responding. Check your ring logic very carefully.
- An interrupt was not cleared correctly. Many chips will not interrupt again until you have cleared the last interrupt.
- If your OS has a separate post for tasks and ISRs (e.g. RTXC), make sure that these are used correctly. tfKernelIsrPostEvent should use the ISR specific post function.
- If your OS cannot post from an ISR at all, you should call tfNotifyInterfaceIsr from the OS's post-ISR function (eg, eCOS). That is, OS's that do not allow you to post to a semaphore or event flag inside an ISR will allow you to register a function that will be run after the ISR. Put your tfNotifyInterfaceIsr inside this function.

Other problems

- System locks up. This is usually caused by memory corruption (especially writing to memory address 0). See the "Corrupted Memory" section above. Getting stuck inside an ISR will also cause this behavior.
- Kernel Error: tfTcpSendPacket: send queue corrupted. This is symptomatic of counting semaphores not working correctly or not initializing the counting semaphore value to zero.