KwikNet®

TFTP Client / Server

User's Guide

Version 3

First Printing: **February 1, 2002**
Last Printing: **September 15, 2005**

Manual Order Number: PN303-9G

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5
Phone: (604) 734-2796
Fax: (604) 734-8114

TECHNICAL SUPPORT

KADAK Products Ltd. is committed to technical support for its software products. Our programs are designed to be easily incorporated in your systems and every effort has been made to eliminate errors.

Engineering Change Notices (ECNs) are provided periodically to repair faults or to improve performance. You will automatically receive these updates during the product's initial support period. For technical support beyond the initial period, you must purchase a Technical Support Subscription. Contact KADAK for details. Please keep us informed of the primary user in your company to whom update notices and other pertinent information should be directed.

Should you require direct technical assistance in your use of this KADAK software product, engineering support is available by telephone, fax or e-mail. KADAK reserves the right to charge for technical support services which it deems to be beyond the normal scope of technical support.

We would be pleased to receive your comments and suggestions concerning this product and its documentation. Your feedback helps in the continuing product evolution.

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5

Phone: (604) 734-2796
Fax: (604) 734-8114
e-mail: amxtech@kadak.com

DISCLAIMER

KADAK Products Ltd. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability and fitness for any particular purpose. Further, KADAK Products Ltd. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of KADAK Products Ltd. to notify any person of such revision or changes.

TRADEMARKS

AMX in the stylized form and KwikNet are registered trademarks of KADAK Products Ltd.
AMX, AMX/FS, InSight, *KwikLook* and KwikPeg are trademarks of KADAK Products Ltd.
UNIX is a registered trademark of AT&T Bell Laboratories.
Microsoft, MS-DOS and Windows are registered trademarks of Microsoft Corporation.
All other trademarked names are the property of their respective owners.

KwikNet TFTP Client / Server User's Guide

Table of Contents

Page

1. KwikNet TFTP Overview **1**

1.1 Introduction 1
1.2 General Operation 2
File Data Transfers 2
The KwikNet TFTP API 3
Transfer Mode 4
Blocking and Non-Blocking Operation 4
Changing Operating Mode 5
End of File Indication 5
Timing Requirements 6
TFTP Error Packets 7
TFTP Descriptor Options 7
TFTP User Variable 7
TFTP Logging 8
1.3 KwikNet TFTP Configuration 9
1.4 TFTP Client Task 11
Multitasking Operation 12
Single Threaded Operation 12
1.5 TFTP Server Task 13
Multitasking Operation 14
Single Threaded Operation 14
1.6 TFTP Sample Program 15
Startup 16
TFTP Client Operation 17
TFTP Server Operation 18
Shutdown 19
KwikNet and TFTP Logging 19
Running the Sample Program 20
1.7 Making the TFTP Sample Program 21
TFTP Sample Program Directories 21
TFTP Sample Program Files 22
TFTP Sample Program Parameter File 23
TFTP Sample Program KwikNet Library 23
The TFTP Sample Program Make Process 24
1.8 Adding TFTP to Your Application 25
KwikNet Library 25
Memory Allocation 25
TFTP Client and Server Tasks 26
Reconstructing Your KwikNet Application 27
AMX Considerations 27
Performance and Timing Considerations 28

2. KwikNet TFTP Services **29**

2.1 Introduction to TFTP Services 29
KwikNet Procedure Descriptions 29
2.2 TFTP Service Procedures 31

This page left blank intentionally.

1. KwikNet TFTP Overview

1.1 Introduction

The Trivial File Transfer Protocol (TFTP) is a standard protocol that uses UDP datagrams for transferring files between networked machines. The KwikNet TFTP option implements this protocol using UDP services provided by the KwikNet® TCP/IP Stack. KwikNet is a compact, reliable, high performance TCP/IP stack, well suited for use in embedded networking applications.

TFTP provides reliable file transfers, despite the fact that it operates using the unreliable User Datagram Protocol (UDP) layer. TFTP implements a simple peer-to-peer handshaking strategy to ensure that UDP datagrams are ordered correctly and are retransmitted if necessary.

A remote target may use TFTP to dynamically load operational parameters from a host computer. The host can use standard TFTP server or client software to provide the data. Using TFTP also eliminates the need to use the Transport Control Protocol (TCP) for reliable data transfer. By eliminating TCP from the application, code size and complexity can be reduced in the target.

The KwikNet TFTP option is best used with a real-time operating system (RTOS) such as KADAK's AMX™ Real-Time Multitasking Kernel. However, the KwikNet TFTP option can also be used in a single threaded environment without an RTOS. The KwikNet Porting Kit User's Guide describes the use of KwikNet with your choice of RT/OS. Note that throughout this manual, the term RT/OS is used to refer to any operating system, be it a multitasking RTOS or a single threaded OS.

You can readily tailor the KwikNet stack to accommodate your TFTP needs by using the KwikNet Configuration Builder, a Windows® utility which makes configuring KwikNet a snap. Your KwikNet stack will only include the TFTP features required by your application.

This manual makes no attempt to describe the Trivial File Transfer Protocol (TFTP), what it is or how it operates. It is assumed that you have a working knowledge of the TFTP protocol as it applies to your needs. Reference materials are provided in Appendix A of the KwikNet TCP/IP Stack User's Guide.

The purpose of this manual is to provide the system designer and applications programmer with the information required to properly configure and implement a networking system using the KwikNet TCP/IP Stack and TFTP. It is assumed that you are familiar with the architecture of the target processor.

KwikNet and its options are available in C source format to ensure that regardless of your development environment, your ability to use and support KwikNet is uninhibited. The source program may also include code fragments programmed in the assembly language of the target processor to improve execution speed.

The C programming language, commonly used in real-time systems, is used throughout this manual to illustrate the features of KwikNet and its TFTP option.

1.2 General Operation

The TFTP protocol is formally defined by the IETF document STD-33. The KwikNet TFTP option is compliant with that specification. The RFC should be consulted for any detailed questions concerning the TFTP protocol. The KwikNet TFTP option implements the subset of TFTP features typically required for use in embedded applications.

TFTP is a peer-to-peer protocol. A file transfer is initiated between two peers in client-server fashion. One machine, the client, initiates a file transfer by sending a request to another machine, the server. The direction of the transfer (to or from the server) is indicated in the request. If the server grants the request, the designations of client and server become meaningless and the transfer proceeds with one peer assuming the role of sender and the other assuming the role of receiver. Once the file sender has completed the transfer, it severs the connection with the receiver. At any time during the transfer, either peer can abort the session and terminate the connection.

TFTP does not support persistent connections. Each connection begins with a file transfer request by one peer (the client) to the other peer (the server). The connection between the two peers lasts only until the transfer has been completed or aborted.

The KwikNet TFTP option provides all of the services necessary to implement one or more TFTP clients and servers. Although multiple clients and multiple servers can coexist and operate concurrently, most applications will require only a single TFTP client or a single TFTP server.

File Data Transfers

As its name implies, the Trivial File Transfer Protocol (TFTP) is designed to allow the transfer of file data between networked machines. However, interpretation of the term "file data" is left to the application clients and servers which use TFTP. The transferred data is only file related if a TFTP peer actually uses a local file system to read (write) the data which it sends (receives).

The KwikNet TFTP option provides a simple collection of services which make the transfer of data between peers independent of any particular file system. In fact, when using KwikNet TFTP, **there is no need for an actual file system**, a feature making KwikNet TFTP ideally suited for use in embedded applications which have no file system. Frequently such systems simply use the file name as an identifier which specifies the particular information (data) to be transferred.

When a TFTP connection is requested by a TFTP client, the client identifies the name of the file to be transferred. It is up to the TFTP server to decode the filename and decide if it can send (or fetch) the named file. Similarly, it is the TFTP client's responsibility to determine the local destination (source) of the file data being transferred.

If a KwikNet TFTP peer (client or server) utilizes a real file system, it is responsible for local file access. When transferring a file from its peer, the KwikNet TFTP peer must open the named local file for write access prior to fetching the file data from its peer. The KwikNet peer must write all received data to the local file. When transferring a file to its peer, the KwikNet TFTP peer must open the named local file for read access prior to sending the file data to its peer. The KwikNet peer must read the file data from the local file and send it to its peer.

The KwikNet TFTP API

The formal STD-33 TFTP specification describes the fundamentals of the TFTP protocol. The specification defines the various TFTP datagrams, their content and their intended uses. The specification hints at connection timeouts and datagram retransmission times but gives no definitive rules for implementing such services. Also omitted from the specification are guidelines for an **application programming interface (API)** for accessing TFTP services. Hence, there is no standard TFTP implementation.

The KwikNet TFTP API draws upon features found in most file systems and TCP socket implementations, making the KwikNet TFTP API familiar to experienced programmers. Following the TCP paradigm, a KwikNet TFTP peer establishes a TFTP connection which lasts for the duration of the transfer.

A KwikNet TFTP client opens a connection to a TFTP server port with a call to procedure *knтf_open()*. The TFTP connection is then referenced using a **TFTP descriptor**, a unique handle assigned by KwikNet to identify the connection.

A KwikNet TFTP server calls *kntf_listen()* to create a TFTP listening port which KwikNet identifies with a special **TFTP server descriptor**. The server can then call *kntf_accept()* to accept or reject a connection request directed to the server's TFTP port from any TFTP client. The connection, once accepted by the server, is identified by the TFTP descriptor assigned by KwikNet to the connection.

Once a TFTP connection has been established, the roles of the two peers are similar. A TFTP client (server) calls *kntf_read()* to fetch file data from a TFTP server (client). A TFTP client (server) calls *kntf_write()* to send file data to a TFTP server (client).

The end of the file transfer is signaled by the sender in its final data transfer to its peer. Once the transfer has been completed, the KwikNet TFTP client (server) must call *kntf_close()* to close the TFTP connection which it opened (accepted). A KwikNet TFTP server can also call *kntf_close()* to close its listening port when it is no longer required.

Other KwikNet services are also available for use by KwikNet TFTP clients and servers. Procedure *kntf_abort()* can be used to abort a transfer with an error indication to the peer describing the reason for the severed connection. Procedure *kntf_option()* can be used to sense or change various operating attributes and parameters associated with a particular TFTP connection. And finally, procedure *kntf_status()* can be used to generate a TFTP status log on the KwikNet logging device summarizing the current state of a particular TFTP connection.

> Note
>
> The KwikNet TFTP option is a KADAK software product, separate from that offered by Treck Inc. Hence the TFTP application programming interface (API) described in Chapter 6 of the Treck TCP/IP User Manual does not apply to the KwikNet TFTP component.

Transfer Mode

TFTP supports three modes of transfer as defined in IETF document STD-33. The transfer mode is specified by the TFTP peer (client) which initiates the transfer. The *netascii* transfer mode is used to indicate that the file data being transferred is considered to be text. The *octet* transfer mode indicates that the file data being transferred is binary in nature. The obsolete *mail* transfer mode is not supported by the KwikNet TFTP option.

The transfer mode does not actually affect the manner in which file data is transferred. KwikNet always transfers text or binary file data as a sequence of 8-bit bytes in a TFTP packet.

The transfer mode is provided for use by a TFTP peer which must adapt its local data access method to match the type of data being transferred. For example, if a TFTP peer utilizes a local file system, it can use the TFTP transfer mode indicator to adjust its file open request to ensure that the local file is accessed in the appropriate text or binary mode.

Blocking and Non-Blocking Operation

A KwikNet TFTP peer can service a TFTP connection in one of two operating modes: blocking or non-blocking. The operating mode is determined by the manner in which the connection is established. A KwikNet TFTP client specifies the operating mode in its call to `kntf_open()` to establish a TFTP connection. A KwikNet TFTP server specifies its preferred operating mode in its call to `kntf_listen()` to create a TFTP listening port. Client connections to that port, when accepted by the server, inherit the server's operating mode.

When servicing a TFTP connection opened in **blocking mode**, a KwikNet TFTP peer will be forced to wait whenever it initiates a TFTP transaction which cannot be completed without waiting for a response from its peer. For example, when opening a connection, a TFTP client will be blocked and forced to wait until the connection is established. A TFTP server will be blocked when it tries to accept a client connection request if none are pending. A client or server will also be blocked while KwikNet transfers file data to or from its peer.

In a multitasking system, lower priority tasks will be allowed to execute while a KwikNet TFTP client or server is blocked. In a single threaded system, KwikNet will continue to operate but your application's App-Task will be blocked waiting for the TFTP operation to complete.

When servicing a TFTP connection opened in **non-blocking mode**, a KwikNet TFTP peer does not have to wait for any TFTP transaction to finish, even if a response from its peer is required to complete the operation. The KwikNet peer will receive an error indication (`KN_ERTFWOULDBLOCK`) if the transaction would normally have forced it to wait for the transaction to complete. For example, when opening a connection, a TFTP client will not be blocked, even though the connection will not be established until the client's request is acknowledged by the server. Similarly, a TFTP server will not be blocked if it tries to accept a client connection request and none are pending. Furthermore, a client or server will not be blocked while KwikNet transfers file data to or from its peer.

In a multitasking system, a KwikNet TFTP client or server which services a non-blocking TFTP connection must periodically relinquish control of the processor to allow lower priority tasks to execute. In a single threaded system, the KwikNet peer must regularly call *kn_yield()* to give the KwikNet Task an opportunity to service the TCP/IP stack.

Changing Operating Mode

The operating mode for a TFTP connection can be changed by a KwikNet TFTP client or server with a call to service procedure *kntf_option()*. A client could open its TFTP connection for non-blocking operation. Then, once the transfer completed, the client could switch the connection to blocking mode, allowing it to wait until its call to *kntf_close()* actually severed the connection.

A server could accept connections in non-blocking mode so that it would never block if a connection request was unavailable. The server would then be free to service outstanding TFTP connections for which it had assumed responsibility. In a multitasking system, the server task might pass each non-blocking connection which it accepted to another task having access to file services. The file task might choose to switch the TFTP connection to blocking mode to simplify its method of operation.

End of File Indication

The TFTP protocol specifies that the end of a file transfer is signaled by the sender. The signal is a TFTP data transfer with less than a full size payload of 512 bytes.

A KwikNet TFTP client or server which is sending file data signals the successful end of the transfer by calling KwikNet service procedure *kntf_write()* with a special end of file indication. All file data previously accepted for transmission is sent to the peer. If no untransmitted data remains, a zero length block of file data is sent to the peer. Either of these data transmissions acts as an end of file indication to the peer.

When fetching file data, a KwikNet TFTP client or server call to KwikNet service procedure *kntf_read()* will return a data count of 0 as an indication that the end of the file has been reached.

Timing Requirements

The TFTP protocol demands a timely response to each transmitted packet. Each transmitted packet acts as a response to the most recently received packet. When a TFTP peer is receiving data, it sends a TFTP acknowledgement packet as the response to each received data packet. When sending data, the peer sends a data packet upon receipt of a TFTP acknowledgement packet. Thus, once started, a TFTP transfer will proceed to completion without excessive delay.

If an expected response fails to materialize, a TFTP peer will retransmit its TFTP packet, be it an acknowledgement packet or a data packet. Eventually, if no response is received, the TFTP peer will sever its connection.

The KwikNet TFTP peer (client or server) establishes the pace at which file data can be transferred to or from its peer. Once a TFTP connection has been established, the KwikNet TFTP peer must service the connection fast enough to prevent its peer from losing the connection. Several factors influence the required connection service rate.

If a KwikNet TFTP peer is sending data, it must have ready access to the data to be sent. If the peer is fetching data, it must be able to accept (store or process) the received data in a timely fashion. In either case, the peer must make regular calls to *`kntf_write()`* or *`kntf_read()`* to sustain the TFTP transfer.

If the KwikNet TFTP connection operates in blocking mode, the connection will be serviced at the TFTP service interval specified in your KwikNet Network Parameter File (see Chapter 1.3). Hence, a brief pause equal to the service interval will occur following each transmission of a data packet or whenever more data is required from the peer.

If the KwikNet TFTP connection operates in non-blocking mode, the connection will only be serviced when your KwikNet TFTP peer calls to *`kntf_write()`* or *`kntf_read()`*. The TFTP data transfer rate will therefore be dictated by your application, to the limit imposed by the actual network and the foreign TFTP peer. Hence, the non-blocking mode of operation offers the best possible TFTP throughput.

TFTP Error Packets

The TFTP specification includes an error reporting mechanism. At any time during a TFTP file transfer session, a TFTP error packet can be sent by either peer to terminate the transfer. The error packet contains an error number and text string which identifies the error condition which led to the severed connection. Since an error packet is never acknowledged by its receiver, it is never retransmitted by its sender.

Your KwikNet TFTP peer can call service procedure *knтf_abort()* to sever a TFTP connection by sending a TFTP error packet to the foreign peer. Your TFTP peer can provide the TFTP error number and text string describing the reason for the termination, both of which will be sent to the foreign peer in the TFTP error packet.

If a KwikNet TFTP connection is terminated by receipt of a TFTP error packet from the foreign peer, the error number and text string, if any, from the error packet are saved for access by the KwikNet peer. When the KwikNet peer services the TFTP connection, its KwikNet function call will be rejected with error code *KN_ERTFPROTO*, indicating that the connection was severed by the foreign peer. The KwikNet peer can call *kntf_option()* to retrieve the peer's error number and descriptive text.

TFTP Descriptor Options

KwikNet assigns a TFTP descriptor for each TFTP connection which it establishes. It also allocates a TFTP server descriptor for listening for client transfer requests. Many of the attributes and parameters associated with a TFTP descriptor can be accessed or modified by your application using KwikNet service procedure *kntf_option()*.

The following TFTP descriptor options are recognized by KwikNet. The option mnemonics are defined in file *KN_TFTP.H*. Options marked R can be read; those marked W can be modified. A parameter of the specified type must be passed to function *kntf_option()* when manipulating a specific option.

Option	Parameter Type	R/W	Purpose
KN_TFTP_OPT_GETMODE	*int **	R	Get operating mode
KN_TFTP_OPT_SETMODE	*long*	W	Set operating mode
KN_TFTP_OPT_GETVAR	*long **	R	Get private user variable
KN_TFTP_OPT_SETVAR	*long*	W	Set private user variable
KN_TFTP_OPT_GETLPORT	*unsigned short **	R	Get local port number
KN_TFTP_OPT_GETFPORT	*unsigned short **	R	Get foreign port number
KN_TFTP_OPT_GETFADDR	*struct in_addr **	R	Get foreign IPv4 address
KN_TFTP_OPT_GETERRNUM	*long **	R	Get TFTP error number
KN_TFTP_OPT_GETERRMSG	*char ***	R	Get TFTP error message

TFTP User Variable

KwikNet allows your TFTP client or server to maintain an application specific variable associated with any KwikNet TFTP descriptor. This user variable is a *long* value whose purpose is dictated by your application. KwikNet function *kntf_option()* must be used to set/get the user variable associated with a particular TFTP descriptor.

TFTP Logging

KwikNet provides a data logging service which can be used to advantage to observe the progress of stack activity. This service is described in Chapter 1.6 of the KwikNet TCP/IP Stack User's Guide.

The KwikNet TFTP client or server can be configured to record error information on the KwikNet logging device. To enable this feature, edit your KwikNet Network Parameter File and check the "Log errors" option on the TFTP property page.

The TFTP server does not log the normal receipt of transfer requests from TFTP clients. Nor do the KwikNet TFTP client and server log the successful transfer of file data between TFTP peers. Since these operations involve normal UDP transactions, they can be observed using standard KwikNet debug logging and trace facilities.

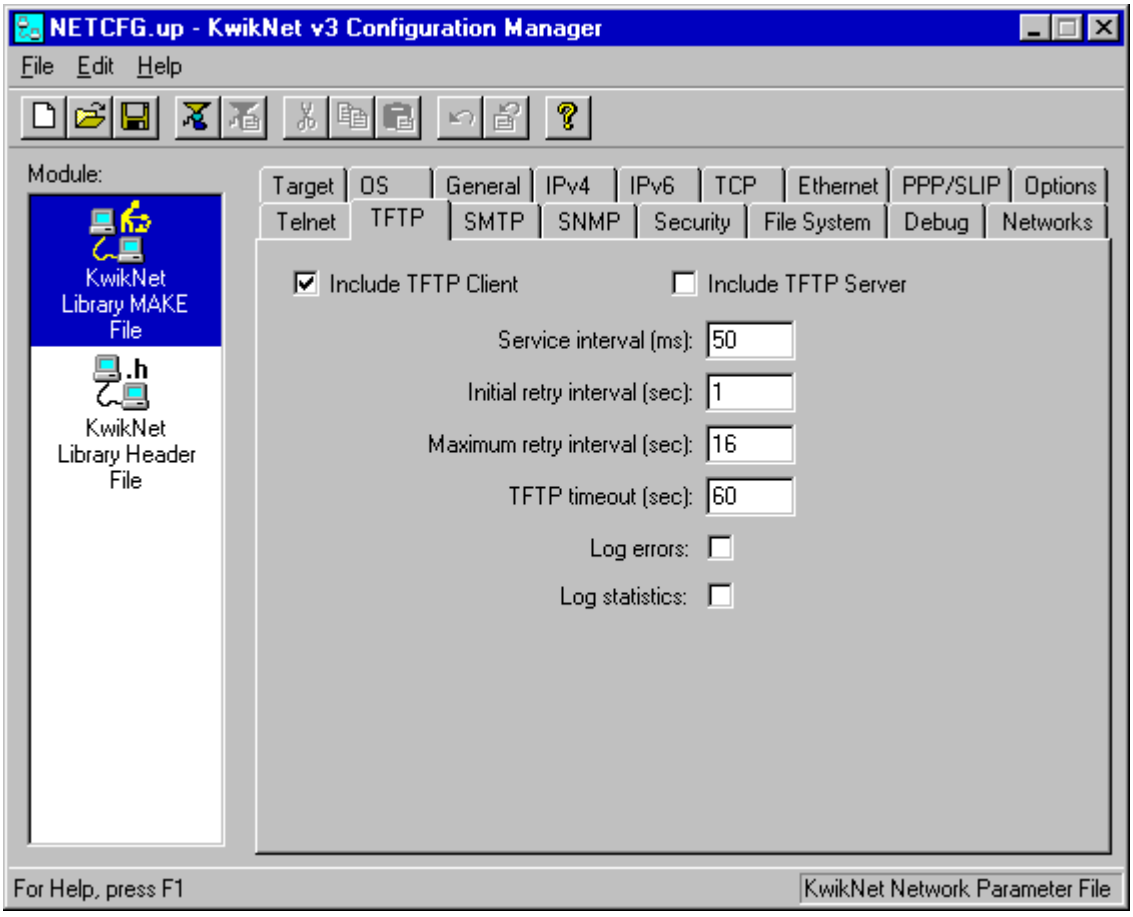
The KwikNet TFTP client and server log all TFTP error packets which are sent to or received from a TFTP peer. They also log abnormal run-time conditions which usually indicate that KwikNet's private TFTP control structures have been compromised (altered or misused) by your application.

If TFTP statistics logging is enabled, your KwikNet TFTP client or server can call procedure *`kntf_status()`* to generate a TFTP status log summarizing the current state of a particular TFTP connection. The status log identifies the TFTP endpoints and describes the operating characteristics of the connection. The log includes a statistics summary, counts of various events detected while the TFTP connection was being serviced. The status summary will be logged on the KwikNet logging device. To enable this feature, edit your KwikNet Network Parameter File and check the "Log statistics" option on the TFTP property page.

1.3 KwikNet TFTP Configuration

You can readily tailor the KwikNet stack to accommodate your TFTP needs by using the KwikNet Configuration Builder to edit your KwikNet Network Parameter File. The KwikNet Library parameters are edited on the TFTP property page. The layout of the window is shown below.

Note that the TCP protocol and a file system are not prerequisites for the TFTP protocol.



TFTP Parameters (continued)

Include TFTP Client

Check this box if your application will include a TFTP client. Otherwise, leave this box unchecked.

Include TFTP Server

Check this box if your application will include a TFTP server. Otherwise, leave this box unchecked.

Service Interval

Specify the interval in milliseconds at which a TFTP client or server should service its TFTP connection when operating in blocking mode.

Initial/Maximum Retry Interval and TFTP Timeout

Specify the initial interval in seconds which a TFTP client or server is allowed to wait for a response from its TFTP peer before initiating a retransmission of the TFTP packet for which a response is required. The retry interval is doubled after each retransmission but is not allowed to exceed the specified maximum retry interval, also measured in seconds.

Transmission retries will continue until the cumulative elapsed time from the initial transmission exceeds the specified TFTP timeout interval.

Log Errors

Check this box if you wish to log all TFTP error messages generated or received by your TFTP client or server. All logged messages will be directed to the KwikNet logging device, if one has been provided. Otherwise, leave this box unchecked. Leaving this box unchecked reduces the code size by eliminating all error message generation logic from the TFTP service procedures in the KwikNet Library.

Log Statistics

Check this box if your TFTP client or server will use function *`kntf_status()`* to generate a TFTP status summary for a TFTP connection. The summary will be directed to the KwikNet logging device, if one has been provided. Otherwise, leave this box unchecked. Leaving this box unchecked reduces the code size by eliminating all status summary generation logic from the KwikNet Library.

1.4 TFTP Client Task

The KwikNet TFTP option includes a set of services for use by one or more TFTP clients. Each TFTP client is an application program which makes use of these services to communicate with a TFTP server. The collection of application procedures which makes up such a program is called a TFTP client task.

If your TFTP client task utilizes an actual file system, it must be initialized and ready for use before the client task uses any KwikNet TFTP client services to transfer its files. All disk drives which the TFTP client will be permitted to access must be mounted and ready for use.

Your TFTP client task must call *knтf_open()* to establish a connection with a particular TFTP server. The IPv4 address of the TFTP server and its TFTP port number must be provided by your client task. The file name and file transfer mode must be provided so that the TFTP server can identify the file being transferred and the direction of the transfer. The client must also specify whether the connection is to operate in blocking or non-blocking mode. All of these parameters must be provided by your client task in its call to *knтf_open()*.

If a file system is being used, the client must open the file of interest in the appropriate access mode. The file should be opened prior to initiating the TFTP connection to avoid unnecessary delay once the connection has been established.

Once connected to the TFTP server, the client task must call *knтf_write()* or *knтf_read()* (see Chapter 2) to transfer file data to or from the server. It is the TFTP client's responsibility to access the data to be sent or save the data received. If a file system is being used, the client must read the file to get data for transmission or write received data to the file.

Using KwikNet TFTP services is much like using a file system. Operations go to completion or until an error condition is encountered. Hence, when a TFTP client makes a call to a TFTP service procedure, that operation must complete before another operation can be initiated. If the TFTP connection operates in blocking mode, the TFTP client will be forced to wait until each operation completes before being allowed to resume. In non-blocking mode, the client will receive an error indication (*KN_ERTFWOULDBLOCK*) if the operation cannot be initiated until a prior operation completes.

When sending file data, the TFTP client task must call *knтf_write()* with a special end of file indication to ensure that a termination packet is sent to the server. This final packet will include any data previously buffered for transmission. When sending data in non-blocking mode, the client must repetitively call *knтf_write()* with the end of file indication until the server acknowledges receipt of the final end of file packet.

When the file transfer is done, the TFTP client task closes the connection with a call to procedure *knтf_close()*. Once the connection is closed, further TFTP file data transfers cannot be performed by the client without first calling *knтf_open()* to establish a new connection.

The TFTP Sample Program provided with the KwikNet TFTP option illustrates a TFTP client task. The client task uses the KwikNet console driver (see Chapter 1.8 of the KwikNet TCP/IP Stack User's Guide) to record its progress on a simple console device.

Multitasking Operation

When used with a real-time operating system (RTOS) such as KADAK's AMX Real-Time Multitasking Kernel, each TFTP client must be an application task. Although one task can be written to service multiple TFTP connections, it is usual, and conceptually simpler, to consider each TFTP client to be a unique task. Such a task is referred to as a TFTP client task.

A TFTP client task is created and started just like any other application task. Once started, the TFTP client task operates as previously described.

> Note
>
> In multitasking systems, each KwikNet TFTP client task MUST execute at a priority below that of the KwikNet Task.

Single Threaded Operation

When used with a single threaded operating system, the TFTP client operates in the user domain as part of your App-Task as described in Chapter 1.2 of the KwikNet TCP/IP Stack User's Guide. When your App-Task is executing your client code, the App-Task is referred to as a TFTP client task.

While executing as a TFTP client, your App-Task must continue to regularly call KwikNet procedure *kn_yield()* to let the KwikNet TCP/IP Stack continue to operate. Fortunately, when your TFTP client initiates any TFTP transaction in blocking mode, KwikNet ensures that the TCP/IP stack continues to operate until the TFTP transaction is complete.

Although KwikNet can support multiple, concurrent TFTP connections, it is up to your TFTP client task to manage the separate TFTP transaction sequences for each of the connections.

1.5 TFTP Server Task

The KwikNet TFTP option includes a set of services which can be used to implement a TFTP server. The TFTP server is an application program which makes use of these services to communicate with one or more TFTP clients.

If your TFTP server task utilizes an actual file system, it must be initialized and ready for use before the server task uses any KwikNet TFTP client services to transfer its files. All disk drives which the TFTP server will be permitted to access must be mounted and ready for use.

Your TFTP server can provide a specific IPv4 address which clients must use to connect to the TFTP server. Alternatively, the TFTP server can service all TFTP transfer requests directed to any of the IP addresses assigned to the server's network node.

By default, the TFTP server accepts requests directed to the well known TFTP command port number 69. However, a KwikNet TFTP server can be setup to service requests on any port number which you choose.

Your TFTP server task must call *`kntf_listen()`* to create a TFTP server descriptor which can be used to detect requests from TFTP clients. The IPv4 address of the TFTP server and its TFTP port number must be provided by the server. The server must also specify whether the server is to operate in blocking or non-blocking mode. Most servers will operate in non-blocking mode so that multiple concurrent connections can be serviced. All of these parameters must be provided by your server task in its call to *`kntf_listen()`*.

Your TFTP server task must then call *`kntf_accept()`* to test for a transfer request from a TFTP client. In blocking mode, the TFTP server will be forced to wait until a transfer request arrives. In non-blocking mode, the server will receive an error indication (*`KN_ERTFWOULDBLOCK`*) if no transfer request is pending. Upon receipt of a request from a TFTP client, a TFTP connection is established with that client. The connection inherits the server's mode of operation, be it blocking or non-blocking.

Your TFTP server task must then service the TFTP connection or delegate that service responsibility to another task. In keeping with the word trivial in the protocol name, your server might simply complete the requested transfer before accepting any other connections. However, most TFTP servers should maintain a list of active TFTP connections and ensure that each of the connections is serviced periodically.

If a file system is being used, the server must open the file of interest in the appropriate access mode. The server task must call *`kntf_write()`* or *`kntf_read()`* (see Chapter 2) to transfer file data to or from the client. It is the TFTP server's responsibility to access the data to be sent or save the data received. If a file system is being used, the server must read the file to get data for transmission or write received data to the file.

Using KwikNet TFTP services to transfer data is much like using a file system. Operations go to completion or until an error condition is encountered. Hence, when a TFTP server makes a call to a TFTP service procedure, that operation must complete before another operation can be initiated. If the TFTP connection operates in blocking mode, the TFTP server will be forced to wait until each operation completes before being allowed to resume. In non-blocking mode, the server will receive an error indication (*`KN_ERTFWOULDBLOCK`*) if the operation cannot be initiated until a prior operation completes.

When sending file data, the TFTP server task must call *kntf_write()* with a special end of file indication to ensure that a termination packet is sent to the client. This final packet will include any data previously buffered for transmission. When sending data in non-blocking mode, the server must repetitively call *kntf_write()* with the end of file indication until the client acknowledges receipt of the final end of file packet.

When the file transfer is done, the TFTP server task closes the connection with a call to procedure *kntf_close()*. If transfers are being serviced sequentially by the TFTP server, it must resume calling *kntf_accept()* to poll for the next client transfer request.

The TFTP Sample Program provided with the KwikNet TFTP option illustrates a TFTP server task. The server task uses the KwikNet console driver (see Chapter 1.8 of the KwikNet TCP/IP Stack User's Guide) to record its progress on a simple console device.

Multitasking Operation

When used with a real-time operating system (RTOS) such as KADAK's AMX Real-Time Multitasking Kernel, the TFTP server operates as an application task. Such a task is referred to as a TFTP server task. Although more than one TFTP server task is allowed, rarely is there such a need.

A TFTP server task is created and started just like any other application task. Once started, the TFTP server task operates as previously described.

> Note
>
> In multitasking systems, each KwikNet TFTP server task MUST execute at a priority below that of the KwikNet Task.

Single Threaded Operation

When used with a single threaded operating system, the TFTP server can operate in the user domain or in the KwikNet domain. When the TFTP server operates in the user domain, it executes as part of your App-Task as described in Chapter 1.2 of the KwikNet TCP/IP Stack User's Guide. When your App-Task is executing your server code, the App-Task is referred to as a TFTP server task.

While executing as a TFTP server, your App-Task must continue to regularly call KwikNet procedure *kn_yield()* to let the KwikNet TCP/IP Stack continue to operate. Fortunately, when your TFTP server initiates any TFTP transaction in blocking mode, KwikNet ensures that the TCP/IP stack continues to operate until the TFTP transaction is complete.

To operate in the KwikNet domain, your application must call *kn_addserver()* (see Chapter 4.6 of the KwikNet TCP/IP User's Guide) from your App-Task to add your TFTP server function to the KwikNet server list. Since your TFTP server function will execute in the context of the KwikNet Task, it must operate in non-blocking mode so that KwikNet can continue to service the TCP/IP stack. Once the TFTP server is operational, your App-Task must regularly call KwikNet procedure *kn_yield()* to let KwikNet and the TFTP server operate.

1.6 TFTP Sample Program

A TFTP Sample Program is provided with the KwikNet TFTP option to illustrate the use of the KwikNet TFTP client and server. The sample program is ready for use with the AMX Real-Time Multitasking Kernel. The sample program can also be used with any of the porting examples provided with the KwikNet Porting Kit.

The sample configuration supports a single network interface. The network uses the KwikNet Ethernet Network Driver. Because the sample must operate on all supported target processors without any specific Ethernet device dependence, KwikNet's Ethernet Loopback Driver is used. Use of this driver allows the TFTP client and server to be tested even if network hardware is not available.

Once the TFTP Sample Program has been tested in loopback fashion, you can replace the Ethernet Loopback Driver with your own network device driver. Then the KwikNet TFTP client and server can be adapted to meet your needs, allowing the client to connect to other TFTP servers and foreign clients to access the server.

The KwikNet TCP/IP Stack requires a clock for proper network timing. All examples provided with the KwikNet Porting Kit illustrate the clock interface. However, the sample program provided for use with AMX has been enhanced to eliminate any dependence on specific target hardware. This sample program includes a very low priority task that can detect if you have added a real AMX clock driver to the sample configuration. If a real hardware clock is not available, this task simulates clock interrupts, thereby providing AMX ticks that meet KwikNet's needs.

The sample includes a TFTP server task and a TFTP client task. In a multitasking system, these tasks are real tasks managed by the RTOS. In a single threaded system, the server is attached to the KwikNet Task's server queue and operates in the KwikNet domain. The client is simply the App-Task executing in the user domain.

The sample TFTP server task creates a TFTP server session and waits for a file transfer request from a TFTP client. The TFTP client task opens a TFTP connection, requesting to send a file to the server. Upon receiving the request from the client, the server grants the request. The TFTP client task then sends the file data and completes the transfer by closing the TFTP connection. The server then waits for another file transfer request.

Next, the TFTP client task requests access to a non-existent file from the server. The server responds with an error message indicating that no such file exists. The client closes the TFTP connection upon receiving the error notification.

Finally, the TFTP client task requests the server to return the original file sent to it by the client. The server honors the request by sending the data. When the transfer is complete the server closes the TFTP connection. The client receives the file data and, when the end of file indication is received, closes its end of the TFTP connection.

The sample uses the KwikNet data logging and message recording services to record messages generated by the TFTP server task, the TFTP client task and the KwikNet TCP/IP Stack. These services are described in Chapters 1.6 and 1.7 of the KwikNet TCP/IP Stack User's Guide. Messages are stored as an array of strings in memory but can be easily echoed to a console terminal (see Chapter 1.8 of the KwikNet TCP/IP Stack User's Guide).

Startup

The manner in which the KwikNet TFTP Sample Program starts and operates is completely dependent upon the underlying operating system with which KwikNet is being used. All sample programs provided with KwikNet and its optional components share a common implementation methodology which is described in Appendix E of the KwikNet TCP/IP Stack User's Guide. Both multitasking and single threaded operation are described.

When used with AMX, the sample program operates as follows. AMX is launched from the *main()* program. Restart Procedure *rrproc()* starts the print task and calls function *app_prep()* which starts KwikNet at its entry point *kn_enter()* and then proceeds to start the TFTP server and client tasks. A low priority background task is also started to simulate clock interrupts in the absence of a hardware clock.

Once the AMX initialization is complete, the high priority print task executes and waits for the arrival of AMX messages in its private mailbox. Each AMX message includes a pointer to a log buffer containing a KwikNet message to be recorded.

Once the print task is ready and waiting, the KwikNet Task begins executing because it is of higher priority than all tasks which can use its service. The KwikNet Task initializes the network and its associated loopback driver and prepares the IP and UDP protocol stacks for use by the sample program.

Once the KwikNet initialization is complete, the TFTP server task begins to execute. When the server reaches a state in which it is waiting for client requests, the lower priority TFTP client task begins to execute.

In a single threaded system, the TFTP client task executes in the user domain. The TFTP Sample Program uses KwikNet function *kn_addserver()* to add the TFTP server procedure to the KwikNet server queue so that it will be executed in the KwikNet domain at periodic intervals to service TFTP client requests.

TFTP Client Operation

Once the TFTP client task begins, it calls *knf_open()* to connect to a TFTP server to send a file to the server. The client opens the connection to operate in blocking mode. The client is therefore forced to wait until the server has indicated that it is willing to accept the file.

The TFTP client then calls *knf_write()* to transfer file data to the server in binary mode. Since the sample does not incorporate an actual file system, the file data is simply a small, fixed size block of data created by the client. When the transfer is complete, the client calls *knf_write()* with an end of file indication to ensure that untransmitted data is sent to the server. The client then generates a TFTP status summary on the KwikNet logging device. Finally, the client calls *knf_close()* to sever the TFTP connection.

The TFTP client then begins an error test. The client calls *knf_open()* to retrieve a non-existent file from the TFTP server. The client receives the server's rejection and records the reported TFTP error number and error message received from the server.

Finally, the TFTP client prepares to retrieve the file previously sent to the server. The client calls *knf_open()* to read the file in binary mode. This time, the connection is opened in non-blocking mode. The client repeatedly calls *knf_read()* to retrieve the file data, delaying briefly between calls. The client ignores *KN_ERTFWOULDBLOCK* errors which simply indicate that data is not yet available for reading. When all file data has been received, the client will observe a transfer count of 0, indicating that the end of the file has been reached. The client then generates a TFTP status summary on the KwikNet logging device and calls *knf_close()* to close the TFTP connection.

To signal that the sample is finished, the TFTP client task sets the TFTP server task's state variable into a state which will force the server task to shut down.

TFTP Server Operation

The sample TFTP server is implemented as a simple state machine which is driven by periodic calls to service function *server_proc()*. In a multitasking system, this function is called at periodic intervals by the TFTP server task. In a single threaded system, this function is called at periodic intervals by the KwikNet Task.

Once the TFTP server has been started, it enters the initialization state. The server calls *kntf_listen()* to create a TFTP server descriptor which can be used to detect requests from TFTP clients. The server descriptor is created to operate in non-blocking mode so that the server will not block in a single threaded system when it executes in the KwikNet domain.

The server then enters the accept state. While in the accept state, the server calls *kntf_accept()* to test for a transfer request from a TFTP client. Upon receipt of a request, a TFTP connection is established. The connection inherits the server's non-blocking mode of operation.

The server accepts the first request which it receives to write to file *data.bin* and enters the fetch state. Any other request to write to file *data.bin* or to any other file is rejected with a call to *kntf_abort()* to send an error indication back to the client and close the TFTP connection.

The server accepts the first request which it receives to read from file *data.bin* and enters the send state. If file *data.bin* has not yet been received by the server or if the client's request specifies a different file name, the request is rejected with a call to *kntf_abort()* to send an error indication back to the client and close the TFTP connection.

When the TFTP client is sending a file to the server, the server operates in the fetch state. The server calls *kntf_read()* to fetch file data from the client. Since the sample does not incorporate an actual file system, the file data is simply recorded in a fixed size local buffer maintained by the server. When the transfer is complete, the TFTP server calls *kntf_close()* to close the TFTP connection. The server then returns to the accept state to await another client request.

When the TFTP client is retrieving a file from the server, the server operates in the send state. The server calls *kntf_write()* to send file data to the client. Since the sample has no file system, the file data is simply extracted from the local buffer used previously by the server to record file *data.bin*. When the transfer is complete, the server repetitively calls *kntf_write()* with an end of file indication to ensure that untransmitted data is sent to the client. Finally, the TFTP server calls *kntf_close()* to sever the TFTP connection. The server then returns to the accept state to await another client request.

When the TFTP client has finished its final transfer, it forces the TFTP server task into the shut down state. In this state, the server generates a TFTP server status summary on the KwikNet logging device and calls *kntf_close()* to close the TFTP server session.

Shutdown

When the sample is finished, the TFTP client task initiates a controlled system shutdown. First, the TFTP client task initiates termination of the TFTP server task by directly forcing the server into its shut down state.

Once the TFTP server task has stopped, the client calls procedure *kn_exit()* to stop operation of the KwikNet TCP/IP Stack.

Finally, the client initiates a shutdown of the underlying operating system (if possible) and a return to the *main()* procedure.

KwikNet and TFTP Logging

The TFTP Sample Program uses the simple KwikNet message recording service to log text messages. The recorder saves the recorded text strings in a 30,000 byte memory buffer until either 500 strings have been recorded or the memory buffer capacity is reached.

The TFTP Sample Program directs messages to this recorder by calling the KwikNet log procedure *kn_dprintf()*. This procedure operates similarly to the C *printf()* function except that an extra integer parameter of value *0* must precede the format string. The TFTP client task uses this feature to record its progress, connection activity and the final shutdown message. The TFTP server task uses this feature to record connection activity and errors as they are detected.

KwikNet formats the message into a log buffer and passes the buffer to an application log function for printing. Log function *sam_record()* in the KwikNet Application OS Interface serves this purpose.

In a multitasking system the log buffer is delivered as part of an RTOS dependent message to a print task. The print task calls *kn_logmsg()* in the KwikNet message recording module to record the message and release the log buffer.

In a single threaded system, the log function *sam_record()* can usually call *kn_logmsg()* to record the message and release the log buffer. However, if the message is being generated while executing in the interrupt domain, the log buffer must be passed to the KwikNet Task to be logged. The sample programs provided with the KwikNet Porting Kit illustrate this process.

Since the recorded strings are just stored in memory, they are not readily visible. If a debugger is used to control execution of the TFTP Sample Program, the program can be stopped and the strings can be viewed in text form in a display window by viewing the array variable *kn_recordlist[]* in module *KNRECORD.C*.

Running the Sample Program

The KwikNet TFTP Sample Program is built as described in Chapter 1.7. The result is an executable load module suitable for testing with a debugger.

Since the KwikNet TFTP Sample Program has no visible output unless operated with a console terminal, its operation can only be confirmed using your debugger. Since the program has no hardware dependence, it can readily be used with a target processor simulator, if one is available.

KwikNet includes a number of debug features (see Chapter 1.9 of the KwikNet TCP/IP Stack User's Guide) which will assist you in running the TFTP Sample Program. With KwikNet's debug features enabled, you can place a breakpoint on procedure *kn_bphit()* to trap all errors detected by KwikNet. Of course, if you are using AMX, it is always wise to execute with a breakpoint on the AMX fatal exit procedure *cjksfatal* (*ajfatl* for AMX 86).

If you breakpoint at the end of the *main()* program, you can examine the messages recorded in memory. The messages are stored sequentially in a character array called *kn_records[]*. Variable *kn_recordlist[]* is an array of string pointers referencing the individual recorded messages. Most debuggers will allow you to dump the strings referenced in *kn_recordlist[]* in text form in a display window. The list of string pointers is terminated with a *NULL* string pointer.

If you are connected to the target processor by a serial link, do not be surprised if the debugger takes quite some time to access and display all of the strings referenced by *kn_recordlist[]*. You may be able to improve the response by limiting the display to the actual number of strings in the array as defined by variable *kn_recordindex*.

Once you are confident that the KwikNet TFTP Sample Program is operating properly, you may wish to breakpoint your way through the TFTP client and server, monitoring the recorded messages as you go.

1.7 Making the TFTP Sample Program

The sheer volume of documentation provided with the TFTP option may at first be daunting. However, constructing the KwikNet TFTP Sample Program is actually a fairly simple process made even simpler by the KwikNet Configuration Manager, a Windows® utility provided with KwikNet.

The TFTP Sample Program includes all of the components needed to build the sample application for a particular target processor. You can take these components and, with minor modifications, adapt them for your particular target processor and development environment.

> Note
>
> The KwikNet TFTP Sample Program for a particular target processor family is provided ready for use on one of the development boards used at KADAK for testing.
>
> The TFTP Sample Program **does not require a file system**.

TFTP Sample Program Directories

When KwikNet and its TFTP option are installed, the following subdirectories on which the sample program construction process depends are created within directory *KNTnnn*.

TCPIP	KwikNet header and source files, Ethernet Network Driver Ethernet and Serial Loopback Drivers
TFTP	TFTP protocol
CFGBLDW	KwikNet Configuration Builder; template files
ERR	Construction error summary
MAKE	KwikNet Library make directory
TOOLXXX	Toolset specific files
TOOLXXX\DRIVERS	KwikNet device drivers and board driver
TOOLXXX\LIB	Toolset specific KwikNet Library will be built here
TOOLXXX\SAM_MAKE	Sample program make directory
TOOLXXX\SAM_TFTP	KwikNet TFTP Sample Program directory
TOOLXXX\SAM_COMN	Common sample program source files

One or more toolset specific directories *TOOLXXX* will be present. There will be one such directory for each of the software development toolsets that KADAK supports. Each toolset vendor is identified by a unique two or three character mnemonic, *XXX*. The mnemonic *UU* identifies the toolset vendor used with the KwikNet Porting Kit.

TFTP Sample Program Files

To build the KwikNet TFTP Sample Program using make file *KNTFTSAM.MAK*, each of the following source files must be present in the indicated destination directory.

Source File	Destination Directory	File Purpose	
.	*CFGBLDW*	KwikNet Configuration Builder; template files	
	KwikNet source directories containing:		
KN_API.H	*TCPIP*	KwikNet Application Interface definitions	
KN_OSIF.H	*TCPIP*	KwikNet OS Interface definitions	
KN_TFTP.H	*TFTP*	KwikNet TFTP definitions	
	Toolset root directory containing:		
KN_OSIF.INC	*TOOLXXX*	OS Interface Make Specification	
KNZZZCC.INC	*TOOLXXX*	Tailoring File (for use with make utility)	
KNZZZCC.H	*TOOLXXX*	Compiler Configuration Header File	
	KwikNet TFTP Sample Program directory containing:		
KNTFTSAM.MAK	*TOOLXXX\SAM_TFTP*	TFTP Sample Program make file	
KNTFTSAM.C	*TOOLXXX\SAM_TFTP*	TFTP Sample Program	
KNZZZAPP.H	*TOOLXXX\SAM_TFTP*	TFTP Sample Program Application Header	
KNTFTLIB.UP	*TOOLXXX\SAM_TFTP*	Network Parameter File	
KNTFTSAM.LKS	*TOOLXXX\SAM_TFTP*	Link Specification File	(toolset dependent)
		Other toolset dependent files may be present.	
KNTFTSCF.UP	*TOOLXXX\SAM_TFTP*	User Parameter File	(for use with AMX)
KNTFTTCF.UP	*TOOLXXX\SAM_TFTP*	Target Parameter File	(for use with AMX)
	Common sample program source files:		
KNSAMOS.C	*TOOLXXX\SAM_COMN*	Application OS Interface	
KNSAMOS.H	*TOOLXXX\SAM_COMN*	Application OS Interface header file	
KNRECORD.C	*TOOLXXX\SAM_COMN*	Message recording services	
KNCONSOL.C	*TOOLXXX\SAM_COMN*	Console driver	
KNCONSOL.H	*TOOLXXX\SAM_COMN*	Console driver header	
		Console driver serial I/O support:	
KN8250S.C	*TOOLXXX\SAM_COMN*	INS8250 (NS16550) UART driver	
KN_BOARD.C	*TOOLXXX\DRIVERS*	Board driver for target hardware	

TFTP Sample Program Parameter File

The Network Parameter File *KNTFTLIB.UP* describes the KwikNet and TFTP options and features illustrated by the sample program. This file is used to construct the KwikNet Library for the TFTP Sample Program.

The Network Parameter File *KNTFTLIB.UP* also describes the network interfaces and the associated device drivers that the sample program needs to operate.

TFTP Sample Program KwikNet Library

Before you can construct the KwikNet TFTP Sample Program, you must first build the associated KwikNet Library.

Use the KwikNet Configuration Builder to edit the sample program Network Parameter File *KNTFTLIB.UP*. Use the Configuration Builder to generate the Network Library Make File *KNTFTLIB.MAK*.

Look for any KwikNet Library Header File *KN_LIB.H* in your toolset library directory *TOOLXXX\LIB*. If the file exists, delete it to ensure that the KwikNet Library is rebuilt to match the needs of the TFTP Sample Program.

Then copy files *KNTFTLIB.UP* and *KNTFTLIB.MAK* into the *MAKE* directory in the KwikNet installation directory *KNTnnn*. Use the Microsoft make utility and your C compiler and librarian to generate the KwikNet Library. Follow the guidelines presented in Chapter 3.2 of the KwikNet TCP/IP Stack User's Guide.

> Note
>
> The KwikNet Library must be built before the TFTP Sample Program can be made. If file *KN_LIB.H* exists in your toolset library directory *TOOLXXX\LIB*, delete it to force the make process to rebuild the KwikNet Library.

The TFTP Sample Program Make Process

Each KwikNet sample program must be constructed from within its own directory in the KwikNet toolset directory. Hence, the KwikNet TFTP Sample Program must be built in directory *TOOLXXX\SAM_TFTP*.

All of the compilers and librarians used at KADAK were tested on a Windows® workstation running Windows NT, 2000 and XP. However, you can build each KwikNet sample program using any recent version of Windows, provided that your software development tools operate on that platform.

To create the KwikNet TFTP Sample Program, proceed as follows. From the Windows Start menu, choose the MS-DOS Command Prompt from the Programs folder. Make the KwikNet toolset *TOOLXXX\SAM_TFTP* directory the current directory.

To use Microsoft's *NMAKE* utility, issue the following command.

```
NMAKE -fKNTFTSAM.MAK "TOOLSET=XXX" "TRKPATH=treckpath"
    "OSPATH=yourospath" "TPATH=toolpath"
```

The make symbol *TOOLSET* is defined to be the toolset mnemonic *XXX* used by KADAK to identify the software tools which you are using.

The symbol *TRKPATH* is defined to be the string *treckpath*, the full path (or the path relative to directory *TOOLXXX\SAM_TFTP*) to your Turbo Treck TCP/IP installation directory.

The symbol *OSPATH* is defined to be the string *yourospath*, the full path (or the path relative to directory *TOOLXXX\SAM_TFTP*) to the directory containing your RT/OS components (header files, libraries and/or object modules). When using AMX, string *yourospath* is the path to your AMX installation directory.

The symbol *TPATH* is defined to be the string *toolpath*, the full path to the directory in which your software development tools have been installed. For some toolsets, *TPATH* is not required. The symbol is only required if it is referenced in file *KNZZZCC.INC*.

The KwikNet TFTP Sample Program load module *KNTFTSAM.xxx* is created in toolset directory *TOOLXXX\SAM_TFTP*. The file extension of the load module will be dictated by the toolset you are using. The extension, such as *OMF*, *ABS*, *EXE*, *EXP* or *HEX*, will match the definition of macro *XEXT* in the tailoring file.

The final step is to use your debugger to load and execute the KwikNet TFTP Sample Program load module *KNTFTSAM.xxx*.

1.8 Adding TFTP to Your Application

Before you can add the TFTP protocol to your application, there are a number of prerequisites that your application must include. You must have a working KwikNet TCP/IP stack operating with your RT/OS. It is imperative that you start with a tested TCP/IP stack with functioning device drivers before you add TFTP. If these components are not operational, the KwikNet TFTP option cannot operate correctly.

KwikNet Library

Begin by deciding whether you need a TFTP client or server or both. Rarely are both required. Then decide which TFTP features must be supported. Review the TFTP property page described in Chapter 1.3.

Use the KwikNet Configuration Manager to edit your application's KwikNet Network Parameter File to include the TFTP protocol. Then rebuild your KwikNet Library. The library extension may be *.A* or *.LIB* or some other extension dictated by the toolset which you are using.

Memory Allocation

A TFTP client or server requires approximately 700 bytes of memory for each TFTP descriptor which it creates to handle a TFTP connection. A TFTP server also creates a TFTP server descriptor of similar size to manage requests from TFTP clients. These TFTP descriptors are allocated by TFTP service functions *kntf_open()*, *kntf_accept()* and *kntf_listen()*.

To meet these demands, you may have to edit your KwikNet Network Parameter File to increase the memory available for allocation.

TFTP Client and Server Tasks

You must provide one task for each TFTP client and server that you wish to incorporate into your application. Usually one TFTP client task or one TFTP server task is required. Rarely are both needed. Even more rarely are two or more clients or servers required.

If necessary, you can create a single TFTP client task which services multiple concurrent TFTP connections. A single TFTP server task can also service multiple concurrent TFTP client connections. In such cases, the TFTP client or server task must use non-blocking TFTP descriptors so that the task is not blocked servicing one TFTP peer to the detriment of all other peers.

In a multitasking system, you may have to increase the total number of tasks allowed by your RTOS in order to add the TFTP tasks.

A stack size of 4K to 8K bytes is considered adequate for most TFTP client or server tasks. The stack size can be trimmed after your TFTP tasks have been tested and actual stack usage observed using your debugger.

In a multitasking system, all TFTP tasks must be of lower execution priority than the KwikNet Task. If both TFTP server and client tasks exist, it is usual to make TFTP server tasks of higher priority than TFTP client tasks.

If you are incorporating a TFTP client or server, then you have the coding responsibility. You must create a TFTP task procedure which performs the TFTP operations required by your application. Only you can define such a procedure. All of the KwikNet TFTP services listed in Chapter 2.2 are at your disposal. You can use the TFTP client and server tasks in the TFTP Sample Program as a guideline for proper form.

Your TFTP client task must first open a TFTP connection. It can then read data from a file and send it to the TFTP server or fetch data from the TFTP server and write it to a file. When the transfer is finished, the client must close the TFTP connection.

Your TFTP server task must first start a TFTP server session. When a TFTP client request is received, the server must accept the request and establish an open TFTP connection or reject the request. It can then read data from a file and send it to the TFTP client or fetch data from the TFTP client and write it to a file. When the transfer is finished, the client must close the TFTP connection.

The TFTP client and server task C source modules must be compiled just like any other KwikNet application module. However, your compiler will also require access to TFTP header file *KN_TFTP.H* in the Treck installation directory, say *C:\TRECK\INCLUDE*. This header file is copied to the Treck directory from the KwikNet *TFTP* installation directory when the KwikNet Library is created. The compilation procedure is described in Chapter 3.4 of the KwikNet TCP/IP Stack User's Guide.

Reconstructing Your KwikNet Application

Since you are adding TFTP to an existing KwikNet application, there is little to be done.

To meet the memory demands of your TFTP client and server, you may have to edit your KwikNet Network Parameter File to increase the memory available for allocation. If you do so, you must then rebuild your KwikNet Library.

Your application link and/or locate specification file must include the KwikNet Library which you built with support for TFTP. The object modules for your TFTP client and server tasks and any support modules that they might require must also be included in your link specification together with your other application object modules.

With these changes in place, you can link and create an updated KwikNet application with TFTP support included.

AMX Considerations

When reconstructing a KwikNet application that uses the AMX Real-Time Multitasking Kernel, adapt the procedure just described to include the following considerations.

You may have to edit your AMX User Parameter File to increase the maximum number of tasks allowed in order to add TFTP client and server tasks.

TFTP client and server tasks can be predefined in your AMX User Parameter File or they can be created dynamically at run-time as is done in the KwikNet TFTP Sample Program. These are simple AMX trigger tasks without message queues.

A stack size of 4K to 8K bytes is considered adequate for use with most device drivers. It should also suffice even if you are using the AMX/FS File System. The stack size can be trimmed after your TFTP tasks have been tested and actual stack usage observed using your debugger.

The TFTP task priorities must be lower than that of the KwikNet Task. If both TFTP server and client tasks exist, it is usual to make TFTP server tasks of higher priority than TFTP client tasks. If you are using AMX 86 to access MS-DOS® file services, the PC Supervisor Task should be below all TFTP client and server tasks in priority.

If you edit your AMX User Parameter File, you must then rebuild and compile your AMX System Configuration Module. If you are using the AMX/FS File System, you should also rebuild and compile your AMX/FS File System Configuration Module.

No changes to your AMX Target Configuration Module are required to support TFTP unless your TFTP client task requires special device support that is not already part of your application.

Performance and Timing Considerations

A meaningful discussion of all of the issues which affect the performance of a TFTP server or client are beyond the scope of this document. Factors affecting the performance of the KwikNet TFTP client and server include the following:

processor speed
memory access speed and caching effects
file system performance and disk access times (if used by your client/server)
competing disk accesses for different users
network type (Ethernet, SLIP, PPP)
network device driver implementation (buffering, polling, DMA support, etc.)
IP packet fragmentation
network hops required for connection
operation of the remote (foreign) connected client or server
KwikNet TCP/IP Stack configuration (clock, memory availability, etc.)
KwikNet TFTP configuration (service/retry/timeout intervals)

Of all these factors, only the last two can be easily adjusted. Increasing the fundamental clock rate for the KwikNet TCP/IP Stack beyond 50Hz will have little effect and will adversely affect systems with slow processors or memory. Increasing the memory available for use by the TCP/IP stack will help if high speed Ethernet devices are in use and the processor is fast enough to keep up.

It is recommended that the TFTP service interval be set to match the TCP/IP stack clock frequency. Although faster TFTP service may improve throughput, it will also introduce further burden on the processor.

Best throughput can be achieved by using a TFTP connection which operates in non-blocking mode. Your TFTP client or server will then set the pace at which data can be transferred to or from the TFTP peer, within the constraints imposed by the actual network. When servicing a non-blocking TFTP connection, your client or server must periodically relinquish control of the processor or lower priority tasks (activities) will never be serviced.

If your TFTP client (server) services a TFTP connection in blocking mode, it will be forced to wait whenever a TFTP transaction must be delayed until a response is received from the TFTP peer. Your client (server) will be forced to delay for the full service interval, even if the TFTP response is received prior to the end of the delay. Throughput on a blocking TFTP connection can therefore be severely curtailed if the service interval is too long.

The TFTP specification advises the use of connection timeouts and recommends exponentially increasing the delay interval between retransmissions but avoids any mention of specific timeout values. The default KwikNet TFTP timeout interval is 60 seconds. The default initial and maximum retry intervals are 1 and 16 seconds respectively. Consequently, in the absence of an expected response from a TFTP peer, KwikNet will retransmit to the peer at intervals of 1, 2, 4, 8, 16 and 16 seconds before declaring the connection broken after the 60 second TFTP timeout interval expires.

2. KwikNet TFTP Services

2.1 Introduction to TFTP Services

The KwikNet TFTP option provides a full set of TFTP services for use by your TFTP client and server. These service procedures reside in the KwikNet Library which you must link with your application.

A description of these KwikNet TFTP service procedures is provided in Chapter 2.2. The descriptions are ordered alphabetically for easy reference.

Italics are used to distinguish programming examples. Procedure names and variable names that appear in narrative text are also italicized. Occasionally a lower case procedure name or variable name may appear capitalized if it occurs as the first word in a sentence.

Vertical ellipses are used in program examples to indicate that a portion of the program code is missing. Most frequently this will occur in examples where fragments of application dependent code are missing.

```
:
: /* Continue processing */
:
```

Capitals are used for all defined KwikNet file names, constants and error codes. All KwikNet procedure, structure and constant names can be readily identified according to the nomenclature introduced in Chapter 1.3 of the KwikNet TCP/IP Stack User's Guide.

KwikNet Procedure Descriptions

A consistent style has been adopted for the description of the KwikNet TFTP service procedures. The procedure name is presented at the extreme top right and left as in a dictionary. This method of presentation has been chosen to make it easy to find procedures since they are ordered alphabetically.

Purpose A one-line statement of purpose is always provided.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

This block is used to indicate which application procedures can call the KwikNet procedure. A filled in box indicates that the procedure is allowed to call the KwikNet procedure. In the above example, only tasks would be allowed to call the procedure.

For AMX users, this block is used to indicate which of your AMX application procedures can call the KwikNet procedure. You are reminded that the term ISP refers to the Interrupt Handler of a conforming ISP.

...more

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

For other multitasking systems, a task is any application task executing at a priority below that of the KwikNet Task. A timer procedure is a function executed by a task that is higher in priority than the KwikNet Task. An ISP is a KwikNet device driver interrupt handler called from an RTOS compatible interrupt service routine. The other procedures do not exist.

For a single threaded system, your App-Task (see glossary in Appendix A of the KwikNet TCP/IP Stack User's Guide) is the only task. An ISP is a KwikNet device driver interrupt handler called from an interrupt service routine. The other procedures do not exist.

Setup The prototype of the KwikNet procedure is shown.
The KwikNet header file in which the prototype is located is identified.
Include KwikNet header files *KN_LIB.H* and *KN_TFTP.H* for compilation.

File *KN_LIB.H* is the KwikNet include file that corresponds to the KwikNet Library that your application uses. This file is created for you by the KwikNet Configuration Manager when you create your KwikNet Library. File *KN_LIB.H* automatically includes the correct subset of the KwikNet header files for a particular target processor.

File *KN_TFTP.H* is the KwikNet include file which you must include if your application uses TFTP client or server services. This file is located in KwikNet installation directory *TFTP*. It is copied to the Treck installation directory, say *C:\TRECK\INCLUDE*, when you build the KwikNet Library.

Description Defines all input parameters to the procedure and expands upon the purpose or method if required.

Returns The outputs, if any, produced by the procedure are always defined. Most KwikNet procedures return an integer error status.

Restrictions If any restrictions on the use of the procedure exist, they are described.

Note Special notes, suggestions or warnings are offered where necessary.

See Also A cross-reference to other related KwikNet procedures is always provided if applicable.

2.2 TFTP Service Procedures

KwikNet provides a collection of service procedures for use by your TFTP client or server. These service procedures reside in the KwikNet Library which you must link with your application.

The following list summarizes these KwikNet TFTP service procedures.

TFTP Client Operations

knтf_open	Open a connection to a TFTP server

TFTP Server Operations

kntf_listen	Create an active TFTP server session
kntf_accept	Accept a connection request from a TFTP client

TFTP Common Operations

kntf_read	Read from a file on a TFTP connection
kntf_write	Write to a file on a TFTP connection
kntf_close	Close a TFTP connection or server session
kntf_abort	Abort (prematurely terminate) a TFTP connection
kntf_option	Read or modify a TFTP descriptor option
kntf_status	Generate a status log for a TFTP descriptor

Purpose **Abort (Prematurely Terminate) a TFTP Connection**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_abort(KN_TFD tfd, unsigned int errnum,
                                const char *errmsg);
```

Description *Tfd* is a TFTP descriptor identifying the TFTP connection to be terminated. The descriptor *tfd* must have been created with a call to *kntf_open()* or *kntf_accept()*.

Errnum is an error number that specifies the reason for the abnormal termination. The error number will be sent to the TFTP peer as the 16-bit error code field in a TFTP error packet. The TFTP specification identifies the following standard error numbers.

0	Transfer aborted (The error message (if any) describes the reason.)
1	File not found
2	Access violation
3	Disk full or allocation exceeded
4	Illegal TFTP operation
5	Unknown transfer ID
6	File already exists
7	No such user

Errmsg is a pointer to a descriptive text string intended for human interpretation. This string is sent to the TFTP peer as the error message field in the TFTP error packet. The string must be terminated with a *'\0'* character. The string length, including the *'\0'* terminator, is limited to the maximum data transfer size allowed in a single TFTP packet (512 bytes).

If parameter *errmsg* is *NULL*, a default string will be derived based upon the value of parameter *errnum*. If *errnum* is 0 to 7, the default string will be one of those listed above. Otherwise, an empty error message string terminated with a *'\0'* character will be used.

Returns If successful, a value of *0* is returned. The TFTP connection is severed and the TFTP descriptor is closed.

On failure, one of the following error codes is returned.

KN_ERTFD	The TFTP descriptor *tfd* is invalid.

See Also *kntf_close()*

Purpose **Accept a Connection Request from a TFTP Client**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_accept(KN_TFD tfd, KN_TFD *newtfdp,
            char *filenamebuf, int filenamesize, char *mode);
```

Description *Tfd* is a TFTP server descriptor identifying the TFTP server from which a file transfer request is to be accepted. The descriptor *tfd* must have been created with a call to *kntf_listen()*.

Newtfdp is a pointer to storage for a new TFTP descriptor to be created to service a transfer request from a TFTP client.

In most cases, the caller will be forced to wait for a connection request if none is pending at the time of the call. However, if server descriptor *tfd* is non-blocking, *kntf_accept()* will return immediately to the caller with error code *KN_ERTFWOULDBLOCK* if a connection request is not yet pending. Descriptor *tfd* remains open listening for connection requests.

If a request for connection from a TFTP client is pending at descriptor *tfd*, a new descriptor is created with the same properties as descriptor *tfd*. The new TFTP descriptor is returned to the caller at **newtfdp*. The new descriptor provides the local end point for the data connection to the TFTP client. The new descriptor inherits the blocking attributes of server descriptor *tfd*. Note that the new descriptor cannot be used to accept more connections.

Filenamebuf is a pointer to storage for the filename specified in the TFTP client's transfer request. The filename stored at **filenamebuf* will be an exact copy of the filename field from the TFTP client request, up to and including the terminating *'\0'* character.

If the filename string received from the TFTP client exceeds the available storage, it will be truncated to *filenamesize-1* characters and the error code *KN_ERTFTRUNC* will be returned to the caller.

If parameter *filenamebuf* is *NULL*, the filename field in the client's TFTP request will be ignored.

...more

Description ...continued

Filenamesize specifies the number of bytes of available storage referenced by parameter *filenamebuf*. The maximum length of a client's filename string is therefore *filenamesize-1*. Parameter *filenamesize* is ignored if *filenamebuf* is *NULL*.

Mode is a pointer to a minimum of 3 bytes of storage for a file access mode string. The *'\0'* terminated string is derived from parameters in the client's transfer request. The file access mode string will be one of the following lower case strings.

"r"	Client wishes to fetch a file in text mode.
"rb"	Client wishes to fetch a file in binary (octet) mode.
"w"	Client wishes to send a file in text mode.
"wb"	Client wishes to send a file in binary (octet) mode.

If your TFTP server uses a real file system, the file access mode string can be used in the server's call to open the local file for reading or writing. When reading from a file for the client, your TFTP server must call *kntf_write()* to send the file data to the client. When writing to a file for the client, your TFTP server must call *kntf_read()* to fetch the file data from the client.

If parameter *mode* is *NULL*, the file access mode information in the client's TFTP transfer request will be ignored.

...more

...continued

Returns If successful, a value of *0* is returned. A valid TFTP descriptor is stored at **newtfdp*. If *filenamebuf* is not *NULL*, the *'\0'* terminated filename (if any) specified by the TFTP client will be stored at **filenamebuf*. If *mode* is not *NULL*, the file access mode (if any) specified by the TFTP client will be stored as a string at **mode*.

On failure, one of the following error codes is returned.

KN_ERTFD	The TFTP descriptor *tfd* is invalid.
KN_ERPARAM	Parameter *newtfdp* is invalid (*NULL*).
KN_ERNOMEM	Memory is not available for a new TFTP descriptor.
KN_ERTFDTYPE	Descriptor *tfd* cannot handle connection requests. (*Tfd* was not created by *kntf_listen()*.)
KN_ERTFTRUNC	The client filename has been truncated to fit the available storage. In all other aspects, the operation is considered to have been successful.
KN_ERTFWOULDBLOCK	Descriptor *tfd* is non-blocking and there are no client transfer requests pending.
KN_ERTFUDP	A KwikNet UDP socket cannot be opened.
KN_ERTFNOPORT	A local UDP port number is not available.
KN_ERTFSTATE	Descriptor *tfd* reached an invalid internal state.

Note Descriptor **newtfdp* must be used by your TFTP server in subsequent calls to *kntf_write()* or *kntf_read()* to transfer data to or from the client to which the new descriptor is connected. Descriptor **newtfdp* must be closed with a call to *kntf_close()* when the transfer is complete.

The transfer can be terminated by your TFTP server prior to completion with a call to *kntf_abort()* using descriptor **newtfdp*. Your server can also call *kntf_abort()* to reject a client request if the filename or file access mode is unacceptable or if other conditions preclude the transfer.

Note If error *KN_ERTFSTATE* is detected, your server must call *kntf_close()* to close the server descriptor *tfd* and release all associated resources.

See Also *kntf_listen()*, *kntf_read()*, *kntf_write()*, *kntf_close()*, *kntf_abort()*

Purpose **Close a TFTP Connection or Server Session**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_close(KN_TFD tfd);
```

Description *Tfd* is a TFTP descriptor identifying the TFTP connection to be closed. *Tfd* can be an active descriptor created by a TFTP client with a call to *kntf_open()* or accepted by a TFTP server with a call to *kntf_accept()*.

A TFTP server can also call *kntf_close()* to close a server descriptor which it created with a call to *kntf_listen()*.

Returns The TFTP connection, if any, is severed and the TFTP descriptor is marked as closed.

If closed successfully, a value of *0* is returned.

On failure, one of the following error codes is returned.

KN_ERTFD	The TFTP descriptor *tfd* is invalid.
KN_ERTFEOF	The transfer has not yet reached end of file. If reading, there is more data to be read. If writing, end of file has not been written.

Note

When successfully finished sending file data, you **MUST** call *kntf_write()* with an end of file indication before closing the connection. If the transfer is operating in non-blocking mode, you must repetitively make end of file calls to *kntf_write()* until the call returns a result other than *KN_ERTFWOULDBLOCK*.

Note In most cases, the TFTP descriptor will linger briefly while the connection (if any) is being closed. If a *KN_ERTFEOF* error is generated, a TFTP error message will have been sent to the foreign peer indicating that the transfer was prematurely aborted.

Restriction The TFTP descriptor is no longer valid after it is closed.

See Also *kntf_open()*, *kntf_accept()*, *kntf_abort()*, *kntf_write()*

Purpose **Create an Active TFTP Server Session**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_listen(KN_TFD *tfdp, struct in_addr *laddrp,
                    int lport, int backlog, int nonblock);
```

Description *Tfdp* is a pointer to storage for a new TFTP server descriptor which your TFTP server can use to accept TFTP requests from clients. The TFTP server descriptor identifies a particular local TFTP server. A server descriptor cannot be used to send or receive data since it does not provide a connection to a TFTP client.

Descriptor **tfdp* can be used by a TFTP server in subsequent calls to *kntf_accept()* to accept connection requests from TFTP clients. The TFTP server must call *kntf_close()* to release descriptor **tfdp* when it is no longer needed.

Laddrp is a pointer to a structure containing the local IPv4 address, in net endian form, to be used by TFTP clients to connect to the TFTP server. If parameter *laddrp* is *NULL* or the IP address at **laddrp* is *INADDR_ANY*, clients will be able to connect to the TFTP server using the IP address of any network interface in the computer on which the TFTP server is operating. The BSD structure *in_addr* is defined as follows in Treck header file *TRSOCKET.H* located in the *TRECK\INCLUDE* directory:

```
struct in_addr {
  u_long s_addr;              /* IPv4 address (net endian)*/
  };
```

Lport identifies the local UDP port on which the TFTP server is to listen for TFTP client connection requests. If parameter *lport* is *0*, the well known TFTP server port number 69 will be assigned to the server.

...more

Description ...continued

Backlog is the maximum number of pending TFTP client connection requests which the server is permitted to queue. Requests are queued in the order received. *Backlog* must be greater than or equal to *1*.

If a request for connection is received while the server's TFTP request queue is full, the request is rejected and a TFTP error message is returned to the client.

When your TFTP server calls *kntf_accept()* to fetch the next available request from the queue, it will be given a new TFTP descriptor with a connection to the client making the request.

Nonblock is a flag which determines the mode in which the server descriptor will operate. If *nonblock* is non-zero, the descriptor will operate in non-blocking mode. If *nonblock* is zero, the descriptor will operate in blocking mode.

If your TFTP server calls *kntf_accept()* to accept a request from a blocking server descriptor, the server will be forced to wait until a client connection request becomes available.

If your TFTP server calls *kntf_accept()* to accept a request from a non-blocking server descriptor, the server will resume immediately with error indication *KN_ERTFWOULDBLOCK* if a client connection request is not already available.

Returns If successful, a value of *0* is returned. The TFTP server descriptor will be stored at **tfdp*.

On failure, one of the following error codes is returned.

KN_ERPARAM	Parameter *tfdp* is invalid (*NULL*) or parameter *backlog* is invalid (*<1*).
KN_ERNOMEM	Memory is not available for a new TFTP descriptor.
KN_ERTFUDP	A KwikNet UDP socket cannot be opened or bound to the specified IPv4 address and/or port.

Restriction The TFTP server descriptor must only be used by your TFTP server in calls to *kntf_accept()*, *kntf_option()* or *kntf_close()*. The TFTP server descriptor must not be closed until your TFTP server is no longer using the descriptor.

See Also *kntf_accept()*, *kntf_close()*

Purpose **Open a Connection to a TFTP Server**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_open(KN_TFD *tfdp, const char *filename,
              const char *mode, struct in_addr *faddrp,
              int fport, int nonblock);
```

Description *Tfdp* is a pointer to storage for a new TFTP descriptor. The new descriptor provides the data connection to the TFTP server.

Filename is a pointer to a text string defining the name of the file to be transferred to or from the TFTP server. The filename string, including the terminating *'\0'* character, is sent to the TFTP server in the request for connection. The filename syntax must be acceptable to the TFTP server. If the length of the filename string exceeds the available storage in a TFTP packet, the filename string will be truncated in the packet.

Mode is a pointer to a *'\0'* terminated text string defining the desired file access mode. The file access mode determines the direction and method of transfer. The file access mode is encoded and sent to the TFTP server in the request for connection. The file access mode must be one of the following lower case strings.

"r"	Fetch a file in text mode from a TFTP server.
"rb"	Fetch a file in binary (octet) mode from a TFTP server.
"w"	Send a file in text mode to a TFTP server.
"wb"	Send a file in binary (octet) mode to a TFTP server.

Faddrp is a pointer to a structure containing the IPv4 address, in net endian form, of the TFTP server with whom a TFTP connection is to be established. The BSD structure *in_addr* is defined as follows in Treck header file *TRSOCKET.H* located in the *TRECK\INCLUDE* directory:

```
struct in_addr {
  u_long s_addr;              /* IPv4 address (net endian)*/
  };
```

Fport is the UDP port number on which the TFTP server is awaiting requests. If parameter *fport* is *0*, the well known TFTP port number 69 will be used to establish the connection.

...more

Description ...continued

Nonblock is a flag which determines the mode in which the TFTP descriptor will operate. If *nonblock* is non-zero, the descriptor will operate in non-blocking mode. If *nonblock* is zero, the descriptor will operate in blocking mode.

If a TFTP function call references a blocking descriptor, the caller will be forced to wait until the requested operation completes, successfully or otherwise.

If a TFTP function call references a non-blocking descriptor, the caller will be given error indication *KN_ERTFWOULDBLOCK* if the caller would have to wait for the requested operation to be completed.

Returns If successful, a value of *0* is returned. A valid TFTP descriptor is stored at **tfdp* and a connection to the TFTP server is established.

If a non-blocking connection is being established (parameter *nonblock* is non-zero), a value of *KN_ERTFWOULDBLOCK* may be returned. In this case, a valid TFTP descriptor will be stored at **tfdp* but a connection to the TFTP server will not yet exist.

If the request to establish a TFTP connection is rejected by the TFTP server, a value of *KN_ERTFPROTO* will be returned to the caller. In this case, a valid TFTP descriptor will be stored at **tfdp* but a connection to the TFTP server will not exist. Call function *kntf_option()* to determine the reason for the TFTP protocol error and then call *kntf_close()* to relinquish the descriptor.

On failure, one of the following error codes is returned.

KN_ERPARAM	Parameter *tdfp*, *filename*, *mode* or *faddrp* is invalid (*NULL*) or file access mode **mode* is not supported or the foreign IPv4 address is 0.0.0.0 which is not allowed.
KN_ERNOMEM	Memory is not available for a new TFTP descriptor.
KN_ERTFTIMEOUT	Connection attempt timed out waiting for a response.
KN_ERTFUDP	A KwikNet UDP socket cannot be opened.
KN_ERTFNOPORT	A local UDP port number is not available.
KN_ERTFSTATE	An invalid internal state transition was detected.
KN_ERTFWOULDBLOCK	Parameter *nonblock* specifies non-blocking operation. The server connection request is in progress but has not yet been established.

...more

...continued

Note Descriptor **tfdp* must be used by your TFTP client in subsequent calls to *kntf_write()* or *kntf_read()* to transfer data to or from the TFTP server to which the new descriptor is connected. Descriptor **tfdp* must be closed with a call to *kntf_close()* when the transfer is complete.

The transfer can be terminated by your TFTP client prior to completion with a call to *kntf_abort()* using descriptor **tfdp*.

Note If you receive error code *KN_ERTFWOULDBLOCK* when opening a non-blocking connection, you can proceed with calls to *kntf_read()* or *kntf_write()* as though a connection exists. These functions will return error code *KN_ERTFWOULDBLOCK* until such time as the connection is established or an error is detected or the data transfer begins.

See Also *kntf_read()*, *kntf_write()*, *kntf_close()*, *kntf_option()*

Purpose **Read or Modify a TFTP Descriptor Option**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_option(KN_TFD tfd, int option, long value);
```

Description *Tfd* is a TFTP descriptor created by *kntf_open()*, *kntf_listen()* or *kntf_accept()*.

Option identifies the TFTP option to be read or modified.

Value is an option specific parameter.

The following options defined in header file *KN_TFTP.H* are supported:

KN_TFTP_OPT_SETMODE
If *value* is *0*, set the descriptor to blocking mode.
If *value* is not *0*, set the descriptor to non-blocking mode.

KN_TFTP_OPT_GETMODE
Get the descriptor blocking mode.
Value is a pointer to an *int* variable to be used as storage for the current blocking mode of the descriptor. Upon return, **value* is not *0* if the descriptor is non-blocking or 0 if the descriptor is blocking.

KN_TFTP_OPT_SETVAR
Set the user-defined private variable in the descriptor to *value*.
The private variable can be used by your application for any purpose.

KN_TFTP_OPT_GETVAR
Get the user-defined private variable from the descriptor.
Value is a pointer to a *long* variable. Upon return, **value* will contain the user-defined private variable from the descriptor.

KN_TFTP_OPT_GETLPORT
Get the local UDP port number for the TFTP connection.
Value is a pointer to an *unsigned short* variable.
Upon return, **value* will contain the local UDP port number.

KN_TFTP_OPT_GETFPORT
Get the foreign UDP port number for the TFTP connection.
Value is a pointer to an *unsigned short* variable.
Upon return, **value* will contain the peer's UDP port number.

...more

Description ...continued

KN_TFTP_OPT_GETFADDR
Get the foreign IPv4 address for the TFTP connection.
Value is a pointer to a structure of type *struct in_addr*.
Upon return, **value* will contain the TFTP peer's IPv4 address.

KN_TFTP_OPT_GETERRNUM
Get the TFTP error number most recently received from or sent to the connected TFTP peer. *Value* is a pointer to a *long* variable. Upon return, **value* will contain the error number merged with one of the following masks from header file *KN_TFTP.H* defining the error source:

KN_TFTP_ERCVD	*0x10000*	Error packet was received
KN_TFTP_ESENT	*0x20000*	Error packet was sent
KN_TFTP_ETIME	*0x40000*	Connection timeout
KN_TFTP_EMASK	*0x0FFFF*	Mask to isolate TFTP error number

The TFTP error number, as defined in the description of function *kntf_abort()*, can be isolated using mask *KN_TFTP_EMASK*.

KN_TFTP_OPT_GETERRMSG
Get the TFTP error string most recently received from the connected TFTP peer. *Value* is a pointer to a pointer variable of type *char **. Upon return, **value* will contain a pointer to the *'\0'* terminated error string. If no error string has been received, **value* will be *NULL*.

KN_TFTP_OPT_SETLINGER
If *value* is *0*, the descriptor will not linger once it is closed.
If *value* is not *0*, the descriptor will linger when closed after reading a file so that an acknowledgement can be retransmitted if the peer requires confirmation that its final transfer was successful.

KN_TFTP_OPT_SETOVREPLY
If *value* is *0*, the TFTP server listening on descriptor *tfd* will silently ignore client requests if its request queue is full. If *value* is not *0*, the server will reject such a client request by sending a TFTP error message to the client.

Returns If successful, a value of *0* is returned.

On failure, one of the following error codes is returned.

KN_ERTFD	The TFTP descriptor *tfd* is invalid.
KN_ERPARAM	Invalid option or parameter *value* is *NULL* when a pointer is required.

Purpose **Read from a File on a TFTP Connection**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_read(KN_TFD tfd, void *bufptr, int buflen);
```

Description *Tfd* is a TFTP descriptor created by *kntf_open()* or *kntf_accept()*. The descriptor identifies the local end of a TFTP connection to be used to acquire file data from a TFTP peer.

Bufptr is a pointer to a storage buffer for the file data to be received from the foreign TFTP peer.

Buflen is the buffer size, measured in bytes.

Returns If successful, the value returned is the number of bytes of file data stored at **bufptr*. The value *0* is returned if all file data has been transferred from the TFTP peer. In this case, the peer is considered to have reached the end of the file being transferred.

If *tfd* is a blocking descriptor, the caller will be forced to wait until at least *buflen* bytes of file data are available from the peer. At that time, *buflen* bytes of file data will be transferred to **bufptr* and the value *buflen* will be returned to the caller. If the peer has reached the logical end of the file being transferred, the value returned will be less than *buflen* (possibly *0*) and will indicate the number of bytes of file data stored at **bufptr* as a result of the final transfer.

If *tfd* is a non-blocking descriptor, all available file data from the peer will be transferred to **bufptr* up to a maximum of *buflen* bytes. The actual number of bytes stored at **bufptr* is returned to the caller. If no file data is available and the peer has not yet reached the end of the file being transferred, the error code *KN_ERTFWOULDBLOCK* will be returned to the caller. The value *0* is returned if no more data is available and the peer has reached the end of the file.

...more

Returns ...continued

On failure, one of the following error codes is returned.

KN_ERTFD	The TFTP descriptor *tfd* is invalid.
KN_ERPARAM	*Bufptr* is invalid (*NULL*) or *buflen* is not >0.
KN_ERTFDTYPE	Descriptor *tfd* was not opened for file reading.
KN_ERTFWOULDBLOCK	The descriptor is non-blocking and data is not yet available for transfer to the caller's buffer.
KN_ERTFTIMEOUT	Connection timed out waiting for data.
KN_ERTFNOCONN	Descriptor *tfd* has no TFTP connection.
KN_ERTFPROTO	Protocol error received or sent. (call function *kntf_option()* for detail.)
KN_ERTFSTATE	An invalid internal state transition was detected.

Note If error *KN_ERTFSTATE* is detected, you must call *kntf_close()* to close descriptor *tfd* and release all associated resources.

See Also *kntf_write()*, *kntf_option()*

Purpose **Generate a Status Log for a TFTP Descriptor**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_status(KN_TFD tfd);
```

Description *Tfd* is the TFTP descriptor created by *kntf_open()*, *kntf_listen()* or *kntf_accept()* for which a status log is to be generated.

Returns If successful, a value of *0* is returned and a TFTP status summary for descriptor *tfd* will be generated on the KwikNet logging device.

On failure, the following error code is returned.

KN_ERTFD The TFTP descriptor *tfd* is invalid.

Note If TFTP statistics logging is not enabled in the KwikNet Library, a TFTP status summary will not be generated, even if KwikNet logging is enabled.

If TFTP statistics logging is enabled in the KwikNet Library, the TFTP status summary will be logged on the KwikNet logging device as a sequence of messages of class *KN_PA_TFTP*.

Purpose **Write to a File on a TFTP Connection**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_TFTP.H*.

```
#include "KN_LIB.H"
#include "KN_TFTP.H"
int kntf_write(KN_TFD tfd, void *bufptr, int buflen);
```

Description *Tfd* is a TFTP descriptor created by *kntf_open()* or *kntf_accept()*. The descriptor identifies the local end of a TFTP connection to be used to send file data to a TFTP peer.

Bufptr is a pointer to a storage buffer providing the file data to be sent to the foreign TFTP peer. When the end of file is reached (*buflen* is *-1*), parameter *bufptr* is ignored.

Buflen is the buffer size, measured in bytes. If *bufptr* is *>0*, it indicates the number of bytes of data to be transferred. A *buflen* value of *0* is invalid. Set *buflen* to *-1* to indicate that the end of the file has been reached and no more data is available for transfer.

When an end of file request is serviced, all file data previously accepted for transmission is sent to the peer. If no untransmitted data remains, a zero length block of file data is sent to the peer. Either of these data transmissions acts as an end of file indication to the foreign peer.

Returns If a data write is successful, the value returned will be the number of bytes of file data accepted from **bufptr* for delivery to the TFTP peer. The value will be greater than zero.

If *tfd* is a blocking descriptor, the caller may be forced to wait until *buflen* bytes of file data have been accepted for delivery to the peer. At that time, the value *buflen* will be returned to the caller.

If *tfd* is a non-blocking descriptor, some of the available file data from **bufptr* up to a maximum of *buflen* bytes may be accepted for delivery to the peer. The actual number of bytes accepted from **bufptr* is returned to the caller. If no file data can be accepted for delivery to the peer, the error code *KN_ERTFWOULDBLOCK* will be returned to the caller.

If an end of file write is successful, the value *0* will be returned. If *tfd* is a non-blocking descriptor, the error code *KN_ERTFWOULDBLOCK* will be returned to the caller until the foreign peer acknowledges receipt of the final end of file indication, at which time the value *0* will be returned.

...more

Returns ...continued

On failure, one of the following error codes is returned.

KN_ERTFD	The TFTP descriptor *tfd* is invalid.
KN_ERPARAM	*Bufptr* value of *NULL* is invalid unless *buflen* is *-1*. *Buflen* is not *>0* or *-1*.
KN_ERTFDTYPE	Descriptor *tfd* was not opened for file writing.
KN_ERTFWOULDBLOCK	The descriptor is non-blocking and 1) data cannot be accepted for transfer until space is available or 2) end of file has not been acknowledged by the peer.
KN_ERTFTIMEOUT	Connection timed out waiting for acknowledgement.
KN_ERTFEOF	Cannot write data once end of file has been indicated.
KN_ERTFNOCONN	Descriptor *tfd* has no TFTP connection.
KN_ERTFPROTO	Protocol error received or sent. (call function *kntf_option()* for detail.)
KN_ERTFSTATE	An invalid internal state transition was detected.

Note

When successfully finished sending file data, you **MUST** call *kntf_write()* with an end of file indication before closing the connection. If the transfer is operating in non-blocking mode, you must repetitively make end of file calls to *kntf_write()* until the call returns a result other than *KN_ERTFWOULDBLOCK*.

Note If error *KN_ERTFSTATE* is detected, you must call *kntf_close()* to close descriptor *tfd* and release all associated resources.

See Also *kntf_read()*, *kntf_option()*