KwikNet®

TELNET Client / Server

User's Guide

Version 3

First Printing: **September 1, 1999**
Last Printing: **September 15, 2005**

Manual Order Number: PN303-9T

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 Telnet Client / Server User's Guide

Table of Contents

Page

Page

1. KwikNet Telnet Operation

1.1 Introduction

The TELNET protocol is a simple remote terminal protocol used for connecting a terminal on one machine to a process on another using TCP/IP based networks. The KwikNet Telnet Option implements this protocol on top of the KwikNet™ TCP/IP Stack, a compact, reliable, high performance TCP/IP stack, well suited for use in embedded networking applications.

The KwikNet Telnet Option is best used with a real-time operating system (RTOS) such as KADAK's AMX™ Real-Time Multitasking Kernel. However, the KwikNet Telnet 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 Telnet needs by using the KwikNet Configuration Builder, a Windows® utility which makes configuring KwikNet a snap. Your KwikNet stack will only include the Telnet features required by your application.

This manual makes no attempt to describe the TELNET protocol, what it is or how it operates. It is assumed that you have a working knowledge of the TELNET 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 Telnet. 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 Telnet Option.

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

1.2 General Operation

The TELNET protocol is a simple remote terminal protocol used for connecting a terminal on one machine to a process on another over TCP/IP based networks such as the Internet. TELNET is formally defined by the IETF documents RFC-854 and RFC-855. Document RFC-1700 lists currently defined TELNET options and provides a reference to the RFC which describes each option. The KwikNet Telnet Option is compliant with those specifications. The RFCs should be consulted for any detailed questions concerning the TELNET protocol. The KwikNet Telnet Option implements the subset of Telnet features typically required for use in embedded applications.

TELNET is a client-server protocol. One machine, the client, initiates a conversation by contacting another machine, the server. The server must be operating before the client initiates its requests. The client and server can then use the TELNET protocol to negotiate how each should operate. Generally, a client communicates with one server at a time while most servers are designed to work concurrently with multiple clients.

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

TELNET Connection

When a Telnet client contacts a Telnet server, a TCP connection is established between the two machines. The server does a passive open by listening on a TCP socket which is bound to the well known Telnet port number 23 for requests from potential clients. The client can then connect its own TCP socket to the server. This connection is referred to as the Telnet connection. The connection persists until broken by either the client or the server. The Telnet connection is used by the client and server to send commands, negotiate terminal options and transfer data (characters) to or from the terminal.

TELNET Commands

A Telnet command is a character string which begins with the Telnet *interpret as command* (IAC) character followed by one or more command characters. The command character names and values are listed in Figure 1.2-1. Those flagged with * are handled automatically by KwikNet. Note that the EL and EC commands provide network equivalents of your line editing characters, not line editing services. All commands flagged with ■ or □ must be handled by your client or server application using services provided by the KwikNet Telnet interface.

It is recommended that the commands flagged with *!* be followed immediately with a DM data mark command. Upon receipt of these commands, you should usually respond with a DM synch command. It is recommended that the DM data mark and synch commands be sent urgently using the TCP out-of-band signaling services provided by the KwikNet Telnet interface.

Telnet commands are defined in Telnet header file *KN_TELN.H* as symbols of the form *KN_TELCMD_xxxx* where *xxxx* is one of the command names shown in Figure 1.2-1.

Name	Value		Command as Intended by Sender
IAC	255	*	Interpret next character as a command
DONT	254	*	Demand that peer not handle an option
			Reject peer WILL request so peer will not handle an option
			Accept peer WONT indication that it has stopped handling an option
DO	253	*	Demand that peer handle an option
			Accept peer WILL request so peer can start handling an option
WONT	252	*	Inform peer that an option can no longer be supported
			Reject peer DO request to handle an option
			Accept peer DONT request and stop handling an option
WILL	251	*	Inform peer that an option can be supported
			Accept peer DO request and start handling an option
SB	250	□	What follows is an option subnegotiation command
GA	249	■	Give peer a go-ahead signal allowing peer to resume sending
EL	248	*	Ask peer to erase the current line of data which it has received
EC	247	*	Ask peer to erase the previous data character which it received
AYT	246	!■	Send peer an "are you there" signal
AO	245	!■	Send peer a signal to "abort output"
IP	244	!■	Send peer a signal to "interrupt process"
BRK	243	■	Send peer a "break" signal
DM	242	>■	Insert data mark at point at which an urgent signal was sent to peer
		>■	Send peer a "synch" signal
NOP	241	■	Ask peer to do "no operation"
SE	240	□	Identify the end of an option subnegotiation command
EOR	239	■	Mark the end of a record in the data stream
ABORT	238	■	Abort Line mode: RFC-1184
SUSP	237	■	Suspend Line mode: RFC-1184
EOF	236	■	End of file Line mode: RFC-1184
0 to	235	■	Undefined commands

Note
- * Telnet command is handled by KwikNet without assistance.
- ■ Telnet command is handled by application using KwikNet services.
- □ Telnet option subnegotiation is handled by application.
- ! Telnet command should be followed by an urgent data mark (DM).
- > Telnet command should be sent with TCP urgent notification.

Figure 1.2-1 TELNET Commands

TELNET Options

Telnet options control the manner in which the client and server agree to operate. The Telnet DO, DONT, WILL and WONT commands are used to express the willingness of the client and server to negotiate a particular option. The negotiation command consists of three characters: IAC, one of the four negotiation command characters and an <option> identification character.

Some options require the client and server to exchange option parameters using a Telnet option subnegotiation command. The option subnegotiation is only allowed after both ends have agreed to support the option. The option subnegotiation command is a character string consisting of the IAC, SB and <option> characters followed by zero or more characters which form the option parameters. The command is terminated with the IAC and SE character pair.

Your application client and server must handle the option negotiation using services provided by the KwikNet Telnet interface. KwikNet handles the low level negotiation in response to requests from your application or from a Telnet peer. Whenever an option negotiation completes, your application is informed of the fact. Subnegotiations, if required, are the responsibility of your application client or server.

KwikNet supports up to 64 Telnet options. By default, the KwikNet Telnet Option handles only the first four basic Telnet options listed in Figure 1.2-2. However, you can extend the number of supported options to the full 64 identified in Figure 1.2-2. To do so, edit your KwikNet Network Parameter File and check the box labeled "Enable Telnet option subnegotiation" on the Telnet property page.

The options currently specified by RFC-1700 are defined in KwikNet header file *KN_TELN.H* as symbols of the form *KN_TELOPT_xxxx*. Figure 1.2-2 lists the specific Telnet options for which KwikNet provides the necessary support. All other options can be dynamically configured by you so that KwikNet will allow the option to be negotiated by your application.

When a Telnet connection is established, KwikNet assumes that none of the Telnet options are operational at either end of the connection. That is the default state for any Network Virtual Terminal (NVT) at the time of its connection. KwikNet will then initiate negotiations with the connected peer to suppress use of the go-ahead (GA) command.

Your application can subsequently initiate the negotiation of alternate options if necessary. If a negotiated option requires subnegotiation, your application must assume responsibility for sending and/or interpreting the option parameters.

Option	Value	Default	Purpose
Transmit binary	0	no	Transmit data as 8-bit binary characters
Echo	1	no	Echo data characters as they are received
Reconnect	2	no	Reserved for use by application per RFC
Suppress GA	3	yes	Suppress sending of GA "go-ahead" command
See RFC-1700	4 to 40	no	Reserved for use by application per RFC
Undefined	41 to 63	no	Reserved for use by application

Figure 1.2-2 TELNET Options

The Network Virtual Terminal

The TELNET protocol is used to interconnect a client terminal and a server process across a network. The TELNET protocol is symmetrical, allowing either end of the connection to look like a Network Virtual Terminal (NVT), a simplified terminal with only those characteristics specified by the TELNET protocol. At each end of the connection, the application client or server must act as though it is an NVT. It is the KwikNet Telnet interface which makes this possible.

The Network Virtual Terminal operates using the 7-bit ASCII character set. However, to improve client-server compatibility, the TELNET protocol uses the two character sequences shown in Figure 1.2-3 to encode a minimal set of line control characters.

Telnet	**Line Control**	**KwikNet Default**
CR LF	End-of-line indication	CR LF (*0x0D*, *0x0A*)
CR NUL	Return character	CR (*0x0D*)
IAC EC	Erase character	BS (*0x08*)
IAC EL	Erase line	ESC (*0x1B*)

Figure 1.2-3 NVT Line Control Characters

The KwikNet Telnet interface maps the Telnet line control sequences to and from your application's character set. The default mappings used by KwikNet are shown in Figure 1.2-3. However, your application can alter the mapping to suit the needs of a particular client, server or Telnet session. The end-of-line indication can be mapped to either CR or LF. The character-erase and line-erase control codes can be mapped to any single 7-bit ASCII character.

Your application can also force KwikNet to treat the CR NUL encoding as an alternate end-of-line indication. When this alternate translation is in effect, KwikNet will always send the CR NUL sequence as the Telnet representation of your end-of-line character sequence. On receipt, KwikNet will translate both CR LF and CR NUL to your end-of-line character sequence.

KwikNet also provides a special end-of-line filtering service to allow your application to cope with connections to clients or servers which behave oddly. Once the end-of-line translation has been performed, KwikNet will strip orphan CR and/or LF characters from the received data stream. It is important to realize that these characters will not be stripped if they are part of a valid end-of-line sequence. It is your application which determines which, if any, of the orphan CR and LF characters are to be stripped.

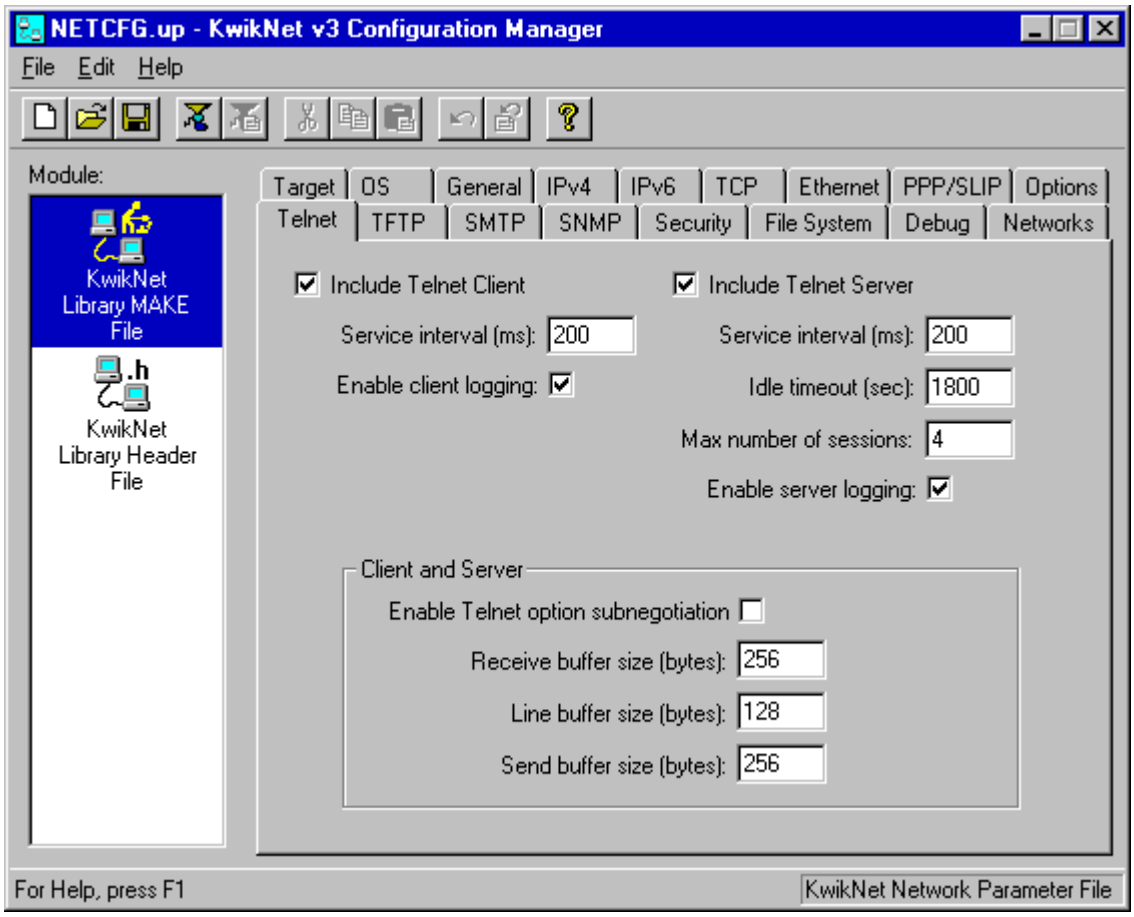
The TELNET protocol dictates that an NVT operates using 7-bit ASCII character encoding. The KwikNet Telnet interface relaxes the specification, allowing you to use 8-bit character encoding. Your application can send and receive all 256 8-bit characters. KwikNet will automatically convert any occurrence of the IAC character (*0xFF*) to the IAC IAC pair as required by the TELNET protocol. This feature is not to be confused with the Telnet 8-bit binary option which two NVT peers can negotiate.

Your application must call KwikNet service procedure *kntn_ioctl()* to adjust these NVT character mappings and operating characteristics.

1.3 KwikNet Telnet Configuration

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

Note that the TCP protocol is a prerequisite for the TELNET protocol. You must include TCP in your KwikNet Network Parameter File in order to use the TELNET protocol.



Telnet Parameters (continued)

Include Telnet Client

Check this box if your application will include a Telnet client which connects to a Telnet server. Otherwise, leave this box unchecked.

Client Service Interval

Specify the interval in milliseconds at which a Telnet client should service its Telnet connection. This interval determines the period at which the client connection to the server will be serviced by KwikNet whenever the client is waiting for data.

Enable Client Logging

Check this box if you wish your Telnet client to log events as they occur during a Telnet session. All logged messages will be directed to the Telnet client's callback function, if one has been provided. Otherwise, leave this box unchecked. Leaving this box unchecked reduces the code size by eliminating all message generation logic from the client service procedures in the KwikNet Library.

Include Telnet Server

Check this box if your application will include a Telnet server to provide network access to processes present in your target system. Otherwise, leave this box unchecked.

Server Service Interval

Specify the interval in milliseconds at which your Telnet server should service all of its Telnet connections.

Idle Timeout

Specify the interval in seconds which your Telnet server should allow before declaring that its session with a Telnet client is at an end because the client has had no interaction with the server during this interval. The Telnet server will break its connection to terminate the session.

Maximum Number of Sessions

This parameter defines the maximum number of Telnet clients which the Telnet server will support at any one time. Once this many sessions are active, the Telnet server will reject further requests for service until one or more of the active sessions terminates.

Telnet Parameters (continued)

Enable Server Logging

Check this box if you wish your Telnet server to log events as they occur during a Telnet session. All logged messages will be directed to the Telnet server's callback function. Otherwise, leave this box unchecked. Leaving this box unchecked reduces the code size by eliminating all message generation logic from the Telnet server procedures in the KwikNet Library.

Enable Telnet Option Subnegotiation

Check this box if your application client or server must be able negotiate all possible Telnet options, including those which require option subnegotiation. Leaving this box unchecked reduces the code and data size by eliminating support for all but the following mandatory Telnet options: binary, echo and suppress-GA.

Receive Buffer Size

This parameter defines the size (in bytes) of the buffer to be allocated for the reception of Telnet commands and application data by the client or server. The receive buffer size must match or exceed the longest Telnet option subnegotiation string which the Telnet client or Telnet server must accept. A value of 256 is typical.

Line Buffer Size

This parameter defines the size (in bytes) of the data buffer used to buffer received data (not Telnet commands) for presentation to your client or server application. The line buffer size establishes the maximum number of characters which KwikNet will ever present to your application in one data transfer. A value of 128 is typical.

Send Buffer Size

This parameter defines the size (in bytes) of the data buffer to be allocated for the buffering of Telnet commands for transmission to a Telnet peer. All command characters destined to a Telnet peer are held in the send buffer until they can be sent via the Telnet TCP socket connection.

The send buffer must be large enough to hold the longest Telnet option subnegotiation command which your Telnet application client or server will ever send. A value of 256 is typical.

1.4 Telnet Client Task

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

Your Telnet client task must call *kntc_create()* to create a Telnet session. The client inherits a Network Virtual Terminal with all of the characteristics described in Chapter 1.2. If necessary, the client task can alter some of its session parameters using service procedure *kntn_ioctl()*. For example, you may wish to revise the default client identification name "*TELC*" assigned by KwikNet to every Telnet client.

Once the client session is ready, using KwikNet client services is much like using a file system interface to interact with a terminal. The client can call *kntc_open()* to establish a connection with a particular Telnet server. The IPv4 address of the Telnet server must be provided by your client task.

Once the connection with a Telnet server has been established, the client task can use the KwikNet Telnet client-specific procedures described in Chapter 2 to converse with the server and the processes which it controls. The client task must use procedure *kntc_receive()* to fetch data received from the server and *kntn_send()* to send data to the server. Procedure *kntn_sendcmd()* can be used to send simple Telnet commands.

Procedure *kntn_ioctl()* can be used to sense the state of a Telnet option, reconfigure a Telnet option or initiate a Telnet option negotiation. If any option other than those listed in Figure 1.2-2 is negotiated, the client task must provide a callback function to handle the notification which occurs when the negotiation terminates. If option subnegotiation is required, the callback function will be expected to handle the option subnegotiation commands received from the server. The client task must use procedure *kntn_sendraw()* to send properly formatted option subnegotiation commands, if required.

Most operations performed for a KwikNet Telnet client task go to completion or until an error condition is encountered. As long as the client task makes calls to KwikNet service procedures *kntn_send()*, *kntn_sendcmd()* or *kntn_sendraw()* to send data or commands, the client session will be serviced. The client task is never blocked by KwikNet while data is being sent. If the client task calls *kntc_receive()* to read data and none is available, the client task will be blocked by KwikNet. However, the client session will continue to be serviced at the interval defined in your KwikNet Network Parameter File (see Chapter 1.3) until data arrives or an error condition is detected.

If necessary, service procedure *kntc_check()* can be called periodically to force KwikNet to service the client's Telnet connection. This procedure can also be used to determine if data is available for reading or to detect if the client's connection to the server has been broken.

When the connection with a particular Telnet server is no longer required, the Telnet client task closes the connection with a call to procedure *kntc_close()*.

When the Telnet client is finished its final Telnet connection, the Telnet session can be terminated with a call to *kntn_delete()*. No further Telnet transactions can be initiated by the client without first calling *kntc_create()* to establish a new client session.

The Telnet Sample Program provided with the KwikNet Telnet Option illustrates an interactive Telnet client task. The client task uses the KwikNet console driver (see Chapter 1.8 of the KwikNet TCP/IP Stack User's Guide) to implement a command line interface with a user at a simple, interactive console device.

Multitasking Client Operation

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

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

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

Single Threaded Client Operation

When used with a single threaded operating system, the Telnet 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 Telnet client task.

While executing as a Telnet 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 Telnet client initiates any Telnet transaction, KwikNet ensures that the TCP/IP stack continues to operate until the Telnet transaction is complete.

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

1.5 Telnet Server Task

The KwikNet Telnet Option includes a set of services which can be used to implement a Telnet server. Each Telnet server is an application program which makes use of these services to communicate with one or more Telnet clients. The collection of application procedures which makes up such a program is called a Telnet server task.

Your Telnet server task must call *knts_create()* to create a KwikNet Telnet server. The server task must provide a callback function with which the KwikNet Telnet server can interact to service Telnet clients. Once the server is started, it is this callback function which provides the interface to your application. The server callback function is described in Chapter 1.6.

The server inherits a Network Virtual Terminal with all of the characteristics described in Chapter 1.2. If necessary, the server task can alter some of its parameters using service procedure *knts_ioctl()*. For example, you may wish to revise the default server identification name "*TELS*" assigned by KwikNet to every Telnet server.

By default, the Telnet server will service all Telnet requests directed to any of the IP addresses assigned to the server's network node. If preferred, your server task can use procedure *knts_ioctl()* to provide a specific IPv4 address which clients must use to connect to your Telnet server.

The Telnet server accepts requests directed to the well known Telnet port number 23. However, your server task can use procedure *knts_ioctl()* to alter its configuration to service requests on any port number which you choose. Of course, only clients which know your alternate port number will be able to connect to your server.

When ready to begin operation, the server task simply calls KwikNet procedure *knts_start()*. Thereafter, the KwikNet Telnet server interacts with the server callback function. The callback function receives notification whenever a client connection is made or broken. Telnet commands and options and data from the client are passed to your application through the callback function. In the absence of any client activity, the callback function is called periodically (at the server's service interval) to allow your server to send data or Telnet commands to the client.

A Telnet server callback function can use the KwikNet Telnet server-specific procedures described in Chapter 2 to converse with its clients. The callback function will automatically receive data from the client. The callback function can call procedure *knts_send()* to send data to the client. Procedure *knts_sendcmd()* can be used to send simple Telnet commands.

The server callback function can call procedure *knts_ioctl()* to sense the state of a Telnet option, reconfigure a Telnet option or initiate a Telnet option negotiation. If any option other than those listed in Figure 1.2-2 is negotiated, the callback function must be prepared to handle the notification which occurs when the negotiation terminates. If option subnegotiation is required, the callback function will also be expected to handle the option subnegotiation commands received from the client. The callback function must use procedure *knts_sendraw()* to send properly formatted option subnegotiation commands, if required.

The Telnet server task will operate until some unrecoverable error condition is detected or until your application calls KwikNet service procedure *knts_stop()* requesting the server to stop. Once the KwikNet Telnet server stops, your Telnet server task can delete the server with a call to *kntn_delete()*.

The Telnet Sample Program provided with the KwikNet Telnet Option includes a working example of a Telnet server.

Telnet Server Identification

A Telnet server must function as a unique, identifiable entity on the network. To achieve this feature, KwikNet allows you to customize the external look of your Telnet server by allowing each server to have its own name.

When a KwikNet Telnet server is created, it is given the name "*TELS*". The server name is a string used in internal KwikNet error logging messages to uniquely identify the particular Telnet server.

Your server task can use service procedure *kntn_ioctl()* to provide an alternate Telnet server name which meets the needs of your application.

Multitasking Server Operation

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

A Telnet server task is created and started just like any other application task. When ready to begin operation, the server task simply calls KwikNet procedures *knts_create()* and *knts_start()* to create a server and begin service. There is no return from procedure *knts_start()* until the server is forced to stop.

The Telnet server task will operate until some unrecoverable error condition is detected or until some other application task calls KwikNet service procedure *knts_stop()* requesting the server to stop. The Telnet server task will abort all of its active Telnet sessions and resume execution following the initial call to *knts_start()*. Your Telnet server task can then delete the server with a call to *kntn_delete()*.

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

Single Threaded Server Operation

When used with a single threaded operating system, the Telnet server operates in the KwikNet domain in the context of the KwikNet Task as described in Chapter 1.2 of the KwikNet TCP/IP Stack User's Guide.

Your App-Task must call KwikNet procedure *knts_create()* to create a Telnet server. When your App-Task calls KwikNet procedure *knts_start()* to start the server, the server is added to the KwikNet server queue. Such a server is referred to as a Telnet server task. Although more than one Telnet server task is allowed, rarely is there such a need.

Once the Telnet server is operational, your App-Task must regularly call KwikNet procedure *kn_yield()* to let the Telnet server operate.

The Telnet server task will operate until some unrecoverable error condition is detected or until your App-Task calls KwikNet service procedure *knts_stop()* requesting the server to stop. The Telnet server task will abort all of its active Telnet sessions and remove itself from the KwikNet server queue. Your App-Task can then delete the server with a call to *kntn_delete()*.

1.6 Server Callback Function

Your interface to a KwikNet Telnet server is the server callback function. The server callback function is the application procedure which you must identify when you call procedure *knts_create()* to create the Telnet server. It is this callback function which provides access by any Telnet client to the services provided by your application. The server callback function is called by the KwikNet Telnet server to interpret and handle all such client requests for service.

The callback function is prototyped as follows. The callback function prototype is defined using a *typedef* declaration named *KN_TELCALLBACK*. The prototype declaration and definitions for callback parameters are provided in KwikNet header file *KN_TELN.H*.

```
void ts_callback(KN_TELND telnd, int opcode,
                 char *p1, int p2, void *param)
```

Parameter ***telnd*** is a Telnet descriptor allocated by KwikNet to identify a particular Telnet client session. This parameter must be used by the callback function to identify the client session in any calls which it makes to KwikNet's Telnet service procedures.

Parameter ***opcode*** is an integer value which identifies the purpose of the call to the server callback function. The following callback codes are defined in Telnet header file *KN_TELN.H*. The interpretation of parameters *p1* and *p2* is dictated by the value of parameter *opcode*.

KN_TELCBC_LOG	Log an error message
KN_TELCBC_CONN	New client connection established
KN_TELCBC_DROP	Client disconnected and session terminated
KN_TELCBC_IDLE	Telnet connection is idle
KN_TELCBC_DATA	Data characters received from client
KN_TELCBC_CMD	Telnet command received from client
KN_TELCBC_OPTION	Telnet option negotiation signal
KN_TELCBC_OPTSUB	Telnet option subnegotiation command received from client

Parameter ***param*** is the application callback parameter. The interpretation of this parameter is up to you. The value received by the callback function is either the server's default callback parameter value or the current callback parameter value.

Whenever a new client session is created (*opcode* is *KN_TELCBC_CONN*), the session inherits the server's **default callback parameter value**. Usually the default value is the original callback parameter value specified to procedure *knts_create()* when the Telnet server was created. However, if your application called procedure *kntn_ioctl()* to alter the callback parameter value prior to calling procedure *knts_start()* to start the server, then the default value will be the parameter value in effect at the time the server was started.

Once a client session is underway, the callback parameter value inherited from the server becomes the **current callback parameter value**. Hence, after the initial call establishing a new client session, the callback function will always receive the current value. Of course, your callback function can always use procedure *kntn_ioctl()* to alter the current callback parameter value for the client session specified by parameter *telnd*.

Callback Function Execution

The server callback function executes in the context of the KwikNet Telnet server task which called it. The callback function must service the request as quickly as possible. Service of all other Telnet activity by the server task will be delayed until the callback function finishes execution.

In a multitasking system, responsibility for time consuming operations such as data transfers to or from files should be passed to other lower priority tasks, if possible. Timed delays and transmission of large blocks of data should be avoided or handled via the idle callback.

The server callback function can call any of the KwikNet Telnet server-specific procedures described in Chapter 2 as long as it adheres to the documented restrictions, if any.

Client Connections

Whenever the Telnet server accepts a new TCP socket connection from a Telnet client, it calls the callback function with callback code *KN_TELCBC_CONN*. Parameters *p1* and *p2* are unused. Parameter *param* will have the server's default callback parameter value.

When a new connection is established, you may wish to use procedure *kntn_ioctl()* to alter the value of the callback parameter for the particular client session. For example, the server provided with the Telnet Sample Program uses this strategy to assign a unique client management block to each new client session for subsequent use throughout the session. The pointer to the client management block is installed as the current callback parameter value for the client's session.

The client session will inherit the server's default NVT character scheme, the characteristics of which were in effect when the Telnet server was started. The callback function can use procedure *kntn_ioctl()* to establish an alternate local character scheme for the particular client session. For example, it can alter the Telnet line editing characters or the local end-of-line representation.

If a client breaks its TCP socket connection, the Telnet server will call the callback function with callback code *KN_TELCBC_DROP*. The Telnet server will also make this callback if it automatically disconnects the client for any reason. Note that this callback will also occur following any disconnect initiated by the callback function itself with a call to procedure *kntn_ioctl()*. The callback function must release the resources, if any, which it has been using to service the client. Upon return to the Telnet server, the client session will be terminated. Parameters *p1* and *p2* are unused. Parameter *param* will have the current callback parameter value.

Idle Notification

If the Telnet server has no data, command or option information to present to your application, it calls the callback function at each service interval with callback code *KN_TELCBC_IDLE*. This call gives your server process an opportunity to initiate or resume an option negotiation sequence or the transmission of data. Parameters *p1* and *p2* are unused. Parameter *param* will have the current callback parameter value.

Data Received from Client

Whenever the Telnet server receives one or more data characters from a connected client, it calls the callback function with callback code *KN_TELCBC_DATA* to present the data for interpretation by your application. Parameter *p1* is a pointer to the data characters in the client session's line buffer. Parameter *p2* specifies the number of data characters stored at **p1*. *P2* will always be greater than *0* and will never be greater than the line buffer size which you declared in your KwikNet Network Parameter File. When operating in text mode (Telnet binary option is disabled locally), count *p2* will include all characters which have been received, up to and including the first end-of-line character string, if any. Parameter *param* will have the current callback parameter value.

Unless you have altered the server's or session's Telnet character representations, all received characters will be 7-bit ASCII characters and Telnet line control characters will have been converted to the defaults specified in Figure 1.2-3.

The server callback function must accept all of the data presented to it or lose it. If the data must be preserved, the callback function must copy it from the session's line buffer.

Telnet Commands from Client

Whenever the Telnet server receives a Telnet command which it cannot handle, it calls the callback function with callback code *KN_TELCBC_CMD* to allow your application to service the command. These are the commands listed in Figure 1.2-1 and flagged with the ▪ character.

Parameter *p2* is the Telnet command identifier, an integer value less than 255. Telnet command identifiers are defined in Telnet header file *KN_TELN.H* as symbols of the form *KN_TELCMD_xxxx* where *xxxx* is one of the command names listed in Figure 1.2-1. Parameter *p1* is unused. Parameter *param* will have the current callback parameter value.

Commands such as "interrupt process" (IP), "abort output" (AO) and "are you there" (AYT) are considered urgent and are usually received as out-of-band TCP requests. If the callback function receives one of these commands, it must assume that all pending data from the client has been discarded up to the point in the data stream at which the client sent the urgent command.

The TELNET protocol specifies that your application may have to send a synch command (DM) following your service of some Telnet commands. It recommends you send a synch command after handling the IP command. It demands that you do so after handling the AO command. It leaves the decision to your handling of the AYT command. No other commands require the synch response. It is recommended that the DM synch command be sent urgently using the TCP out-of-band signaling services provided by the KwikNet Telnet interface.

Telnet Option Notifications

Whenever the Telnet server detects a significant option negotiation event, it calls the callback function with callback code *KN_TELCBC_OPTION* to notify your application. Such a call is made whenever the Telnet server detects that a locally initiated negotiable option has been accepted or rejected by the client. Similarly, such a call is made whenever the Telnet server accepts or rejects a client request to negotiate an option. Parameter *param* will have the current callback parameter value. Parameter *p1* is unused.

Parameter *p2* identifies the option and the nature of the negotiation event being reported. The option value is a number in the range *0* to *63*, the maximum option value supported by KwikNet. The options currently specified by RFC-1700 are defined in KwikNet header file *KN_TELN.H* as symbols of the form *KN_TELOPT_xxxx*.

The option and its event flags can be isolated using the bit masks defined in KwikNet header file *KN_TELN.H* as symbols of the form *KN_TELOPV_xxxx*. The masks and their meanings are as follows:

KN_TELOPV_OPTION	Option mask used to isolate the option value
KN_TELOPV_STATE	Option is disabled/enabled (0/1)
KN_TELOPV_END	Option negotiated at local/remote (0/1) end of connection
KN_TELOPV_RESULT	Option negotiated was rejected/accepted (0/1)

If the option negotiated requires an option subnegotiation sequence, the callback function must use procedure *kntn_sendraw()* to send a properly formatted subnegotiation command, if your end of the negotiation bears that responsibility. Otherwise, the callback function must be prepared to accept a possible option subnegotiation command from the client.

Telnet Option Subnegotiation Commands from Client

An option subnegotiation command consists of the IAC SB character pair followed by an option identifier character, option parameter characters and a terminating IAC SE character pair. Properly embedded IAC parameter characters, if any, will appear as an IAC IAC character pair.

Whenever the Telnet server receives an option subnegotiation command from a client, it calls the callback function with callback code *KN_TELCBC_OPTSUB* to notify your application. Parameter *param* will have the current callback parameter value.

Parameter *p1* is a pointer to the first character (the IAC character) of a command buffer containing an exact copy of the option subnegotiation command as received by the Telnet server. Hence, the character referenced by **(p1+2)* identifies the negotiated option. The option identifier will have a value in the range *0* to *63*, the maximum option value supported by KwikNet. The options are defined in KwikNet header file *KN_TELN.H* as symbols of the form *KN_TELOPT_xxxx*.

Parameter *p2* specifies the total number of characters stored in the command buffer referenced by *p1*, including all embedded Telnet command characters. *P2* will always be greater than *0* and will never be greater than the receive buffer size which you declared in your KwikNet Network Parameter File.

If the option subnegotiation command in the command buffer is not properly terminated with an IAC SE character pair and the character at *p1 is the '\0' character, then the Telnet server is reporting an overlength option subnegotiation command which it was forced to discard for lack of room in its receive buffer.

Server Message Logging

The KwikNet Telnet server can be configured to log error information. To enable this feature, edit your KwikNet Network Parameter File and check the server Logging option on the Telnet property page.

The Telnet server does not log the normal receipt and delivery of Telnet commands or data. Since these operations involve normal TCP transactions, they can be observed using standard KwikNet debug logging and trace facilities.

If logging is permitted, the Telnet server calls the callback function with callback code *KN_TELCBC_LOG* to record error information. Since server error messages can be generated at any time, parameter *param* will have either the default or current callback parameter value. It is recommended that the callback function ignore parameter *param*.

Parameter *p1* is a pointer to a string buffer containing the message to be logged. Most strings include a final '\n' character. All are terminated by a '\0' character. Note that this string is a KwikNet error message, not a data string derived from Telnet.

Parameter *p2* is the KwikNet message print attribute identifying a Telnet server message (i.e. class *KN_PA_S_TELN*). Its use is illustrated below.

The callback function must accept the string and record it as quickly as possible. The Telnet server is unable to resume until the function returns. If strings must be buffered for presentation to the recording device, it is the logging function's responsibility.

KwikNet provides a data logging service which can be used to advantage to record these error messages. This service is described in Chapter 1.6 of the KwikNet TCP/IP Stack User's Guide. The callback function can log the error message on the KwikNet logging device by calling KwikNet service procedure *kn_dprintf()* giving it the attribute parameter *p2* and the message string pointer *p1* as follows:

```
kn_dprintf(p2, "%s", p1);
```

1.7 Client Callback Function

A KwikNet Telnet client can also have its own callback function. The callback function can be provided at the time the client task calls *kntc_create()* to create the client session. Alternatively, the callback function and its parameter can be installed using procedure *kntn_ioctl()* at the time callback services are actually needed by the client.

The client callback function executes in the context of the Telnet client task. It is called by KwikNet from within client task procedures such as *kntc_receive()*, *kntn_send()*, *kntn_sendcmd()*, *kntn_sendraw()* or *kntc_check()*, all of which force KwikNet to service the Telnet client session.

Since the client callback function operates in the same manner as the server callback function described in Chapter 1.6, you should review that material. However, minor differences do exist. For example, the client callback function always receives the client's current callback parameter value.

Since the client initiates its own connection, the client callback function never receives a *KN_TELCBC_CONN* connection callback. And, because the client always closes its own session, it only receives a *KN_TELCBC_DROP* disconnect callback if the client session is terminated by the server.

The *KN_TELCBC_IDLE* idle callback occurs at the client's service interval if, and only if, the client is idle within procedure *kntc_receive()* awaiting data from the server.

The Telnet client task does not automatically receive data. It must call procedure *kntc_receive()* to fetch data from its server. Hence, the client callback function will never see a *KN_TELCBC_DATA* data callback.

Telnet commands, option negotiation signals and option subnegotiation commands will never be seen by a client task if the client's session has no callback function. Hence, the client task must provide a callback function if it wishes to handle commands from its server or to participate in an option negotiation sequence with its server.

Client Message Logging

The KwikNet Telnet client can be configured to log error information. To enable this feature, edit your KwikNet Network Parameter File and check the client Logging option on the Telnet property page. However, even if permitted, client logging will not occur unless your client task provides a client callback function.

The Telnet client does not log the normal receipt and delivery of Telnet commands or data. Since these operations involve normal TCP transactions, they can be observed using standard KwikNet debug logging and trace facilities.

If logging is permitted and a callback function is available, the Telnet client calls the callback function with callback code *KN_TELCBC_LOG* to record error information. Parameter *param* will have the current callback parameter value.

Parameter *p1* is a pointer to a string buffer containing the message to be logged. Most strings include a final *'\n'* character. All are terminated by a *'\0'* character. Note that this string is a KwikNet error message, not a data string derived from Telnet.

Parameter *p2* is the KwikNet message print attribute identifying a Telnet client message (i.e. class *KN_PA_C_TELN*). Its use is illustrated below.

The callback function must accept the string and record it as quickly as possible. The Telnet client is unable to resume until the function returns. If strings must be buffered for presentation to the recording device, it is the logging function's responsibility.

As an example, the Telnet client provided with the KwikNet Telnet Sample Program accepts each error message and displays it on the client's console device.

Alternatively, the KwikNet data logging service can be used to advantage by a Telnet client to record these error messages. This service is described in Chapter 1.6 of the KwikNet TCP/IP Stack User's Guide. The callback function can log the error message on the KwikNet logging device by calling KwikNet service procedure *kn_dprintf()* giving it the attribute parameter *p2* and the message string pointer *p1* as follows:

```
kn_dprintf(p2, "%s", p1);
```

1.8 Telnet Option Negotiation

The KwikNet Telnet Option supports up to 64 Telnet options. By default, the KwikNet Telnet Option handles only the first four basic Telnet options listed in Figure 1.2-2. You can extend the number of supported options to the full 64 by editing your KwikNet Network Parameter File and checking the box labeled "Enable Telnet option subnegotiation" on the Telnet property page.

The options currently specified by RFC-1700 are defined in KwikNet header file *KN_TELN.H* as symbols of the form *KN_TELOPT_xxxx*. Figure 1.2-2 lists the specific Telnet options for which KwikNet provides built-in support. All other options must be dynamically configured by you so that KwikNet will allow the option to be negotiated.

When a KwikNet Telnet client or server is created, it is given an array of option specifications for each of the 4 or 64 options which KwikNet is configured to support. The options for which KwikNet provides built-in support are set ready to be negotiated into their default states. All other options in the array are initially considered unused. All options in the array retain their initial specification unless modified by your client or server with a call to procedure *kntn_ioctl()*.

Once a KwikNet Telnet server has been created, the server task can alter its option specifications with calls to procedure *kntn_ioctl()*. When the server is started with a call to procedure *knts_start()*, the state of the server's option array at that instant becomes the server's default. Thereafter, every client session established by the server inherits the server's default option settings. Of course, the server's callback function can adjust any of the options for any client session which it services. However, such adjustments apply only to the particular client session identified by the callback function.

When a KwikNet Telnet client task calls procedure *kntc_open()* to connect to a server, the client's option array is initialized to the default state previously described. Any subsequent option changes made by the client with calls to procedure *kntn_ioctl()* apply only to that client session.

Option Specifications

A Telnet option is used by the two ends of a Telnet connection to determine how each end will interpret and manage a particular terminal feature. Each end of the connection must agree with the other end as to how each will operate. Hence, both the local and remote end of each option must be negotiated. When the two ends agree to allow one end to support the option's feature, that end of the option is said to be enabled.

KwikNet uses an 8-bit option specification to control each Telnet option. Two 4-bit masks are used to control each end of the option. The masks are designated as local and remote option masks. In the descriptions which follow, replace *x* with *L* for local masks and *R* for remote masks. The option specification masks are defined in KwikNet header file *KN_TELN.H* as follows:

KN_TELOPM_xCFG	End *x* of the option can be negotiated	(0 = no; 1 = yes)
KN_TELOPM_xNEG	End *x* of the option is to be negotiated	(0 = no; 1 = yes)
KN_TELOPM_xEN	End *x* of the option is currently enabled	(0 = no; 1 = yes)
KN_TELOPM_xPEND	End *x* of the option is being negotiated	(0 = no; 1 = yes)

Manipulating Telnet Options

Service procedure *kntn_ioctl()* is used by a KwikNet Telnet client or server to read or initiate a change of state of a Telnet option for a particular client session.

The read operation fetches the option specification mask for a particular option. The read provides the complete current state of both ends of the option, ready for interpretation.

Effecting option changes is more complex. In the discussion which follows, only one end of the option is considered. However, the discussion applies equally to the local (*x* = *L*) end or remote (*x* = *R*) end of the option. In fact, both ends can be adjusted at once.

End *x* of an option cannot be altered from its default disabled, non-negotiable state without first being made configurable by setting the *KN_TELOPM_xCFG* bit. Once end *x* is configurable, it can be made non-configurable again, provided that end *x* of the option is disabled at the time.

Once an option is configurable, you can request that the option be negotiated into its opposite state. Hence, if an option is disabled, you can request that it be enabled through negotiation with your peer. If the option is enabled, you can request that it be disabled. In the latter case, refusal by the peer is not allowed by the TELNET protocol.

To request that the state of end *x* of an option be toggled, you must issue a negotiation request by calling procedure *kntn_ioctl()* to set both the *KN_TELOPM_xCFG* and *KN_TELOPM_xNEG* bits. The request will be denied if a negotiation request is already in place (*KN_TELOPM_xNEG* bit is set) or in progress (*KN_TELOPM_xPEND* bit is set).

If negotiation of an option is initiated by a server's callback function, the callback function will receive notification when the negotiation completes.

If negotiation of an option is initiated by a client task, the client must poll the option to determine if the option negotiation succeeded or failed. Negotiation is complete when the *KN_TELOPM_xNEG* and *KN_TELOPM_xPEND* bits are both 0. At that time, the option state specified by the *KN_TELOPM_xEN* bit determines the success or failure of the operation.

Alternatively, if the client provides a client callback function, that function will receive notification when the negotiation completes.

Finally, if a negotiated option requires an option subnegotiation sequence, your client must provide a client callback function. Your client or server callback function will bear full responsibility for sending or accepting all option subnegotiation commands as described in Chapter 1.8.

1.9 Telnet Sample Program

A Telnet Sample Program is provided with the KwikNet Telnet Option to illustrate the use of the KwikNet Telnet 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 Telnet client and server to be tested even if network hardware is not available. Once the Telnet Sample Program has been tested in loopback fashion, you can replace the Ethernet Loopback Driver with your own network device driver. Then the KwikNet Telnet client will be able to connect to other Telnet servers and foreign clients will be able to access the KwikNet Telnet server.

The KwikNet TCP/IP Stack requires a clock for proper network timing. The examples provided with the KwikNet Porting Kit all illustrate the clock interface. If you are using KwikNet with AMX, you must provide an AMX clock driver. If you have ported the AMX Sample Program to your hardware platform, you can use its AMX Clock Driver.

The sample includes a Telnet server task and a Telnet client task. The client uses the KwikNet console driver to provide a command line interface with a user. The console driver can be configured as described in Chapter 1.8 of the KwikNet TCP/IP Stack User's Guide to use any of several possible terminal devices as an interactive terminal. If you are using KwikNet with AMX and have ported the AMX Sample Program to your hardware platform, you can use its serial UART driver for console I/O.

The sample also uses the KwikNet data logging and message recording services to record messages generated by the Telnet server 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. The messages are recorded into an array of strings in memory. The Telnet client's interactive *dump* directive can be used to list these messages on the console device and empty the recording array.

Startup

The manner in which the KwikNet Telnet 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, creates the Telnet server task and then creates and starts the Telnet client task. The Telnet server task remains idle until started by the Telnet client task as will be described.

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 Telnet client task finally begins to execute. It starts KwikNet at its entry point *kn_enter()*. KwikNet self starts and forces the KwikNet Task to execute. Because the KwikNet Task operates at a priority above all tasks which use its services, it temporarily preempts the Telnet client task. The KwikNet Task initializes the network and its associated loopback driver and prepares the IP and TCP protocol stacks for use by the sample program.

Once the KwikNet initialization is complete, the Telnet client task resumes execution.

Telnet Client Operation

Once the KwikNet initialization is complete, the Telnet client task resumes execution. It initializes the KwikNet console driver and generates a signon message on the console device. It then calls *knto_create()* to create a Telnet client session.

The sample program's Telnet client operates in two modes: local and terminal. The client starts in local mode and goes to terminal mode when so directed by its user.

In **local mode**, the Telnet client generates a command line prompt *"Local>"* and waits for a user to enter a lower case directive and any parameters required by that directive. The directive is terminated by the Enter key (*'\r'* character).

The Telnet client decodes the directive and performs the requested action. Since most client service procedures are exercised by the Telnet client task, its code can serve as an excellent programming model for your own Telnet client software.

The following complete list of local mode directives will be presented if either *help* or *?* is entered as the command line directive.

```
help -  Display this text.
exit -  Terminate this sample program.
open <server IP address> <port>
          Connect to a Telnet server.  If port is 0,
          Telnet server port 23 will be used.
close  - End a previously opened Telnet session.
text   - Transfer in text mode.   Server echoes reformatted lines.
binary - Transfer in binary mode. Server operates in echo mode.
show   - Show summary of client session statistics.
ok     - See if connected to Telnet server.

serv - Start the KwikNet Telnet server on this machine.
stop - Stop  the KwikNet Telnet server.
dump [stat] - Dump KwikNet recorded log [and statistics].

glossary:
<text> - String you must provide.
[optional] - Parameter(s) within [] can be omitted.
         (omit the <,>,[ and ] characters).
```

The *open* directive is used to establish a Telnet session with a server. It is this directive which forces the sample program client to leave local mode and enter terminal mode.

In **terminal mode**, all command line input from the client's console keyboard continues to be echoed to the local console screen and buffered locally by the client. If the command line string is a valid local directive, it is acted upon by the client. Otherwise, it is assumed to be terminal data and is sent by the client to the server using the *kntn_send()* procedure. The client's end-of-line character (`'\r'`) is used to mark the end of these transmitted character strings. Hence, the client continues to handle its local directives even while in terminal mode.

In terminal mode, the sample program client regularly calls procedure *kntc_check()* to ensure that its KwikNet client session is properly serviced by KwikNet. As long as data continues to be received from the server, it is echoed by the client to its console screen. As a result, the server's response (if any) to strings received from the client will appear on the client's screen. The client then awaits the next user command from the console.

Several of the client's local directives are only valid in terminal mode. The *show* directive is used to generate a summary of client session statistics on the console. The *close* directive forces the sample program client to terminate its Telnet session with the server, leave terminal mode and revert to local mode. The *text* directive causes the client to send each line of data to the server as text using the Telnet text transfer mode. The *binary* directive causes the client to send each line of data to the server using the Telnet binary transfer mode.

Text and Binary Modes

The Telnet Sample Program can be used to confirm the correct operation of the KwikNet Telnet client and server when using Telnet's text or binary methods of transfer. In text mode, the client sends text to the server a line at a time and the server echoes each received line with simple, visible modifications. In binary mode, the client continues to send text to the server. However, the server discards all binary data, relying on the Telnet echo option to reflect each character to the client without modification.

Text mode of operation is initiated by the client when it decodes the *text* directive while operating in terminal mode. If the client is already operating in text mode, the directive is ignored. Otherwise, the client negotiates with the server to disable the character echo and binary transfer options at both ends of the connection. When the server detects that its binary option is disabled, it operates in its text mode in which it reformats and transmits each received line of text.

Binary mode of operation is initiated by the client when it decodes the *binary* directive while operating in terminal mode. If the client is already operating in binary mode, the directive is ignored. Otherwise, the client negotiates with the server to enable the server's character echo option. The client then negotiates the enabling of the binary transfer option at both ends of the connection. When the server detects that its binary option is enabled, it operates in its binary mode, discarding all data which it receives. However, since the server's echo option is enabled, each received character is automatically echoed back to the client without modification.

Telnet Server Operation

The Telnet client cannot open a connection to a Telnet server unless such a server exists on the network. Unless you have replaced the Ethernet Loopback Driver with a real device driver, the Telnet Sample Program has no direct network connection. Hence no Telnet server is accessible.

So why not use the KwikNet Telnet server? If you give the Telnet client the *serv* directive, it will start the KwikNet Telnet server task which will immediately begin operation since it is of higher priority than the client task. After starting the server task, the client task pauses briefly before it calls KwikNet procedure *kntn_ioctl()* to fetch the local KwikNet Telnet server task's IPv4 address and port number.

The Telnet client task displays the server's IPv4 address and port number giving you, the user, the chance to see that a server now exists to whom you can connect. Use the *open* directive to connect to the server at that IP address and port number. The KwikNet Telnet client is now conversing with the KwikNet Telnet server *across the network* even though both are executing on the same processor.

The sample program Telnet server operates in its text mode using Telnet's 7-bit ASCII data transfer mode unless otherwise negotiated with its connected client. The server's callback function buffers all received data until its end-of-line character (*'\n'*) is received. Note that the sample program server uses *'\n'* as its end-of-line character even though the sample program client uses *'\r'*.

When the server callback function is notified that a new client has connected to the server, it allocates one of its two client management blocks for use during that client's Telnet session. Only two blocks are provided because the Telnet Sample Program server is configured to concurrently handle a maximum of two clients.

When the sample program server has accumulated a line of data, it records the string on the KwikNet logging device and echoes the line back to the client. The echoed string is formatted specially so that the client's user can distinguish the server's response from the user's original input string. The following example illustrates this process.

"*Hi there.*"	Received by server from client
"*UserA |Hi there.|*"	Displayed by server on the KwikNet logging device
"*nnnn |Hi there.|*"	Echoed by server to the client

The server's end-of-line character *'\n'* is not shown in this example. The character | is used to bracket the echo of the received characters, excluding the end-of-line character. Note that the " character is neither received or sent. Also observe that no C end-of-string character (*'\0'*) is implied by this example.

The string *UserA* (or *UserB*) is used to identify which of the two supported clients generated the string received by the server.

In text mode, the server counts the number of lines of data received from each client. The line number is echoed to the client as the 4-digit string *nnnn*.

When the server callback function receives notification that a client has successfully negotiated the enabling of the server's Telnet binary option, the server begins operating in its binary mode. It stops interpreting data from the client as lines of text and simply discards all received characters. However, since the client also negotiates the enabling of the server's echo option, the server's echo of all received binary data continues until the client negotiates the disabling of the Telnet binary option and the return to text mode by both client and server.

At any time, you can enter the client *stop* directive which causes the Telnet client to call KwikNet procedure *knts_stop()* requesting the Telnet server task to stop execution. If any client still has an open connection to the server, the connection will be broken by the server before the server ceases to operate. In this case, if you issue the client's *ok* directive, you will probably observe a message indicating that the client's connection to the server has been lost.

The Telnet client calls procedure *knts_stop()* to stop the local Telnet server. The client provides a stoppage function which the server will call after it has stopped. The stoppage function allows your application (the client task in this case) to detect when the local KwikNet server has finally stopped, successfully or otherwise.

When the Telnet server stops, the client task's stoppage function calls KwikNet procedure *kntn_showstat()* to record the server's network statistics on the KwikNet logging device. The client task's *dump* directive can then be used to view the recorded server information.

Shutdown

When the Telnet client task decodes the *exit* directive, it closes any open connection which it may have had with a Telnet server. It then terminates the Telnet client session for which it has been responsible.

If the Telnet server task is still operating, the Telnet client task requests it to stop.

Next, the Telnet client task generates a signoff message and relinquishes use of the KwikNet console driver. It then calls procedure *kn_exit()* to stop operation of the KwikNet TCP/IP Stack.

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

KwikNet and Telnet Server Logging

The Telnet Sample Program uses the simple KwikNet message recording service to log various 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 Telnet 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 Telnet client task uses this feature to record a shutdown message. The Telnet server task also 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. To overcome this restriction, you can use the sample program Telnet client's interactive *dump* directive to list all of the recorded messages on the client's console device and empty the recording array.

Alternatively, if a debugger is used to control execution of the Telnet 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*.

Client Logging

The Telnet Sample Program's client task provides its own callback function which is used to record activities which occur during the Telnet client session. This logging function directs its output to the same console terminal which is used by the client task for its command line user interface.

Running the Sample Program

The KwikNet Telnet Sample Program requires that your target hardware include an interface to a console terminal. The Telnet client task will use its command line interface to interact with you at that terminal. You are therefore the real user behind the Telnet client task.

Each action which you initiate using a command line directive will generate a response on the terminal as the client task handles your request. The Telnet server task will only run if you use the *serv* directive to start it. The Telnet Sample Program runs until you issue the *exit* directive to shut it down.

KwikNet includes a number of debug features (see Chapter 1.9 of the KwikNet TCP/IP Stack User's Guide) which can assist you in using the Telnet 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).

The Telnet server task uses the KwikNet message recording service to log messages concerning its operation. KwikNet also records selected debug and trace information if any of these features are enabled. Unless you have modified the KwikNet recording mechanism, these messages are simply saved in memory and are therefore not visible. However, you can use the Telnet client's interactive *dump* directive to list all of the recorded messages on the client's console device and empty the recording array.

If you issue the *show* directive, the client task will call KwikNet procedure *kntn_showstat()* to display a summary of client session statistics on its console device. Of course, the *show* directive is only meaningful when a client Telnet terminal session is in progress.

If you use the *stop* directive to stop the Telnet server and then issue a *dump* directive, you can observe the server's network statistics which were recorded when the server stopped.

If you issue the *dump* directive with the *stat* parameter, the client task will call KwikNet procedure *kn_netstats()* to record network statistics on the KwikNet logging device. Only those network statistics which you enabled in your KwikNet Network Parameter File will be recorded. The client task will then dump these recorded statistics on its console.

1.10 Making the Telnet Sample Program

The sheer volume of detail necessary to understand and use Telnet with TCP/IP may at first be daunting. However, constructing the KwikNet Telnet Sample Program is actually a fairly simple process made even simpler by the KwikNet Configuration Manager, a Windows® utility provided with KwikNet.

The Telnet 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 Telnet Sample Program for a particular target processor family is provided ready for use on one of the development boards used at KADAK for testing.

Telnet Sample Program Directories

When KwikNet and its Telnet 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
TELNET	TELNET 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_TEL	KwikNet Telnet 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 which 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.

Telnet Sample Program Files

To build the KwikNet Telnet Sample Program using make file *KNTELSAM.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_SOCK.H	*TCPIP*	KwikNet Socket Interface definitions	
KN_TELN.H	*TELNET*	KwikNet Telnet 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 Telnet Sample Program directory containing:		
KNTELSAM.MAK	*TOOLXXX\SAM_TEL*	Telnet Sample Program make file	
KNTELSAM.C	*TOOLXXX\SAM_TEL*	Telnet Sample Program	
KNZZZAPP.H	*TOOLXXX\SAM_TEL*	Telnet Sample Program Application Header	
KNTELLIB.UP	*TOOLXXX\SAM_TEL*	Network Parameter File	
KNTELSAM.LKS	*TOOLXXX\SAM_TEL*	Link Specification File	(toolset dependent)
		Other toolset dependent files may be present.	
KNTELSCF.UP	*TOOLXXX\SAM_TEL*	User Parameter File	(for use with AMX)
KNTELTCF.UP	*TOOLXXX\SAM_TEL*	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	

Telnet Sample Program Parameter File

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

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

Telnet Sample Program KwikNet Library

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

Use the KwikNet Configuration Builder to edit the sample program Network Parameter File *KNTELLIB.UP*. Use the Configuration Builder to generate the Network Library Make File *KNTELLIB.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 Telnet Sample Program.

Then copy files *KNTELLIB.UP* and *KNTELLIB.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 Telnet 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 Telnet Sample Program Make Process

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

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 Telnet 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_TEL* directory the current directory.

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

```
NMAKE -fKNTELSAM.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_TEL*) 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_TEL*) 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 Telnet Sample Program load module *KNTELSAM.xxx* is created in toolset directory *TOOLXXX\SAM_TEL*. 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 Telnet Sample Program load module *KNTELSAM.xxx*.

1.11 Adding Telnet to Your Application

Before you can add the TELNET protocol to your application, there are a number of prerequisites which 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 Telnet. If these components are not operational, the KwikNet Telnet Option cannot operate correctly.

KwikNet Library

Begin by deciding whether you need a Telnet client or server or both. Rarely are both required. Then decide which Telnet features must be supported. Review the Telnet property page described in Chapter 1.3. In particular, omit support for Telnet option negotiation unless you actually have such a need.

If you are incorporating a Telnet server, you may wish to customize the external look of your server by providing your own server name as described in Chapter 1.5.

Armed with your Telnet feature list, use the KwikNet Configuration Manager to edit your application's KwikNet Network Parameter File to include the TELNET 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

Each Telnet client that you implement will allocate a block of memory for its dedicated use. Each Telnet server will allocate one such block for each client that it concurrently services. A rough estimate of the block size is *256+rcvbuf+linebuf+sendbuf* where:

rcvbuf	is the configured Telnet receive buffer size,
linebuf	is the configured Telnet line buffer size and
sendbuf	is the configured Telnet send buffer size.

Each of these parameters is adjustable in your KwikNet Network Parameter File on the Telnet property page. To meet these requirements, you may have to edit your KwikNet Network Parameter File to increase the memory available for allocation.

Telnet Client and Server Tasks

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

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

A stack size of 4K to 8K bytes is considered adequate for use with most RT/OS implementations. The stack size can be trimmed after your Telnet tasks have been tested and actual stack usage observed using your debugger.

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

If you are incorporating a Telnet client, you must create a Telnet client task procedure which performs the Telnet operations required by your application. Only you can define such a procedure. All of the KwikNet Telnet client-specific procedures listed in Chapter 2 are at your disposal. You can use the Telnet client task in the Telnet Sample Program as a guideline for proper form.

If you are incorporating a Telnet server, then you may have a significant coding responsibility. You must create a Telnet server task procedure which creates and starts a KwikNet Telnet server. You must also provide the server callback function to interpret and handle all client requests for service. Only you can define these requirements. It is this callback function which provides access by a Telnet client to your application services. All of the KwikNet Telnet server-specific procedures listed in Chapter 2 are at your disposal. You can use the Telnet server task in the Telnet Sample Program as a guideline for proper form.

The Telnet 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 Telnet header file *KN_TELN.H* in the Treck installation directory, say *C:\TRECK\INCLUDE*. This header file is copied to the Treck directory from the KwikNet *TELNET* 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 Telnet to an existing KwikNet application, there is little to be done.

To meet the memory demands of your Telnet client and servers, 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 files must include the KwikNet Library which you built with support for Telnet. The object modules for your Telnet client and server tasks and any support modules which 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 Telnet support included.

AMX Considerations

When reconstructing a KwikNet application which 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 Telnet client and server tasks.

Telnet 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 Telnet Sample Program. These are simple AMX trigger tasks without message queues.

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

The Telnet task priorities must be lower than that of the KwikNet Task. If both Telnet server and client tasks exist, it is usual to make Telnet server tasks of higher priority than Telnet client tasks. If you are using AMX 86 to access MS-DOS® file services, the PC Supervisor Task should be below all Telnet 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 Telnet unless your Telnet client or server task requires special device support which is not already part of your application.

Performance Considerations

A meaningful discussion of all of the issues which affect the performance of a Telnet server or client are beyond the scope of this document. Factors affecting the performance of the KwikNet Telnet 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.)
TCP protocol effects (window size adaptations)
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, sockets, etc.)
KwikNet Telnet configuration (service intervals, Telnet buffer sizes)

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.

Setting the Telnet service rate to match the TCP/IP stack clock frequency is the best you can accomplish. Any faster Telnet service will serve little purpose and will simply introduce further burden on the processor.

2. KwikNet Telnet Services

2.1 Introduction to Telnet Services

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

A description of these KwikNet Telnet 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 which appear in narrative text are also displayed in italics. 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 Telnet 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 □ Client Task □ Client Callback ■ Server Task □ Server Callback □ ISP □ Timer 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 your application server task 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. AMX Timer Procedures, Restart Procedures and Exit Procedures must not call the KwikNet Telnet service procedures unless documented otherwise.

...more

Used by ☐ Client Task ☐ Client Callback ■ Server Task ☐ Server Callback ☐ ISP ☐ Timer Procedure

For other multitasking systems, the client or server task is an application task executing at a priority below that of the KwikNet Task. A timer procedure is any function executed by a task of higher priority than the KwikNet Task. An ISP is a KwikNet device driver interrupt handler called from an RTOS compatible interrupt service routine.

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. The client task executes as part of your App-Task. The server task is executed by the KwikNet Task in the KwikNet domain. An ISP is a KwikNet device driver interrupt handler called from an interrupt service routine. Timer 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_TELN.H* for compilation.

File *KN_LIB.H* is the KwikNet include file which corresponds to the KwikNet Library which 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_TELN.H* is the KwikNet include file which you must include if your application uses Telnet client or server services. This file is located in KwikNet installation directory *TELNET*. 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 Telnet Service Procedures

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

The following list summarizes these KwikNet Telnet service procedures. They are grouped functionally for easy reference.

Client Operations

kntc_create	Create an instance of a KwikNet Telnet client
kntc_open	Open a connection to a Telnet server
kntc_close	Close the connection to the Telnet server
kntc_check	Check for data and commands from the Telnet server
kntc_receive	Get application data from the Telnet server

Server Operations

knts_create	Create an instance of a KwikNet Telnet server
knts_start	Start a KwikNet Telnet server
knts_stop	Stop a KwikNet Telnet server
knts_status	Fetch the status of a KwikNet Telnet server

Common Client and/or Server Operations

kntn_delete	Delete an instance of a KwikNet Telnet client or server
kntn_errno	Fetch most recent status result (error) recorded for a KwikNet Telnet client, server or client session
kntn_ioctl	Read or modify Telnet client, server or session parameters: Network addresses Callback functions and parameters Line editing character encoding specifications End-of-line attributes Initiate a Telnet option negotiation sequence Disconnect a Telnet server's client Miscellaneous operations: Reset error indicator Get type of the entity identified by a Telnet descriptor Read or modify Telnet client, server or client session name
kntn_send	Send application data to a Telnet peer
kntn_sendcmd	Send a simple Telnet command to a Telnet peer
kntn_sendraw	Send raw Telnet command to a Telnet peer Used to send Telnet option subnegotiation commands
kntn_showstat	Generate a statistics summary for a KwikNet Telnet client, server or client session

Purpose **Check for Data and Commands from the Telnet Server**

Used by ■ Client Task □ Client Callback □ Server Task □ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntc_check(KN_TELND telnd);
```

Description *Telnd* is a Telnet descriptor identifying the KwikNet Telnet client task which wants to fetch data from its server.

This procedure can be used by the KwikNet Telnet client task to periodically poll for received data. The call forces KwikNet to service the client session in the absence of other requests to wait for received data or send data or commands.

The client callback function, if one exists, will be called if any Telnet commands or option events require service.

Any command characters buffered for transmission will be sent to the server to the extent permitted by the TCP socket connection.

Returns The number of bytes of application data available from the server is returned. If there is no data available and the connection to the server is still valid, the value *0* is returned.

On failure, the error status *-1* is returned.

The error indicator for Telnet descriptor *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELNOSESS	A client session has not been opened.
KN_ERTELNOCONN	There is no socket connection.
else	TCP socket error encountered.

Note The received data, if any, is available for reading. However, the caller must use procedure *kntc_receive()* to fetch the received data.

See Also *kntc_receive()*

Purpose **Close the Connection to the Telnet Server**

Used by ■ Client Task □ Client Callback □ Server Task □ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntc_close(KN_TELND telnd);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet client session with an open connection which is to be closed.

Returns If successful, a value of *0* is returned. The Telnet session with a Telnet server is terminated and the socket associated with the session is closed.

The Telnet descriptor remains valid and can be used to open another Telnet connection.

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELNOSESS	A client session has not been opened.
KN_ERTELNOCONN	There is no socket connection.
else	TCP socket error encountered.

See Also *kntc_open()*

Purpose **Create an Instance of a Telnet Client**

Used by ■ Client Task □ Client Callback □ Server Task □ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntc_create(KN_TELND *telndp,
            void (*cbfn)(KN_TELND telnd, int opcode,
                         char *p1, int p2, void *param),
            void *param);
```

Description *Telndp* is a pointer to storage for the Telnet descriptor which will be assigned by KwikNet to identify the Telnet client.

Cbfn is a pointer to a client callback function which the Telnet client will call to report Telnet session activity. This function must be coded to operate as described in Chapter 1.7. If a callback function is not required, set parameter *cbfn* to *(KN_TELCALLBACK)0L*.

Param is a parameter which the caller wishes to pass to the callback function *cbfn()*. If no parameter is required, set *param* to *NULL*.

Returns If successful, a value of *0* is returned. A valid Telnet descriptor is stored at **telndp*. This descriptor must be used in all subsequent calls by the Telnet client task to identify the particular Telnet client.

The Telnet client is created with the following default KwikNet Telnet attributes in effect:

7-bit ASCII data transfers
Character echo disabled
Suppress-GA to be enabled at both ends of connection if possible
Line control characters set per Figure 1.2-3

On failure, the error status *-1* is returned and **telndp* is undefined.

If a Telnet client cannot be created, the error indicator defining the reason for failure cannot be recorded. You cannot use *kntn_errno()* to retrieve the error code since you have no Telnet descriptor to interrogate. The most probable reason for failure is that sufficient memory is not available for use by the Telnet client.

Note If Telnet client logging is not enabled in the KwikNet Library, Telnet logging will not occur, even if a callback function has been provided.

If Telnet client logging is enabled in the KwikNet Library and a callback function is provided, that function will be called to log KwikNet messages describing Telnet client session activity.

See Also *kntc_open()*, *kntn_delete()*

Purpose **Open a Connection to a Telnet Server**

Used by ■ Client Task □ Client Callback □ Server Task □ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntc_open(KN_TELND telnd, struct in_addr *inadrp, int port);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet client wishing to make the connection to its server.

Inadrp is a pointer to a structure containing the IPv4 address, in net endian form, of the Telnet server with whom a Telnet session 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)*/
  };
```

Port is the port number for the Telnet server. If parameter *port* is *0*, the well known Telnet port number 23 will be used.

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

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELINSESS	Client session is already in progress.
KN_ERTELSOCK	Cannot allocate socket for connection.
else	Cannot connect; TCP socket error encountered.

Note If a Telnet client already has a connection to a server, any attempt to open another connection without first closing the existing connection will be rejected with a *KN_ERTELINSESS* error indication.

Note A TCP/IP connection will be established with the specified Telnet server. The local and foreign IPv4 addresses and port numbers associated with the connection can be read using procedure *kntn_ioctl()* (network parameters).

...more

Note The Telnet client session is opened with the following default KwikNet Telnet attributes in effect:

7-bit ASCII data transfers
Character echo disabled
Suppress-GA to be enabled at both ends of connection if possible
Line control characters set per Figure 1.2-3

See Also *kntc_close()*

Purpose **Receive Application Data from a Telnet Server**

Used by ■ Client Task □ Client Callback □ Server Task □ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntc_receive(KN_TELND telnd, char *buf, int len);
```

Description *Telnd* is a Telnet descriptor identifying the KwikNet Telnet client task which wants to fetch data from its server.

Buf is a pointer to a storage buffer for the received data.

Len is the buffer size, measured in bytes.

Returns If successful, the number of bytes of data stored at **buf* is returned. The number of data bytes will be <= *len*. When operating in text mode (Telnet binary option is disabled locally), the transfer of data characters to the buffer at **bufp* will cease after the first end-of-line character string, if any, has been stored in the buffer.

The KwikNet Telnet client will not return to the caller until an error condition is detected or some data has been received from the server. KwikNet will continue to service the client session at its regular service interval while awaiting data from the server.

On failure, the error status *-1* is returned.

The error indicator for Telnet descriptor *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELNOSESS	A client session has not been opened.
KN_ERTELNOCONN	There is no socket connection.
else	TCP socket error encountered.

Restrictions This procedure must only be called by a Telnet client task.

Note If the current settings of the peer's Telnet options indicate that the peer has binary transmission enabled, data will be received as 8-bit characters without translation. Otherwise, characters are interpreted as ASCII according to the current attributes specified by Telnet descriptor *telnd*. If 8-bit characters are received, they are accepted without translation even though the TELNET protocol only allows 7-bit characters. All ASCII line control characters are subject to conversion from their Telnet equivalents. In either mode, the IAC IAC pair shrinks to one IAC character.

See Also *kntc_check()*, *kntn_send()*, *kntn_sendcmd()*, *kntn_sendraw()*

Purpose **Delete a KwikNet Telnet Client or Server**

Used by ■ Client Task □ Client Callback ■ Server Task □ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_delete(KN_TELND telnd);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet client or server which is to be deleted.

Returns If successful, a value of *0* is returned. The Telnet client or server is deleted and all resources associated with it are released.

On failure, the error status *-1* is returned.

The error indicator for client or server *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELINSESS	The Telnet client or server still has an active session. The server must be stopped before it can be deleted. The client session must be closed.

Restrictions A Telnet server must be stopped before it can be deleted.

A Telnet client's session must be closed before the client can be deleted.

Note It is recommended that only a server (client) task delete its instance of a Telnet server (client).

See Also *kntc_create()*, *knts_create()*

Purpose **Get Error Code from Recent Telnet Operation**

Used by ■ Client Task ■ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_errno(KN_TELND telnd);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet entity for which error information is to be retrieved. Telnet entities include a KwikNet Telnet client or server or any client session which either manages.

Returns If successful, a KwikNet error code is returned. These error codes are summarized in Appendix B of the KwikNet TCP/IP Stack User's Guide. The error code identifies the reason for the failure, if any, of the most recent Telnet operation attempted for entity *telnd*. Note that TCP/IP socket error codes can also be encountered because the Telnet client and server use TCP/IP socket connections to communicate with their peers.

An error status of *KN_ERTELND* is returned if the Telnet descriptor *telnd* is invalid, precluding the interrogation of the Telnet entity.

Note The error code associated with entity *telnd* remains unaltered. To reset the error code to *0*, use procedure *kntn_ioctl()* (miscellaneous).

Crosstalk If a Telnet descriptor is used indiscriminately, it is possible to generate error code crosstalk. For example, suppose a server callback function somehow manages to use a server descriptor (the one generated by procedure *knts_create()*) instead of the client session descriptor presented to the callback function by the server. If an error is detected when the callback function calls a KwikNet procedure (and an error is highly likely), the error code will be recorded in the server descriptor, not in the client session descriptor. The errant callback function can call *kntn_error()* to read the error code and, provided it continues to use the server descriptor, will get the correct error code. However, since it is the server descriptor that has the error code, the next time the server examines its own error code it will observe a false error condition.

Even worse, crosstalk can occur if you have both a client and server operating in the same processor. If the client task uses a server descriptor, or vice-versa, each can cause error codes to be recorded in the other's descriptor. The result is confusion at best, chaos at worst.

However, for error code crosstalk to occur, your client, server and client sessions must have granted access to each other's descriptors, a condition easily remedied by good design and programming practice.

Purpose **Read or Modify Telnet Network Parameters**

Used by ■ Client Task ■ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
#include "KN_SOCK.H"
int kntn_ioctl(KN_TELND telnd, int opcode, void *addrp);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet entity of interest. Telnet entities include a KwikNet Telnet client or server or any client session which either manages.

Opcode is an operation code identifying the network parameter to be read or modified. The allowable operations are specified by the following ioctl values which are defined in header file *KN_TELN.H*.

KN_TELIOC_GETLADR	Get local network IPv4 address and port
KN_TELIOC_GETRADR	Get remote network IPv4 address and port
KN_TELIOC_SETLADR	Set local server network IPv4 address and port

Addrp is a pointer to a socket address structure used to specify the IPv4 address and port number for one end of a socket connection. The socket address structure *sockaddr_in* is defined in the Treck sockets header file *TRSOCKET.H* located in the *TRECK\INCLUDE* directory.

When setting the network address, the structure must be initialized to contain the IPv4 address in structure member *addrp->sin_addr* and the port number in member *addrp->sin_port*. When reading the network address, these structure members will be filled upon return. Both IP address and port number are specified in net endian form. All other structure members are unused by this procedure.

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

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERPARAM	*Opcode* is invalid **or** *addrp* is *NULL*.
KN_ERTELREJECT	*Opcode* is invalid for the entity specified by *telnd*
KN_ERTELACTIVE	*Opcode* is invalid; Telnet server already started.
KN_ERTELNOCONN	There is no socket connection.

Restriction A Telnet server task can set its local network address prior to starting the Telnet server. No other local or foreign network address can be modified.

Purpose **Read or Modify Telnet Callback Settings**

Used by ■ Client Task ■ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_ioctl(KN_TELND telnd, int opcode, void *cbvarp);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet entity of interest. Telnet entities include a KwikNet Telnet client or server or any client session which either manages.

Opcode is an operation code identifying the callback setting to be read or modified. The allowable operations are specified by the following ioctl values which are defined in header file *KN_TELN.H*.

KN_TELIOC_GETCBF	Get callback function pointer
KN_TELIOC_SETCBF	Set callback function pointer
KN_TELIOC_GETCBP	Get callback parameter
KN_TELIOC_SETCBP	Set callback parameter

Cbvarp is a pointer to a callback variable. The callback variable provides the new value or storage for a copy of the current value of one of the callback settings.

When reading or installing the pointer to a callback function, parameter *cbvarp* is a pointer to a callback variable of type *KN_TELCALLBACK*. The callback variable is therefore a pointer to a callback function. A Telnet client can remove its the callback function by setting the callback function pointer to *(KN_TELCALLBACK)0L*.

When reading or installing a callback parameter, *cbvarp* is a pointer to an actual callback parameter, which, by definition, is a pointer to *void*.

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

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERPARAM	*Opcode* is invalid **or** *cbvarp* is *NULL*.
KN_ERTELREJECT	A Telnet server cannot remove its callback function.

Restriction You cannot remove the callback function for a Telnet server or any client session which it is handling.

kntn_ioctl
(line editing characters)

Purpose **Read or Modify Telnet Line Editing Characters**

Used by ■ Client Task ■ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_ioctl(KN_TELND telnd, int opcode, void *charp);
```

Description This procedure is used to read or modify the definition of the ASCII characters which your application wishes to use for the EC (erase character) and EL (erase line) line editing commands supported by the TELNET protocol. The specified ASCII characters are translated to/from the equivalent Telnet commands as data is sent/received by your application.

Telnd is the Telnet descriptor identifying the Telnet entity of interest. Telnet entities include a KwikNet Telnet client or server or any client session which either manages.

Opcode is an operation code identifying the editing character to be read or modified. The allowable operations are specified by the following ioctl values which are defined in header file *KN_TELN.H*.

KN_TELIOC_GETEC	Get character used for erase-character
KN_TELIOC_SETEC	Set character used for erase-character
KN_TELIOC_GETEL	Get character used for erase-line
KN_TELIOC_SETEL	Set character used for erase-line

Charp is a pointer to a character of type *unsigned char*. The character provides the new value or storage for a copy of the current value of one of the characters used by your application for line editing. A value of *'\0'* for either line editing character will force the equivalent Telnet command to be ignored if received. A value of *'\0'* will also preclude conversion of any transmitted character to that Telnet command.

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

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERPARAM	*Opcode* is invalid **or** *cbvarp* is *NULL*.

Note The line editing characters established for a KwikNet Telnet client or server are inherited by any client session which either handles. Subsequent changes for a client session affect only that session.

kntn_ioctl
(end-of-line attributes)

Purpose **Read or Modify Telnet End-of-Line Attributes**

Used by ■ Client Task ■ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_ioctl(KN_TELND telnd, int opcode, void *attribp);
```

Description This procedure is used to read or modify the attributes used to specify how the ASCII end-of-line characters *LF* (*0x0A*) and *CR* (*0x0A*) are to be interpreted by your application. The specified attributes determine how the Telnet end-of-line command (CR LF) and return command (CR NUL) are generated and interpreted.

Telnd is the Telnet descriptor identifying the Telnet entity of interest. Telnet entities include a KwikNet Telnet client or server or any client session which either manages.

Opcode is an operation code which indicates that end-of-line attributes are to be read or modified. The allowable operations are specified by the following ioctl values which are defined in header file *KN_TELN.H*.

KN_TELIOC_GETEOL	Get end-of-line attributes
KN_TELIOC_SETEOL	Set end-of-line attributes

Attribp is a pointer to an *unsigned int* which provides the new value or storage for a copy of the current value of the end-of-line attributes. The allowable attributes are specified by ORing together one or more of the following bit mask values which are defined in header file *KN_TELN.H*.

KN_TELEOL_CRLF	Application uses *CR LF* for end-of-line
KN_TELEOL_CR	Application uses *CR* for end-of-line
KN_TELEOL_LF	Application uses *LF* for end-of-line
KN_TELEOL_STRIPCR	Strip orphan *CR* characters
KN_TELEOL_STRIPLF	Strip orphan *LF* characters
KN_TELEOL_TXCRNUL	Transmit CR NUL as end-of-line command instead of the default CR LF command
KN_TELEOL_RXCRNUL	Convert received CR NUL to application's end-of-line character sequence instead of the default *CR* character
KN_TELEOL_RXLF	Convert received *LF* character to application's end-of-line character sequence

...more

Description ...continued

Only one of the three application end-of-line attributes *KN_TELEOL_CRLF*, *KN_TELEOL_CR* or *KN_TELEOL_LF* must be specified. The attribute *KN_TELEOL_CRLF* can also be used as a mask to isolate the end-of-line attribute from all other attributes.

If attribute *KN_TELEOL_STRIPCR* or *KN_TELEOL_STRIPLF* is specified, the corresponding ASCII character (*LF* or *CR*) will be stripped from the data stream accepted from or delivered to your application. Only orphans, *CR* or *LF* characters which are not part of a valid application end-of-line character sequence, will be stripped.

By default, any occurrence of the application end-of-line character sequence will be translated to the Telnet CR LF pair for transmission. You can set the *KN_TELEOL_TXCRNUL* attribute to alter this behavior and force the Telnet CR NUL pair to be sent instead.

By default, any Telnet CR NUL pair which is received will be translated to the ASCII character *CR* and be subject to orphan stripping prior to presentation to your application. You can alter this behavior by setting the *KN_TELEOL_RXCRNUL* attribute to force the Telnet CR NUL pair to be treated as an end-of-line signal in addition to the normal Telnet CR LF pair.

By default, any ASCII character *LF* which is received will be subject to orphan stripping before it is passed to your application untranslated. You can alter this behavior by setting the *KN_TELEOL_RXLF* attribute to force any received *LF* character to be treated as an additional end-of-line signal.

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

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERPARAM	*Opcode* is invalid **or** *attribp* is *NULL*.

Note The end-of-line attributes established for a KwikNet Telnet client or server are inherited by any client session which either handles. Subsequent changes for a client session affect only that session.

Purpose **Read or Modify Telnet Option Settings**

Used by ■ Client Task ■ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_ioctl(KN_TELND telnd, int opcode, void *optionp);
```

Description This procedure is used to read or initiate modification of a Telnet option. The option negotiation methodology is described in Chapter 1.8.

Telnd is the Telnet descriptor identifying the Telnet entity of interest. Telnet entities include a KwikNet Telnet client or server or any client session which either manages.

Opcode is an operation code indicating that the current state of a Telnet option is to be read or its modification negotiated. The allowable operations are specified by the following ioctl values which are defined in header file *KN_TELN.H*.

KN_TELIOC_GETOPT	Get current state of a Telnet option
KN_TELIOC_SETOPT	Set (negotiate) state of a Telnet option

Optionp is a pointer to an array of two integer values (type *int*). The first value is the Telnet option identifier, a number in the range 0 to 63. The options currently specified by RFC-1700 are defined in KwikNet header file *KN_TELN.H* as symbols of the form *KN_TELOPT_xxxx*.

The second value in the array is the option state. When reading the state of a Telnet option, a value defining the state of both ends of the option is stored in this integer upon return. When adjusting a Telnet option, the value of the second integer specifies the change, if any, to be initiated and which ends of the option are to be affected.

...more

Description ...continued

KwikNet uses an 8-bit option specification to control each Telnet option. Two 4-bit masks are used to control each end of the option. The masks are designated as local and remote option masks. In the descriptions which follow, replace *x* with *L* for local masks and *R* for remote masks. The option specification masks are defined in KwikNet header file *KN_TELN.H* as follows:

KN_TELOPM_xCFG	End *x* of the option can be negotiated	(0 = no; 1 = yes)
KN_TELOPM_xNEG	End *x* of the option is to be negotiated	(0 = no; 1 = yes)
KN_TELOPM_xEN	End *x* of the option is currently enabled	(0 = no; 1 = yes)
KN_TELOPM_xPEND	End *x* of the option is being negotiated	(0 = no; 1 = yes)

These option masks can be used to interpret the current state of a Telnet option after it is read. When changing the state of an option, only masks *KN_TELOPM_xCFG* and *KN_TELOPM_xNEG* can be specified. Other mask bits, if set, will be rejected as invalid parameters.

End *x* of an option must be negotiable (mask bit *KN_TELOPM_xCFG* set) before the option state can be altered. Once negotiable, the state of end *x* of the option can be changed (toggled) through negotiation with the connected peer. To initiate such a change, you must specify the mask *KN_TELOPM_xCFG|KN_TELOPM_xNEG* to stay negotiable and start the negotiation.

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

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERPARAM	*Opcode* is invalid **or** *optionp* is *NULL* **or** Telnet option identifier is out of range.
KN_ERTELREJECT	Current option state precludes requested action.

Note Since you can initiate negotiation of both ends of an option at once, it is essential that you use care when dealing with only one end of the option. For example, assume you wish to adjust the local (*x = L*) end of an option. If the remote end is also negotiable (its mask bit *KN_TELOPM_RCFG* is set), you must be sure to include its negotiable mask *KN_TELOPM_RCFG* in your state mask when adjusting the local end of the option so that the remote end will not inadvertently be rendered non-negotiable.

...more

Note You can only initiate a change in the option state. If the option is disabled, you can ask that it be enabled and vice-versa.

Restrictions End *x* of an option must be disabled before you can request that it be made non-negotiable by setting its four mask bits to *0*. Be careful not to upset the other end of the option if it is not also being disabled.

Any options specified for a KwikNet Telnet server before the server starts do not actually initiate negotiation. They simply become the default options to be negotiated for any client which connects to the server.

Any options specified for a KwikNet Telnet client before the client opens a connection to a server do not actually initiate negotiation. They simply become the default options to be negotiated by the client when it first connects to a server.

All KwikNet Telnet client options are reset to their KwikNet defaults when a client session is closed. Hence, if common Telnet options are required for all of a client's connections, they must be specified by the KwikNet Telnet client prior to establishing each connection.

kntn_ioctl
(disconnect)

Purpose **Disconnect a Telnet Server's Client**

Used by □ Client Task □ Client Callback □ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_ioctl(KN_TELND telnd, int opcode, void *unused);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet client session being handled by the server callback function. The connection with the client is to be broken by the server.

Opcode is the operation code indicating that a client is to be disconnected. This ioctl value is defined in header file *KN_TELN.H*.

KN_TELIOC_DROP	Disconnect a client

The dummy parameter *unused* can be set to *NULL*.

Returns If successful, a value of *0* is returned.
The disconnection will occur when the server callback function completes.

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELNOCONN	There is no connection.

Restriction Only the server callback function for a KwikNet Telnet server can disconnect a client being handled by the server.

Purpose **Miscellaneous Telnet Client and Server Operations**

Used by ■ Client Task ■ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_ioctl(KN_TELND telnd, int opcode, void *paramp);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet entity of interest. Telnet entities include a KwikNet Telnet client or server or any client session which either manages.

Opcode is an operation code identifying the operation to be performed. The allowable operations are specified by the following ioctl values which are defined in header file *KN_TELN.H*.

KN_TELIOC_NOERR	Reset error number
KN_TELIOC_GETTYPE	Get type of Telnet entity
KN_TELIOC_GETNAME	Get name of Telnet entity
KN_TELIOC_SETNAME	Set name of Telnet entity

Paramp is an operation dependent pointer parameter.

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

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERPARAM	*Opcode* is invalid **or** *paramp* is *NULL*.

...more

Returns ...continued

If *opcode* is *KN_TELIOC_NOERR*, parameter *paramp* is unused and should be set to *NULL*. The error number recorded for the Telnet entity identified by *telnd* is reset to *0*.

If *opcode* is *KN_TELIOC_GETTYPE*, parameter *paramp* is a pointer to storage for an integer value (*int*) representing the type of the Telnet entity identified by *telnd*. Upon return, the stored type will be one of following values which are defined in header file *KN_TELN.H*.

KN_TELNTY_SERVER	Telnet server
KN_TELNTY_SSESS	Client session for a Telnet server
KN_TELNTY_CLIENT	Telnet client
KN_TELNTY_CSESS	Client session for a Telnet client

If *opcode* is *KN_TELIOC_GETNAME*, parameter *paramp* is a pointer to storage for a string pointer. Upon return, the string pointer references a constant, *'\0'* terminated string giving the name of the Telnet entity identified by *telnd*.

If *opcode* is *KN_TELIOC_SETNAME*, parameter *paramp* is a string pointer which references a constant, *'\0'* terminated string providing the name to be applied to the Telnet entity identified by *telnd*.

Purpose **Send Application Data to a Telnet Peer**

Used by ■ Client Task ■ Client Callback □ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_send(KN_TELND telnd, const char *buf, int len);
```

Description *Telnd* is a Telnet descriptor identifying the Telnet entity which is initiating the data transmission. A KwikNet Telnet client task or its callback function can send data to its server. A KwikNet Telnet server callback function can send data to any client which it is servicing.

Buf is a pointer to the buffer of application data to be sent.

Len is the buffer size, measured in bytes.

Returns If successful, the number of bytes of data sent from **buf* is returned.

If none of the data can be accepted for transmission via the TCP socket, an error status of *0* will be returned.

On failure, the error status *-1* is returned.

The error indicator for Telnet descriptor *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELNOSESS	A client session has not been opened.
KN_ERTELNOCONN	There is no socket connection.
else	TCP socket error encountered.

Note If the current settings of the sender's Telnet options indicate that binary transmission is enabled, data is transmitted as 8-bit characters without translation. Otherwise, characters are interpreted as ASCII according to the current attributes specified by Telnet descriptor *telnd*. If 8-bit characters are encountered, they are sent without translation even though the TELNET protocol only allows 7-bit characters. All ASCII line control characters are subject to conversion to their Telnet equivalents. In either mode, a valid IAC character is always sent as an IAC IAC pair.

See Also *kntn_sendcmd()*, *kntn_sendraw()*

Purpose **Send a Telnet Command to a Telnet Peer**

Used by ■ Client Task ■ Client Callback □ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_sendcmd(KN_TELND telnd, int cmd);
```

Description Use this procedure to send simple Telnet commands.

Telnd is a Telnet descriptor identifying the Telnet entity which is initiating the transmission. A KwikNet Telnet client task or its callback function can send a Telnet command to its server. A KwikNet Telnet server callback function can send a Telnet command to any client which it is servicing.

Cmd is the command identifier used to specify the Telnet command to be sent. Valid command identifiers for the commands listed in Figure 1.2-1 are defined as symbols of the form *KN_TELCMD_xxxx* in KwikNet header file *KN_TELN.H*. The string *xxxx* corresponds to the Telnet command name shown in Figure 1.2-1. The commands marked ■ can be sent by your application using this procedure.

The synch command (*KN_TELCMD_DM*) is usually only sent following, or in response to, one of the commands marked *!*. It is usually sent using the TCP urgent out-of-band feature which you can indicate by ORing the qualifier *KN_TELCMD_URGENT* with the command identifier.

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

On failure, the error status *-1* is returned.

The error indicator for Telnet descriptor *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERPARAM	*Cmd* is not a valid command identifier.
KN_ERTELNOSESS	A client session has not been opened.
KN_ERTELNOCONN	There is no socket connection.
KN_ERTELBUSY	Some prior command has not yet been transmitted.
else	TCP socket error encountered.

See Also *kntn_send()*, *kntn_sendraw()*

Purpose **Send a Raw Telnet Command to a Telnet Peer**

Used by ■ Client Task ■ Client Callback □ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_sendraw(KN_TELND telnd, const char *buf, int len);
```

Description Use this procedure to send properly formatted Telnet commands such as those needed for Telnet option subnegotiation.

Telnd is a Telnet descriptor identifying the Telnet entity which is initiating the transmission. A KwikNet Telnet client task or its callback function can send a Telnet command to its server. A KwikNet Telnet server callback function can send a Telnet command to any client which it is servicing.

Buf is a pointer to the buffer of Telnet command data to be sent.

Len is the buffer size, measured in bytes.

Returns If successful, the value *len* is returned and all *len* of bytes of data from **buf* will have been accepted for transmission.

On failure, the error status *-1* is returned.

The error indicator for Telnet descriptor *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELNOSESS	A client session has not been opened.
KN_ERTELNOCONN	There is no socket connection.
KN_ERTELBUSY	Some prior command has not yet been transmitted.
KN_ERTELNOSPC	Command length exceeds send buffer size.
else	TCP socket error encountered.

Restriction The request will be rejected if all *len* bytes cannot be accepted for transmission at the time of the call. The command will therefore be rejected if it exceeds the configured size of the send buffer. It will also be rejected if a prior command is buffered waiting for transmission because the TCP socket is currently full.

Note Procedure *kntn_sendraw()* must only be used to send properly formatted Telnet command strings. Any IAC character which is not part of a Telnet command sequence must appear in the data buffer as an IAC IAC pair. The *len* bytes of data are sent without modification to the sender's peer.

See Also *kntn_sendcmd()*, *kntn_send()*

Purpose **Generate a Statistics Summary for a Telnet Client, Server or Session**

Used by ■ Client Task □ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int kntn_showstat(KN_TELND telnd);
```

Description *Telnd* is the Telnet descriptor identifying the Telnet client, server or client session for which a statistics summary is to be generated. The client session can be that of a KwikNet Telnet client or any of those being serviced by a KwikNet Telnet server.

The current statistics for the particular Telnet entity will presented as a sequence of strings for logging or display by the KwikNet Telnet client or server handling the client session. The strings are presented in a series of calls with opcode *KN_TELCBC_LOG* to the client or server callback function.

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

On failure, the error status *-1* is returned.

The error indicator for session *telnd* is set to define the reason for failure. Use *kntn_errno()* to retrieve the error code.

KN_ERTELND	The Telnet descriptor *telnd* is invalid.
KN_ERTELREJECT	Operation invalid from a client callback function.
KN_ERTELNOLOG	Logging by the Telnet server (client) is not enabled.
KN_ERTELNOCBF	The Telnet client has no callback function.

Note If Telnet server (client) logging is not enabled in the KwikNet Library, a Telnet statistics summary for a server (client), or for a client session being handled by a server (client), will not be generated.

If Telnet server logging is enabled, the server callback function will be called to generate the server or client session statistics summary as a sequence of messages of class *KN_PA_S_TELN*.

If Telnet client logging is enabled and a callback function is provided, that function will be called to generate the client or client session statistics summary as a sequence of messages of class *KN_PA_C_TELN*.

Restriction If a server callback function initiates a client session statistics summary, the log will be generated by the Telnet server at the first idle opportunity.

Purpose **Create an Instance of a Telnet Server**

Used by □ Client Task □ Client Callback ■ Server Task □ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int knts_create(KN_TELND *telndp,
            void (*cbfn)(KN_TELND telnd, int opcode,
                         char *p1, int p2, void *param),
            void *param);
```

Description *Telndp* is a pointer to storage for the Telnet descriptor which will be assigned by KwikNet to identify the Telnet server.

Cbfn is a pointer to the server callback function which the Telnet server will call to report Telnet session activity. This function must be coded to operate as described in Chapter 1.6.

Param is a parameter which the caller wishes to pass to the callback function *cbfn()*. If no parameter is required, set *param* to *NULL*.

Returns If successful, a value of *0* is returned. A valid Telnet descriptor is stored at **telndp*. This descriptor must be used in all subsequent calls by the Telnet server task to identify the particular Telnet server.

The Telnet server is created with the following default KwikNet Telnet attributes in effect:

7-bit ASCII data transfers
Character echo disabled
Suppress-GA to be enabled at both ends of connection if possible
Line control characters set per Figure 1.2-3

On failure, the error status *-1* is returned and **telndp* is undefined.

If a Telnet server cannot be created, the error indicator defining the reason for failure cannot be recorded. You cannot use *kntn_errno()* to retrieve the error code since you have no Telnet descriptor to interrogate. The most probable reason for failure is that sufficient memory is not available for use by the Telnet server.

Note If Telnet server logging is not enabled in the KwikNet Library, Telnet logging will not occur.

If Telnet server logging is enabled in the KwikNet Library, the callback function will be called to log KwikNet messages describing Telnet server session activity.

See Also *kntn_delete()*, *knts_start()*, *knts_status()*

Purpose **Start a KwikNet Telnet Server**

Used by □ Client Task □ Client Callback ■ Server Task □ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int knts_start(KN_TELND telnd);
```

Description *Telnd* is the Telnet descriptor identifying a KwikNet Telnet server created with a prior call to procedure *knts_create()*.

Note Before the KwikNet Telnet server is started, you must identify the network IPv4 address and port number which it is to serve. When a KwikNet Telnet server is created, it is assigned the IP address *INADDR_ANY* and the well known Telnet port number 23. Hence, clients will be able to connect to the Telnet server using the IP address of any network interface in the computer on which the Telnet server is operating. If you wish, you can use service procedure *kntn_ioctl()* (network parameters) to alter the server's IPv4 address and port number prior to starting the server.

Returns (Multitasking Operation)

This procedure must only be called by the application task which will assume the role of Telnet server. Usually that is the task which created the server instance by calling procedure *knts_create()*. All KwikNet Telnet server operations will be performed in the context of this task.

If the server is successfully started, there will be no return from this procedure until the KwikNet Telnet server is requested to stop. At that time there will be a return to the *knts_start()* caller.

A value of *0* is returned if, and only if, the KwikNet Telnet server starts successfully and eventually stops without error.

On failure, the error status *-1* is returned. Failure indicates that the KwikNet Telnet server cannot be started, was forced to abort because of a serious, unrecoverable fault or terminated with an error condition when requested to stop.

...more

Returns (Single Threaded Operation)

This procedure must be called from your App-Task. The Telnet server will be added to the KwikNet server queue. Thereafter, all KwikNet Telnet server operations will be performed in the KwikNet domain in the context of the KwikNet Task.

A value of *0* is returned if the KwikNet Telnet server is successfully started.

If the KwikNet Telnet server cannot be started, the error status *-1* is returned.

See Also *knts_status()*, *knts_stop()*

Purpose **Fetch the Status of a KwikNet Telnet Server**

Used by □ Client Task □ Client Callback ■ Server Task ■ Server Callback □ ISP □ Timer Procedure

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int knts_status(KN_TELND telnd);
```

Description *Telnd* is the Telnet descriptor of an active Telnet server.

Returns If successful, a value of *n >= 0* is returned where *n* is the total number of Telnet clients currently being served by the Telnet server. A Telnet client is included in the count even if its initial option negotiation sequence is in progress but has not yet completed.

On failure, the error status *-1* is returned. Failure indicates that Telnet descriptor *telnd* does not reference an active Telnet server.

See Also *knts_start()*, *knts_stop()*

Purpose **Stop a KwikNet Telnet Server**

Used by □ Client Task □ Client Callback □ Server Task ■ Server Callback □ ISP □ Timer Procedure ■ Application Task

Setup Prototype is in file *KN_TELN.H*.

```
#include "KN_LIB.H"
#include "KN_TELN.H"
int knts_stop(KN_TELND telnd,
              void (*shutdownfn)(int, unsigned long),
              unsigned long param);
```

Description *Telnd* is the Telnet descriptor of the active Telnet server which is to be stopped.

Shutdownfn is a pointer to a stoppage function which the Telnet server will call when it has shut down and is about to return to the point at which a call to *knts_start()* started the server. The stoppage function will execute in the context of the stopped Telnet server.

If you do not require notification when the Telnet server shutdown is complete, set *shutdownfn* to *(void (*)(int, unsigned long))0L*.

Param is a parameter which will be passed to your stoppage function *shutdownfn()*. If a stoppage function is not provided, set *param* to *0L*. The call to the stoppage function is of the form:

```
(*shutdownfn)(error, param);
```

Parameter *error* is the last KwikNet error code, if any, recorded by the KwikNet Telnet server before it stopped operating.

Returns A value of *0* is returned if, and only if, the KwikNet Telnet server accepts the request, thereby acknowledging that the Telnet server is willing to shut down. The value *0* will be returned even if the KwikNet server eventually terminates with an error condition.

On failure, the error status *-1* is returned. Failure indicates that Telnet descriptor *telnd* does not reference an active Telnet server.

Note If one or more Telnet client connections are open at the time a server is stopped, all such client sessions will be closed before the server stops.

See Also *knts_start()*, *knts_status()*

This page left blank intentionally.