

KwikNet[®]

TCP/IP Stack

USER'S GUIDE

First Printing: May 15, 1998

Last Printing: September 15, 2002

Manual Order Number: PN303-9

Copyright © 1997 - 2002

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 for a period of one year. After that period, you may purchase additional updates. Please keep us informed of the primary user in your company to whom these 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 without charge. 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

**Copyright © 1997-2002 by KADAK Products Ltd.
All rights reserved.**

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of KADAK Products Ltd., Vancouver, BC, CANADA.

DISCLAIMER

KADAK Products Ltd. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties or merchantability or 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.

Copyright Notice

The KwikNet TCP/IP Stack is derived from the University of California's Berkeley Software Distribution (BSD). Some components have been adapted from software made available by the Massachusetts Institute of Technology and Carnegie Mellon University. Use of this software requires the following software copyright acknowledgements.

Copyright © 1982, 1986 Regents of the University of California All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright © 1988, 1989 Carnegie Mellon University All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of CMU not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

CMU DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL CMU BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

This page left blank intentionally.

KwikNet TCP/IP Stack User's Guide

Table of Contents

	Page
1. KwikNet Overview	1
1.1 Introduction	1
1.2 General Operation	3
KwikNet Operation	5
Multitasking Operation	6
Single Threaded Operation	7
The Single Threaded Server Queue	8
1.3 KwikNet Nomenclature.....	9
1.4 Byte Ordering and Endianness	10
1.5 Memory Allocation Requirements	12
Memory Heap Assignment	12
Memory Acquisition Function	12
Memory Allocation Protection.....	13
1.6 KwikNet Data Logging Service	14
Message Formatting.....	14
Message Print Attributes.....	15
KwikNet Data Log Function	16
1.7 KwikNet Message Recording Service.....	17
1.8 KwikNet Console Driver.....	18
Serial I/O Terminal as the Console Device	19
PC Display/Keyboard as the Console Device	19
Telnet as the Console Device.....	19
AMX Console Devices	19
1.9 Debugging Aids	20
Debug Logging	20
Breakpoint Traps.....	21
Fatal Errors	21
Debug Mask.....	22
1.10 KwikNet TCP/IP Sample Program - A Tutorial.....	23
Startup.....	24
Client - Server Using TCP Sockets	24
Client - Server Using UDP Sockets	26
Logging	27
Shutdown	27
Running the TCP/IP Sample Program	28

KwikNet TCP/IP Stack User's Guide

Table of Contents (continued)

	Page
2. KwikNet System Configuration	29
2.1 Introduction	29
KwikNet Libraries	29
Network Configuration Module	31
2.2 KwikNet Configuration Builder	33
Starting the Builder	33
Screen Layout	34
Menus.....	35
Field Editing.....	36
Add, Edit and Delete KwikNet Objects	37
2.3 KwikNet Library Parameter File	39
Target Parameters	40
OS Parameters.....	42
IP Stack Parameters	45
TCP Stack Parameters.....	48
Ethernet / SLIP Parameters	50
Modem Parameters	52
DNS / DHCP Client Parameters	54
Debug and Trace Parameters	57
2.4 KwikNet Network Parameter File	59
General Application Parameters	60
Ethernet Network Definition.....	62
SLIP Network Definition	64
Network Device Driver Definition.....	66
Network IP Address Definition.....	68
Modem Options	70
 3. KwikNet System Construction	 73
3.1 Building an Application	73
3.2 Making the KwikNet Libraries.....	74
KwikNet Directories and Files.....	76
Getting Ready	76
Network Library Make File	77
Gathering Files.....	77
Creating the KwikNet Libraries	78
Generated KwikNet Library Modules.....	79
3.3 Compiling the Network Configuration Module	80
3.4 Compiling Application Modules	81
3.5 Linking the Application	82
3.6 Making the TCP/IP Sample Program.....	83
TCP/IP Sample Program Directories	83
TCP/IP Sample Program Files	84
TCP/IP Sample Program Parameter Files	85
TCP/IP Sample Program KwikNet Libraries	85
The TCP/IP Sample Program Make Process.....	86

KwikNet TCP/IP Stack User's Guide

Table of Contents (continued)

	Page
3. KwikNet System Construction (continued)	87
3.7 Using KwikNet with AMX	87
3.7.1 AMX System Configuration	87
KwikNet Task	87
AMX Interrupt Stack	88
KwikNet Semaphores	88
KwikNet Memory Pool.....	88
KwikNet Timer	88
KwikNet Restart and Exit Procedures.....	89
AMX 86 and AMX 386/EP PC Supervisor	89
3.7.2 AMX Target Configuration.....	90
32-Bit AMX Systems.....	90
16-Bit AMX 86 Systems.....	90
3.7.3 Toolset Considerations.....	91
Tailoring Files	91
Compiler Configuration Header File.....	91
OS Interface Make File.....	91
3.7.4 AMX Application Construction Summary.....	92
 4. KwikNet IP/UDP Services	93
4.1 The UDP Programming Interface.....	93
The UDP Channel	93
Receiving UDP Datagrams	94
Processing Received UDP Datagrams	95
Broadcast UDP Datagrams	95
UDP Echo Requests.....	95
UDP Sockets	95
4.2 The DHCP (BOOTP) Client	96
DHCP Operation	96
DHCP Timeout	97
DHCP Leases	97
DNS Server Support	97
DHCP Option Request.....	97
4.3 The DNS Client.....	98
DNS Server List.....	98
DNS Queries	99
DNS Name Lookup.....	99
Get Host By Name	100
Interpreting DNS Results	101
4.4 ICMP Protocol	102
ICMP Destination Unreachable Hook.....	102
Using PING.....	103
Initiating a Ping.....	103
Handling a Ping Reply	104
4.5 KwikNet State Management	105
Network States	105
KwikNet States	106
4.6 KwikNet IP and UDP Library Services.....	107

KwikNet TCP/IP Stack User's Guide
Table of Contents (continued)

	Page
5. KwikNet TCP/IP Sockets	149
5.1 Introduction to KwikNet Sockets	149
KwikNet Procedure Descriptions.....	149
KwikNet Sockets API	150
Socket Addresses	150
Non-Blocking Sockets	151
KwikNet Error Codes	151
5.2 Socket Types	152
Stream Socket (for TCP).....	152
Datagram Socket (for UDP).....	152
Using UDP Sockets.....	153
UDP Sockets Examples	154
5.3 Socket Options	155
Unsupported Socket Options.....	157
5.4 KwikNet Socket Services.....	159

KwikNet TCP/IP Stack User's Guide

Appendices

	Page
Appendix A. Reference Materials and Glossary	A-1
A.1 Reference Materials.....	A-1
Books	A-1
Internet Sources	A-1
A.2 KwikNet Glossary.....	A-3
 Appendix B. KwikNet Error Codes	 B-1
 Appendix C. KwikNet Universal File System Interface	 C-1
C.1 Introduction	C-1
C.2 KwikNet File System Parameters	C-2
C.3 Using the AMX/FS File System	C-4
C.4 Using the MS-DOS File System	C-6
C.5 Using a Custom File System	C-7
 Appendix D. KwikNet Administration Interface	 D-1
D.1 Introduction	D-1
User Definitions	D-1
User Access Rights	D-2
Customizing Administration Services.....	D-2
D.2 KwikNet Administration Parameters	D-3
 Appendix E. KwikNet Sample Program Architecture	 E-1
Console Interface	E-1
KwikNet Sample Program Operation with AMX	E-3
KwikNet Porting Kit Sample Program - Multitasking Operation	E-5
KwikNet Porting Kit Sample Program - Single Threaded Operation	E-7

KwikNet TCP/IP Stack User's Guide

Table of Figures

	Page
Figure 1.2-1 KwikNet Application Block Diagram	4
Figure 1.2-2 KwikNet Operation	5
 Figure 2.1-1 Creating the KwikNet Network Library Make File.....	 30
Figure 2.1-2 Creating the KwikNet Network Configuration Module	32
Figure 2.2-1 Configuration Manager Screen Layout.....	34
 Figure 3.2-1 KwikNet Library Construction.....	 75
Figure E-1 KwikNet Sample Program Procedures.....	E-2

This page left blank intentionally.

1. KwikNet Overview

1.1 Introduction

The KwikNet[®] TCP/IP Stack is a compact, reliable, high performance TCP/IP stack, well suited for use in embedded networking applications.

The KwikNet TCP/IP Stack includes a complete complement of protocols, some of which are optional. You can readily tailor the KwikNet stack to accommodate your needs by using the KwikNet Configuration Builder, a Windows[®] utility which makes configuring KwikNet a snap. Your KwikNet stack will only include the features required by your application.

KwikNet is best used with a real-time operating system (RTOS) such as KADAK's AMX[™] Real-Time Multitasking Kernel. However, KwikNet can also be used in a single threaded environment without an RTOS.

When used with the AMX multitasking kernel, KwikNet is delivered to you ready for use on a particular target processor with any of the software development tools which KADAK supports for that target. You can concentrate on your application, knowing that the integration of KwikNet with AMX is fully functional.

KwikNet can also be provided in a form most suitable for porting to your own operating system, target hardware and software development tools. The KwikNet Porting Kit permits KwikNet to be used with your own in-house RTOS or with the commercial RTOS of your choice. The kit includes an RTOS example illustrating the use of KwikNet with a custom RTOS and three examples of single threaded use: one for MS-DOS, one for the Tenberry DOS/4GW DOS Extender and one for a custom operating system. Detailed porting instructions are provided in the KwikNet Porting Kit User's Guide.

This manual makes no attempt to describe TCP/IP, what it is or how it operates. It is assumed that you have a working knowledge of the TCP/IP protocol suite as it applies to your needs. Reference materials are provided in Appendix A.

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. It is assumed that you are familiar with the architecture of the target processor. It is further assumed that you are familiar with the rudiments of microprocessor programming including the concepts of code, data and stack separation.

KwikNet is 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.

Manual Overview

This chapter provides an overview of the KwikNet TCP/IP Stack. The general operation of KwikNet is described and the nomenclature used by KADAK is introduced. Appendix A includes a glossary which will help when you are stuck trying to remember what one of the many protocol mnemonics means. A number of topics unrelated to network issues are covered in this chapter. KwikNet memory allocation requirements are examined. KwikNet data logging, message recording, console support and debugging features are also presented. Finally, the KwikNet TCP/IP Sample Program used to exercise KwikNet and illustrate its proper use is described in a tutorial fashion.

Chapter 2 is your system configuration guide. You may wish to read this chapter to learn how easy it is to use the KwikNet Configuration Builder to customize KwikNet for your use.

Chapter 3 describes how KwikNet is configured for use and combined with your application to form an executable load module. It also describes how users of KADAK's AMX Real-Time Multitasking Kernel must adapt their AMX configuration for use with KwikNet. If you are porting KwikNet to your own operating environment, the material provided in this chapter will simply augment the more detailed description presented in the KwikNet Porting Kit User' Guide.

Chapter 4 presents the KwikNet IP/UDP application programming interface (API). This chapter includes descriptions of the lower level (IP and UDP) services which are available for applications which choose not to use the TCP protocol and its socket interface. Topics such as the UDP programming interface, the DHCP and DNS clients and PING are also covered. It also describes common utility procedures which are available for use by applications and device drivers.

Chapter 5 presents the KwikNet socket application programming interface (socket API). It includes an alphabetic summary of all the KwikNet socket procedures at your disposal.

The KwikNet device driver interface is described in the KwikNet Device Driver Technical Reference Manual. The use of the KwikNet Modem Driver and serial device drivers with SLIP or PPP networks is described in that manual.

Optional KwikNet components such as PPP, FTP, HTTP and SNMP are described in separate reference manuals. These components will be of interest only if you have purchased and are using the relevant KwikNet options.

Note

Throughout this manual the term RT/OS is used to refer to any operating system (OS), be it a multitasking RTOS or a single threaded OS.

1.2 General Operation

The KwikNet TCP/IP Stack and your application operate as illustrated in Figure 1.2-1. If you are using KwikNet with AMX, all of the components shown in the block diagram are provided with KwikNet, ready to use with AMX. You simply provide the application.

If you are using the KwikNet Porting Kit to port KwikNet to your operating environment, then the shaded blocks indicate modules which will require modification to adapt KwikNet for use with your application. As you can see, very few modules require adaptation.

The KwikNet TCP/IP Stack sits between your application and the network. In some cases, your application may exclude the TCP stack and interface directly with the UDP and IP stack using the services described in Chapter 4. In most cases, your application will use the KwikNet TCP or UDP socket services presented in Chapter 5. The KwikNet application interface shields you from any direct involvement with the underlying network, device drivers or operating system.

KwikNet includes an operating system interface which makes it suitable for use with or without a real-time operating system. KwikNet is connected to the RT/OS by an OS Interface Module, a C file containing procedures which provide access to the services of the RT/OS. This module is incorporated into the KwikNet IP Library so that it is always available for use by your application.

The KwikNet TCP/IP Stack consists of one or more KwikNet Libraries built according to your specifications to meet your particular needs. The stack interacts directly with one or more KwikNet device drivers, each of which connects KwikNet to a particular network. Each network and its associated device driver is identified in the KwikNet Network Configuration Module. Your KwikNet Libraries and the KwikNet Network Configuration Module are derived from parameter files generated by the KwikNet Configuration Builder (see Chapter 2).

KwikNet communicates with an external network through the device driver which handles the hardware device physically connected to the network. The KwikNet device driver interface is described in the KwikNet Device Driver Technical Reference Manual. In most applications, the device driver is interrupt driven. Only in the simplest of systems can the device driver afford to use a polling strategy. The device driver interface allows KwikNet to call the driver to initiate transmissions on the network. It also allows the driver to signal KwikNet upon receipt of a packet from the network. A separate board driver connects KwikNet, its device drivers and the OS Interface Module to your target hardware in an RT/OS independent manner.

Figure 1.2-1 also shows an application OS interface, a C module used by KADAK to provide a standard interface between the RT/OS and the sample programs (applications) provided with KwikNet and its options. If you port the KwikNet sample programs (and it is recommended that you do so), you will have to use this module. You may also find that portions of this module can, with very little adaptation, be used by your own application.

Finally, the RT/OS must provide a timing source. Although the RT/OS clock driver is shown as a separate component, it may be implemented as an interrupt service routine which resides in the OS Interface Module or in the application OS interface. When AMX is used, your AMX clock driver will generate the fundamental timing needed by KwikNet.

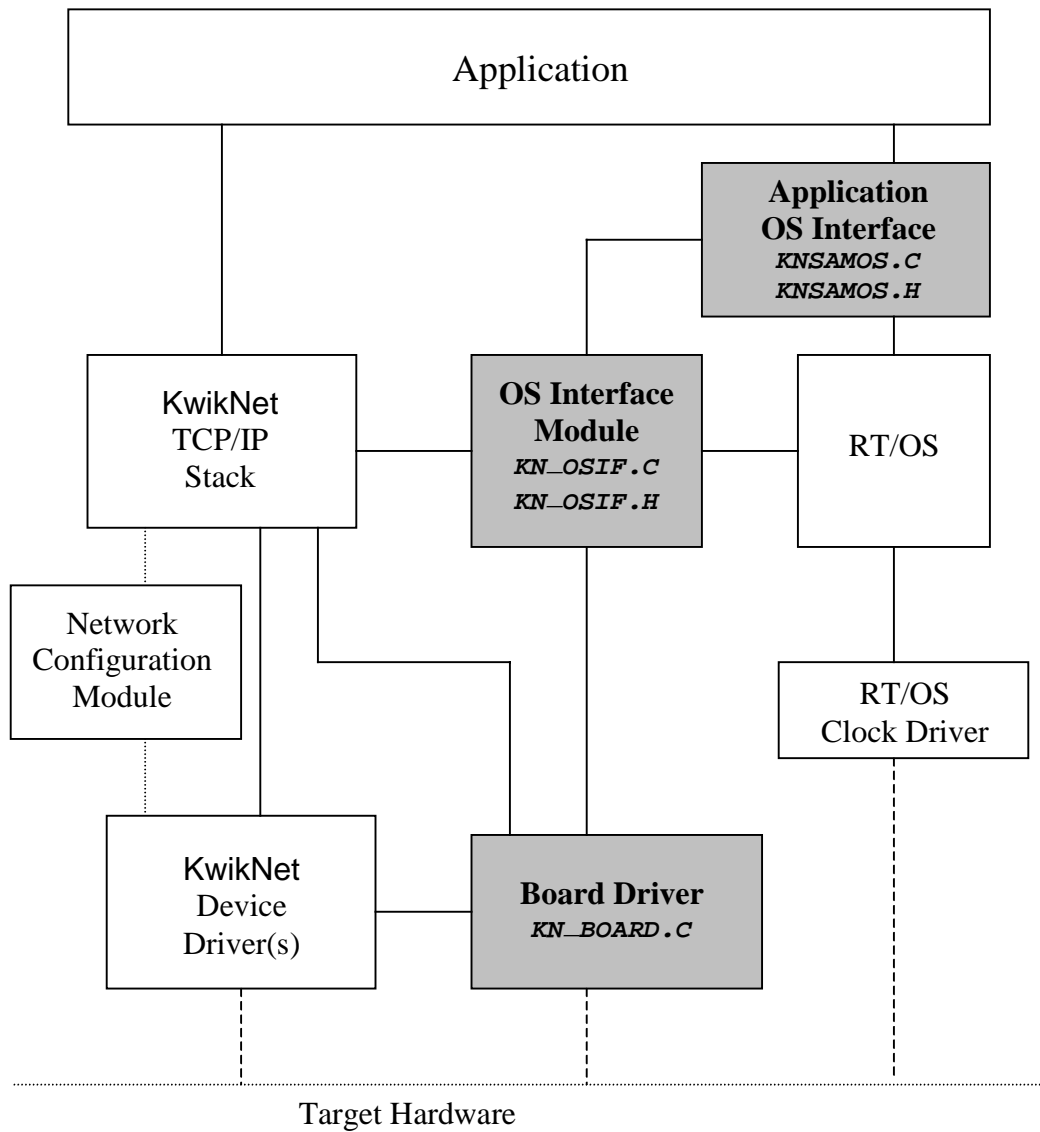


Figure 1.2-1 KwikNet Application Block Diagram

KwikNet Operation

The KwikNet TCP/IP Stack can be used with either a multitasking RTOS or a single threaded operating system. KwikNet and your application operate together as illustrated in Figure 1.2-2. Although the application interface with KwikNet is the same in both cases, the way it executes is quite different.

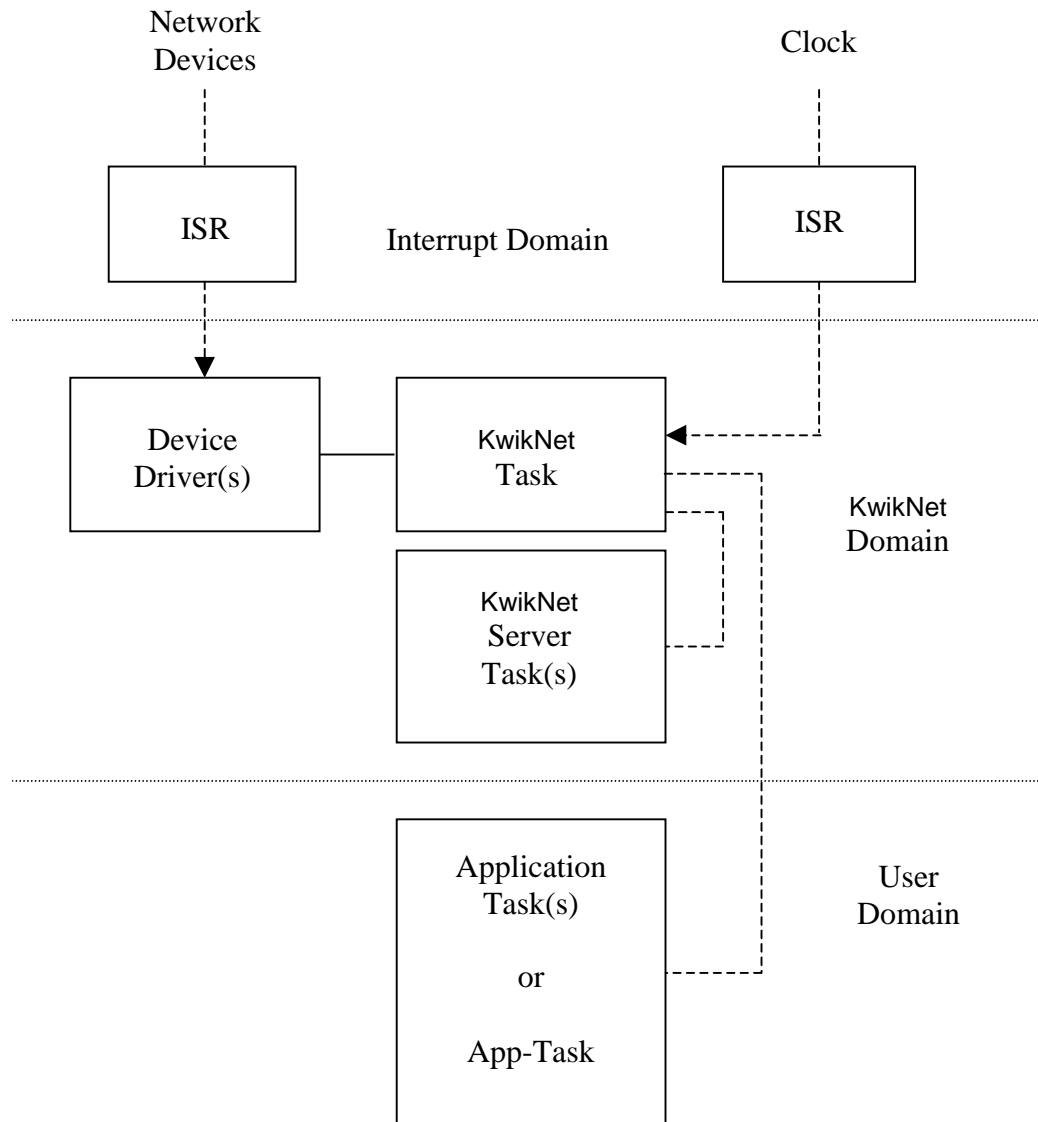


Figure 1.2-2 KwikNet Operation

Multitasking Operation

In a multitasking system which uses an RTOS, operation of the TCP/IP stack is controlled by a single task called the KwikNet Task. This task begins execution after your application calls procedure `kn_enter()` to start KwikNet. The KwikNet Task executes at a priority above that of all other tasks which use KwikNet services.

The KwikNet Task receives timer ticks from the RTOS through the KwikNet OS interface. These ticks, KwikNet's fundamental timing source, occur at the frequency which you specify when you configure your KwikNet Libraries.

KwikNet uses its Ethernet, SLIP or PPP network driver to interact with a particular network's device driver. The KwikNet Task and the device driver cooperate to ensure that network packet transmission and reception occur in a timely fashion. Interrupts generated by the device's hardware interface are serviced by an RTOS compatible interrupt service routine (ISR) which calls the device driver's interrupt handler.

Your application tasks interact with KwikNet using the UDP or IP programming interface described in Chapter 4 or the TCP or UDP socket services presented in Chapter 5.

At times, KwikNet may be forced to suspend your application task pending completion of a requested service. How this is done depends on the *magic* of the KwikNet OS interface. In a multitasking environment, only the task requesting service is suspended. Other tasks are free to execute and use KwikNet services. KwikNet and its OS interface resolve the problems, if any, which may occur when multiple tasks make conflicting demands on the use of the stack.

If you use any of the optional KwikNet components such as the FTP Server or Web Server, you will observe that these servers are also implemented as tasks running under the auspices of the RTOS. These tasks will be of lower priority than the KwikNet Task but will usually be of higher priority than your application tasks which use KwikNet services.

Finally, note that most applications will probably include one or more tasks of higher priority than the KwikNet Task. These tasks, although critical for the success of your application, must not starve the KwikNet Task's demands for execution time.

Note

All application tasks which use KwikNet services **MUST** execute at a priority below that of the KwikNet Task.

Single Threaded Operation

Single threaded operation is only supported by KwikNet if you are using the KwikNet Porting Kit.

In a single threaded system, there is a single application task which, for reference purposes, is called the **App-Task**.

The App-Task starts in your *main()* function and executes in what will be referred to as the **user domain**. Once the App-Task starts, the thread of execution is sequential, flowing back and forth between your application and KwikNet. When KwikNet code is being executed, your application is said to be in the **KwikNet domain**.

An interrupt can occur while executing in either the user domain or the KwikNet domain. When an interrupt occurs, an interrupt service routine (ISR) begins execution in what is called the **interrupt domain**. All interrupts, even if nested, are serviced in the interrupt domain. When service of an interrupt is finished, execution resumes in the domain which was in effect when the interrupt occurred.

Operation of the TCP/IP stack is controlled by a single body of KwikNet code which, by definition, executes in the KwikNet domain. This body of code is called the KwikNet Task, to distinguish it from your App-Task.

Your App-Task controls the flow of execution within your application. The KwikNet Task can only execute when your App-Task permits. The KwikNet Task does not begin until your application calls procedure *kn_enter()* to start KwikNet.

Once KwikNet has been started, your App-Task must regularly call KwikNet procedure *kn_yield()* to let the KwikNet Task service the TCP/IP stack. Procedure *kn_yield()* is included in the KwikNet IP Library and is described in Chapter 4.6.

The KwikNet Task receives timer ticks from the clock ISR through the KwikNet OS interface. These ticks, KwikNet's fundamental timing source, occur at the frequency which you specify when you configure your KwikNet Libraries. For best performance, your App-Task should yield to the KwikNet Task at this frequency or higher.

KwikNet uses its Ethernet, SLIP or PPP network driver to interact with a particular network's device driver. The KwikNet Task and the device driver cooperate to ensure that network packet transmission and reception occur in a timely fashion. Interrupts generated by the device's hardware interface are serviced by an interrupt service routine (ISR) which calls the device driver's interrupt handler.

Warning

Your application **MUST** regularly yield to the KwikNet Task by calling procedure *kn_yield()*. Failure to yield at least at the defined KwikNet clock frequency may result in poor performance of the TCP/IP stack.

Your App-Task can interact with KwikNet using the UDP or IP programming interface described in Chapter 4 or the TCP or UDP socket services presented in Chapter 5.

At times, KwikNet will be forced to suspend your App-Task pending completion of a requested service. How this is done depends on the *magic* of the KwikNet OS interface which ensures that the KwikNet Task continues to service the TCP/IP stack while waiting for the event of interest. Execution continues in the KwikNet domain until the service completes and KwikNet returns to the App-Task in the user domain.

If your App-Task makes requests for non-blocking data transfers across high speed Ethernet networks, it must increase the frequency at which it yields to the KwikNet Task in order to achieve optimum network performance.

The Single Threaded Server Queue

In a single threaded system, the KwikNet Task maintains a server queue to support the optional KwikNet components such as the FTP Server or Web Server. The KwikNet Task periodically executes each of the servers on its server queue, thereby letting these servers operate much as they would in a multitasking system.

Your application can append its own servers to the KwikNet server queue, adding a very primitive form of non-preemptive multitasking to an otherwise single threaded system.

The KwikNet TCP/IP Sample Program illustrates the process. The App-Task calls procedure `kn_addserver()` to add a TCP server to the KwikNet Task server queue. The App-Task then acts as a client using TCP to communicate with the server. When the sample is finished, the TCP server removes itself from the server queue.

Procedure `kn_addserver()` is included in the KwikNet IP Library and is described in Chapter 4.6.

1.3 KwikNet Nomenclature

The following nomenclature standards have been adopted throughout this manual.

Numbers used in this manual are decimal unless otherwise indicated. Hexadecimal numbers are indicated in the format *0xABCD*.

The terminology *A(Table XYZ)* is used to define addresses. It is read as "the address of Table XYZ".

Read/write memory is referred to as RAM. Read only memory (non-volatile storage) is referred to as ROM.

KwikNet symbol names and reserved words are identified as follows:

<i>kn_pppp</i>	KwikNet C procedure name <i>pppp</i>
<i>knxtttt</i>	KwikNet structure name of type <i>tttt</i>
<i>xttttyyy</i>	Member <i>yyy</i> of a KwikNet structure of type <i>tttt</i>
<i>KN_ssssss</i>	Reserved symbols defined in KwikNet header files
<i>KN_EReeee</i>	KwikNet Error Code <i>eeee</i>
<i>KN_WReeee</i>	KwikNet Warning Code <i>eeee</i>
<i>KN_FEeeee</i>	KwikNet Fatal Error Code <i>eeee</i>
<i>KN_FFFFF.xxx</i>	KwikNet reserved filenames
<i>kkkkFFFF.xxx</i>	KwikNet reserved filenames for the following protocols <i>kkkk</i> (standard: <i>IP, TCP, UDP, SLIP, MDM, DNS, DHCP</i> client) (options: <i>PPP, FTP, HTTP, SNMP, SMTP</i> , etc.)
<i>KNnnnFFF.xxx</i>	KwikNet target and toolset specific filenames
<i>KNZZZFFF.xxx</i>	KwikNet filenames for application portability
<i>KN_LIB.H</i>	KwikNet Library Configuration Module (an include file)

The *nnn* in a KwikNet filename is the 3-digit KwikNet part number used by KADAK to identify a particular version of KwikNet. For example, file *KN713CM.EXE* is the KwikNet Configuration Manager provided with the KwikNet Porting Kit which is identified by KADAK part number 713.

Files with names of the form *KNZZZFFF.xxx* are intended to make KwikNet less sensitive to the environment in which it is used. For example, the KwikNet compiler configuration header file *KNZZZCC.H* is used to identify the particular characteristics of the compiler being used to construct your KwikNet application.

File *KN_LIB.H* is the KwikNet Library Configuration Module, an include file which includes the subset of KwikNet header files needed for compilation of your application C code. By including file *KN_LIB.H* in your source modules, your KwikNet application becomes readily portable to other target processors.

Throughout this manual examples are provided in C. Code examples are presented in lower case. File names are shown in upper case. C code assumes that an *int* is 32 bits on 32-bit processors or 16 bits on 16-bit processors as is common for most C compilers.

1.4 Byte Ordering and Endianness

To use a TCP/IP stack, you must adhere to the byte ordering rules defined by the TCP/IP protocol suite. Doing so is complicated by the fact that not all target processors follow these rules.

The TCP/IP network uses the big endian model for byte ordering. Sequential bytes in the data stream always appear in sequential bytes in memory. The initial byte of the data stream is at the lowest memory address and the final byte is at the highest memory address. The network over which the transfer takes place is said to be big endian.

If a multi-byte value appears within the TCP/IP data stream, the most significant byte of that data value will always appear first in the stream. When stored in memory, the most significant byte of a multi-byte value will always appear at the lower memory address with successive bytes of the value stored at sequential, higher memory addresses. Data which resides in memory in this fashion is said to be in net endian form.

When TCP/IP is used on a processor such as the Motorola 68xxx, the network endianness matches the processor endianness. Both are big endian. Consequently, the natural storage of multi-byte values in memory matches that required by the TCP/IP data stream.

However, when TCP/IP is used on a processor such as the Intel 80x86, the network endianness conflicts with the processor endianness. The network requires big endian values in the data stream but the processor's natural storage of multi-byte values in memory is little endian.

Fortunately the KwikNet TCP/IP Stack can be used on target processors which are either big or little endian. The endianness of the target processor is a configuration parameter in the KwikNet Library Parameter File used in the construction of the KwikNet Libraries as described in Chapter 2.3.

Although KwikNet may be used with either big or little endian processors, it does not alleviate your application from the responsibility for correct presentation or interpretation of data delivered over the TCP/IP network. KwikNet treats your application data as a byte stream with no particular endian characteristics. It is up to your application to present the data for delivery to a remote destination in a byte ordered format that the remote end can understand.

For example, two little endian machines can send and receive data streams containing multi-byte values ordered in little endian form. The multi-byte values can be directly fetched from or stored into memory. However, if one of these machines is replaced by a big endian machine, suddenly the application will fail even though KwikNet will continue to successfully deliver the data streams between the two machines. Of course, the newer big endian machine could be reprogrammed to properly store and retrieve the little endian values expected by the other machine.

This example illustrates the absolute need for applications to agree upon the manner in which data values will be delivered to each other. Conventional wisdom suggests that if multi-byte values are always stored in net endian form, then any machine can participate in the conversation, regardless of the machine's endianness.

Net Endian Data

KwikNet provides a set of macros (or functions) which can be used by applications to convert 2-byte (short) values and 4-byte (long) values between net endian form and host endian form. These macros assume the host endianness defined in the KwikNet Library Parameter File.

The following macros are available in the KwikNet IP Library.

```
nlv = htonl(hlv)   Convert long from host to network endian form
nsv = htons(hsv)   Convert short from host to network endian form
hlv = ntohl(nlv)   Convert long from network to host endian form
hsv = ntohs(nsv)   Convert short from network to host endian form
```

On big endian machines, these macros leave the data parameter unaltered since the network is also big endian. On little endian machines, these macros reverse the order of the bytes in the macro parameter.

It should be obvious that *htonl* and *ntohl* are equivalent as are *htons* and *ntohs*. So which macro should be used if two are identical? Although it does not matter, it is recommended that the macro be chosen for best meaning in the context of its use. For example, when storing a *long* value *hlv* into memory for delivery in the data stream, use *htonl*(*hlv*) to indicate the conversion of the data from host to net endian form. Similarly, when fetching a *long* value *hlv* from a received data packet, use *ntohl*(*nlv*) to indicate the conversion of the data from net to host endian form.

So how does your application know which data values require conversion? There is no simple answer. The content of the data portion of any packet delivered on the network is known only to the sender and receiver. Both must agree to the method of interpretation.

Of greater concern is the management of the data while it is under the control of your application. Most hosts prefer to operate with data in the natural form dictated by the target processor. For this reason, data is usually converted to net endian form prior to sending and from net endian form upon receipt.

There are several data values which applications frequently use which, by convention, are always maintained in net endian form. These are network parameters such as IP addresses, subnet masks and default gateway addresses. It is good programming practice to always use comments to identify variables which are assumed to be kept in net endian form. All other variables can then safely be assumed to be in host endian form.

1.5 Memory Allocation Requirements

KwikNet must be able to dynamically allocate and free blocks of memory of varying sizes. KwikNet supports two memory allocation strategies, both of which are implemented in the OS Interface Module.

The first strategy uses standard C library functions *calloc()* and *free()* to allocate and free memory. The second alternative is to use the memory allocation services provided by the operating system (RT/OS) with which KwikNet is being used.

The KwikNet Library must be configured to select one of the two memory allocation strategies. The strategy is defined by the parameters in your Library Parameter File. The choices are made on the OS property page using the KwikNet Configuration Builder (see Chapter 2.3).

When KwikNet is used with the AMX Real-Time Multitasking Kernel, either strategy can be used. The examples provided with the KwikNet Porting Kit support standard C, but can be easily modified to use memory allocation services in your RT/OS.

Memory Heap Assignment

You may be using an RT/OS which requires a fixed region of memory for use as a heap. KwikNet can be configured so that the memory heap is provided by your application from any of the following sources:

- (1) a static array in the KwikNet Configuration Module,
- (2) an absolute address in memory or
- (3) a region provided by a memory acquisition procedure *kn_memacquire()*.

Memory Acquisition Function

If you choose to dynamically assign the memory heap to your RT/OS, you must provide a memory acquisition function *kn_memacquire()* which KwikNet will call when it first starts. The prototype for this function is as follows:

```
unsigned long kn_memacquire(char **memp);
```

The memory acquisition function must provide access to a fixed region of *n* bytes of memory for use by the RT/OS as a heap. The function must install a pointer to the memory region into the pointer variable referenced as **memp* and return the value *n*, the size of the region.

When KwikNet shuts down it will call a function *kn_memreturn()* which you must provide to dispose of the memory region, if necessary. The prototype for this function is as follows:

```
int kn_memreturn(char *memp);
```

Parameter *memp* is a pointer to the region of memory previously acquired from your *kn_memacquire()* function. If this function successfully handles the memory region disposal, it must return the value 0. Otherwise, it must return a non-zero value.

Memory Allocation Protection

When operating in a multitasking environment, the memory allocation services must be thread-safe. If the services you have chosen to use are not thread-safe, KwikNet can be configured to use the memory locking mechanism in its OS Interface Module to protect access to the unsafe memory allocation services.

The KwikNet sample programs provided for use with AMX use the AMX Memory Manager to allocate memory from a static array located in the KwikNet Configuration Module. These AMX memory allocation services are inherently thread-safe.

When KwikNet is used with AMX and standard C is used for memory allocation, you must enable the memory locking protection in the KwikNet OS Interface to protect the unsafe services in the C library.

The examples provided with the KwikNet Porting Kit are ready for use with standard C. If you port KwikNet to a multitasking RTOS, be sure to implement the memory protection mechanism or provide access to thread-safe services within the RTOS.

1.6 KwikNet Data Logging Service

Like most TCP/IP stacks, KwikNet can generate a variety of messages to assist you in your use of the stack. The messages can provide debug information and trace execution of the stack through its various protocol layers. Your application can also generate a statistics summary of event counts by calling KwikNet procedure *kn_netstats()*.

For debug and trace messages to be generated, the KwikNet Libraries must be configured accordingly. Statistics for a particular KwikNet component will only be available for logging if the KwikNet Libraries are appropriately configured. To enable these features, use the KwikNet Configuration Builder to edit your Library Parameter File as described in Chapter 2.3.

Even with the message sources are enabled, KwikNet will only log the messages if your KwikNet Network Parameter File has data logging enabled. This file is edited using the KwikNet Configuration Builder as described in Chapter 2.4.

Edit the Network Parameter File as follows. Go to the Application property page and check the box which enables data logging. On the same page, define the amount of memory you are willing to reserve for KwikNet to use for data logging buffers. Specify the maximum allowable message line length, usually about 80 characters. Finally, enter the name of the data log function to be used by KwikNet to *print* each message. This function will be described shortly.

The KwikNet Libraries and your Network Configuration Module, ready for data logging use, will be generated and linked with your application as described in Chapter 3.

Message Formatting

Many TCP/IP stacks produce these data logging messages using the C library *printf()* function which is often not even available in embedded systems. KwikNet provides its own procedure called *kn_dprintf()* which, although similar to *printf()*, has several special features not found in the latter. This procedure is fully described in Chapter 4.6. The prototype is as follows.

```
int kn_dprintf(int attrib, const char *fmt, ...);
```

Unlike *printf()*, KwikNet's procedure receives a parameter *attrib* which defines the print attributes of the message. This encoded parameter defines the severity of the message, the message class and the message source. These characteristics will be described shortly.

The parameter *fmt* is a pointer to a conventional format string which can be followed by zero or more parameters as required by the format specification. Not all formats are supported. For example, parameters of type *float* and *double* are not permitted. However, a new format "%a" is introduced which greatly simplifies the formatting of network IP addresses in dotted decimal notation.

For a complete specification of the formatting features supported by KwikNet procedure *kn_dprintf()*, see the description of format procedure *kn_fmt()* in Chapter 4.6.

Message Print Attributes

The parameter defining the message print attributes includes three fields of interest to the user: severity level, class and source type. These fields can be isolated using the following symbolic masks defined in KwikNet header file *KN_COMN.H*. All other bits in the parameter are reserved for the private use of KwikNet.

<i>KN_PA_LEVEL</i>	Severity level
<i>KN_PA_CLASS</i>	Message class
<i>KN_PA_TYPE</i>	Source type

The severity levels are defined as follows:

<i>KN_PA_INFO</i>	General information and application messages
<i>KN_PA_WARN</i>	KwikNet warnings
<i>KN_PA_FATAL</i>	KwikNet fatal error messages

The message classes, defined as follows, can be used to identify the device to which the messages should be directed.

<i>KN_PA_APP</i>	General information and application messages
<i>KN_PA_DEBUG</i>	KwikNet debug logging
<i>KN_PA_STATS</i>	KwikNet network statistics
<i>KN_PA_MDMLOG</i>	KwikNet modem event log
<i>KN_PA_PPPLOG</i>	KwikNet PPP protocol traces

The message source types define the module, network, device, protocol layer or service which was executing when the message was generated. The list is extensive and subject to change. The source types, defined in KwikNet header file *KN_COMN.H*, will generally be of little interest to your application. The source types could be used to provide a sub-classification if you wish to archive messages in some manner.

The message print attributes are defined such that an attribute of 0 will always describe an application message of lowest severity and with no known source type. Hence, applications can easily call *kn_dprintf()* with an attribute of 0 to log messages.

KwikNet Data Log Function

When data logging is enabled, the KwikNet message generation procedure *kn_dprintf()* calls the data log function specified in your Network Parameter File. It is the purpose of this function to record (and display or print) the message contained in the KwikNet log buffer which it receives.

The Application OS Interface module *KNSAMOS.C* provided for use with KwikNet sample programs includes a working example of a data log function called *sam_record()*. With some modifications, this procedure may be suitable for use by your application. At the very least, it will provide a good model for you to use.

The data log function must be declared as follows:

```
int sam_record(int attrib, char *bufp, int count);
```

The character buffer referenced by pointer *bufp* is a KwikNet log buffer. It contains a '\0' terminated string. The length of the string in bytes is specified by parameter *count*. Message strings are limited to the line length which you specified in your configuration. The newline character '\n' is used as the end of line indication in all KwikNet messages.

Parameter *attrib* defines the message print attributes. This is the same parameter presented to KwikNet's *kn_dprintf()* procedure. Your log function can decode the message class to determine the device on which the message must be recorded or displayed. It can also decide if any special action is required because of the message severity or source.

Finally, your log function must assume responsibility for the KwikNet log buffer. If your function accepts the log buffer, it must eventually release it by passing the pointer *bufp* to KwikNet procedure *kn_logbuffree()*. In this case, your log function must return the value 0 to KwikNet indicating your acceptance of the log buffer.

If your log function cannot accept the log buffer for some reason, it must return the value -1 to KwikNet. In this case, KwikNet will free the log buffer.

In a multitasking system, the log function should add the log buffer to a message queue for eventual recording (and printing or display) by a print task which services the message queue. The examples provided for use with AMX and with the KwikNet Porting Kit pass the log buffer to a print task which uses the KwikNet message recording service described in Chapter 1.7 to dispose of each message.

In a single threaded system, the log function should add the log buffer to a message queue for eventual recording (and printing or display) by the App-Task. However, if performance is not an issue, the log function can actually record the message and release the log buffer itself. Care must be taken to ensure that such an action is not allowed to occur while executing within the interrupt domain. The examples provided with the KwikNet Porting Kit operate in the latter fashion, using the KwikNet message recording service described in Chapter 1.7 to dispose of each message.

1.7 KwikNet Message Recording Service

Recognizing that embedded systems may not be able to display or print messages, KADAK provides an alternate message recording service. This service is provided in module *KNRECORD.C* which is located in the toolset dependent installation directory *TOOLXXX\SAM_COMN* (see Chapter 3.6).

The KwikNet message recording service, used by all KwikNet sample programs, accepts a message contained in a KwikNet log buffer. The message is copied from the log buffer into a memory array and the log buffer is released.

The messages are stored sequentially in a character array called *kn_records[]*. As each message is recorded, a pointer to the copy of the message is stored into the next available entry in variable *kn_recordlist[]*, an array of string pointers. The list of string pointers is terminated with a *NULL* string pointer. Message recording ceases as soon as either array becomes full.

Procedure *kn_loginit()* in module *KNRECORD.C* must be called by your application before the message recording service can be used by KwikNet. For this reason, your *main()* function should call *kn_loginit()* as one of its earliest operations.

Once the service is ready, procedure *kn_logmsg()* can be called to record a message contained in a KwikNet log buffer. The Application OS Interface module *KNSAMOS.C* used by KwikNet sample programs provides an example. The data log function *sam_record()* in that module ensures that each KwikNet log buffer is eventually delivered to procedure *kn_logmsg()* which records the message and releases the log buffer.

The data recording service can be adapted to your needs by editing the definitions in the sample program's application header file *KNZZZAPP.H*. A unique header file is provided with each KwikNet sample program. Symbol *KN_REC_MEMORY* must be set to 1 to enable recording of messages into character array *kn_records[]*. Symbol *KN_REC_MEMSIZE* defines the size of that array. Symbol *KN_REC_NUM* defines the maximum number of message string pointers which can be recorded into array *kn_recordlist[]*. If symbol *KN_REC_CONSOLE* is set to 1, each recorded message will also be echoed to the KwikNet console driver as described in Chapter 1.8.

Some of the KwikNet sample programs implement a dump command to display the recorded messages. These applications call procedure *kn_loggets()* to extract each message string from the recording array. After displaying all messages in the order in which they were recorded, procedure *kn_loginit()* is called to reset the array.

Also note that some debuggers will allow you to dump the strings in text form in a display window by viewing the array variable *kn_recordlist[]*.

Warning

The procedures in the recording module *KNRECORD.C* are NOT reentrant. Hence, in multitasking systems, you must ensure that, if one task calls any one of these procedures, no other task can execute any of the procedures until that task completes its use of the recording service.

1.8 KwikNet Console Driver

The KwikNet sample programs provide support for a simple, interactive console device. The console driver, module *KNCONSOL.C*, is located in the toolset dependent installation directory *TOOLXXX\SAM_COMN* (see Chapter 3.6). The console driver can be adapted to use any of several possible console devices, including a terminal connected by a serial UART interface, a PC screen and keyboard or a remote Telnet terminal.

To select a particular console device, edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* as instructed in the file. Note that a unique application header file *KNZZZAPP.H* is provided with each KwikNet sample program.

The basic KwikNet TCP/IP Sample Program uses the console device for displaying messages logged by KwikNet and the application. The data recording procedure *kn_logmsg()* in module *KNRECORD.C* echoes each message it receives to the console device. You can disable this display of recorded messages by setting the value of symbol *KN_REC_CONSOLE* to 0 in the sample program's application header file *KNZZZAPP.H*.

Other KwikNet sample programs (FTP Option, Web Server, etc) provide a simple command interpreter which allows you to interact with the program to control its operation. Since the console device is used by the application, it cannot be used by the recording service to display KwikNet messages. Hence, for these programs, symbol *KN_REC_CONSOLE* is defined to be 0 in the sample program's application header file *KNZZZAPP.H*.

The interactive KwikNet sample programs implement a dump command to display the recorded messages. These applications call procedure *kn_loggets()* in module *KNRECORD.C* to extract all of the message strings from the recording array. The extracted messages are displayed on the console device.

Warning

The message recording services are not reentrant. Hence, the dump command implemented by some KwikNet sample programs should only be used when KwikNet is not active since the extraction of messages for display may occur concurrently with the generation of messages by KwikNet.

If you use the Telnet console device, the dump command must be used with caution. Since KwikNet must be active for the Telnet console driver to operate, KwikNet may generate several messages for every message that is dumped, especially if you have enabled most of the KwikNet debug and trace options.

Serial I/O Terminal as the Console Device

The KwikNet sample program includes a UART serial I/O driver which can be used with the KwikNet console driver to provide access to a terminal. The driver supports the INS8250 (NS16550) USART as implemented in PC compatible hardware. To select this device for console driver use, edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVUART*. This serial I/O driver can also be used with the KwikNet sample programs for AMX.

The UART driver *KN8250S.C* is located in the toolset dependent installation directory *TOOLXXX\SAM_COMN* (see Chapter 3.6). Compile the console driver *KNCONSOL.C* and the UART driver *KN8250S.C* and link the resulting object modules with the sample program.

PC Display/Keyboard as the Console Device

When used on PC hardware with MS-DOS, the KwikNet console driver can be directed to use the PC display and keyboard as a terminal. Edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVPC*. The PC display and keyboard can only be used with a C library that supports the non-standard *_putch()*, *_kbhit()* and *_getch()* functions. This console device can also be used with the KwikNet sample programs for AMX 86.

Telnet as the Console Device

If the KwikNet sample program is modified to provide access to a real network, the KwikNet console driver can be directed to use the Telnet protocol to access a remote terminal. Edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVTELNET*. The KwikNet Libraries must have TCP support included. The console driver will listen on the well known Telnet port number 23 for a connection. It then uses the TCP socket to communicate with the remote terminal to which it is connected. The Telnet console device can also be used with the KwikNet sample programs for AMX.

AMX Console Devices

The KwikNet console driver provided with AMX can be used with the KwikNet serial UART driver described above. However, if you have already ported the AMX Sample Program serial I/O driver to your hardware, you can direct the console driver to use it to access a terminal. Edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVAMX*. Compile the console driver *KNCONSOL.C* and link the resulting object module and your AMX serial driver object module with the sample program.

If you are using AMX 86 or AMX 386/EP, the KwikNet console driver can use the AMX PC Supervisor to access the PC display and keyboard as a terminal. Edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVAMXPCS*. Be sure to link the sample program with the AMX PCS Configuration Module and the PC Supervisor Library.

1.9 Debugging Aids

KwikNet includes a number of debug features which, if used effectively, can help you test your networking application. These features can also be used to provide information to KADAK's technical support staff should you require their assistance.

KwikNet's debugging services fall into the following categories: debug logging, breakpoint traps and fatal error detection. Most of these features can only be used to best advantage if your application provides a data log function as described in Chapter 1.6.

Debug Logging

To use KwikNet's debug logging features, you must first build the KwikNet Libraries to include the extra code necessary to detect and record the events of interest. To do so, use the KwikNet Configuration Builder to edit your KwikNet Library Parameter File and view the Debug property page (see Chapter 2.3). Check the box labeled "Enable debug logging". Doing so enables the full range of debug capabilities within the KwikNet Libraries.

You can also choose the specific debug data which you wish KwikNet to log. You simply check the features of interest from the list presented on the Debug property page. As you will learn shortly, you can also dynamically define or alter your selections at runtime.

You also have the option on the Debug property page to enable code tracing as KwikNet executes through any of several higher level protocol layers such as TCP. Tracing through the PPP layers will only be operative if you have purchased and are using the optional KwikNet PPP component. Unlike debug logging features, code trace selections cannot be dynamically adjusted. To change your trace selections, you must edit your Library Parameter File and rebuild your KwikNet Libraries.

Breakpoint Traps

KwikNet can generate a debug trap when it encounters an error condition which is generally not expected in the normal course of events. Such errors are often the result of modifications of private KwikNet data by errant applications which result in decision conflicts which KwikNet cannot resolve.

Other KwikNet debug traps are also included whenever debug logging is enabled.

To use KwikNet's debug trap, you must first build the KwikNet Libraries to include the extra code necessary to generate the trap. To do so, use the KwikNet Configuration Builder to edit your KwikNet Library Parameter File and view the Debug property page (see Chapter 2.3). Check the box labeled "Enable debug breakpoint".

Each debug trap generates a call to the KwikNet breakpoint procedure *kn_bphit()*. When testing your application, you can place a breakpoint on this procedure to trap all errors detected by KwikNet.

Fatal Errors

Some of the errors detected by KwikNet are serious enough to require that KwikNet cease operation. To proceed would risk further corruption and would probably lead to a catastrophic collapse in an unpredictable fashion.

When KwikNet detects such a fatal error, it calls procedure *kn_panic()* which attempts to log a message describing the fault and then stop the RT/OS. When used with AMX, KwikNet forces an AMX fatal exit.

When testing, it is always wise to execute with a breakpoint on procedure *kn_panic()*. If you are using KwikNet with AMX, you should also have a breakpoint on the AMX fatal exit procedure *cjksfatal* (*ajfat1* and *AAFATL* for AMX 86).

Debug Mask

KwikNet maintains a public debug control variable, an unsigned integer named *kn_dbgflags*. The bits in this variable are used to determine which, if any, KwikNet debug data is logged. The bit masks *KN_DBxxxxx* used to access this variable are defined in KwikNet header file *KN_COMN.H*. Most of the bit masks correlate to the debug logging options available on the Debug property page.

If bit *KN_DBENABLE* in variable *kn_dbgflags* is set to 0, all debug logging is disabled. In this case, the remaining bits in the debug mask are ignored.

If bit *KN_DBENABLE* in variable *kn_dbgflags* is set to 1, debug logging is enabled. In this case, data logging will occur for a particular feature only if the bit in the mask corresponding to that feature is set to 1.

If logging is enabled and a warning or fatal message is logged (see Chapter 1.6), the action taken depends on the state of the *KN_DBHALT* bit in variable *kn_dbgflags*. If this bit is set to 1, KwikNet will enter a permanent loop, unconditionally calling its breakpoint procedure *kn_bphit()*.

The debug halt bit *KN_DBHALT* is never set by KwikNet. It is provided to allow you to stop further KwikNet execution when a warning or fatal message is logged.

Note

Although the KwikNet variable *kn_dbgflags* is always present, its content will only have an effect if your application has been linked with a KwikNet Library configured to support debug logging.

1.10 KwikNet TCP/IP Sample Program - A Tutorial

A TCP/IP Sample Program is provided with KwikNet to illustrate the use of the TCP/IP stack within an application. The sample program is ready for use with the AMX Real-Time Multitasking Kernel. The sample program has also been tested with each of the five 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 provides two benefits: the illustration of a very simple device driver and an example of its use for testing purposes when network hardware is not available.

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. However, the sample program provided for use with AMX has been enhanced to eliminate any dependence on specific target hardware. This sample program includes a very low priority task which can detect if you have added a real AMX clock driver to the sample configuration. If a real hardware clock is not available, this task simulates clock interrupts, thereby providing AMX ticks which meet KwikNet's needs.

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

The client and server use the KwikNet sockets application programming interface (API) to communicate. Two scenarios are followed, one after the other.

In the first scenario, the server creates a streaming socket, listens to the socket for a connection request, accepts a message from the client and generates the correct response. The client creates a streaming socket, establishes a connection with the server, sends its request and verifies the proper response.

In the second scenario, the server creates a UDP (connectionless) socket and listens for incoming requests. The server then uses the socket select feature to wait for the availability of a message from the client. The server reads the message, identifies the message source (client) and sends the correct response back to that client. The client creates a UDP socket, sends its request to the server, waits for the availability of a message from the server and verifies the proper response.

Once the final scenario has completed, the client calls KwikNet to log a summary of the network statistics accumulated during the session.

The sample uses the KwikNet message recording service (see Chapter 1.7) to record messages generated by the server, the client and KwikNet. Messages are stored as an array of strings in memory but can be easily echoed to a console terminal (see Chapter 1.8).

Startup

The manner in which the KwikNet TCP/IP 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. Both multitasking and single threaded operation are described in detail.

When used with AMX, the sample program operates as follows. AMX is launched from the *main()* program. Restart Procedure *rrproc()* starts the client task and a print task. A low priority background task is also started to simulate clock interrupts in the absence of a hardware clock.

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

Once the print task is ready and waiting, the 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 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 client resumes execution, starts the server and begins the first of two scenarios.

Client - Server Using TCP Sockets

This example illustrates the use of KwikNet's TCP/IP socket interface to establish a connection between two end points for the reliable transfer of data. Although the end points happen to be tasks running on the same host computer, the actions required by each are still the same as would be required if they resided on separate hosts interconnected by a real network.

The server's first scenario is embodied in function *server1()*. The corresponding function executed by the client is *client1()*.

The **server** calls *kn_socket()* to create a streaming socket. The server calls *kn_setsockopt()* to revise the socket's attributes to permit the eventual reuse of the unique server port number of 5001. It then calls *kn_bind()* to bind itself to the socket, identifying itself as port 5001, but allowing KwikNet to assign its IP address. KwikNet assigns IP address 192.168.1.73, the IP address of the only network present in the sample configuration. The server then calls *kn_listen()* to prime the socket to listen for incoming connection requests. Finally, the server calls *kn_accept()* to wait for such a connection to be established. KwikNet gives the server a new socket which corresponds to the server's end of the connection to its client.

Once the connection with the client has been made, the server uses procedure *kn_recv()* to receive a 4 byte message, a *long* value which defines the length of a block of data which the client intends to send to the server.

The server uses `kn_send()` to echo the 4 byte value back to the client as an acknowledgement that the server is prepared to accept that amount of data from the client. Then the server calls `kn_recv()` to acquire the data block from the client and `kn_send()` to echo the data block back to the client.

Once the data has been echoed to the client, the server uses `kn_shutdown()` to terminate all send operations on its connected socket and then repetitively calls `kn_recv()` until all unexpected data, if any, from the client has been discarded. The connected socket and the socket used for listening are then closed using `kn_close()`.

The **client** calls `kn_socket()` to create a streaming socket. It then calls `kn_bind()` to bind itself to the socket, allowing KwikNet to assign a port number and IP address.

The client then calls `kn_connect()` to connect to the server with port number 5001 and IP address 192.168.1.73. Note that the client has to know the port number of the server to which it is trying to connect. The IP address of the server happens to be the IP address assigned to the Ethernet network in the KwikNet Network Configuration Module linked with the TCP/IP Sample Program. The sample illustrates the use of procedure `kn_inet_addr()` to convert an IP address in dotted decimal form to an equivalent network compatible form.

Once the connection has been established, the client sends the `long` value 26 (defined as `SAMMSGSZ`) as a 4 byte message to the server. The number `SAMMSGSZ` is the number of data bytes which the client intends to send to the server. It happens to be the number of characters in the alphabet, the data which will be sent.

The client then uses procedure `kn_recv()` to receive a 4 byte message, a `long` value which confirms the length of the block of data which the server is willing to accept from the client.

The client then prepares to send the 26 characters of the alphabet to the server. The client does so using the `kn_sendmsg()` procedure which permits the data to be gathered from disjoint locations in memory but delivered as a sequential byte stream. The 26 characters are gathered from the following two strings: 10 from the first and 16 from the second.

```
"ABCDEFGH IJ1234"  
"KLMNOPQRSTUVWXYZ5678"
```

The client then waits for an echo from the server of the data actually received by the server. The client does so using the `kn_recvmsg()` procedure which permits the received data to be scattered into disjoint locations in memory even though received as a sequential byte stream. The data is scattered into a zero filled character buffer: 7 bytes at offset 0, 11 bytes at offset 20 and 8 bytes at offset 40. The three strings at offsets 0, 20 and 40 are then expected to match as follows.

```
"ABCDEFGH"  
"HIJKLMN OPQR"  
"STUVWXYZ"
```

Finally, the client uses `kn_shutdown()` to terminate all send operations on its connected socket and then repetitively calls `kn_recv()` until all unexpected data, if any, from the server has been discarded. The socket is then closed using `kn_close()`.

Client - Server Using UDP Sockets

This example illustrates the use of KwikNet's UDP socket interface to deliver UDP datagrams between two end points. This data transfer mechanism is not considered reliable. Furthermore, since TCP is not used, a logical connection between the end points does not exist. Although the end points happen to be running on the same host computer, the actions required by each are still the same as would be required if they resided on separate hosts interconnected by a real network.

The server's second scenario is embodied in function *server2()*. The corresponding function executed by the client is *client2()*.

The **server** calls *kn_socket()* to create a connectionless datagram socket. The server calls *kn_setsockopt()* to permit the ongoing reuse of the unique server port number of 5001. It then calls *kn_bind()* to bind itself to the socket, identifying itself as port 5001, but allowing KwikNet to assign its IP address. KwikNet assigns IP address 192.168.1.73, the IP address of the only network present in the sample configuration. Once the bind is complete, the server can immediately receive data directed to IP address 192.168.1.73.

The server then uses procedure *kn_select()* to wait until some data from a client is available for reading. The sample illustrates the proper use of the macros *FD_ZERO*, *FD_SET* and *FD_ISSET* for manipulating socket sets.

Once data from the client is available, the server uses *kn_recvfrom()* to learn the client's network address and to receive a 4 byte message, a *long* value, from the client.

The server then uses procedure *kn_select()* to wait until the server's socket is ready to accept data for transmission. The value received from the client is incremented by two and echoed to the client using the *kn_sendto()* procedure to send the 4 byte *long* value.

Once the value has been echoed to the client, the server uses *kn_shutdown()* to terminate all send and receive operations on its socket. The socket is then closed using *kn_close()*.

The **client** calls *kn_socket()* to create a connectionless datagram socket. It then calls *kn_bind()* to bind itself to the socket, allowing KwikNet to assign a port number and IP address.

The client then uses procedure *kn_select()* to wait until the client's socket is ready to accept data for transmission. The arbitrary value 8 is sent to the server using the *kn_sendto()* procedure to send the 4 byte *long* value. The value is sent to the server with port number 5001 and IP address 192.168.1.73.

Note that the client has to know the port number of the server to which it is trying to connect. The IP address of the server happens to be the IP address assigned to the Ethernet network in the KwikNet Network Configuration Module linked with the TCP/IP Sample Program. The sample illustrates the use of procedure *kn_inet_addr()* to convert an IP address in dotted decimal form to an equivalent network compatible form.

The client then uses procedure *kn_select()* to wait until some data from the server is available for reading. The sample illustrates the proper use of the macros *FD_ZERO*, *FD_SET* and *FD_ISSET* for manipulating socket sets. The client then uses procedure *kn_recvfrom()* to receive a 4 byte message, a *long* value from the server and confirms that the received value is 10, the value sent incremented by 2.

Once the echoed value has been received from the server, the client uses *kn_shutdown()* to terminate all send and receive operations on its socket. The socket is then closed using *kn_close()*.

Logging

The KwikNet TCP/IP Sample Program includes a simple recorder for logging 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 application can generate messages 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 sample program uses this feature to record startup and shutdown messages. The client and server record progress messages and log 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.

Shutdown

Once the client and server have completed their second scenario, the client calls KwikNet procedure *kn_netstats()* to record all network statistics gathered by KwikNet during the session.

The client then calls procedure *kn_exit()* to stop operation of the KwikNet TCP/IP Stack. A final completion message is then logged. Note that the KwikNet data logging services continue to be used by the application even though the stack itself has ceased operation. Finally, the client requests the operating system to shut down and return to the *main()* function.

Running the TCP/IP Sample Program

The KwikNet TCP/IP Sample Program load module is built just like any other KwikNet application program. The KwikNet Library Parameter File and Network Parameter File are first created using the KwikNet Configuration Builder as described in Chapter 2. Then the KwikNet Libraries and the Network Configuration Module must be produced. Finally these KwikNet modules must be linked with the application, the RT/OS libraries and the C runtime library to create a load module suitable for testing with a debugger. This construction process is explained in Chapter 3.

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

KwikNet includes a number of debug features (see Chapter 1.9) which will assist you in running the TCP/IP 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* (*ajfatal* for AMX 86).

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

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

Once you are confident that the KwikNet TCP/IP Sample Program is operating properly, you may wish to breakpoint your way through the client and server (functions *clientN()* and *serverN()*), monitoring the recorded messages as you go.

Do not be surprised by the large number of statistics messages generated by the client's call to *kn_netstats()*. If you use the KwikNet Configuration Builder to view the Library Parameter File *KNSAMLIB.UP* you will observe that all TCP and IP statistics gathering options have been enabled.

2. KwikNet System Configuration

2.1 Introduction

Creating an application which uses the KwikNet TCP/IP Stack is a three step process. First, the KwikNet Libraries must be constructed to reflect the options and features which your application will require. Then a Network Configuration Module must be created which describes the networks and devices present in your target hardware. Finally, your application modules must be linked with the KwikNet Libraries and the Network Configuration Module to create a load module suitable for execution in the target processor.

With many portable network stacks, this process requires the tedious and error prone task of editing a collection of files with which you have little familiarity. With KwikNet you simply point and click using the KwikNet Configuration Builder, a Windows[®] utility which greatly simplifies the process. You still have to pick the correct set of options and define your particular network requirements but at least you are concentrating on what you know best, your application.

KwikNet Libraries

The KwikNet Libraries must be constructed to reflect the options and features which your application will require. This information is kept in a text file called the KwikNet Library Parameter File which is created and edited for you by the KwikNet Configuration Builder. This editing process is illustrated in Figures 2.1-1.

From the KwikNet Library Parameter File, say *NETLIB.UP*, the Builder generates a single text file, a make file which can be used to create (make) the KwikNet Libraries as described in Chapter 3.2. This file, called the Network Library Make File *NETLIB.MAK*, is created by merging information from the Library Parameter File with the Network Library Make Template File *KNnnnLIB.MT*.

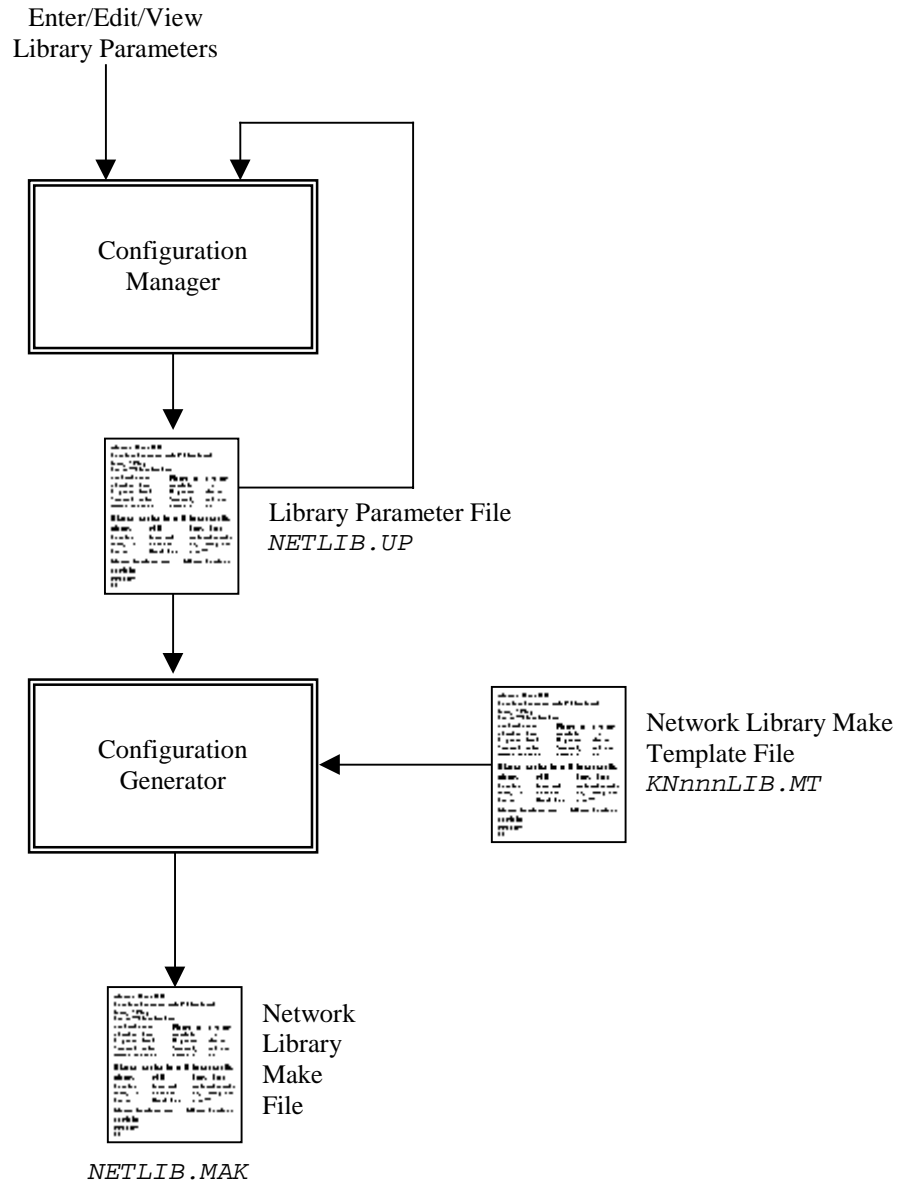


Figure 2.1-1 Creating the KwikNet Network Library Make File

Network Configuration Module

A Network Configuration Module must also be created which describes the networks and devices present in your target hardware. This information is kept in a text file called the Network Parameter File which is created and edited for you by the KwikNet Configuration Builder. This editing process is illustrated in Figure 2.1-2.

From the Network Parameter File, say *NETCFG.UP*, the Builder generates a single C source file. This file, called the Network Configuration Module *NETCFG.C*, is created by merging information from the Network Parameter File with the Network Configuration Template File *KNnnnnCFG.CT*.

This Network Configuration Module *NETCFG.C* is a C source file which completely identifies for KwikNet the particular networks and device drivers present in your application. This file must be compiled and linked as part of your KwikNet system.

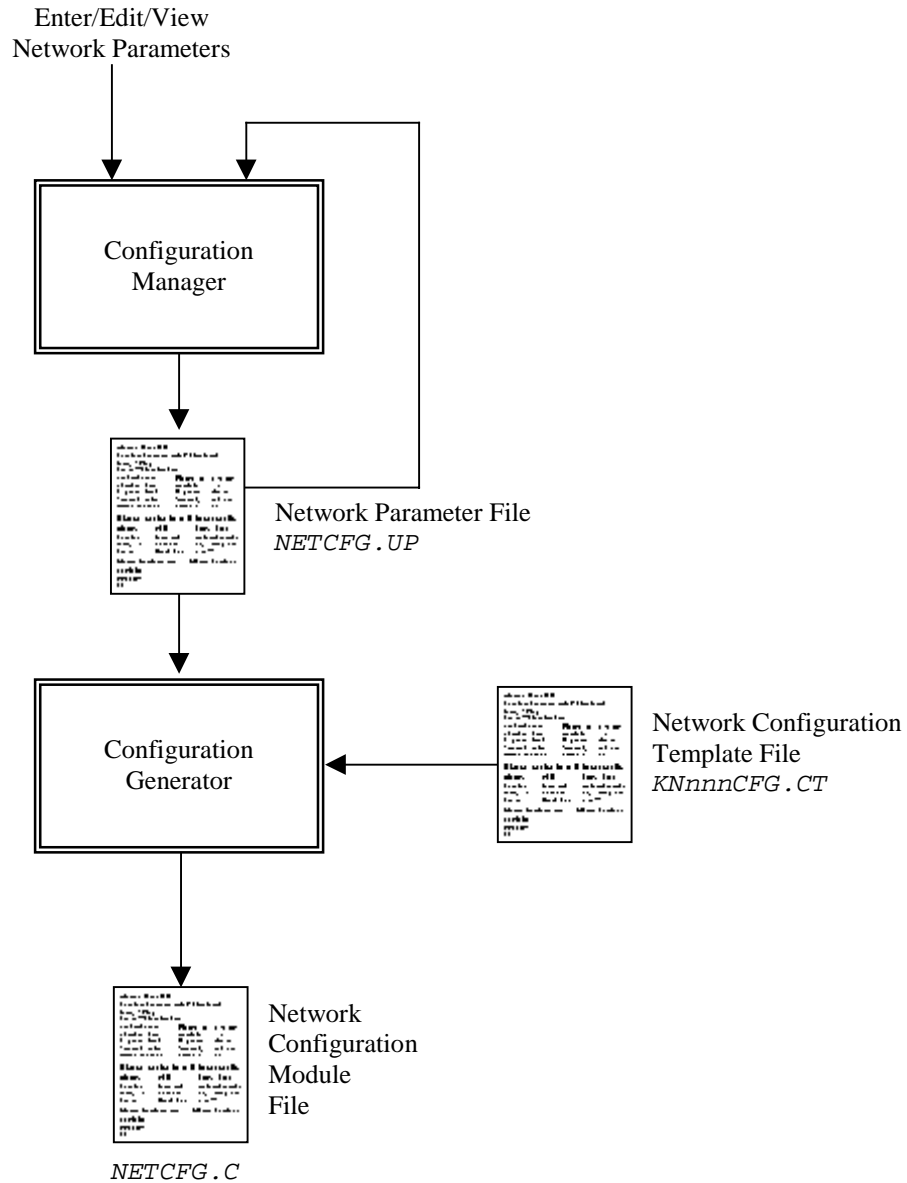


Figure 2.1-2 Creating the KwikNet Network Configuration Module

2.2 KwikNet Configuration Builder

The KwikNet Configuration Builder is a software generation tool which can be used to help create your KwikNet Libraries and your Network Configuration Module. The Builder consists of two components: the Configuration Manager and the Configuration Generator. The Configuration Manager is an interactive utility which allows you to create and edit your Library Parameter File and your Network Parameter File. You can think of the Builder as a very specialized editor.

For convenience, the Configuration Manager has the ability to directly invoke its own copy of the Configuration Generator. The Configuration Generator reads your Parameter File and merges the information from it with a template file to produce an output text file. This process has been described in Chapter 2.1.

The Configuration Generator is also available as a separate, stand alone DOS utility. This utility program can be used within your make files to generate, from your Parameter Files and the KwikNet Template Files, any of the output text files which the Configuration Builder generates.

Starting the Builder

The KwikNet Configuration Builder will operate on a PC or compatible running the Microsoft® Windows® 9x or NT operating system. It can also be used under Windows 3.1 with Win32s extensions.

The KwikNet Configuration Builder is delivered with the following files.

File	Purpose
<i>KNnnnnCM .EXE</i>	KwikNet Configuration Manager (utility program)
<i>KNnnnnCM .CNT</i>	KwikNet Configuration Manager Help Content File
<i>KNnnnnCM .HLP</i>	KwikNet Configuration Manager Help File
<i>KNnnnnCG .EXE</i>	KwikNet Configuration Generator (utility program)
<i>KNnnnnLIB.HT</i>	Network Library Configuration Template File
<i>KNnnnnLIB.MT</i>	Network Library Make Template File
<i>KNnnnnCFG .CT</i>	Network Configuration Template File

When KwikNet is installed on your hard disk, the KwikNet Configuration Manager utility program and its related files are stored in directory *CFGBLDW* in your KwikNet installation directory. To start the Configuration Manager, double click on its filename, *KNnnnnCM.EXE*. Alternatively, you can create a Windows shortcut to the Manager's filename and then simply double click the shortcut's icon.

Screen Layout

Figure 2.2-1 illustrates the Configuration Manager's screen layout. The title bar identifies the parameter file being created or edited. Below the title bar is the menu bar from which the operations you wish the Manager to perform can be selected.

Below the menu bar is an optional Toolbar with buttons for many of the most frequently used menu commands. The Toolbar is hidden or made visible using the View Toolbar command on the Edit menu.

The leftmost Toolbar button labeled LIB is used to create a new Library Parameter File. The button next to it labeled APP is used to create a new Network Parameter File.

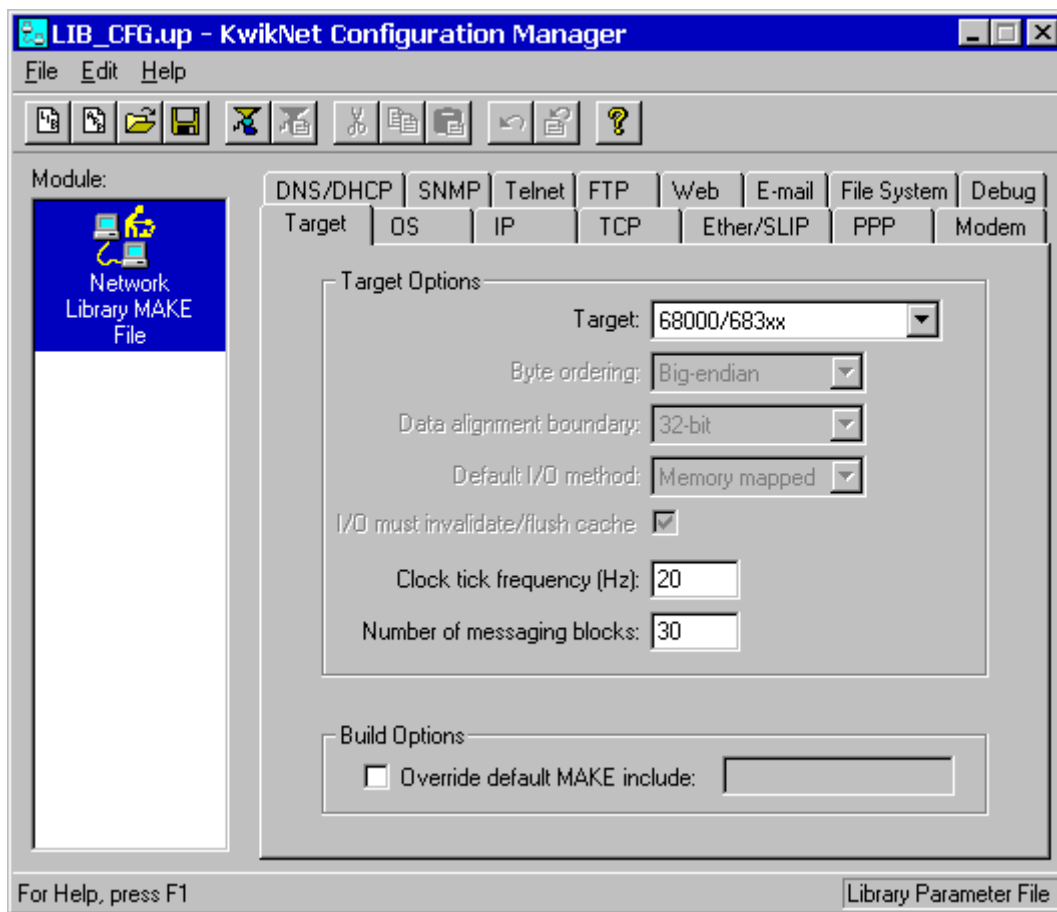


Figure 2.2-1 Configuration Manager Screen Layout

At the bottom of the screen is the status bar. As you select menu items, a brief description of their purpose is displayed in the status bar. If the Configuration Manager encounters an error condition, it presents an error message on the status bar describing the problem and, in many cases, the recommended solution.

Along the left margin of the screen are a set of one or more selector icons. These icons identify the type of output files which the Manager can produce from the parameter file being edited. The example illustrates that when editing a Library Parameter File, only the Network Library MAKE File selector is visible.

The center of the screen is used as an interactive viewing window through which you can view and modify your KwikNet library and network configuration parameters.

Menus

All commands to the Configuration Manager are available as items on the menus present on the menu bar. The **File menu** provides the conventional New, Open, Save and Save As... commands for creating and editing your parameter file. It also provides the Exit command.

Once a particular selector icon has been chosen as the currently active selector, the Generate... command on the File menu can be used to generate the corresponding output product. The path to the template file required by the generator to create this product can be defined using the Templates... command on the File Menu.

The **Edit menu** provides the conventional Cut, Copy, Paste and Undo editing commands. It also includes an Undo Page command to restore the content of all fields on a property page to undo a series of unwanted edits to the page. The Toolbar is hidden or made visible using the View Toolbar command on the Edit menu.

The **Help menu** provides access to the complete KwikNet Configuration Manager reference manual. Context sensitive help is also available by pressing the F1 function key or clicking the ? button on the Toolbar.

Field Editing

When editing a parameter file, a collection of tabbed property sheets is displayed in the central region of the screen. Each tab provides access to a particular property page through which your library or network configuration parameters can be declared. For instance, if while editing your Network Parameter File, you select the Networks tab, the Configuration Manager will present a network definition window (property page) containing all of the parameters you must provide to completely define a network.

Some fields are boolean options in which all you can do is turn the option on or off by checking or unchecking the associated check box.

Some fields are option fields in which you must select one of a set of options identified with radio buttons or pull down lists. Click on the option button or pick the list item which meets your preference.

Other fields may require numeric or text entry. Parameters are entered or edited in these fields by typing new values or text to replace the current field content. Only displayable characters can be entered. New characters which you enter are inserted at the current cursor position in the field. Right and left arrow, backspace and delete keys may be used to edit the field.

When you are altering a numeric or text field, you tell the Configuration Manager that you are finished editing the field by striking the Enter key. At that point, the Configuration Manager checks the numeric value or text string that you have entered for correctness in the context of the current field. If the value or text string that you have entered is invalid, an error indication is provided on the status bar at the bottom of the screen suggesting how the fault should be corrected.

The Tab and Shift-Tab keys can also be used to complete the editing of a field and move to the next or previous field.

If you have modified some of the fields on a property page and then decide that these modified values are not correct, use the Undo Page command on the Edit menu or Toolbar to force the Configuration Manager to restore the content of all fields on the page to the values which were in effect when you moved to that property page.

When you go to save your parameter file or prepare to move to another property page, the Configuration Manager will validate all parameters on the page which you are leaving. If any parameters are incomplete or inconsistent with each other, you will be forced to fix the problem before being allowed to proceed.

Add, Edit and Delete KwikNet Objects

Separate property pages are provided to allow your definition of one or more KwikNet objects such as networks and device drivers.

Pages of this type include a list box at the left side of the property page in which the currently defined objects are listed. At the bottom of the list box there may be a counter showing the number of objects in the list and the allowable maximum number of such objects.

Also below the list are two control buttons labeled Add and Delete. If the allowable maximum number of objects is 0 or if all such objects have already been defined, the Add button will be disabled. If there are no objects defined, the Delete button and all other fields on the page will be disabled.

To add a new object, click on the Add button. A new object with a default identifier will appear at the bottom of the list and will be opened ready for editing. When you enter a valid identifier for the object, your identifier will replace the default in the object list.

To edit an existing object's definition, double click on the object's identifier in the object list. The current values of all of that object's parameters will appear in the property page and the object will be opened ready for editing.

To delete an existing object, click on the object's identifier in the object list. Then click on the Delete button. Be careful because you cannot undo an object deletion.

The objects in the object list can be rearranged by dragging an object's identifier to the desired position in the list. You cannot drag an object directly to the end of the list. To do so, first drag the object to precede the last object on the list. Then drag the last object on the list to precede its predecessor on the list.

This page left blank intentionally.

2.3 KwikNet Library Parameter File

When the Network Library MAKE File selector icon is the currently active selector, the Library Configuration's tabbed property sheet is displayed in the central region of the screen. Each tab provides access to a particular property page through which your library configuration parameters can be declared. For instance, if you select the IP tab, the Configuration Manager will present an IP definition window (property page) containing all of the parameters you can adjust to completely define your use of the IP protocol.

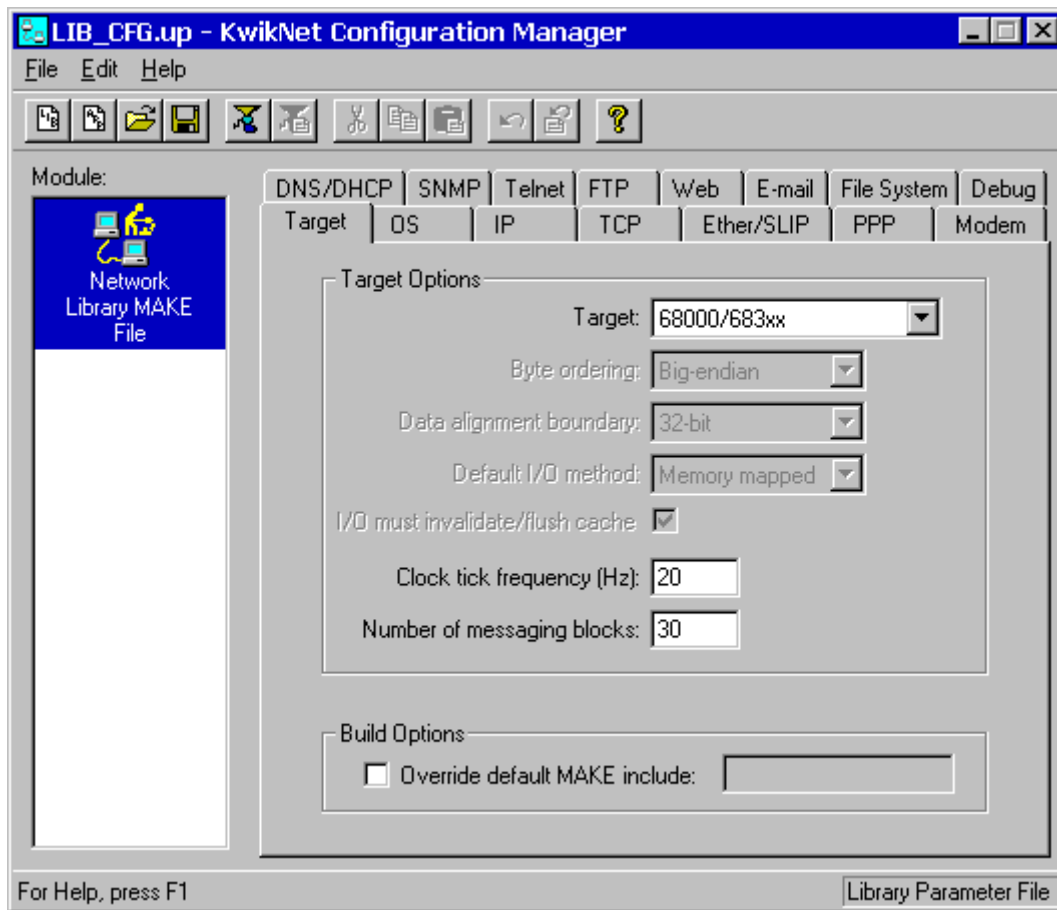
To create a new Library Parameter File, select New Library Parameter File from the File menu. The Configuration Manager will create a new, as yet unnamed, file using its default KwikNet library configuration parameters. When you have finished defining or editing your library configuration, select Save As... from the File menu. The Configuration Manager will save your Library Parameter File in the location which you identify using the filename which you provide.

To open an existing Library Parameter File, say *NETLIB.UP*, select Open... from the File menu and enter the file's name and location or browse to find the file. When you have finished defining or editing your library configuration, select Save from the File menu. The Configuration Manager will rename your original Library Parameter File to be *NETLIB.BAK* and create an updated version of the file called *NETLIB.UP*.

When the Network Library MAKE File selector icon is the currently active selector, the Generate... command on the File menu can be used to generate your Network Library Make File, say *NETLIB.MAK*. The path to the template file required by the generator to create this product can be defined using the Templates... command on the File Menu.

Target Parameters

The KwikNet Libraries must be tailored to operate on a particular target processor. These KwikNet parameters are edited using the Target property page. The layout of the window is shown below.



Target Processor

Select the target processor of interest from those available on the pull down list.

Byte Ordering

From the pull down list, choose the byte ordering scheme (big endian or little endian) used by the target processor's memory system. If the byte ordering method is dictated by the processor, this field will be preset and unalterable.

Target Parameters (continued)

Data Alignment Boundary

From the pull down list, choose the target processor's natural data alignment (16-bit or 32-bit) for long variables and structures. If the natural data alignment is dictated by the processor, this field will be preset and unalterable.

Default I/O Method and I/O Cache

From the pull down list, choose the method (memory mapped or I/O ports) used for device I/O addressing for the target processor. If the I/O addressing method is dictated by the processor, this field will be preset and unalterable.

When memory mapped I/O is used on processors like the PowerPC, it may be necessary to invalidate the data cache before reads and flush the data cache after writes. For such systems, check the I/O...cache box.

KwikNet Clock Tick Frequency

Enter the frequency of the fundamental KwikNet clock tick. All KwikNet timing measurements will be based on this frequency. The KwikNet Task will perform its stack polling operations at this frequency.

The KwikNet clock frequency must be at least 2 Hz. A frequency of 10 Hz or 20 Hz is recommended. Any frequency much above 50 Hz will simply introduce unnecessary execution overhead with little noticeable improvement in network throughput.

Note that KwikNet must achieve the specified clock frequency using timing services provided by the underlying operating system. You must therefore choose a KwikNet clock frequency which is derivable by that operating system. Set the KwikNet clock frequency so that the corresponding period is an integral number of OS system ticks.

Number of Messaging Blocks

KwikNet uses messaging blocks for its private internal communication. You may have to increase the number of messaging blocks if any of the following conditions exist.

- You have many tasks using network services
- You service several networks concurrently
- You expect high levels of network packet activity

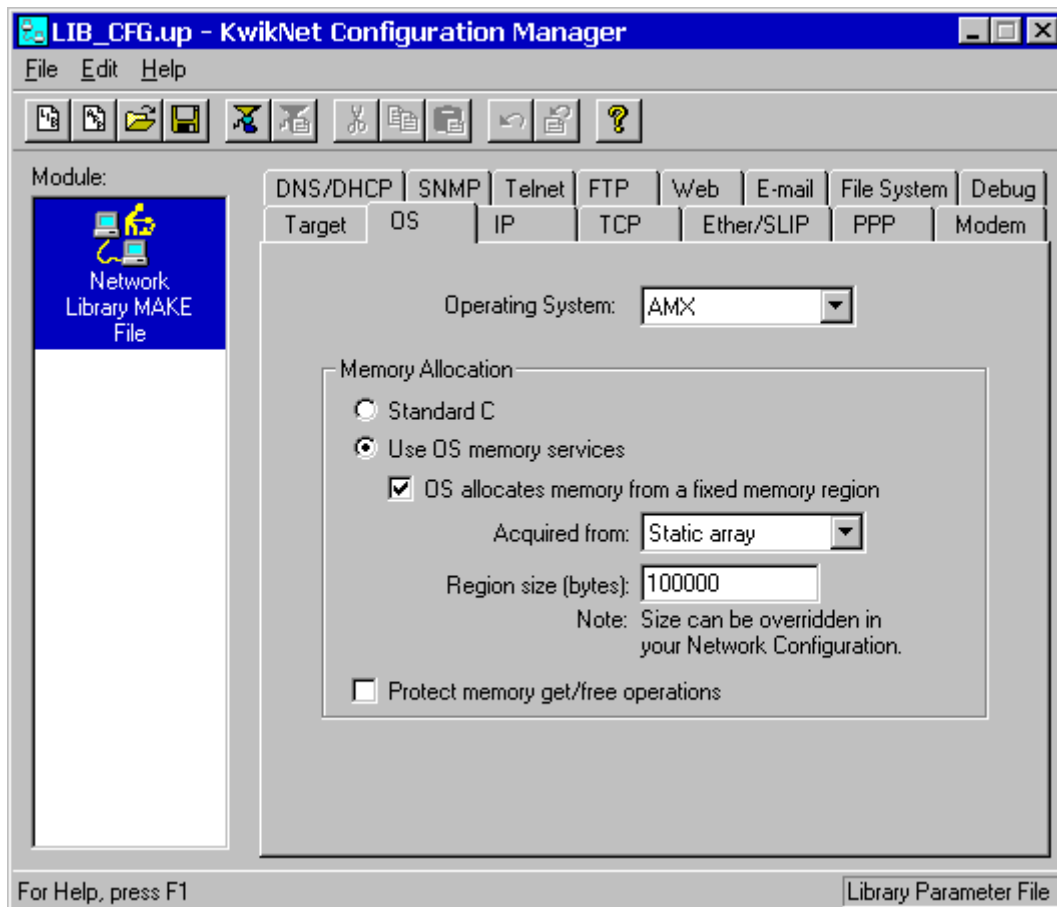
Override Make Include File

The override option is provided to allow an alternate tailoring file to override the make include file *KNZZZCC.INC* used in the construction of the KwikNet Libraries. Tailoring files are described in Chapter 3.1 of the KwikNet TCP/IP Stack User's Guide.

If you must employ an alternate tailoring file for some reason, check the box and enter the name of your tailoring file. Otherwise, leave this box unchecked.

OS Parameters

The KwikNet Libraries must be tailored to operate with a particular operating system. These KwikNet parameters are edited using the OS property page. The layout of the window is shown below.



OS Parameters (continued)

Operating System

From the pull down list, choose the underlying operating system upon which KwikNet must rely. KwikNet is ready for use without modification with KADAK's AMX kernels. Choose AMX for any of the 32-bit AMX kernels. Choose AMX 86 for 16-bit, real mode operation on any 80x86/88 processor.

The KwikNet Porting Kit can be used to port KwikNet to your operating environment, with or without an RTOS. The kit includes examples of KwikNet which work with the following single threaded operating systems: MS-DOS and the Tenberry DOS/4GW DOS Extender for MS-DOS. Custom implementations for a user defined in-house or commercial RTOS and for a single threaded OS are also provided with the kit.

Memory Allocation

KwikNet must be able to dynamically allocate and free blocks of memory of varying sizes. KwikNet uses the memory allocation services in the OS Interface Module *KN_OSIF.C*. KwikNet calls OS interface procedure *kn_osmeminit()* to initialize the memory allocator. Thereafter, KwikNet calls *kn_osmemget()* to get a block of memory and *kn_osmemrls()* to free a previously allocated block.

If you are porting KwikNet to your own operating environment, you may wish to edit these procedures to use your memory allocation services as described in Chapter 2 of the KwikNet Porting Kit User's Guide.

Two methods of memory allocation are supported.

If you check the Standard C radio button, the OS Interface Module provided with KwikNet will use standard C library functions *calloc()* and *free()* to allocate and free memory.

If you are using an operating system which provides its own memory allocation services, check the radio button labelled "Use OS memory services".

Protect Memory Get/Free Operations

When operating in a multitasking environment, the memory allocation services must be thread-safe. If the memory allocation services you have chosen to use are safe, leave this box unchecked. Otherwise, check this box and KwikNet will use its memory locking mechanism to protect access to the unsafe memory allocation services.

When operating in a single threaded environment, memory allocation services are inherently thread-safe. Hence, leave this box unchecked.

OS Parameters (continued)

OS Fixed Memory Region (Heap)

When using the operating system's memory allocation services, you may have to provide memory for use as a heap. If so, check the radio button labelled "OS allocates memory from a fixed region". For example, when AMX memory management services are used, a private memory pool is created for network memory allocation, isolating KwikNet from conflicts arising from memory allocation for other uses.

Source and Size of Fixed Memory Region (Heap)

If the operating system requires a private region of memory for use as a heap, you must provide that memory. From the pull down list, choose the method to be used to allocate such a memory region for use by the memory allocator.

If you choose the **Static array** option, you must enter the array size (in bytes) in the Region size field. A character array *kn_memstore[]* of that size will be declared in the KwikNet Network Configuration Module. KwikNet will allocate memory from that array.

If you choose the **malloc()** option, use the Region size field to define the number of bytes to be allocated for use as a heap. During KwikNet's startup initialization, the KwikNet Task will call C library function *malloc()* once to allocate a memory block of the specified size. KwikNet will subsequently allocate memory from that single block. If you specify a memory size of 0, KwikNet will estimate its total memory requirement and *malloc()* a memory block accordingly.

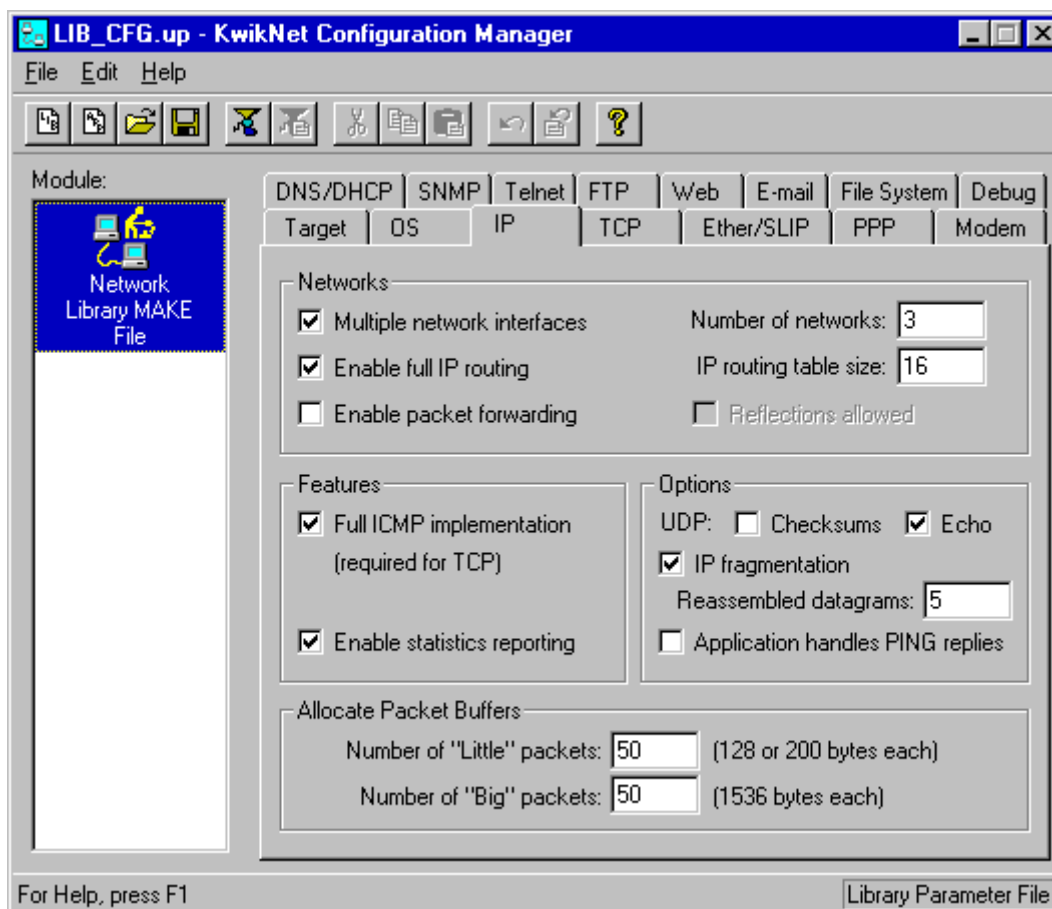
When either of the above methods are chosen, the declared memory size becomes the default value for that particular KwikNet Library. However, an override is allowed to permit the library to be used with several different network configurations, each with its own memory requirements. To override the default memory size for a particular network configuration, edit its Network Parameter File and enter the required memory region size in the field provided on the Application property page.

An alternate approach is to choose the **User function** option and provide a function called *kn_memacquire()* which is prototyped as shown below. The Region size field is unused in this case. When KwikNet starts up, it will call the function to get the size of your memory region and a pointer to it. The function must install a pointer to a block of *n* bytes of memory into **mempp* and return the value *n*.

```
unsigned long kn_memacquire(char **mempp);
```


IP Stack Parameters

The KwikNet IP Stack parameters are edited using the IP property page. The layout of the window is shown below.



IP Stack Parameters (continued)

Multiple Networks

Leave this box unchecked if you have only one network interface through which you can interconnect.

Check this box if your application must support more than one network interface. Such configurations are called multi-homed. Enter the maximum number of network interfaces which KwikNet must be able to support.

Enable Full IP Routing

IP routing is always optional, even if you have multiple network interfaces. Check the box to enable full IP routing and define the maximum number of IP addresses for which you wish KwikNet to be able to retain routing information. If you leave the box unchecked, KwikNet will not retain any IP routing information as it processes IP datagrams.

Enable Packet Forwarding

To enable IP packet forwarding, check this box. Otherwise, leave this box unchecked.

When IP routing and packet forwarding are enabled, KwikNet will attempt to forward packets destined to IP addresses which are not present on the host machine being serviced by KwikNet. IP packets which arrive on one network interface will be forwarded out another network interface. You usually will not require forwarding unless you have multiple network interfaces. However, if you only have one network interface, you can still enable packet forwarding as long as you also allow reflections.

Reflections Allowed

Check this box if you wish to allow KwikNet to forward an IP packet, if necessary, by reflecting the packet back onto the network interface from which the packet arrived. If this box is left unchecked, KwikNet will not forward an IP packet back onto the originating network. If you only have one network interface, reflections must be allowed in order to provide packet forwarding.

Full ICMP

The IP stack will support the ICMP protocol for error and control message handling only if this box is checked.

If you wish to use the TCP protocol or UDP with sockets, you must check this box. You will not be allowed to remove the check from this box if either TCP or UDP sockets is enabled.

IP Stack Parameters (continued)

Enable Statistics Reporting

Check this box if you want KwikNet to be able to log the statistical information which it maintains about events occurring at all layers on each supported network. This statistical data will only be logged if your application calls the KwikNet procedure *kn_netstats()*.

Leave this box unchecked to omit all statistics logging code from the KwikNet Libraries, thereby completely precluding any logging of statistical data.

UDP Checksums

Datagrams sent and received using the User Datagram Protocol can include an optional UDP checksum. UDP checksums, if present, are always checked upon reception of a UDP datagram.

Check this box if you want UDP checksums to be generated on each UDP datagram sent by KwikNet on any supported network. Otherwise, leave this box unchecked.

UDP Echo

Check this box if you want KwikNet to act as a UDP echo server, echoing UDP packets received on the well known echo port number 7 back to the sender. Otherwise, leave this box unchecked.

IP Fragmentation

If you wish KwikNet to be able to split IP datagrams for transmission and reassemble IP datagrams which arrive as multiple IP datagram fragments, check this box and enter the maximum number of fragmented datagrams which KwikNet must be able to concurrently reassemble. Otherwise, leave this box unchecked.

Application Handles PING Replies

Your application can call KwikNet procedure *kn_pingsend()* to send a PING request. The PING response, if any, will normally be discarded by KwikNet. If your application wishes to receive notification of the PING response, you must check this box. Your application can then call KwikNet procedure *kn_pinguserfn()* to post a pointer to the function which KwikNet will call to deliver the PING response to your application.

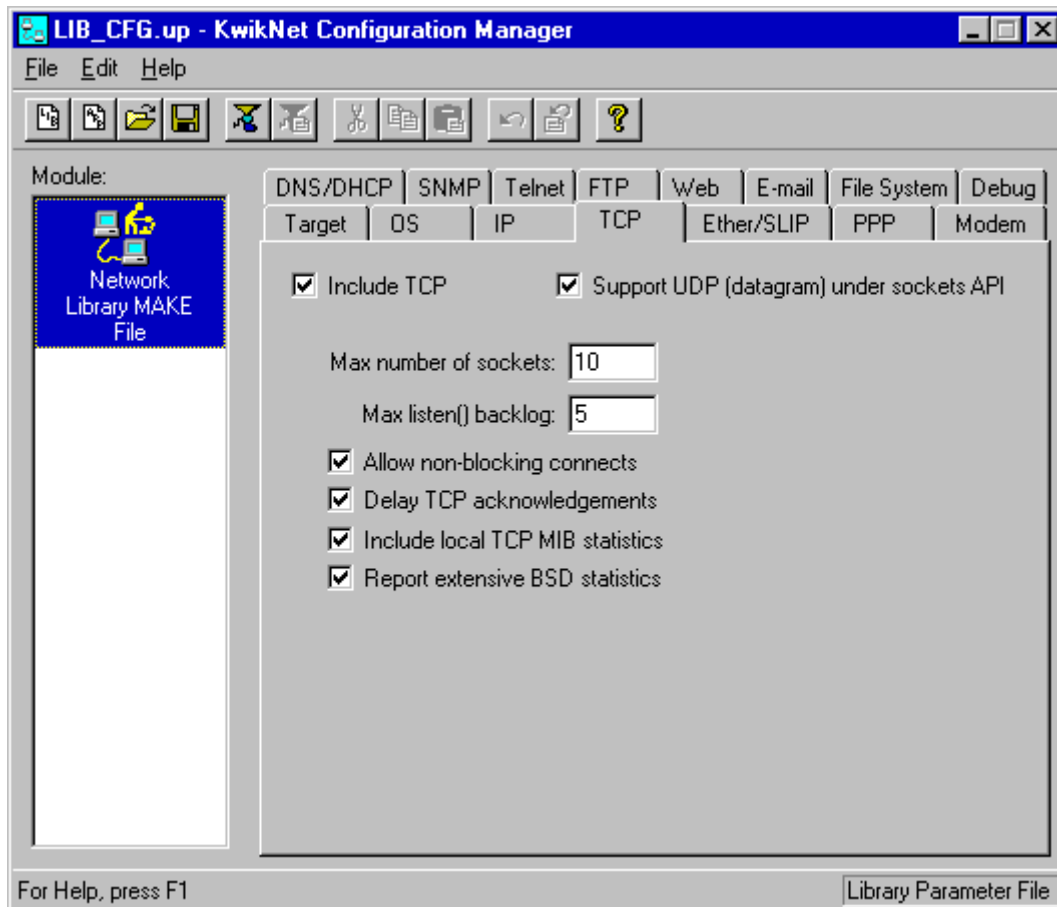
If your application does not use PING or has no need to examine the PING response, leave this box unchecked.

Little and Big Packet Buffers

Enter the total number of "little packet" and "big packet" buffers which KwikNet must allocate. By default little buffers are 128 bytes each. Little buffers will be 200 bytes each if the KwikNet Web Server is used. Big buffers are 1536 bytes each.

TCP Stack Parameters

The KwikNet TCP Stack parameters are edited using the TCP property page. The layout of the window is shown below.



Include TCP

Check this box if you intend to use the TCP protocol and its KwikNet socket interface. Otherwise, leave this box unchecked.

Support UDP Datagrams

Check this box if you wish to be able to use the KwikNet socket interface to communicate using the User Datagram Protocol. Otherwise, leave this box unchecked.

TCP Stack Parameters (continued)

Maximum Number of Sockets

Enter the maximum number of sockets which your application can have in use at any one time. Remember that a TCP server needs one socket to listen for connection requests and one socket for each accepted connection.

TCP Options

Maximum Listen Backlog

Enter the maximum number of backlogged connection requests which KwikNet must be able to queue for a socket being used to listen for connections. Once this number of connection requests have been queued for acceptance by the server, additional client requests for connection will be rejected by KwikNet.

Allow Non-blocking Connects

Check this box if you wish the KwikNet procedure *kn_connect()* to process a request for connection to another socket without forcing the caller to block waiting for the connection to be established. Otherwise, leave this box unchecked.

Delay TCP Acknowledgements

Check this box if you wish KwikNet to delay sending its TCP acknowledgement packets as long as possible while still adhering to the TCP protocol rules for packet acknowledgement. If you leave this box unchecked, KwikNet will acknowledge each TCP packet which it receives. Use of this option can yield improved throughput under some, but not necessarily all, network conditions.

Include Local TCP MIB Statistics

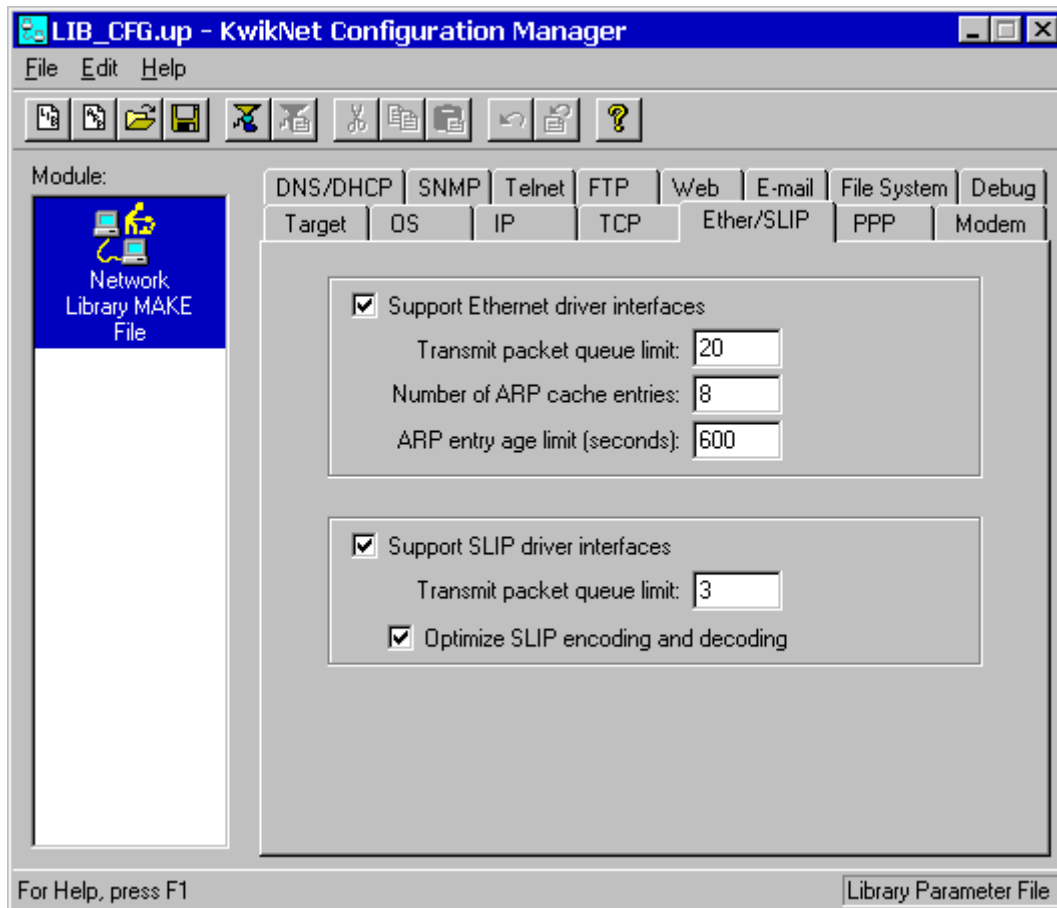
Check this box if you wish the KwikNet TCP Stack to maintain a local Management Information Base (MIB) database for inclusion in the network statistics log generated by the KwikNet procedure *kn_netstats()*. To omit these statistics, and the code which gathers and logs them, leave this box unchecked.

Report Extensive BSD Statistics

Check this box if you wish the KwikNet TCP Stack to maintain the extensive list of statistics usually found in the Berkeley Software Distribution (BSD) TCP/IP stack. These statistics will be included in the network statistics log generated by the KwikNet procedure *kn_netstats()*. To omit these statistics, and the code which gathers and logs them, leave this box unchecked.

Ethernet / SLIP Parameters

KwikNet includes an Ethernet and a SLIP network driver. Each of these drivers can support multiple networks of the same type. From the Ether/SLIP property page, you can select these network drivers for inclusion in the KwikNet Libraries and edit their operating parameters. The layout of the window is shown below.



Support Ethernet

If you have one or more networks with Ethernet device drivers, check this box to include the KwikNet Ethernet Network Driver in your KwikNet Library. Otherwise, leave this box unchecked.

Transmit Packet Queue Limit

The KwikNet Ethernet Network Driver queues packets for transmission until they can be accepted for transmission by the Ethernet device driver. This parameter defines the maximum number of packets which the network driver will queue.

Ethernet / SLIP Parameters (continued)

Number of ARP Cache Entries

The KwikNet Ethernet Network Driver uses the Address Resolution Protocol (ARP) to associate a specific hardware Ethernet address with a particular IP address. This parameter defines the maximum number of ARP address pairs which KwikNet can maintain in its ARP data cache for all supported networks.

If your network interfaces are interconnected with only a few other hosts, set this parameter to the number of such interconnections.

ARP Entry Age Limit

This parameter defines the length of time, measured in seconds, that an ARP address pair can reside in the ARP cache before it is purged. Any ARP entry whose age exceeds this time limit will not be used for address resolution.

Support SLIP

If you have one or more SLIP networks with serial UART device drivers, check this box to include the KwikNet SLIP Network Driver in your KwikNet Library. Otherwise, leave this box unchecked.

Transmit Packet Queue Limit

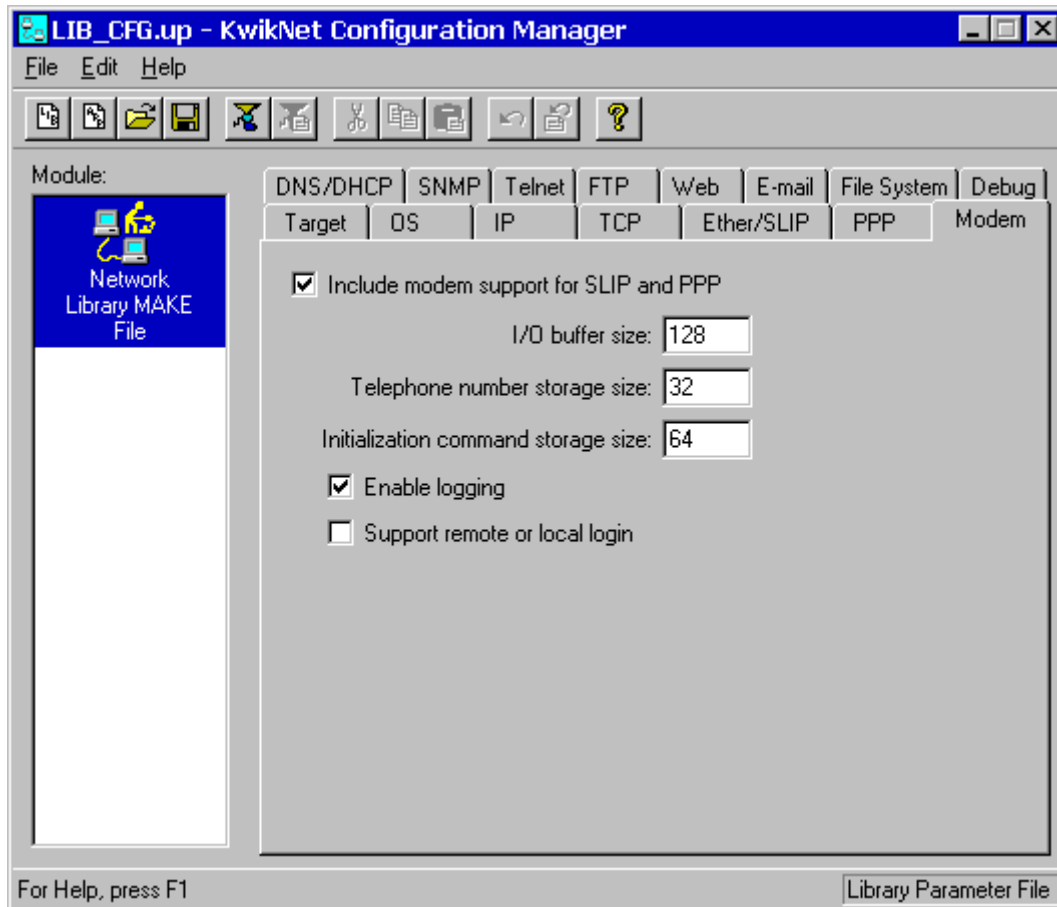
The KwikNet SLIP Network Driver queues packets for transmission until they can be accepted for transmission by the serial device driver. This parameter defines the maximum number of packets which the network driver will queue.

Optimize SLIP Encoding and Decoding

The SLIP protocol translates certain octets occurring in the data stream into an octet pair to avoid conflicts with the octet values reserved for use by the protocol. This encoding and decoding can be optimized to avoid unnecessary data movement in buffers if the data stream has no reserved octet conflicts. Check this box to enable this optimization. To omit this optimization code, leave this box unchecked.

Modem Parameters

KwikNet includes a modem driver for use with any SLIP or PPP network driver. The modem driver can operate concurrently on multiple networks. The modem driver is selected and its parameters edited using the Modem property page. The layout of the window is shown below.



Modem Parameters (continued)

Include Modem Support

If you have one or more SLIP or PPP networks with serial UART device drivers requiring modem support, check this box to include the KwikNet Modem Driver in your KwikNet Library. Otherwise, leave this box unchecked.

I/O Buffer Size

This parameter defines the size of the data buffers allocated for use by the Modem Driver. A transmit buffer and a receive buffer of this size are allocated for each modem which is attached to a UART device driver.

Telephone Number Storage Size

This parameter defines the size of the storage buffer required to hold a complete telephone number which the modem may dial. A buffer of this size is allocated for each modem which is attached to a UART device driver.

Initialization Command Storage

This parameter defines the size of the storage buffer required to hold the complete modem initialization command string required to initialize the modem. A buffer of this size is allocated for each modem which is attached to a UART device driver.

Enable Logging

Check this box if you wish the KwikNet Modem Driver to be able to record debug tracing information on the debug logging device and to log script operations on the modem logging device. If this box is checked, modem logging will occur for every modem which is attached to a UART device driver.

Leave this box unchecked to omit all logging code from the KwikNet Modem Driver, thereby completely precluding any modem logging.

Support Remote or Local Login

The KwikNet Modem Driver can support local or remote login scripts. If this box is checked, then a local login script can be used by the Modem Driver to control the dialing and login sequence when a local user attempts to login to a remote server. A remote login script can be used to control the answering and login sequence when a remote user attempts to login to a local server.

Leave this box unchecked if login scripting is not required on any network serviced by a modem.

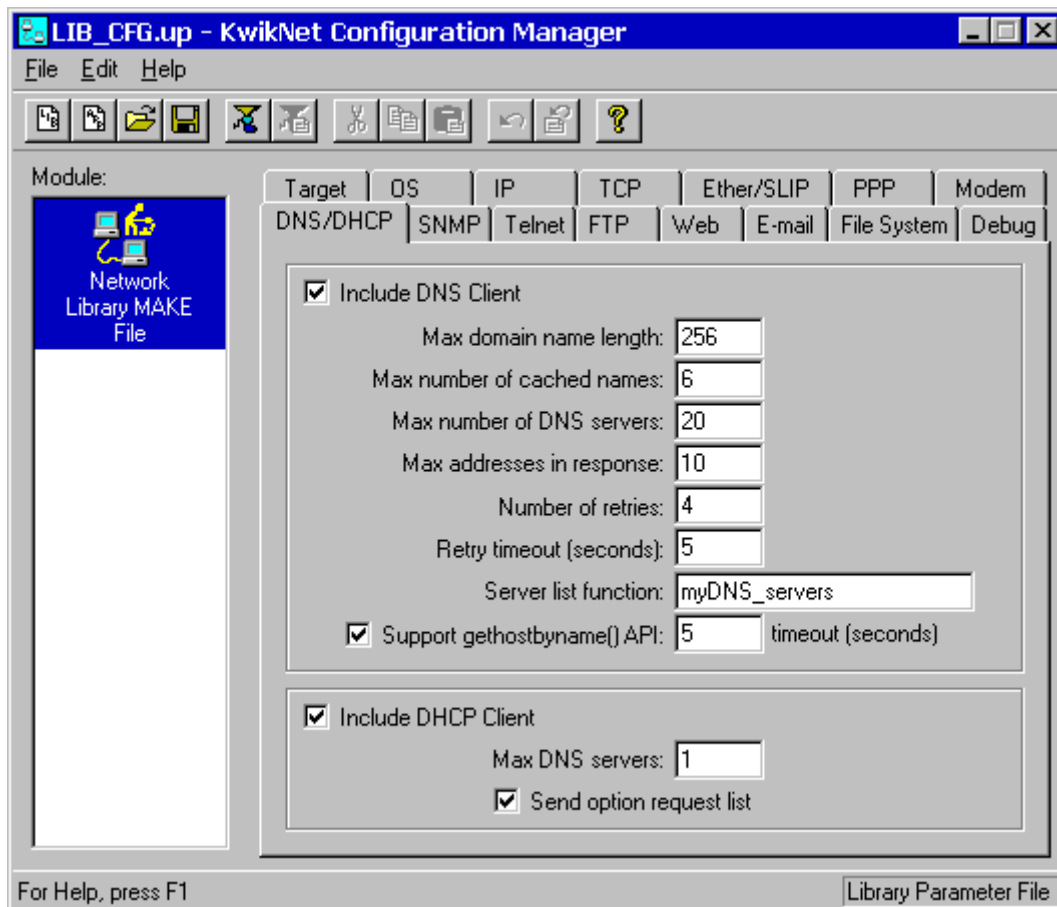
The KwikNet Modem Driver is described in Chapter 1.7 of the KwikNet Device Driver Technical Reference Manual.

DNS / DHCP Client Parameters

KwikNet includes a DNS client for accessing Domain Name System services on interconnected networks. The DNS client is described in Chapter 4.3. The DNS client is selected and its parameters edited using the DNS/DHCP property page.

KwikNet includes a DHCP client for using the Dynamic Host Configuration Protocol (DHCP) to derive an IP address, subnet mask and default gateway for any of your network interfaces. The DHCP client is described in Chapter 4.2. The DHCP client is selected and its parameters edited using the DNS/DHCP property page.

The layout of the DNS/DHCP window is shown below.



DNS / DHCP Client Parameters (continued)

Include DNS Client

If your application wishes to use the KwikNet DNS client to access Domain Name System services on interconnected networks, check this box. Otherwise, leave this box unchecked.

Maximum Domain Name Length

Domain names are strings consisting of dot separated labels such as `www.kadak.com`. This parameter defines the longest string, including the terminating `'\0'` character, which you expect the DNS client to be able to handle.

Maximum Number of Cached Names

The KwikNet DNS client maintains a list of the domain names which your application has queried. This parameter defines the maximum number of domain names which you wish the DNS client to cache. The larger you make this parameter, the more memory will be allocated for caching. The amount of memory is also affected by the maximum domain name length which you define.

Maximum Number of DNS Servers

The KwikNet DNS client maintains a list of the IP addresses of the DNS servers which it can query to resolve a domain name. KwikNet and your application can add and remove DNS servers from this list. This parameter defines the maximum number of DNS servers allowed in the list.

Maximum Addresses in Response

A DNS server can provide more than one IP address which corresponds to a specific domain name. This parameter defines the maximum number of IP addresses which KwikNet will keep from any DNS server response. Additional IP addresses provided by a DNS server will be ignored.

Number of Retries

This parameter defines the number of attempts which the DNS client will make to acquire the IP address for a particular domain name before declaring failure.

Retry Timeout

This parameter defines the number of seconds which the DNS client will wait for a response to a DNS query. If no response is received within this timeout period, the DNS client resends its query. This process repeats up to the maximum number of retries which you have specified.

DNS / DHCP Client Parameters (continued)

Server List Function

Your application can dynamically add and remove DNS servers from the list maintained by the KwikNet DNS client. Alternatively, you can provide a function which KwikNet will call upon startup to fetch a pointer to a list of DNS server IP addresses. This parameter is the name of your function. This function must operate as described in Chapter 4.3 of the KwikNet TCP/IP Stack User's Guide. If you have no such function, leave this field empty.

Support Gethostbyname() API

If your application must use the common function *gethostbyname()* or its KwikNet reentrant equivalent *kn_gethostbyname()*, check this box and indicate the maximum interval, in seconds, which you will allow for the host name to be resolved. Otherwise, leave this box unchecked.

Include DHCP Client

If any of your network interfaces require the use of the Dynamic Host Configuration Protocol (DHCP) to derive an IP address, subnet mask and default gateway, check this box. Otherwise, leave this box unchecked.

Maximum DNS Servers

When a DHCP server responds to an IP address query, it can also provide a list of known DNS server IP addresses. This parameter defines the maximum number of DNS server IP addresses which will be accepted from the DHCP server. Additional DNS server IP addresses provided by the DHCP server will be ignored. If you want the DHCP client to ignore all DNS servers identified by any DHCP server, set this parameter to 0.

This parameter cannot be used unless you also include the KwikNet DNS client.

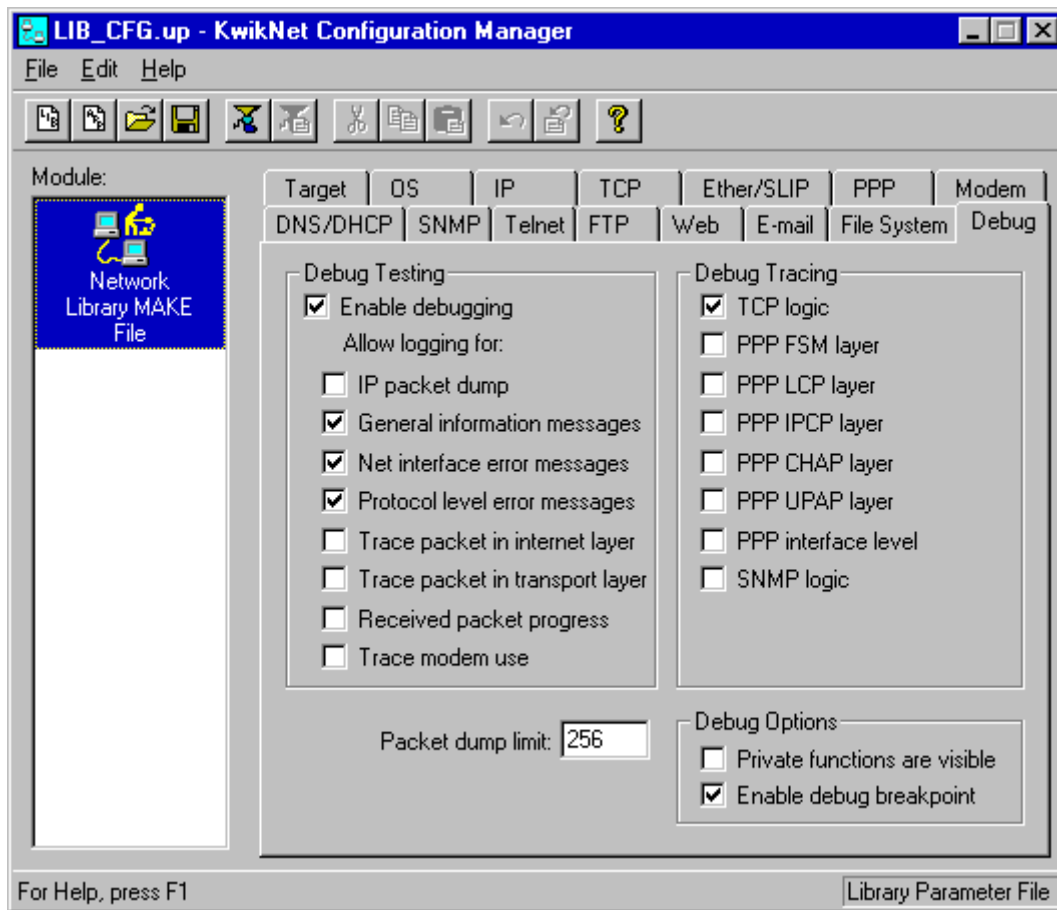
Send Option Request List

The KwikNet DHCP client can include an option request list in its DHCP query. The option list explicitly indicates the DHCP options to which the DHCP server may choose to respond. The KwikNet option list indicates that KwikNet will accept subnet masks, gateways and DNS servers from the DHCP server. The DNS server option will only be presented in the option list if you have included the KwikNet DNS client.

Check this box if you want the KwikNet DHCP client to send its option request list with each DHCP query. Otherwise, leave this box unchecked.

Debug and Trace Parameters

KwikNet includes a number of debugging and execution tracing options. These options are enabled and selected using the Debug property page. The layout of the window is shown below.



Debug and Trace Parameters (continued)

Enable Debugging

Check this box to enable debugging code. If this box is checked, code is inserted into the KwikNet Libraries to generate progress messages and to log errors encountered as the stack executes.

Leave this box unchecked to omit all debug logging code from the KwikNet Libraries, thereby completely precluding any debug logging.

If this box is checked, you can qualify the specific debug information which is to be logged on the debug recording device. Although code will exist to handle each option listed, only those options which are checked will cause debug messages to be logged.

Debug Tracing

Check a particular protocol trace box to enable the logging of trace information for that protocol layer on the debug recording device. If a trace box is checked, code will be inserted into the KwikNet Libraries to log progress messages as that particular protocol layer executes.

Leave a trace box unchecked to omit that particular protocol trace logging code from the KwikNet Libraries, thereby completely precluding trace logging through that protocol layer.

Packet Dump Limit

If the debug logging of IP packet dumps is enabled, KwikNet will log the content of each IP datagram as it is sent or received. The entire IP datagram up to the limit defined by this parameter will be dumped. The dump limit is specified in bytes. The debug log will show the datagram bytes as two-character hexadecimal values, separated by spaces and listed in net endian order. The data log will be split into lines according to the line length specified on the Application property page in your Network Parameter File.

Make Private Functions Visible

Many KwikNet procedures are declared *static* and are hence not visible in link maps or accessible by a debugger. The *static* declaration can be removed from many of these procedures by checking this box.

Enable Debug Breakpoint

Many of the procedures in the KwikNet TCP/IP Stack and its optional modules can generate a debug trap after logging a debug message when an error is detected. Check this box if you wish all such debug traps to be vectored to the KwikNet breakpoint procedure *kn_bphit()* on which you can place a debugger breakpoint. Otherwise, leave this box unchecked.

2.4 KwikNet Network Parameter File

When the Network Configuration Module selector icon is the currently active selector, the Network Configuration Module's tabbed property sheet is displayed in the central region of the screen. Each tab provides access to a particular property page through which your network configuration parameters can be declared. For instance, if you select the Networks tab, the Configuration Manager will present a network definition window (property page) containing all of the parameters you must provide to completely define a network.

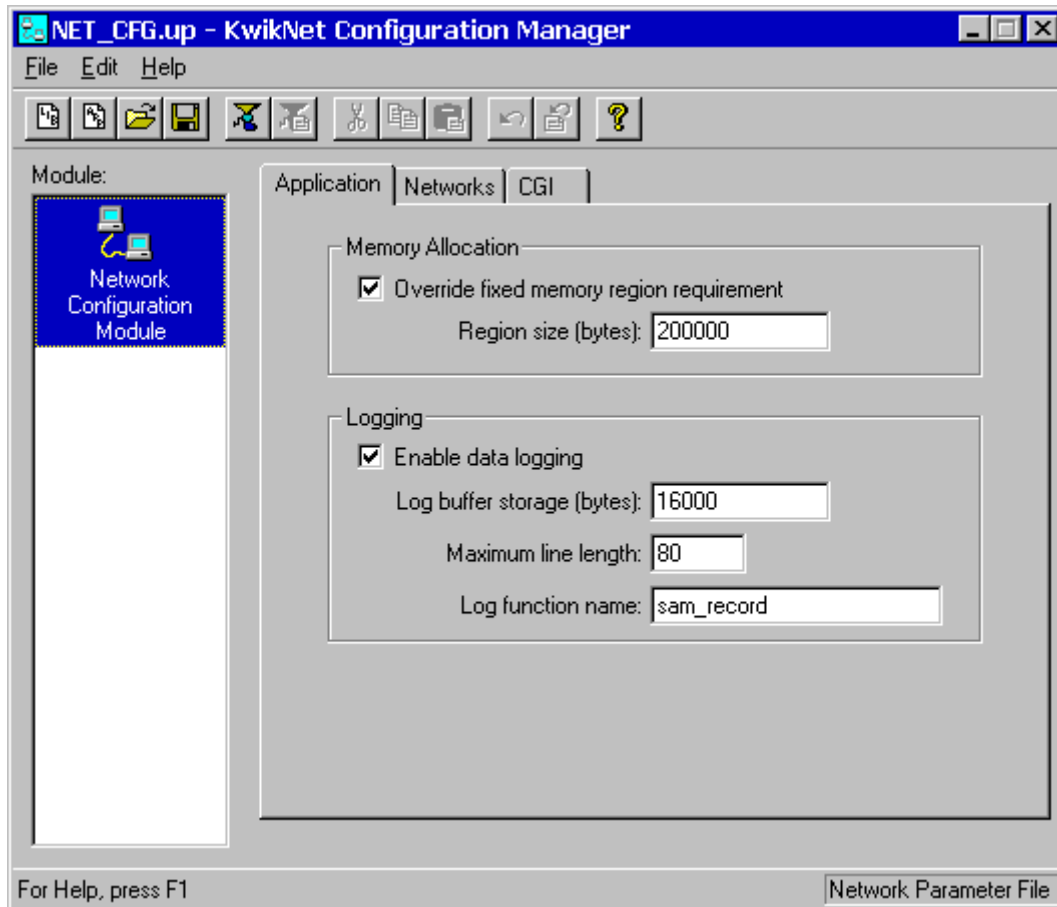
To create a new Network Parameter File, select New Network Parameter File from the File menu. The Configuration Manager will create a new, as yet unnamed, file using its default KwikNet network configuration parameters. When you have finished defining or editing your network configuration, select Save As... from the File menu. The Configuration Manager will save your Network Parameter File in the location which you identify using the filename which you provide.

To open an existing Network Parameter File, say *NETCFG.UP*, select Open... from the File menu and enter the file's name and location or browse to find the file. When you have finished defining or editing your network configuration, select Save from the File menu. The Configuration Manager will rename your original Network Parameter File to be *NETCFG.BAK* and create an updated version of the file called *NETCFG.UP*.

When the Network Configuration Module selector icon is the currently active selector, the Generate... command on the File menu can be used to generate your Network Configuration Module, say *NETCFG.C*. The path to the template file required by the generator to create this product can be defined using the Templates... command on the File Menu.

General Application Parameters

Your Network Parameter File must provide a number of general parameters to adapt KwikNet to meet the needs of your particular networking application. These KwikNet parameters are edited using the Application property page. The layout of the window is shown below.



General Application Parameters (continued)

OS Fixed Memory Region Override

If your KwikNet library has been configured to use a memory allocator which requires a private region of memory for use as a heap, you must provide that memory. If the library is configured so that the memory is to be provided using a **Static array** or **malloc()**, the memory size declared in the library becomes the default memory region size.

To use the default region size from the library, leave the Override box unchecked.

To use an alternate memory region size for this particular network configuration, check the Override box and enter the required size in the Region size field.

Enable Data Logging

Check this box if you wish to provide a logging function to record (display) text messages generated by KwikNet's data logging services. When this box is checked, you must provide all of the Logging parameters listed. If you do not want to provide a data logging function, leave this box unchecked.

Log Buffer Storage

Specify the amount of memory to be reserved for use as data log buffers. A character array of this size will be declared in the KwikNet Network Configuration Module.

Maximum Line Length

Specify the maximum number of characters that are allowed in each line of text logged to your recording (display) device.

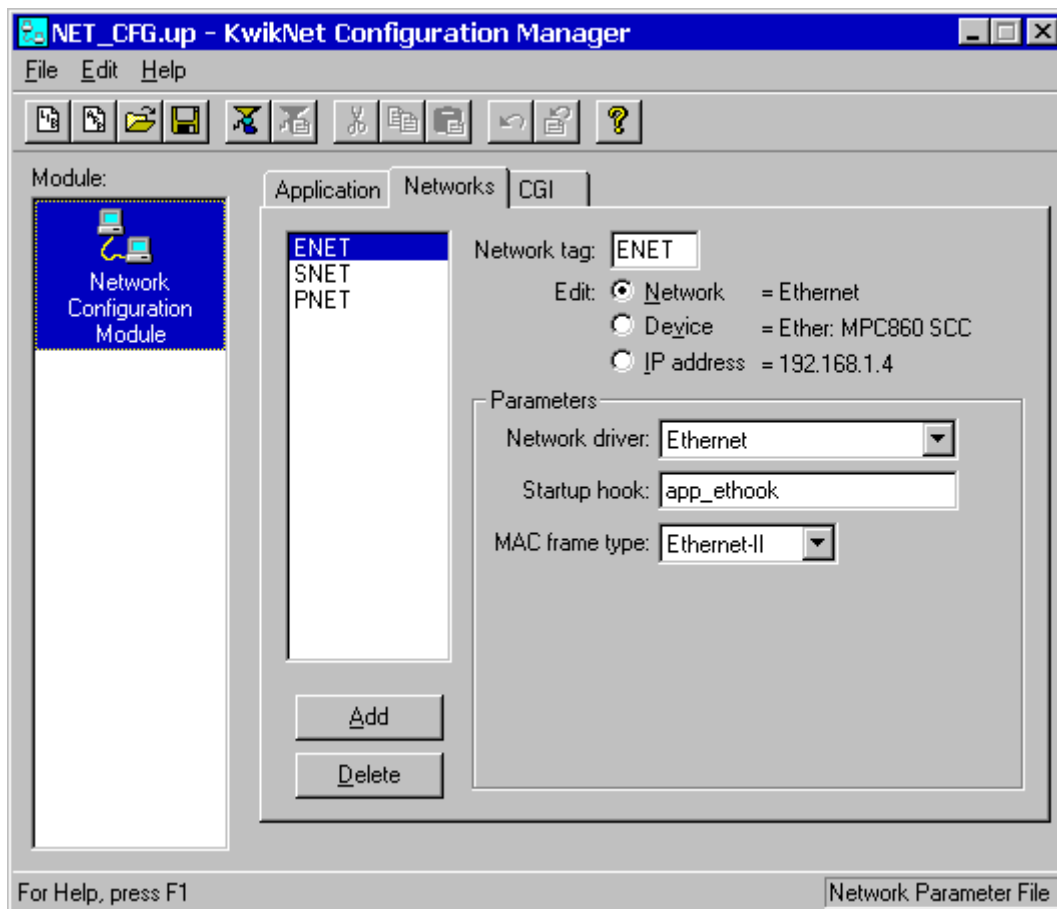
Log Function Name

This parameter provides the name of an application function which will be called to log each line of text generated by KwikNet's data logging services. This function must operate as specified in Chapter 1.6 of the KwikNet TCP/IP Stack User's Guide.

The sample programs provided with KwikNet and its optional components use data logging procedure *sam_record()* in the Application OS Interface module *KNSAMOS.C*. To use that procedure, enter the function name *sam_record* in this field.

Ethernet Network Definition

You must predefine each Ethernet network which your application must support. A separate definition is required for each such network. The total number of networks must not exceed the maximum number of networks which your KwikNet Library Parameter File allows. Each Ethernet network is defined using the Networks property page. The layout of the window is shown below.



Ethernet Network Definition (continued)

Tag

Each network must have a unique 4-character network tag. This parameter defines that tag. Although KwikNet does not restrict the content of the tag in a network description, the Configuration Manager only supports 4 ASCII characters as a tag.

Edit: Network

You must select the Edit: Network radio button to define the network parameters.

Network Driver

You must choose Ethernet from the pull down list to attach the KwikNet Ethernet Network Driver to your Ethernet network.

Startup Hook

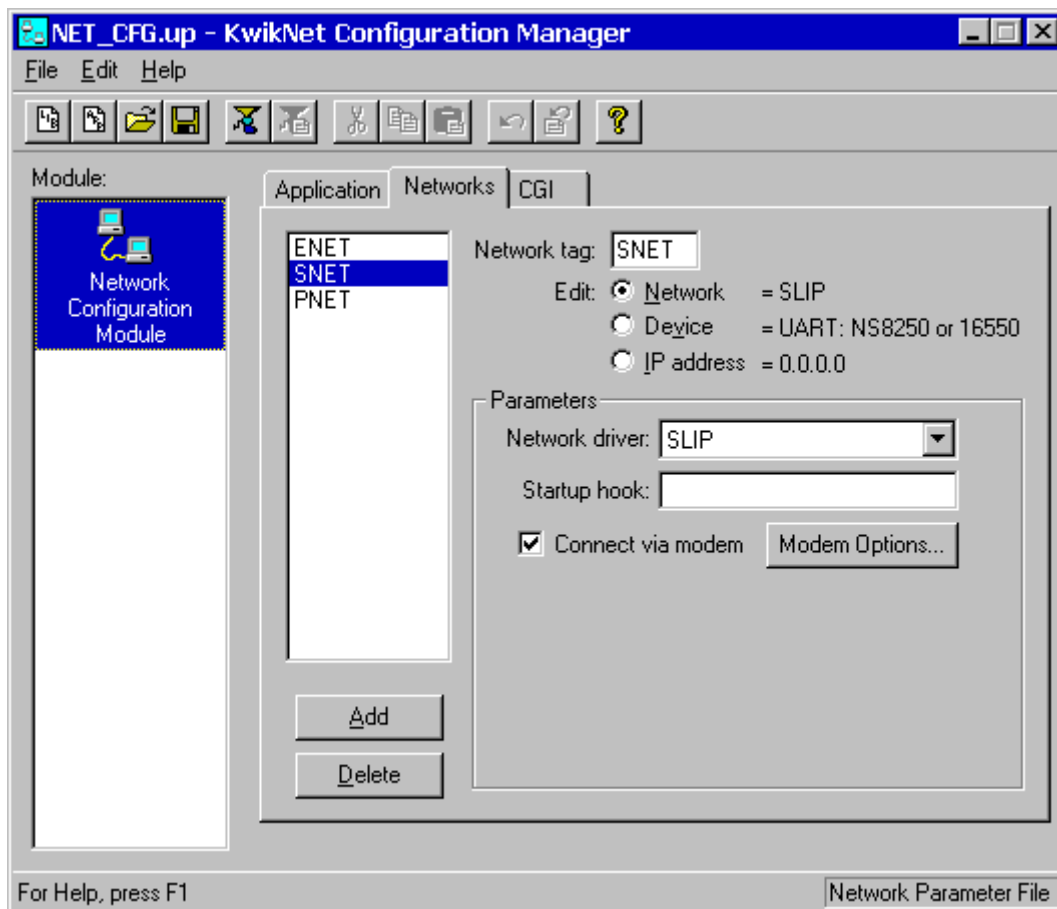
This parameter provides the name of an application function which will be called when the network driver is being initialized. This function can modify the network's configuration parameters and IP address information before the network has been fully initialized. If your application does not require a startup hook for this network, leave this field empty. The Ethernet network driver startup hook is described in Appendix A.1 of the KwikNet Device Driver Technical Reference Manual.

MAC Frame Type

The KwikNet Ethernet Network Driver supports two standards for framing Ethernet data packets, Ethernet-II and IEEE 802.3. An Ethernet network can only communicate with other nodes that use the same framing standard. Select the type of framing to use from the pull down list.

SLIP Network Definition

You must predefine each SLIP network which your application must support. A separate definition is required for each such network. The total number of networks must not exceed the maximum number of networks which your KwikNet Library Parameter File allows. Each SLIP network is defined using the Networks property page. The layout of the window is shown below.



SLIP Network Definition (continued)

Tag

Each network must have a unique 4-character network tag. This parameter defines that tag. Although KwikNet does not restrict the content of the tag in a network description, the Configuration Manager only supports 4 ASCII characters as a tag.

Edit: Network

You must select the Edit: Network radio button to define the network parameters.

Network Driver

You must choose SLIP from the pull down list to attach the KwikNet SLIP Network Driver to your SLIP network.

Startup Hook

This parameter provides the name of an application function which will be called when the network driver is being initialized. This function can modify the network's configuration parameters and IP address information before the network has been fully initialized. If your application does not require a startup hook for this network, leave this field empty. The SLIP network driver startup hook is described in Appendix A.1 of the KwikNet Device Driver Technical Reference Manual.

Modem Connection

The SLIP network driver supports remote connections using the KwikNet Modem Driver. Check this box to attach the Modem Driver to this network. Otherwise, leave this box unchecked. The KwikNet Modem Driver is described in Chapter 1.7 of the KwikNet Device Driver Technical Reference Manual.

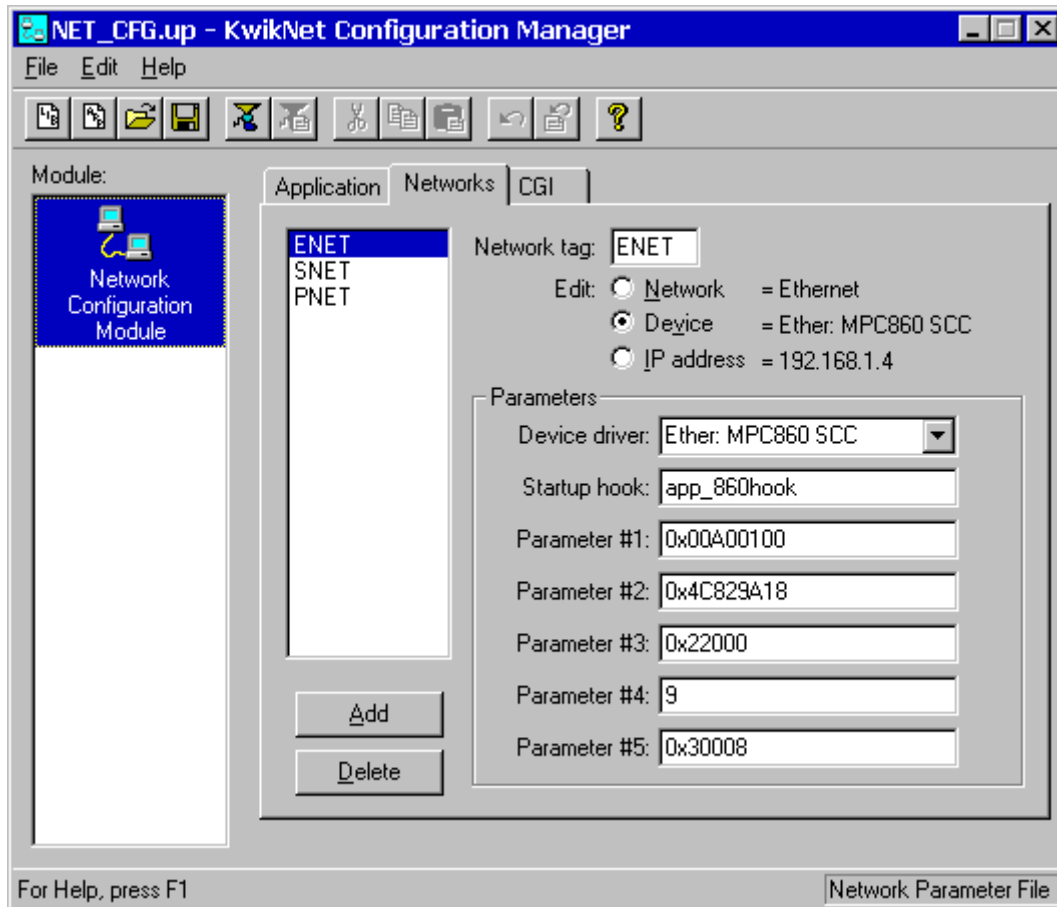
Note that the network still requires a device driver even if the Modem Driver is used.

Modem Options

If you have attached the Modem Driver to this network, then click the Modem Options... button to open the Modem Options Dialog. Within this dialog you can configure the modem to meet your requirements.

Network Device Driver Definition

You must define the device driver attached to each network which your application supports. A separate device driver definition is required for each network. Each network device driver is defined using the Networks property page. The layout of the window is shown below.



Network Device Driver Definition (continued)

Tag

Each device driver inherits the unique 4-character network tag assigned to the network to which the device driver is attached. This parameter defines that tag.

Edit: Device

You must select the Edit: Device radio button to define the network's device driver parameters.

Device Driver

To use any of the KwikNet device drivers which are available from KADAK, select its name from the pull down list.

If you are using your own custom KwikNet device driver or one only recently available from KADAK, you must edit the text region of the list to identify the driver. Replace the text in the list box with the name of the device driver's Device Preparation Function *dddd_prep*. The string *dddd* in the function name is the mnemonic used to uniquely identify the particular device driver.

The selected device driver must match its network driver. Device drivers for Ethernet interface devices can only be used with the Ethernet Network Driver. Device drivers for UART interface devices can only be used with the SLIP or PPP Network Driver. The Configuration Manager is not able to perform this consistency check.

Startup Hook

This parameter provides the name of an application function which will be called when the device driver is being initialized. This function can modify the device's configuration parameters before the device has been fully initialized. If your application does not require a startup hook for this device, leave this field empty. The device driver startup hook is described in Appendix A.2 of the KwikNet Device Driver Technical Reference Manual.

Parameter #1 through #5

There are five optional parameters which can be used to configure the device driver. Each parameter can provide a 32-bit value. Unused parameters can be left empty. The use and meaning of each parameter is completely defined by the device driver.

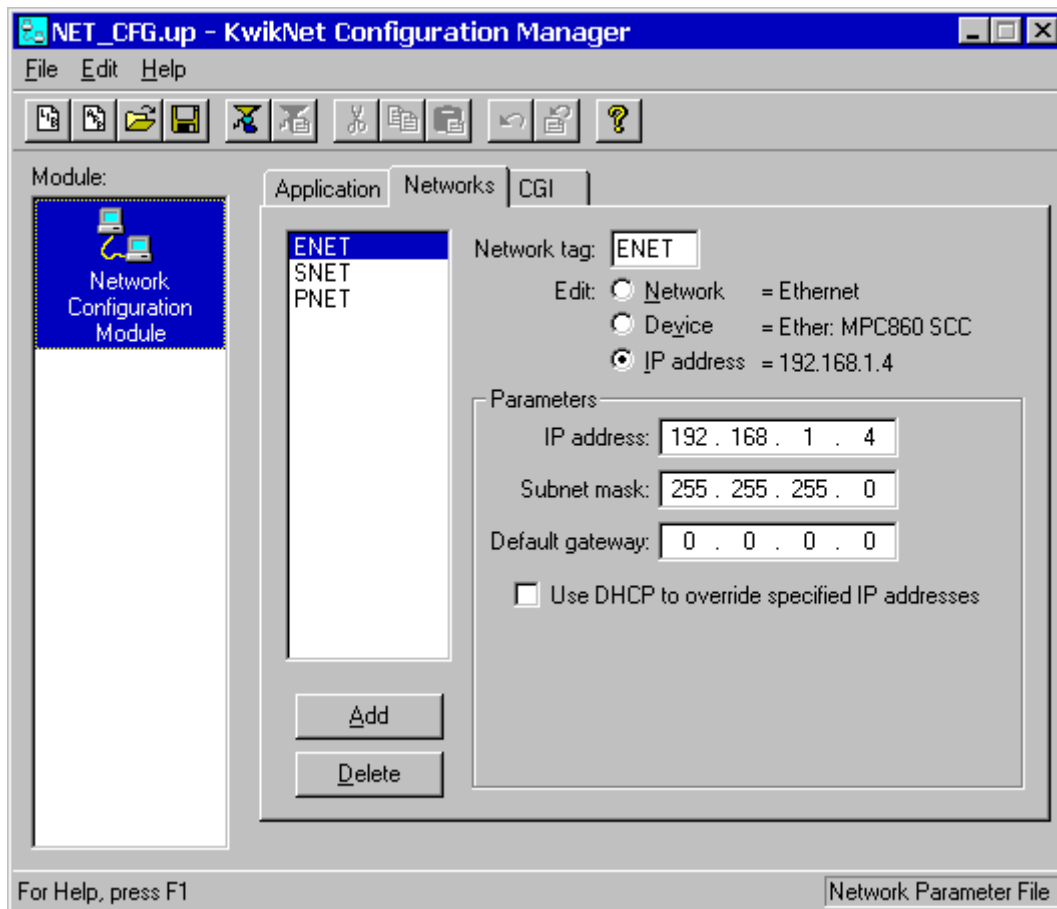
If you are using one of the KwikNet device drivers which are available from KADAK, refer to its manual for its definition of these fields. Otherwise, refer to the data sheets which you created for your custom KwikNet device driver.

Network IP Address Definition

You must provide a network IP address for each network which your application supports. Each network IP address is defined using the Networks property page. The layout of the window is shown below.

You can specify a network's IP address using this property page. Alternatively, you can assign the IP address in your network startup hook, if one is provided. Of course, IP addresses can also be assigned dynamically using DHCP.

If you intend to assign the network IP address at runtime, then you can enter 0.0.0.0 for all of the address fields on this page.



Network Device Driver Definition (continued)

Edit: IP Address

You must select the Edit: IP Address radio button to define the network's IP address and related parameters.

IP Address

Enter the IP address of the network interface. This must be a unique, valid IP address which can be used to identify the host computer attached to this network interface.

Subnet Mask

Enter the subnet mask of the network. The subnet mask defines the manner in which IP addresses on this network are decoded to distinguish between the physical net address and host identifiers. Enter 0.0.0.0 if subnet addressing is not used on this network.

Default Gateway

Enter the IP address of the default gateway to be used in the absence of specific routing information. This value must be a valid host IP address or 0.0.0.0. When sending data, all packets that cannot be sent to a node on the local network will be sent to the default gateway. If the default gateway field is 0.0.0.0, then there is no gateway defined for this network.

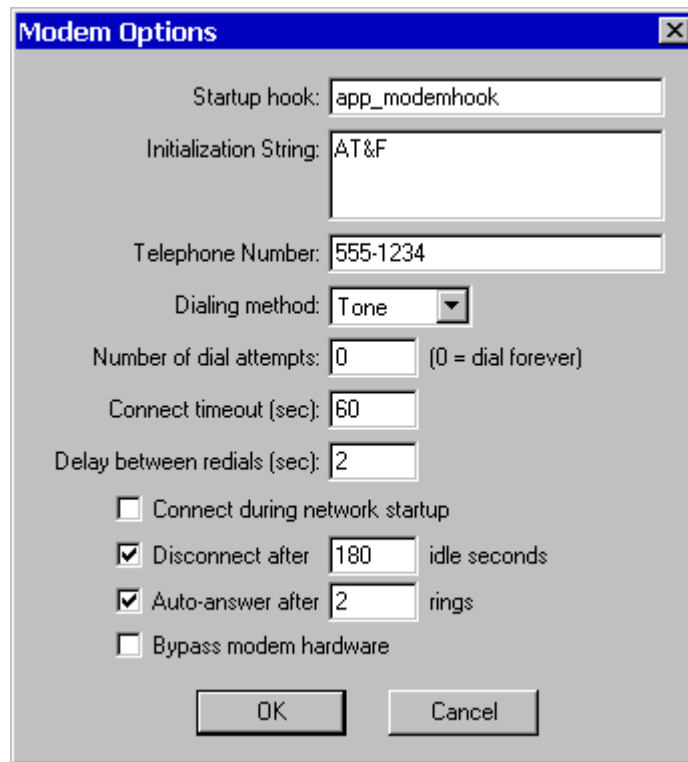
DHCP Override

If this box is checked, the network will use the Dynamic Host Configuration Protocol to derive an IP address, subnet mask and gateway definition for the network. To use this option, there must be a DHCP server located on this network. If dynamic IP address assignment is not required, leave this box unchecked.

If you check this box, be sure to configure your KwikNet IP Library to include the KwikNet DHCP client. To do so, edit your KwikNet Library Parameter File and check the box labeled Include DHCP Client on the IP property page. Failure to do so will result in a compilation error when you attempt to compile your KwikNet Network Configuration Module. The error will inform you that your library does not include the DHCP client.

Modem Options

If a KwikNet network requires modem support, you must define the modem parameters which govern its use. These modem parameters are edited using the Modem Dialog Box entered via the Modem Options... button on the Networks definition property page. The layout of the dialog box is shown below.

The image shows a Windows-style dialog box titled "Modem Options". It contains several fields and checkboxes for configuring modem settings. The fields are: "Startup hook:" with the value "app_modemhook", "Initialization String:" with the value "AT&F", "Telephone Number:" with the value "555-1234", "Dialing method:" with a dropdown menu set to "Tone", "Number of dial attempts:" with the value "0" and a note "(0 = dial forever)", "Connect timeout (sec):" with the value "60", and "Delay between redials (sec):" with the value "2". There are four checkboxes: "Connect during network startup" (unchecked), "Disconnect after" (checked) with a value of "180" and the text "idle seconds", "Auto-answer after" (checked) with a value of "2" and the text "rings", and "Bypass modem hardware" (unchecked). At the bottom are "OK" and "Cancel" buttons.

Modem Options

Startup hook: app_modemhook

Initialization String: AT&F

Telephone Number: 555-1234

Dialing method: Tone

Number of dial attempts: 0 (0 = dial forever)

Connect timeout (sec): 60

Delay between redials (sec): 2

☐ Connect during network startup

☒ Disconnect after 180 idle seconds

☒ Auto-answer after 2 rings

☐ Bypass modem hardware

OK Cancel

Startup Hook

This parameter provides the name of an application function which will be called when the KwikNet Modem Driver is being initialized. This function can modify the modem's configuration parameters before the modem has been fully initialized. If your application does not require a startup hook for this modem, leave this field empty. The modem driver startup hook is described in Appendix A.3 of the KwikNet Device Driver Technical Reference Manual.

Initialization String

The modem initialization string is sent to the modem before each dialing attempt. The command resets the modem and prepares it for dialing. This string should include the Hayes standard "AT" command prefix. Some common initialization strings are "AT&F" and "ATZ". For more information, consult your modem's reference manual.

Modem Options (continued)

Telephone Number

Enter the telephone number of the remote system to which the modem driver will connect. This string may contain special characters that are used to modify the modem's dialing operation. Do not include the Hayes standard "ATDx" prefix in this string.

Some common dialing commands are listed below. Be sure to consult your modem's reference manual for the actual commands that it supports.

<i>0 through 9</i>	Dial this number
<i>() - space</i>	Brackets, minus and space characters are ignored
<i>comma</i>	Timed wait
<i>W</i>	Timed wait for a dial tone
<i>@</i>	Timed wait for remote pickup
<i>S</i>	Dial a number stored in the modem's internal memory

Dialing Method

From the pull down list, choose the dialing method to use with the attached telephone system. Most telephone systems support tone dialing but some older systems may only support pulse dialing.

Number of Dial Attempts

Enter the number of times that the modem driver will attempt to connect to the remote system before declaring a connection error.

Connection Timeout

This field defines the maximum number of seconds that the modem driver will wait for a modem response after it has started a connection attempt. If the modem does not respond within this time, the dialing attempt will be aborted.

Delay Between Redials

This field defines the number of seconds that the modem driver will wait after a connection attempt has failed before it attempts to redial.

Connect During Network Startup

Check this box if you want to connect to the remote system when the KwikNet stack is started. If you leave this box unchecked, KwikNet will not connect to the remote system until your application tries to send data over this network or calls `kn_inet_ifstate()` to start up this network.

Modem Options (continued)

Disconnect When Idle

If you want to automatically disconnect from the remote system when the connection has not been used recently, check this box and enter the number of seconds that the line must remain idle before disconnecting. If you leave this box unchecked, the line will remain active until the remote system terminates the connection, your application calls `kn_inet_ifstate()` to shut down the network being used or KwikNet is shut down.

Auto-answer

If you want the modem driver to automatically answer incoming calls, check this box and enter the number of rings to wait before answering the call. If you leave this box unchecked, the modem driver will not answer incoming calls.

Bypass Modem Hardware

The KwikNet modem driver is required if you need to support remote or local login scripts. Check this box if you wish to use the modem driver to support login scripts but do not actually require a modem to communicate. If this box is checked, all other modem parameters on this screen, except the startup hook, will be ignored by KwikNet.

Note that if you intend to use a login script, the modem scripting feature must be enabled in your KwikNet Library Parameter File. The scripting feature is enabled on the Modem property page.

3. KwikNet System Construction

3.1 Building an Application

If you are using KwikNet with AMX or have ported KwikNet to your operating environment, you are now ready to construct the KwikNet Libraries and build an actual KwikNet application. The sample program(s) provided with KwikNet and its optional components are working examples which you can use either for guidance or as a starting point for your own application.

To build a KwikNet application you must perform the following steps.

1. Using the KwikNet Configuration Builder, create and/or edit a Library Parameter File to select the KwikNet features which your application requires. On the Debug property page, enable some or all of KwikNet's debug features to assist you during initial testing. Use the builder to generate your KwikNet Library Make File. Using that file, create your KwikNet Libraries following the procedure to be described in Chapter 3.2.
2. If none of the available KwikNet device drivers meet your needs, create a custom device driver as described in the KwikNet Device Driver Technical Reference Manual.
3. If necessary, adapt the KwikNet board driver *KN_BOARD.C* to accommodate your target processor, device interfaces and interrupt management scheme. The board driver is also described in the KwikNet Device Driver Technical Reference Manual.
4. Using the KwikNet Configuration Builder, create and/or edit a Network Parameter File to describe your network interfaces and their associated device drivers. Use the builder to generate a KwikNet Network Configuration Module, a C file describing your networks.
5. Finally, create a make file which your make utility can use to build your application. It must compile your application modules, your KwikNet device drivers, your KwikNet board driver and your KwikNet Network Configuration Module. It can then link the resulting object modules with your KwikNet libraries, your RT/OS libraries and your C run-time library to create an executable load module. Follow the compilation and linking recommendations presented in Chapters 3.3, 3.4 and 3.5.
6. Use your software debugger and/or in-circuit emulator tools to transfer your load module to your target hardware. When testing, you should execute your application with a breakpoint on KwikNet procedure *kn_bphit()* so that you can readily detect fatal configuration or programming errors or unusual operation of the KwikNet TCP/IP Stack. Follow the testing guidelines presented in Chapter 1.9.

3.2 Making the KwikNet Libraries

To build the KwikNet Libraries, you will need a make utility capable of running your C compiler and object librarian (archiver). The library construction process is illustrated in Figure 3.2-1. If you have ported KwikNet to your operating environment, the shaded blocks indicate modules which you have already modified to adapt the make process to accommodate your software development tools. If you are using KwikNet with AMX, these modules are ready for use without modification.

Your custom KwikNet Libraries are created from the KwikNet Library Parameter File, a text file describing the TCP/IP features and options which your application requires. This file is created and edited using the KwikNet Configuration Builder as described in Chapter 2.

The KwikNet Configuration Builder uses the information in your Library Parameter File to generate a Network Library Make File. This make file is suitable for use with either Borland's *MAKE* or Microsoft's *NMAKE* utility. The make file purposely avoids constructs and directives that tend to vary among make utilities. Hence, you should have little difficulty using this make file with your own make utility if you so choose.

The make utility uses your C compiler and object librarian to generate the KwikNet Libraries from the KwikNet source modules and the OS Interface Module.

All KwikNet C files include a KwikNet compiler configuration header file *KNZZZCC.H*. This file identifies the characteristics of your C compiler. This file is also used to optimize code sequences within KwikNet modules by taking advantage of compiler specific features such as in-line code, assembly language functions and C library macros or functions. A number of variants of this module are provided with KwikNet ready for use with popular compilers on a variety of target processors.

The OS Interface Module *KN_OSIF.C* is the module which connects KwikNet to your RT/OS (see Figure 1.2-1 in Chapter 1). This module is merged into the KwikNet IP Library. The make process automatically includes the OS Interface Make File *KN_OSIF.INC* to determine the make dependencies and rules which control the compilation of the OS Interface source file *KN_OSIF.C*.

As you would probably expect, the make file does not know how to run your C compiler and object librarian. This information is provided in a file called *KNZZZCC.INC* which the make process automatically includes. This file, called a **tailoring file**, is used to tailor the library construction process to accommodate your make utility's syntax for implicit rules. It also provides the command sequences necessary to invoke your C compiler and object librarian. KwikNet is shipped with a number of tailoring files ready for use with many popular compilers using either Borland's *MAKE* or Microsoft's *NMAKE* utility.

Note

When KwikNet is used with AMX, the compiler configuration header file *KNZZZCC.H*, the OS Interface Make File *KN_OSIF.INC* and the tailoring file *KNZZZCC.INC* provided with KwikNet are ready for use without modification as described in Chapter 3.7.3.

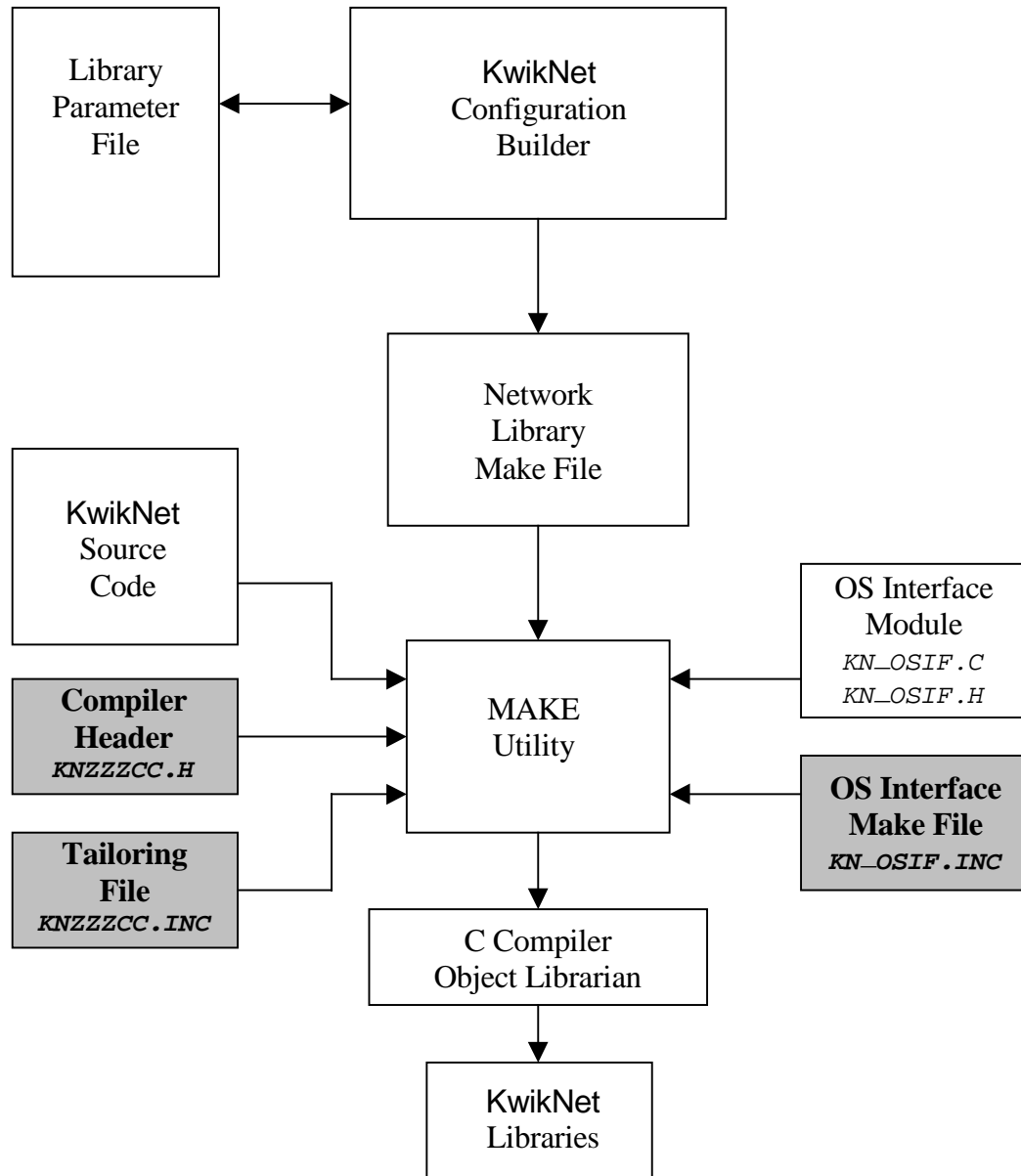


Figure 3.2-1 KwikNet Library Construction

KwikNet Directories and Files

The make process depends upon the structure of the KwikNet installation directory *KNTnnn*. When KwikNet is installed, the following subdirectories are created within directory *KNTnnn*.

<i>INET</i>	IP, UDP and related protocols; DHCP client; DNS client Ethernet and SLIP Network Drivers; Modem Driver Ethernet and Serial Loopback Drivers Universal File System Interface; Administration Interface
<i>TCP</i>	TCP protocol
<i>MAKE</i>	KwikNet make directory
<i>CFGBLDW</i>	KwikNet Configuration Builder; template files
<i>ERR</i>	Construction error summary
<i>TOOLXXX</i>	Toolset specific files
<i>TOOLXXX\LIB</i>	Toolset specific libraries will be built here
<i>TOOLXXX\DRIVERS</i>	KwikNet device drivers

Other subdirectories such as *PPP*, *FTP*, *HTTP* or *SNMP* will also be present if you have purchased the corresponding optional KwikNet components.

Other directories containing sample programs will also be present but are not involved in the library construction process.

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.

Within directory *TOOLXXX* you will find a collection of files of the form *B_tttXXX.vvv* and *M_tttXXX.vvv*. These files, called **tailoring files**, are used to tailor the library construction process for either Borland (*B_tttXXX.vvv*) or Microsoft (*M_tttXXX.vvv*) make utilities. The mnemonic *ttt* identifies the target processor. The extension *vvv* identifies the first version of the compiler for toolset *xxx* with which the tailoring file was tested. The tailoring file can be used with subsequent versions of the tools until some change in their method of operation requires a tailoring file update. For example, file *B_PPCMW.361* was first used to create the PowerPC KwikNet Libraries using Borland's *MAKE* and the MetaWare v3.61 C compiler.

Getting Ready

Before creating the KwikNet Libraries, you must pick your make utility, C compiler and object module librarian (archiver). Be aware that KADAK has observed that not all compilers operate correctly with every version of the Microsoft or Borland make utilities. If the make process inexplicably fails, it will most frequently be because of incompatibilities between these tools.

Pick the tailoring file which matches your choice of make utility, toolset and compiler version. Copy that file into the toolset directory but with name *KNZZZCC.INC*. You may have to overwrite the default copy created in that directory when KwikNet was installed.

Network Library Make File

The KwikNet Configuration Builder is used to create and edit your Library Parameter File, say *NETLIB.UP*. It is this file which describes the KwikNet options and features which your application requires. From this parameter file, the Configuration Builder generates the Network Library Make File, say *NETLIB.MAK*.

The Network Library Make File *NETLIB.MAK* is a make file which can be used to create the KwikNet Libraries tailored to your specifications. This make file is suitable for use with either Borland's *MAKE* or Microsoft's *NMAKE* utility.

Gathering Files

The block diagram in Figure 3.2-1 summarizes the components which are required to build the KwikNet Libraries. All of these files must be present in the appropriate KwikNet installation directories prior to making the KwikNet Libraries. Each of the following source files must be present in the indicated destination directory.

Source File	Destination Directory	File Purpose
<i>NETLIB.UP</i>	<i>MAKE</i>	KwikNet Library Parameter File
<i>NETLIB.MAK</i>	<i>MAKE</i>	KwikNet Library Make File
<i>KN_OSIF.C</i>	<i>INET</i>	OS Interface Module
<i>KN_OSIF.H</i>	<i>INET</i>	OS Interface Header File
<i>KN_OSIF.INC</i>	<i>TOOLXXX</i>	OS Interface Make Specification
<i>KNZZZCC.INC</i>	<i>TOOLXXX</i>	Tailoring File (for use with make utility)
<i>KNZZZCC.H</i>	<i>TOOLXXX</i>	Compiler Configuration Header File
<i>KNnnnnIP.LBM</i>	<i>TOOLXXX\LIB</i>	KwikNet IP Library Specification File
<i>KNnnnnTCP.LBM</i>	<i>TOOLXXX\LIB</i>	KwikNet TCP Library Specification File
<i>KNnnnn*.LBM</i>	<i>TOOLXXX\LIB</i>	Library Specification Files (for optional KwikNet Libraries)

Creating the KwikNet Libraries

The KwikNet Libraries must be constructed from within directory *MAKE* in the KwikNet installation directory. Your Library Parameter File, say *NETLIB.UP*, and your Network Library Make File, say *NETLIB.MAK*, must be present in the KwikNet *MAKE* directory.

All of the compilers and librarians used at KADAK were tested under Windows[®] NT. Many actually require Windows 9x or NT to operate.

To create the KwikNet Libraries, proceed as follows. From the Windows NT Start menu, choose the MS-DOS Command Prompt from the Programs folder. From the Windows 9x Start menu, choose the MS-DOS Prompt from the Programs folder. Make the KwikNet installation *MAKE* directory the current directory.

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

```
NMAKE -fNETLIB.MAK "TOOLSET=XXX" "OSPATH=yourospath" "KPF=NETLIB.UP"
```

To use Borland's *MAKE* utility, issue the following command.

```
MAKE -fNETLIB.MAK -DTOOLSET=XXX -DOSPATH=yourospath -DKPF=NETLIB.UP
```

In each case, the make symbol *TOOLSET* is defined to be the toolset mnemonic *xxx*. The symbol *OSPATH* is defined to be the string *yourospath*, the full path (or the path relative to directory *INET*) 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 make symbol *KPF* is defined to identify the name of the Library Parameter File *NETLIB.UP* from which the Network Library Make File *NETLIB.MAK* was generated. Both of these files must be present in the KwikNet *MAKE* directory.

By default, the KwikNet Libraries will be created in toolset dependent directory *TOOLXXX\LIB*. You can force the libraries to be created elsewhere by defining symbol *NETLIB=libpath* on the make command line. The string *libpath* is the full path (or the path relative to directory *INET*) to the directory in which you wish the libraries to be created. You must copy all library specification files (*.LBM) from toolset *xxx* directory *TOOLXXX\LIB* to your alternate library directory *libpath*.

Generated KwikNet Library Modules

All KwikNet source files will be compiled and the resulting object modules will be placed in directory *TOOLXXX\LIB*. The following KwikNet Libraries will be created from these object files and placed in directory *TOOLXXX\LIB*. Only those libraries needed to meet your library requirements will be created. Note that the library file extension will be *.A* or *.LIB* or some other extension as dictated by the toolset which you are using.

<i>KNnnnIP.A</i>	KwikNet IP Library
<i>KNnnnTCP.A</i>	KwikNet TCP Library
<i>KNnnnOPT.A</i>	KwikNet Library for optional KwikNet component <i>OPT</i>
	For example, <i>OPT</i> may be one of <i>PPP</i> , <i>FTP</i> , <i>WEB</i> or <i>SNM</i> .

In addition to the library modules and the object modules used to create them, the following files will also be created in directory *TOOLXXX\LIB*.

<i>KN_LIB.UP</i>	KwikNet Library Parameter File
<i>KN_LIB.MAK</i>	KwikNet Network Library Make File
<i>KN_LIB.H</i>	KwikNet Library Configuration Module

File *KN_LIB.UP* is a copy of the Library Parameter File *NETLIB.UP* which you identified on your make command line. It is copied to the *LIB* directory so that you have a record of the parameters used to produce the libraries present in the directory.

File *KN_LIB.MAK* is a KwikNet Network Library Make File which can be used to reproduce the libraries. It is generated in the *LIB* directory so that you have a record of the make file used to produce the libraries present in the directory. This file is derived from the KwikNet Library Make Template file *KNnnnLIB.MT* and the parameters in Library Parameter File *KN_LIB.UP*. It should match the make file *NETLIB.MAK* which you passed to your make utility to start the make process.

File *KN_LIB.H* is the KwikNet Library Configuration Module, a C header file generated by the make process. This file is derived from the KwikNet Library Configuration Template file *KNnnnLIB.HT* and the parameters in Library Parameter File *KN_LIB.UP*.

A copy of header file *KN_LIB.H* will also be found in the *INET* directory. The make process copies the file there so that it is available for inclusion in the compilation of all C files in the libraries.

A copy of the toolset dependent header file *TOOLXXX\KNZZZCC.H* will also be found in the *INET* directory. The make process copies the file there so that it is also available for inclusion in the compilation of all C files in the libraries.

Note

If your library specification requires KwikNet components which you have not purchased and installed, the make process will terminate because of the missing source files.

3.3 Compiling the Network Configuration Module

Each of the physical network connections in your target hardware must be described in the KwikNet Network Configuration Module.

The KwikNet Configuration Builder is used to create and edit your Network Parameter File, say *NETCFG.UP* (see Chapter 2.4). It is this file which describes each network and its associated device driver. From this parameter file, the Configuration Builder generates the Network Configuration Module. This process has been described in Chapter 2.1.

The Network Configuration Module is a C source file which must be compiled to produce an object module which is then linked as part of your application.

In order to compile the Network Configuration Module, say *NETCFG.C*, the following KwikNet header files must be present in the same directory as file *NETCFG.C*. Alternatively, you may choose to define the path to these header files using compiler switches or environment variables.

<i>KN_LIB .H</i>	KwikNet Library Configuration Module
<i>KN_API .H</i>	KwikNet Application Interface definitions
<i>KN_COMN .H</i>	KwikNet Common Interface definitions
<i>KN_OSIF .H</i>	KwikNet OS Interface definitions
<i>KNZZZCC .H</i>	KwikNet compiler specific definitions

Header file *KN_LIB.H* is a copy of your KwikNet Library Configuration Module from your KwikNet library directory. This file is created as a byproduct of the KwikNet Library construction process described in Chapter 3.2.

Header files *KN_API.H*, *KN_COMN.H* and *KN_OSIF.H* are the KwikNet files with which all application modules must be compiled. These files will be found in KwikNet installation directory *INET*.

Header file *KNZZZCC.H* is the compiler specific file which will be found in KwikNet installation directory *TOOLXXX*, where *xxx* is KADAK's three character mnemonic for a particular vendor's C tools.

The KwikNet Network Configuration Module is compiled using exactly the same C command line switches as are used for compiling the C modules in the KwikNet libraries. These command line switches are defined in the tailoring file *KNZZZCC.INC* which you used to create your KwikNet libraries with your particular C compiler (see Chapter 3.2).

Note

The make files for the sample programs provided with KwikNet and its optional components automatically create and compile the sample's Network Configuration Module from the sample Network Parameter File.

3.4 Compiling Application Modules

In order to compile an application C source file, say *MYFILE.C*, which makes use of KwikNet services, the following KwikNet header files must be present in the same directory as file *MYFILE.C*. Alternatively, you may choose to define the path to these header files using compiler switches or environment variables.

<i>KN_LIB .H</i>	KwikNet Library Configuration Module
<i>KN_API .H</i>	KwikNet Application Interface definitions
<i>KN_COMN .H</i>	KwikNet Common Interface definitions
<i>KN_OSIF .H</i>	KwikNet OS Interface definitions
<i>KNZZZCC .H</i>	KwikNet compiler specific definitions
<i>KN_SOCK .H</i>	KwikNet Socket Interface definitions

Header file *KN_LIB.H* is a copy of your KwikNet Library Configuration Module from your KwikNet library directory. This file is created as a byproduct of the KwikNet Library construction process described in Chapter 3.2.

Header files *KN_API.H*, *KN_COMN.H* and *KN_OSIF.H* are the KwikNet files with which all application modules must be compiled. Any module which includes *KN_LIB.H* will automatically include these header files as well. These files will be found in KwikNet installation directory *INET*.

Header file *KNZZZCC.H* is the compiler specific file which will be found in KwikNet installation directory *TOOLXXX*, where *XXX* is KADAK's three character mnemonic for a particular vendor's C tools.

Header file *KN_SOCK.H* must be included by all applications which use TCP or UDP socket services. This file will be found in KwikNet installation directory *TCP*.

If source file *MYFILE.C* makes calls to RT/OS service procedures, you must also have access to all of the required RT/OS header files.

You must also have access to your C library header files so that KwikNet header files can reference them.

With these header files in place, your application module *MYFILE.C* is ready to be compiled. If you are using KwikNet with AMX, the procedure is exactly as described in the toolset specific chapter of the AMX Tool Guide. If you have ported KwikNet using your own software development tools, the procedure will be the same as you used to compile the sample program source files.

Note

The make files for the sample programs provided with KwikNet and its optional components compile all of the modules which make up the sample program.

3.5 Linking the Application

To add KwikNet to your application, the following KwikNet object modules and libraries must be added to your link specification. It is recommended that these modules be included in the order listed.

<i>NETCFG</i> .O	KwikNet Network Configuration Module
<i>KNnnnnOPT</i> .A	KwikNet Library for optional KwikNet component <i>OPT</i> For example, <i>OPT</i> may be one of <i>PPP</i> , <i>FTP</i> , <i>WEB</i> or <i>SNM</i> .
<i>KNnnnnTCP</i> .A	KwikNet TCP Library
<i>KNnnnnIP</i> .A	KwikNet IP Library

Note that object file extensions may be .O or .OBJ and library file extensions may be .A or .LIB. Either may be some other extension as dictated by the toolset which you are using.

The Network Configuration Module *NETCFG.O* is the module created and compiled as described in Chapter 3.3. When using AMX, this module should precede your AMX System Configuration Module in the link sequence.

The KwikNet IP Library must always be included in the link. The TCP Library is only required if your application uses the sockets API or if another optional protocol is used which depends on TCP. The KwikNet Libraries should precede the AMX or RT/OS libraries in the link sequence.

Although every effort has been made to ensure that each module in a KwikNet Library contains only forward references to other modules in the same library, the goal has proved impossible to attain. Hence, some library modules do include backward references. This characteristic requires that the libraries be searched recursively until all resolvable references have been satisfied. Most linkers will meet this requirement. If yours does not, you will be forced to include some of the KwikNet libraries more than once in your link specification.

If you are using KwikNet with AMX, there is little difference in linking an AMX application with or without KwikNet. Instructions for linking an AMX system are provided in the toolset specific chapters of the AMX Tool Guide.

Warning

The KwikNet Libraries must be linked in the order specified. The TCP Library must precede the IP Library. Although the libraries for optional KwikNet components must precede the TCP Library, the specific order of each is not critical.

3.6 Making the TCP/IP Sample Program

The KwikNet TCP/IP Stack includes a sample program, a working application that you can use to confirm the operation of KwikNet. Other sample programs are provided with optional KwikNet components such as the FTP Option and the Web Server.

The KwikNet Application Block Diagram (see Figure 1.2-1 in Chapter 1) summarizes the components which are fundamental to any KwikNet application. All of these components must be present in the appropriate KwikNet installation directories prior to making any of the KwikNet sample programs.

TCP/IP Sample Program Directories

When KwikNet is installed, the following subdirectories on which the TCP/IP Sample Program construction process depends are created within directory *KNTnnn*.

<i>INET</i>	IP, UDP and related protocols Ethernet Network Driver Ethernet and Serial Loopback Drivers
<i>TCP</i>	TCP protocol
<i>CFGBLDW</i>	KwikNet Configuration Builder; template files
<i>ERR</i>	Construction error summary
<i>TOOLXXX</i>	Toolset specific files
<i>TOOLXXX\DRIVERS</i>	KwikNet device drivers and board driver
<i>TOOLXXX\LIB</i>	Toolset specific libraries will be built here
<i>TOOLXXX\SAM_MAKE</i>	Sample program make directory
<i>TOOLXXX\SAM_TCP</i>	KwikNet TCP/IP Sample Program directory
<i>TOOLXXX\SAM_COMN</i>	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.

Other subdirectories such as *TOOLXXX\SAM_FTP* or *TOOLXXX\SAM_WEB* will also be present if you have purchased the corresponding optional KwikNet components.

TCP/IP Sample Program Files

To build the KwikNet TCP/IP Sample Program using make file *KNSAMPLE.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 <i>INET</i> and <i>TCP</i> directories containing:	
<i>KN_API.H</i>	<i>INET</i>	KwikNet Application Interface definitions
<i>KN_COMN.H</i>	<i>INET</i>	KwikNet Common definitions
<i>KN_OSIF.H</i>	<i>INET</i>	KwikNet OS Interface definitions
<i>KN_SOCK.H</i>	<i>TCP</i>	KwikNet Socket Interface definitions
	Toolset root directory containing:	
<i>KN_OSIF.INC</i>	<i>TOOLXXX</i>	OS Interface Make Specification
<i>KNZZZCC.INC</i>	<i>TOOLXXX</i>	Tailoring File (for use with make utility)
<i>KNZZZCC.H</i>	<i>TOOLXXX</i>	Compiler Configuration Header File
	KwikNet TCP/IP Sample Program directory containing:	
<i>KNSAMPLE.MAK</i>	<i>TOOLXXX\SAM_TCP</i>	TCP/IP Sample Program make file
<i>KNSAMPLE.C</i>	<i>TOOLXXX\SAM_TCP</i>	TCP/IP Sample Program
<i>KNZZZAPP.H</i>	<i>TOOLXXX\SAM_TCP</i>	TCP/IP Sample Program Application Header
<i>KNSAMLIB.UP</i>	<i>TOOLXXX\SAM_TCP</i>	Library Parameter File
<i>KNSAMNCF.UP</i>	<i>TOOLXXX\SAM_TCP</i>	Network Parameter File
<i>KNSAMPLE.LKS</i>	<i>TOOLXXX\SAM_TCP</i>	Link Specification File (toolset dependent)
		Other toolset dependent files may be present.
<i>KNSAMSCF.UP</i>	<i>TOOLXXX\SAM_TCP</i>	User Parameter File (for use with AMX)
<i>KNSAMTCF.UP</i>	<i>TOOLXXX\SAM_TCP</i>	Target Parameter File (for use with AMX)
	Common sample program source files:	
<i>KNSAMOS.C</i>	<i>TOOLXXX\SAM_COMN</i>	Application OS Interface
<i>KNSAMOS.H</i>	<i>TOOLXXX\SAM_COMN</i>	Application OS Interface header file
<i>KNRECORD.C</i>	<i>TOOLXXX\SAM_COMN</i>	Message recording services
<i>KNCONSOL.C</i>	<i>TOOLXXX\SAM_COMN</i>	Console driver
<i>KNCONSOL.H</i>	<i>TOOLXXX\SAM_COMN</i>	Console driver header
		Console driver serial I/O support:
<i>KN8250S.C</i>	<i>TOOLXXX\SAM_COMN</i>	INS8250 (NS16550) UART driver
<i>KN_BOARD.C</i>	<i>TOOLXXX\DRIVERS</i>	Board driver for target hardware
<i>KN_DVCIO.H</i>	<i>TOOLXXX\DRIVERS</i>	Common device I/O definitions

TCP/IP Sample Program Parameter Files

Two KwikNet parameter files are provided with the KwikNet TCP/IP Sample Program.

The Library Parameter File *KNSAMLIB.UP* describes the KwikNet options and features illustrated by the sample program. This file is used to construct the KwikNet Libraries for the TCP/IP Sample Program.

The Network Parameter File *KNSAMNCF.UP* describes the network interfaces and the associated device drivers which the sample program needs to operate. This file is used to construct the KwikNet Network Configuration Module for the TCP/IP Sample Program.

TCP/IP Sample Program KwikNet Libraries

Before you can construct the KwikNet TCP/IP Sample Program, you must first build the associated KwikNet Libraries.

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

Look for any KwikNet Library Configuration Module *KN_LIB.H* in your toolset library directory *TOOLXXX\LIB*. If the file exists, delete it to ensure that the KwikNet Libraries are rebuilt to match the needs of the TCP/IP Sample Program.

Then copy files *KNSAMLIB.UP* and *KNSAMLIB.MAK* into the *MAKE* directory in the KwikNet installation directory *KNTnnn*. Use either the Borland or Microsoft make utility and your C compiler and librarian to generate the KwikNet TCP and IP Libraries. Follow the guidelines presented in Chapter 3.2.

Note

The KwikNet Libraries must be built before the TCP/IP 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 Libraries.

The TCP/IP Sample Program Make Process

Each KwikNet sample program must be constructed from within the sample program directory in the KwikNet toolset directory. For example, the KwikNet TCP/IP Sample Program must be built in directory *TOOLXXX\SAM_TCP*.

All of the compilers and librarians used at KADAK were tested under Windows® NT. Many actually require Windows 9x or NT to operate.

To create the KwikNet TCP/IP Sample Program, proceed as follows. From the Windows NT Start menu, choose the MS-DOS Command Prompt from the Programs folder. From the Windows 9x Start menu, choose the MS-DOS Prompt from the Programs folder. Make the KwikNet toolset *TOOLXXX\SAM_TCP* directory the current directory.

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

```
NMAKE -fKNSAMPLE.MAK "TOOLSET=XXX" "OSPATH=yourospath" "TPATH=toolpath"
```

To use Borland's *MAKE* utility, issue the following command.

```
MAKE -fKNSAMPLE.MAK -DTOOLSET=XXX -DOSPATH=yourospath -DTPATH=toolpath
```

In each case, the make symbol *TOOLSET* is defined to be the toolset mnemonic *xxx*. The symbol *OSPATH* is defined to be the string *yourospath*, the full path (or the path relative to directory *TOOLXXX\SAM_TCP*) 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.

The make process uses the sample program Network Parameter File *KNSAMNCF.UP* to create Network Configuration Module *KNSAMNCF.C* from the template file *KNnnnCFG.CT* in directory *CFGBLDW*. The file is left in the sample program directory *TOOLXXX\SAM_TCP*.

The KwikNet TCP/IP Sample Program load module *KNSAMPLE.xxx* is created in toolset directory *TOOLXXX\SAM_TCP*. 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 TCP/IP Sample Program load module *KNSAMPLE.xxx*.

3.7 Using KwikNet with AMX

3.7.1 AMX System Configuration

KwikNet includes its own interface to the underlying operating system. The KwikNet OS Interface for AMX is ready for use without modification or customization.

KwikNet makes few demands for AMX resources. Consequently, there are few changes to your AMX System Configuration Module required to accommodate KwikNet.

KwikNet Task

A single KwikNet Task drives the KwikNet TCP/IP Stack. You must add this task to your list of predefined tasks in your AMX System Configuration Module. You can use the AMX Configuration Builder to do so. The task's definition is as follows:

Tag	<i>KNET</i>
Procedure name	<i>kn_task</i>
Priority	Above all tasks which use KwikNet
Task stack size	AMX minimum plus 1024
Queue 0	0
Queue 1	0
Queue 2	0
Queue 3	0

With AMX 86 and some toolsets, you may have to add a leading or trailing underscore (*_*) character to the task procedure name. Note that the KwikNet Task must execute at a priority above all tasks which use KwikNet services.

The task stack size requirement will vary with the particular version of AMX you are using. As a good rule of thumb, choose a stack size which is approximately 1024 bytes more than the minimum stack size required for an AMX task. Add more stack if any of the following conditions exist.

- Target is a RISC processor with increased stack demands
- KwikNet options are used which make use of file system services
- KwikNet debugging aids are enabled in your KwikNet Libraries

For all versions of AMX, the KwikNet Task is a simple task with no AMX message queues. KwikNet uses private messaging blocks for internal communication with the KwikNet Task. The total number of available messaging blocks is defined by you on the Target property page when you create your KwikNet Library Parameter File. You may have to grow the number of messaging blocks if any of the following conditions exist.

- You have many tasks using network services
- You service several networks concurrently
- You expect high levels of network packet activity

AMX Interrupt Stack

You may have to grow the size of your AMX Interrupt Stack, the stack used by all Interrupt Service Procedures. The stack must be large enough to meet the needs of each of the KwikNet device drivers which service your physical network connections. As a good rule of thumb, choose a stack size which is approximately 500 bytes more than the minimum required AMX Interrupt Stack size.

KwikNet Semaphores

KwikNet requires one semaphore for its operation. This semaphore will be created dynamically by KwikNet during its initialization phase. You must include the AMX Semaphore Manager in your AMX System Configuration Module by declaring a requirement for at least one (1) semaphore. If you configure KwikNet to use standard C for memory allocation with memory locking enabled, an additional semaphore will be needed. If you use a file system other than AMX/FS and require file access locking, allocate one more semaphore.

KwikNet Memory Pool

If you configure KwikNet to use AMX memory management services, one AMX memory pool will be required. The memory pool will be created dynamically by KwikNet during its initialization phase. You must include the AMX Memory Manager in your AMX System Configuration Module by declaring a requirement for at least one memory pool. The memory for the pool must be allocated by you. Use the KwikNet Configuration Builder to edit your Network Parameter File and select one of the memory assignment techniques specified on the OS property page.

KwikNet Timer

KwikNet requires one AMX timer for its operation. This timer will be created dynamically by KwikNet during its initialization phase. You must include the AMX Timer Manager in your AMX System Configuration Module by declaring a requirement for at least one timer. Of course, to support timing you must also include an AMX Clock Handler as part of your application.

The KwikNet timer operates at the network clock frequency defined by you in your KwikNet Library Parameter File. The period of the network clock must correspond to an integer multiple of AMX system ticks. For example, you may have a hardware clock interrupt frequency of 1KHz with an AMX tick frequency of 100 Hz and a KwikNet network frequency of 10 Hz. In this case, the KwikNet timer will operate at 100 ms intervals measured with 10 ms resolution.

KwikNet Restart and Exit Procedures

You must include KwikNet Restart Procedure *kn_osready* first (or near first) in your list of Restart Procedures in your AMX System Configuration Module.

KwikNet includes a startup procedure *kn_enter* and a shutdown procedure *kn_exit*. You can include the KwikNet startup procedure *kn_enter* in your list of Restart Procedures in your AMX System Configuration Module. It is this procedure which starts the KwikNet Task to initialize the KwikNet TCP/IP Stack. Alternatively, one of your own Restart Procedures can call *kn_enter()*. The position of this procedure in the list of Restart Procedures is not critical since no KwikNet services can be used by tasks until the KwikNet Task has executed. Another approach is to have a task call *kn_enter()* to start KwikNet. Be certain that no other task tries to use KwikNet services until KwikNet is started.

If your AMX application allows an orderly shutdown and exit from AMX, you can add the KwikNet shutdown procedure *kn_exit* to your list of Exit Procedures in your AMX System Configuration Module. Alternatively, one of your own Exit Procedures can call *kn_exit()*. Insert this procedure into the list at the point in the exit sequence at which the KwikNet TCP/IP Stack is no longer required. You must ensure that all tasks have stopped using KwikNet services before you allow KwikNet to shut down. You can use KwikNet service procedure *kn_state()* for this purpose.

AMX 86 and AMX 386/EP PC Supervisor

Both AMX 86 and AMX 386/EP include a component called the PC Supervisor which permits these versions of AMX to be used with DOS on PC platforms. Special care must be taken when using the PC Supervisor with AMX and KwikNet.

The PC Supervisor's Clock Tick Task and Keyboard Task must be of higher priority than the KwikNet Task to ensure that they operate without interference from network activity.

All tasks which use KwikNet services must be of lower priority than the KwikNet Task. The PC Supervisor Task must be of lower priority than all application tasks which use KwikNet so that it does not interfere with their use of the network.

These task prioritization rules work provided that tasks which use KwikNet services never go compute bound. For example, if a task continuously polls KwikNet to test for the completion of some network operation, then any higher priority task which attempts to use DOS services will appear to hang because the low priority PC Supervisor Task is unable to execute to service the DOS request. In such cases, you will have no choice but to raise the priority of the PC Supervisor Task and accept the fact that DOS operations can temporarily block tasks of lower priority.

If you examine the KwikNet Sample Program provided with AMX 86 or AMX 386/EP, you will observe that the PC Supervisor Task has actually been placed at a priority above the KwikNet Task. This violation of the priority rules was done intentionally for the following reason. The Sample Program can operate without any physical network interfaces. Consequently, the application tasks can execute in a compute bound fashion because they never have to wait for real devices to respond. This scenario prevents the PC Supervisor Task from servicing any request by the Sample Program's Print Task to present messages on the PC display screen. By raising the priority of the PC Supervisor Task above that of the KwikNet Task, all messages appear on the PC display screen as soon as they are generated, making it easier for you to observe the actual sequence of operations.

3.7.2 AMX Target Configuration

Each KwikNet device driver for AMX includes an Interrupt Service Procedure consisting of two (sometimes three) parts. All drivers require an ISP root and an Interrupt Handler. Some versions of AMX also require the driver to provide an ISP stem.

An ISP root is required for each device interrupt source which the KwikNet board driver module *KN_BOARD.C* is configured to support. Unless modified by you, the board driver supports four ISP roots with names of the form *kn_isprootX()* (*X* is *A*, *B*, *C* or *D*). KwikNet dynamically assigns each network device to one of these ISP roots when the network is initialized.

32-Bit AMX Systems

Each ISP root is serviced by a common Interrupt Handler *kn_isphandler()* located in the KwikNet board driver module *KN_BOARD.C*. The handler is called with a single pointer parameter which identifies the network device which generated the interrupt. Four parameters with names of the form *kn_ispparamX* (*X* is *A*, *B*, *C* or *D*) are provided in the board driver module, one for each of the four ISP roots with like names.

An ISP stem *kn_ispstem()* in the KwikNet board driver module *KN_BOARD.C* is provided when required by AMX. The ISP stem also receives the device specific parameter *kn_ispparamX* from the ISP root.

For all 32-bit implementations of AMX, the ISP must be described in the AMX Target Configuration Module. With older 32-bit versions of AMX, this was done by editing a Target Parameter File to include a *...ISPC* directive. Newer versions of AMX include an updated AMX Configuration Builder, a Windows® utility which allows the Target Parameter File to be easily edited to add ISP definitions.

There must be one ISP definition for each of the device interrupt sources which the KwikNet board driver module *KN_BOARD.C* is configured to support. Each ISP definition identifies the names of the ISP root, the ISP Handler and the ISP stem if applicable. Each ISP definition also provides the appropriate pointer parameter *kn_ispparamX*. No interrupt vector is included in the definition since each KwikNet device driver automatically installs the pointer to its ISP root into the AMX Vector Table when KwikNet is initialized by the KwikNet Task.

See the KwikNet TCP/IP Sample Program Target Parameter File *KNSAMTCF.UP* for an example of the definition of the four ISPs supported by the KwikNet board driver module *KN_BOARD.C*.

16-Bit AMX 86 Systems

AMX 86 does not utilize a Target Configuration Module. The KwikNet board driver provided with AMX 86 creates an ISP root named *kn_isprootX()* (*X* is *A*, *B*, *C* or *D*) for each of the device interrupt sources which it is configured to support. Each ISP root *kn_isprootX()* calls its Interrupt Handler *kn_ispsrcX()* which in turn calls procedure *kn_isphandler()* with the device specific *kn_ispparamX* parameter. All of these procedures are located in the board driver module *KN_BOARD.C*. Each KwikNet device driver for AMX 86 automatically installs the pointer to one of these ISP roots into the AMX Vector Table when KwikNet is initialized by the KwikNet Task.

3.7.3 Toolset Considerations

Tailoring Files

The KwikNet Libraries are constructed using your make utility, C compiler and object module librarian. A file which KADAK calls a **tailoring file** is used to tailor the library construction process for a particular C compiler and object librarian. Separate tailoring files are available for each toolset combination which KADAK supports. These tailoring files are provided ready for use with either Borland's *MAKE* or Microsoft's *NMAKE* utility.

KADAK uses a 2 or 3 character toolset mnemonic to identify each supported toolset combination. The tailoring files for toolset *xxx* are located in directory *TOOLxxx* in the KwikNet installation directory *KNTnnn*. Use tailoring files *B_tttxxx.vvv* with Borland's *MAKE* and *M_tttxxx.vvv* with Microsoft's *NMAKE*. The mnemonic *ttt* identifies the target processor. The extension *vvv* identifies the first version of the compiler for toolset *xxx* with which the tailoring file was tested. The tailoring file can be used with subsequent versions of the tools until some change in their method of operation requires a tailoring file update. For example, file *M_PPCDA.42* was first used to create the PowerPC KwikNet Libraries using Microsoft's *NMAKE* and the Diab Data (toolset *DA*) v4.2 (42) C compiler.

Note

Pick the tailoring file which matches your choice of make utility, toolset and compiler version. Copy that file into toolset directory *TOOLxxx* but with name *KNZZZCC.INC*. You may have to overwrite the default copy created when KwikNet was installed.

Compiler Configuration Header File

All KwikNet C files include a KwikNet compiler configuration header file *KNZZZCC.H*. This file identifies the characteristics of your C compiler. When KwikNet is used with AMX, the compiler configuration header file *KNZZZCC.H* installed in KwikNet directory *TOOLxxx* is ready for use with the C compiler for toolset *xxx* without modification.

OS Interface Make File

The OS Interface Module *KN_OSIF.C* is the module which connects KwikNet to AMX (see Figure 1.2-1 in Chapter 1). This module is merged into the KwikNet IP Library. The make process automatically includes the OS Interface Make File *KN_OSIF.INC* to determine the make dependencies and rules which control the compilation of the OS Interface source file *KN_OSIF.C*. When KwikNet is used with AMX, the OS Interface Make File *KN_OSIF.INC* installed in KwikNet directory *TOOLxxx* is ready for use with toolset *xxx* without modification.

3.7.4 AMX Application Construction Summary

Construction of any KwikNet application for use with AMX will closely follow the steps needed to build the KwikNet TCP/IP Sample Program. These steps are summarized below. Note that the make file provided with KwikNet sample programs actually does steps 2, 6, 7, 8 and 9.

1. Using the KwikNet Configuration Builder, open the Sample Program's Library Parameter File *KNSAMLIB.UP* (see Chapter 2.3). Use the builder to generate your KwikNet Library Make File *KNSAMLIB.MAK*. Use this file, either the Borland or Microsoft make utility, and your C compiler and librarian to generate the KwikNet TCP and IP Libraries (see Chapter 3.2).
2. Using the KwikNet Configuration Builder, open the Sample Program's Network Parameter File *KNSAMNCF.UP* (see Chapter 2.4). Use the builder to generate your KwikNet Network Configuration Module *KNSAMNCF.C*. Use your C compiler to compile the Network Configuration Module (see Chapter 3.3).
3. If you wish to use your own clock driver, do step 4. If you wish to use your own serial driver for logging messages to a terminal, do step 5. Otherwise, go to step 6.
4. If you wish to use your own working AMX Clock Driver instead of the simulated clock provided by the sample program, edit the Sample Program User Parameter File and Target Parameter File (not required for AMX 86) to accommodate your clock driver. Edit the KwikNet Sample Program Link Specification File *KNSAMPLE.LKS* to include your clock driver object module.
5. If you ported the AMX Sample Program serial I/O driver to your hardware and wish to use it to log messages to a terminal, edit the sample program Application Header file *KNZZZAPP.H* and define symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVAMX*. Edit the KwikNet Sample Program Link Specification File *KNSAMPLE.LKS* to include your serial driver object module.
6. Using the AMX Configuration Builder, open the Sample Program's User Parameter File *KNSAMSCF.UP*. Use the builder to generate the AMX System Configuration Module *KNSAMSCF.C*. Compile the module as described in the AMX Tool Guide for the toolset which you are using.
7. If you are using AMX 86, go to step 8. Otherwise, using the AMX Configuration Builder, open the Sample Program's Target Parameter File. Use the builder to generate the AMX Target Configuration Module. Assemble the module as described in the AMX Tool Guide for the toolset which you are using.
8. Compile the KwikNet TCP/IP Sample Program application modules listed in Chapter 3.6. Compile these modules with full debug information to improve your view when running the sample with your debugger.
9. Link the modules listed in the KwikNet Sample Program Link Specification File *KNSAMPLE.LKS* together with your C Library to create your KwikNet application load module (see Chapter 3.5).
10. Use your debugger to load and execute the KwikNet Sample Program.

4. KwikNet IP/UDP Services

4.1 The UDP Programming Interface

Applications which are memory constrained or which have no need to use TCP can exclude the TCP stack and use only the KwikNet IP stack and its UDP application programming interface (API). Be careful not to confuse this low level UDP API with the UDP sockets interface available through the TCP sockets API described in Chapter 5.

The low level UDP API will only be present in your load module if your application makes calls to it. Of course, it will also be present if you enable features such as the DHCP client or DNS client which depend on it.

UDP is a connectionless protocol which uses only the unreliable IP layer for UDP datagram delivery. The KwikNet UDP implementation can be configured to checksum UDP datagrams. It is recommended that you enable UDP checksums when you create your KwikNet Libraries so that messages which you send using UDP can be validated. Received UDP datagrams which include a UDP checksum are always validated by KwikNet. Note that even if you enable UDP checksums, you may still receive an incorrect datagram from a host which does not generate UDP checksums.

The UDP Channel

KwikNet defines an abstraction called a UDP channel which it uses to control the sending and receiving of UDP datagrams on the network. UDP datagrams cannot be sent without first acquiring a UDP channel. Received UDP datagrams are rejected if an associated UDP channel does not exist.

To send or receive UDP datagrams, you must first call KwikNet procedure *kn_udpopen()* to open a UDP channel. In the call you must provide the IP address of the foreign host with whom you wish to correspond. An IP address of 0.0.0.0 is used to indicate that you will accept UDP datagrams from any foreign host. If you will only accept UDP datagrams from a specific foreign host, you must also provide the protocol port number for the foreign host. A foreign port number of 0 can be used to indicate that you will accept a UDP datagram from any port at that foreign host. Also required is the local protocol port number by which you wish your UDP channel to be identified. This latter parameter will be used by the foreign host to direct its response back to you.

The KwikNet procedure *kn_udpopen()* returns a handle which uniquely identifies the UDP channel allocated by KwikNet for your use. This handle can be used in calls to *kn_udpsend()* to send data through the UDP channel to any foreign host. However, you will only be able to receive UDP datagrams from the foreign host identified when the UDP channel was opened.

A foreign host requires your local host IP address to send it a UDP datagram. You can use KwikNet procedure *kn_udpbind()* to bind your UDP channel to a specific local IP address. Having done so, you can receive UDP datagrams on your UDP channel without first having to send a UDP datagram to the foreign host to identify your IP address.

Once your application is finished conversing with the foreign host, it must call KwikNet procedure *kn_udpclose* to close the UDP channel. The handle used to access the UDP channel becomes invalid once the channel is closed.

Receiving UDP Datagrams

If you expect to receive a UDP datagram from a foreign host, your open request must provide a pointer to an application callback (upcall) function which KwikNet can call upon receipt of such a UDP datagram. The callback function is prototyped as follows:

```
int user_udprecv(struct knx_udpmsg *msgp, void *userp);
```

Parameter *userp* is an application pointer provided by you in your request to open the channel on which this UDP datagram was received. It is a copy of the parameter found at *msgp->xudpm_user*.

Parameter *msgp* is a pointer to a KwikNet UDP message descriptor. Structure *knx_udpmsg* is defined in KwikNet header file *KN_API.H* as follows:

```
struct knx_udpmsg {
    struct in_addr xudpm_src;          /* IP address of source          */
    struct in_addr xudpm_dest;         /* IP address of destination    */
    int            xudpm_fport;        /* Foreign port (source)        */
    int            xudpm_lport;        /* Local port (destination)     */
    char           *xudpm_datap;       /* Pointer to UDP data          */
    int            xudpm_length;       /* Length of UDP data           */
    int            xudpm_rsv1;         /* Reserved for alignment       */
    void           *xudpm_user;        /* User parameter               */
    struct knx_ip   *xudpm_ip;         /* IP header pointer            */
    struct knx_udp  *xudpm_udp;        /* UDP header pointer           */
    void           *xudpm_pkt;        /* Packet pointer (reserved)    */
};
```

The UDP message structure describes the received UDP datagram. Fields *xudpm_src*, *xudpm_dest*, *xudpm_fport*, and *xudpm_lport* are extracted from the received packet and presented to your application in an easy to use form. The source and destination IP addresses are presented in net endian form in field *s_addr* of structure *in_addr*. The foreign and local ports are provided as integers in host endian form.

The data within the UDP datagram is located in the packet at the memory address specified by field *xudpm_datap*. The length of the data region is specified by field *xudpm_length*. Both of these fields are in host endian form.

If necessary, you can access the IP header and UDP header in the packet using the pointers provided. The packet pointer is reserved for the private use of KwikNet.

Your UDP callback function must return 0 if it accepts the UDP message descriptor. In this case, once your application has finished processing the UDP datagram, it must call KwikNet procedure *kn_udpfree* to release the UDP message descriptor and free the associated data packet for reuse by KwikNet.

Your UDP callback function must return -1 if it cannot accept the message descriptor. In this case, KwikNet will release the UDP message descriptor and free the associated data packet. It is important to note that KwikNet will not send an ICMP destination unreachable message to the originator of the rejected UDP datagram.

Processing Received UDP Datagrams

Your UDP callback function executes in the context of the KwikNet Task. Your function must not initiate any operation which would force the KwikNet Task to be blocked waiting for some event.

In most multitasking applications, it is recommended that your UDP callback function pass the UDP message descriptor to some other application task for processing. Often that task will be the same task that opened the UDP channel and initiated the conversation in the first place.

Even in single threaded systems, your UDP callback function should pass the UDP message descriptor to your App-Task for processing.

Note that your task has access to the copy of your application parameter located in field *xudpm_user* in the UDP message descriptor.

When your task finishes processing the UDP datagram, it must call KwikNet procedure *kn_udpfree* to release the UDP message descriptor and free the associated data packet for reuse by KwikNet.

Broadcast UDP Datagrams

A broadcast UDP datagram is a message directed to IP address 255.255.255.255. You can send and receive broadcast UDP datagrams on a UDP channel. When a broadcast UDP datagram is received, it is delivered to the first UDP channel which KwikNet can find with a matching local port number. If you have multiple UDP channels bound to different IP addresses but using the same local port number, a broadcast UDP datagram directed to that port will be delivered to the UDP channel which was opened first.

UDP Echo Requests

A UDP datagram directed to well known port 7 is called a UDP Echo Request. If KwikNet has been configured to support the UDP echo feature, KwikNet will act as a UDP Echo Server and echo the UDP datagram back to the sender's port.

Your application can handle UDP echo requests by opening a UDP channel on port 7. Any UDP datagram received for port 7 will be passed to your UDP callback function.

UDP Sockets

KwikNet supports the use of the UDP protocol with the same sockets interface used with the TCP/IP protocol (see Chapter 5). When a socket of type *SOCK_DGRAM* is created, KwikNet attaches a UDP channel to the socket and then uses that channel for all transactions which reference the socket. Standard KwikNet sockets calls can then be used by your application to communicate with a foreign host using the UDP socket.

4.2 The DHCP (BOOTP) Client

KwikNet includes support for the Dynamic Host Configuration Protocol (DHCP) which permits a network's IP address to be dynamically assigned when the network is first started. This feature is enabled by a configuration parameter in the KwikNet Library Parameter File used in the construction of the KwikNet Libraries as described in Chapter 2.3. When you define your KwikNet configuration, simply check the option box labeled "Include DHCP Client" on the DNS/DHCP property page.

Once you have enabled DHCP support, KwikNet allows each network interface to be individually DHCP enabled. To do so, check the DHCP Override option in the network's IP address definition in your Network Parameter File as described in Chapter 2.4.

The KwikNet DHCP client uses a DHCP message to discover its IP address. DHCP is the refined version of the early BOOT Protocol (BOOTP) which is now rarely used. When a DHCP query is used, it still operates correctly even if the only servers available are BOOTP servers. The format of the DHCP query is acceptable to a BOOTP server and the KwikNet DHCP client is able to distinguish the BOOTP reply from that of a DHCP server.

DHCP Operation

When KwikNet begins operation, it automatically starts its DHCP client to service each DHCP enabled network. The DHCP client automatically requests an IP address for each Ethernet network. For a SLIP or PPP network, the request for an IP address is deferred until the network is first used and a connection has been established.

The DHCP client uses UDP datagrams for the transmission of DHCP messages. Responses are expected to be in UDP datagrams.

The DHCP client broadcasts a DHCP query to all DHCP servers. If the network has been configured with an IP address other than 0.0.0.0, the IP address is sent in the DHCP options as a request to use that particular network IP address. Note that the broadcast DHCP query looks like a valid BOOTP query to any server which only supports the older BOOTP format.

Once an IP address offer is received from a DHCP server, the DHCP client responds with a request to unconditionally accept the offer. If the DHCP server acknowledges the acceptance of the offer, the DHCP client adopts the IP address thereby making the network ready for use by the application.

If a BOOTP server responds to the initial DHCP query, the DHCP client simply adopts the IP address provided by the BOOTP server thereby making the network ready for use by the application. There is no need for the acceptance and acknowledgment handshake.

The KwikNet DHCP client does not make use of the server host name or boot file name, if any, provided in the DHCP or BOOTP server response.

DHCP Timeout

If no response is received from any DHCP or BOOTP server within the timeout interval (initially four seconds), the DHCP client resends its broadcast query and increases its timeout interval by a factor of 2^n where n is the number of failed attempts thus far. This process continues forever if an IP address cannot be acquired. The retry timeout value is not allowed to exceed an upper limit of 64 seconds.

DHCP Leases

When a DHCP server provides an IP address, it grants the network interface a lease to use that address for a specific interval. The DHCP client always requests a permanent lease but can live with a limited lease interval if that is all that is granted by the DHCP server. Note that the lease granted by a BOOTP response is considered to be permanent.

The DHCP client always tries to renew a limited time lease by negotiating with the DHCP server which granted the lease. If the lease interval is L seconds, the DHCP client begins a lease renewal negotiation after $L/2$ seconds. Negotiation requests are repeated at intervals until $7L/8$ seconds into the lease. Each interval is half the period from the time of the last request to the $7L/8$ seconds mark, but never less than 60 seconds.

If a new lease is not granted, the DHCP server will attempt to negotiate the same IP address from another DHCP server on the same network. Failing that, the DHCP client will initiate a query for a new IP address from any DHCP server.

DNS Server Support

When a DHCP server responds to an IP address query, it can also provide a list of IP addresses for known Domain Name System (DNS) servers. If your KwikNet configuration includes the DNS client, the DHCP client can accept these IP addresses from the DHCP server and pass them to the DNS client for inclusion in its list of DNS servers.

To enable this feature, your KwikNet configuration must specify the maximum number of DNS server IP addresses which the DHCP client can accept. Additional DNS server IP addresses provided by the DHCP server will be ignored.

DHCP Option Request

The KwikNet DHCP client can include an option request list in its DHCP query. The option list indicates specific DHCP options which may be used at the discretion of the DHCP server. By sending the DHCP option request list, your DHCP client identifies the DHCP options which it is equipped to handle.

The KwikNet DHCP client option list indicates that subnet masks, gateways and DNS servers can be accepted from the DHCP server. The DNS server option will only be presented in the option list if your KwikNet configuration includes the DNS client.

4.3 The DNS Client

KwikNet includes support for the Domain Name System (DNS) which permits a network's IP address to be derived from a name string. For example, the IP address for KADAK's Internet website can be derived by doing a DNS query for the name string `www.kadak.com` using function `kn_gethostbyname()`.

This feature is enabled by setting the DNS configuration parameters in the KwikNet Library Parameter File used in the construction of the KwikNet Libraries as described in Chapter 2.3. The component within KwikNet which provides this service is call the DNS client. The DNS client executes in response to queries from your application and then relies upon the KwikNet Task to resolve the query, if necessary.

DNS Server List

The KwikNet DNS client maintains a list of the IP addresses of the DNS servers which it can query to resolve a domain name. Your application can add and remove DNS servers from this list using functions `kn_dns_srvadd()` and `kn_dns_srvdel()`. The KwikNet DHCP client uses these services to add DNS servers identified during DHCP negotiations and to remove them upon lease expiry. The PPP network driver also uses these services to add DNS servers identified during PPP negotiations and to remove them when a PPP network is closed.

When configuring your KwikNet Libraries, you can provide the name of a function which the KwikNet DNS client will call at startup to acquire an initial list of DNS servers which it can interrogate to translate names to IP addresses. Use of such a function is optional.

The function must return a pointer to an array of server IP addresses. The following example illustrates the requirement. Note that the IP addresses in the array must be stored in net endian form and that the list is terminated by an IP address of 0.0.0.0. If you use this feature, the name of your function, say `myDNSservers`, must be entered on the DNS/DHCP property page when you configure your KwikNet Libraries.

```
struct {                                /* IP address structure */
    unsigned char xIPAddr[4];
    } DNSservers[] =                    /* Declare an array of them */
{
    {192, 168, 0, 1},                  /* List server IP addresses */
    {192, 48, 132, 12},
    {192, 74, 3, 1},
    {192, 212, 66, 10},
    {0, 0, 0, 0}                       /* Terminate the list */
};

void *myDNSservers(void)                /* Fetch list of DNS servers*/
{
    return ((void *)DNSservers);
}
```

DNS Queries

The results of each DNS query are kept in a name cache maintained by the DNS client. Each cached entry includes one or more IP addresses provided by the DNS server which resolved the name. The total number of cached names and the maximum number of IP addresses per name are determined by you when you configure your KwikNet Libraries.

Once the KwikNet TCP/IP Stack has been initialized, your application can call KwikNet procedure *kn_dns_query()* to make DNS name queries. In a multitasking system, any task making such a query must be of lower priority than the KwikNet Task. If the name is already in the DNS client's name cache, you will be given the first available IP address immediately. If there is no room in the name cache, the oldest cached name will be purged so that your request can be granted. The KwikNet DNS client will then initiate the query and return with an indication that your query is underway.

When making the query, you can provide a callback function which the KwikNet DNS client will call once an IP address has been acquired. Each query can return one or more IP addresses. The callback function is prototyped as follows:

```
void UserDNS_callback(struct in_addr *answer, int status,  
                      void *userp);
```

If parameter *status* is 0, a list of IP addresses will be presented in the array of *in_addr* structures referenced by parameter *answer*. Each IP address will be presented in net endian form in field *s_addr* of the structure *in_addr*. The array will be terminated by an entry containing an IP address of 0. Your callback function must save the content of *answer* array before returning to the DNS client.

If parameter *status* is not 0, the IP address is unknown. In this case, the content of structure *in_addr* referenced by parameter *answer* will be undefined.

Parameter *userp* is a copy of the application pointer which you provided in your call to *kn_dns_query()* when you initiated the DNS query.

Your DNS callback function executes in the context of the KwikNet Task. Your function must not initiate any operation which would force the KwikNet Task to be blocked waiting for some event.

In most multitasking applications, it is recommended that your DNS callback function pass the IP address, or list of IP addresses, to some other application task for processing. Often that task will be the same task that initiated the DNS query in the first place.

Even in single threaded systems, your DNS callback function should pass the IP address, or list of IP addresses, to your App-Task for processing.

DNS Name Lookup

If your application wishes to determine if the IP address for a particular DNS name is already in the DNS client's cache without initiating a query, it can do so by calling KwikNet procedure *kn_dns_lookup()*. This procedure can also be used to poll for the result of a query if you chose not to provide a DNS callback function when you initiated the query. In a multitasking system, any task doing such a lookup must be of lower priority than the KwikNet Task.

Get Host By Name

For compatibility with other networking systems, KwikNet provides the function *gethostbyname()* which finds the IP address for a host with a specific domain name. Since this function is inherently non-reentrant, KwikNet provides an alternate, reentrant equivalent *kn_gethostbyname()* better suited for use in multitasking systems. These functions are only present in the KwikNet IP Library if the *gethostbyname()* API is enabled on the DNS/DHCP property page when you configured your KwikNet Libraries.

Once the KwikNet TCP/IP Stack has been initialized, your application can call KwikNet procedure *gethostbyname()* to make DNS name queries. In a multitasking system, any task making such a query must be of lower priority than the KwikNet Task. If the name is already in the DNS client's name cache, you will be given a list of the available IP addresses.

If there is no room in the name cache, the oldest cached name will be purged so that your request can be granted. The KwikNet DNS client will then initiate the DNS query.

Once the query is underway, KwikNet will poll at 200 ms. intervals for the results of the query. In a multitasking system, the task initiating the request will be blocked between polls. In a single threaded system, your App-Task will be blocked between polls. Polling will continue for some maximum interval or until the DNS query is resolved.

Applications which call procedure *kn_gethostbyname()* to make a DNS name query can specify the maximum interval to wait for a response to the query. If procedure *gethostbyname()* is used, the maximum wait interval is the value specified on the DNS/DHCP property page when you configured your KwikNet Libraries.

Interpreting DNS Results

Function `gethostbyname()` returns the results of a DNS name query in a static *hostent* structure maintained by KwikNet. Applications which call function `kn_gethostbyname()` must provide a *hostent* structure which has been initialized using procedure `kn_gethostprep()` prior to first use.

Structure *hostent* is defined in KwikNet header file *KN_API.H* as follows:

```
struct hostent {
    const char    *h_name;           /* Official name of host      */
    char          **h_aliases;       /* Alias list                  */
    int           h_addrtype;        /* Host address type          */
    int           h_length;          /* Length of address          */
    char          **h_addr_list;     /* List of IP address pointers */
};
```

Upon return from a successful DNS name query, the *hostent* structure will be filled as follows. Member *h_name* will reference the domain name string used in the DNS query. Since KwikNet does not support DNS aliases, member *h_aliases* will reference a *NULL* pointer. Member *h_addrtype* will be set to 2 (*AF_INET*), the IP address family type. Member *h_length* will be set to `sizeof(struct in_addr)`, the length of an IP address.

Member *h_addr_list* will reference an array of pointers to *in_addr* structures containing the IP addresses. Each IP address will be presented in net endian form in field *s_addr* of the *in_addr* structure. The pointer array will be terminated by a *NULL* pointer. The number of IP addresses in the list will never exceed *KN_DNSMAXADDRES*, the maximum number of IP addresses which your DNS client has been configured to accept from a DNS server.

The specification of function `gethostbyname()` allows it to return an array of pointers to network addresses of any type and length. For this reason, member *h_addr_list* is declared to be *char ***. Hence, a cast of the following form must be used to fetch each IP address.

```
((struct in_addr *) (hentp->h_addr_list[i]))->s_addr
```

Note

An example is provided in the description of function `kn_gethostbyname()` in Chapter 4.6. The example uses `kn_gethostprep()` to initialize a *hostent* structure and then interprets the results of the DNS query.

4.4 ICMP Protocol

KwikNet includes support for the subset of Internet Control Message Protocol (ICMP) services needed for proper network operation.

KwikNet always replies to an ICMP echo request (a PING) with an ICMP echo reply. Your application can make use of this facility as described later in this chapter.

Unless you have configured your KwikNet IP Library to include Full ICMP support, only ICMP echo requests and replies are supported. All other received ICMP datagrams are ignored; no other ICMP datagrams are transmitted.

When full ICMP is enabled, KwikNet supports timestamp requests. When KwikNet receives a timestamp request, it responds with a timestamp reply in which the time of receipt and time of reply are both set to `-1L`, indicating that timestamps are not available.

When full ICMP is enabled, KwikNet supports ICMP destination unreachable datagrams. KwikNet accepts such an ICMP datagram and makes it accessible to an application hook function. If KwikNet discards an IP datagram because it cannot be handled properly, an ICMP destination unreachable message is sent to the host from which the rejected IP datagram was received.

KwikNet maintains counts of the various ICMP datagrams which it sends and receives. These ICMP statistics are included in the network statistics log which you can enable by checking Enable network statistics in your KwikNet IP Library configuration.

ICMP Destination Unreachable Hook

KwikNet permits an application hook function to peek at each ICMP destination unreachable datagram received on any of your network interfaces. To use this feature, you must configure your KwikNet IP Library to include Full ICMP support.

The ICMP destination unreachable hook function is prototyped as follows:

```
void icmpduhook(const unsigned char *datap);
```

Parameter *datap* is a pointer to the ICMP destination unreachable message in the received IP datagram. Hence, *datap* is a pointer to the ICMP message header and data. Your application can examine but not modify the ICMP message content. Interpretation of the ICMP message content per RFC-792 is left to your application.

To enable your hook, your application must install a pointer to the hook function into the KwikNet public variable *kn_icmpdu_hook* which is declared in KwikNet header file *KN_API.H*. Your hook must be installed **after** KwikNet has been successfully started.

Your hook function executes in the context of the KwikNet Task. Your function must not initiate any operation which would force the KwikNet Task to be blocked waiting for some event. It is recommended that your hook function copy the relevant information from the ICMP message for processing by some other application task or by your App-Task.

Using PING

KwikNet includes support for the Packet InterNet Groper (PING), a process by which your application can determine if a particular destination is reachable. The term ping is used to describe the process by which a ping, an ICMP echo request datagram, is sent to a destination address. The destination host, if it exists, is expected to generate a ping reply, an ICMP echo response datagram.

This feature is always available in the KwikNet IP Library. KwikNet will always respond to a ping request received on any of its network interfaces by sending a valid ICMP echo response datagram.

Note that your application memory image (load module) will not include the additional PING support needed to generate a ping and handle a ping response unless your application actually initiates a ping.

Initiating a Ping

To initiate a ping, your application must call KwikNet procedure *kn_pingsend()*. In a multitasking system, any task making such a query must be of lower priority than the KwikNet Task.

When you make your ping request, you must provide the IP address of the destination which you wish to ping. You can also identify a specific data block which you wish to send as part of the ping message. In the absence of such a data block, KwikNet will send a data string which identifies KwikNet as the source of the ping.

When you issue a ping, you can also include a sequence number. The sequence number is sent to the destination but is ignored in the response.

Warning!

In a multitasking system, only one task at a time can issue a ping. It is your responsibility to ensure that two tasks do not attempt to concurrently use KwikNet's ping services.

Handling a Ping Reply

By default, KwikNet simply discards the ping response when it is received. If your application wants to examine the ping response, you must build the KwikNet IP Library accordingly. Check the option box labeled "Application handles PING replies" in the IP configuration parameters in the KwikNet Library Parameter File used in the construction of the KwikNet Libraries as described in Chapter 2.3.

When configured to allow your application to handle ping replies, KwikNet will pass the ping response to your ping callback function. To use this feature, you must call KwikNet procedure *kn_pinguserfn()* to identify your ping callback function before you call *kn_pingsend()* to issue the ping. When you initiate the ping, you can specify the maximum interval that you are prepared to wait for a ping reply to be delivered to your callback function.

The ping callback function is prototyped as follows:

```
void UserPING_callback(char *datap, int length);
```

Parameter *datap* is a pointer to the data block in the datagram received from the pinged destination. Parameter *length* specifies the number of bytes in that data block. The data block is expected to be a copy of the data block sent in the initial ping message.

If parameter *length* is 0, then one of two conditions exist. Either no ping reply was received within the timeout interval or a ping response was received with no data.

Your callback function remains active until such time as you cancel it. You can cancel a callback function by calling KwikNet procedure *kn_pinguserfn()* with a *NULL* callback procedure pointer.

Note that your callback procedure gets to view every ping response received on each of the network interfaces managed by KwikNet. It is therefore advisable to cancel your procedure as soon as it receives the ping response of interest.

4.5 KwikNet State Management

Network States

When KwikNet starts, it initializes each of the network drivers (Ethernet, SLIP or PPP) needed to support your network interfaces.

The Ethernet network driver, once started, remains operational until KwikNet is shut down, if ever.

The SLIP and/or PPP serial network drivers, once initialized for use, normally remain idle until first needed to support data transfer requests by your application. At that time the network is automatically brought up and made ready for use. This method of operation is termed an autostart. However, KwikNet provides a network state management service which permits these networks to be made operational prior to first use. The service also allows these networks to be shut down when no longer required.

KwikNet procedure *kn_inet_ifstate()* implements this network state management service. Using this procedure you can start and stop any SLIP or PPP network and determine the current operational state of any network.

By default, PPP and SLIP networks autostart as soon as your application tries to use the network. The autostart feature works well for serial network interfaces which are directly connected to their peer. However, the autostart feature can introduce difficulties when used with networks that are regularly brought up and shut down, such as those connected by modem. Therefore, a network's autostart ability is permanently disabled whenever procedure *kn_inet_ifstate()* is used to alter the network's state. Hence, after shutting down a PPP or SLIP network, you must explicitly bring that network up again before it will be available for use.

KwikNet States

Most applications start KwikNet with a call to *kn_enter()* and allow KwikNet to run forever. Some use *kn_exit()* to stop KwikNet in preparation for a termination of the entire application. Others find it necessary to start and stop KwikNet as required to adapt to changing network conditions or to recover from serious network faults. The ability to start and stop KwikNet on demand is also useful when testing your network application.

KwikNet is started with a call to *kn_enter()*. Procedure *kn_state()* can then be used to detect when KwikNet is fully operational.

Before KwikNet can be stopped, your application must cease using all KwikNet services. You must ensure that all KwikNet resources such as UDP channels and TCP or UDP sockets have been relinquished. Furthermore, all hooks such as your ICMP hook or PING callback must be removed or otherwise rendered inoperative. Do not forget that all dedicated KwikNet clients and servers (such as those for FTP, HTTP, TELNET and SNMP) must have been stopped in an orderly fashion as specified in their documentation. The private KwikNet DHCP and DNS clients will be stopped by KwikNet.

KwikNet is stopped in two steps. The first step is to wait long enough for all unfinished network transactions to complete. KwikNet procedure *kn_godown()* provides this service. Your application can call *kn_godown()* to start the shutdown process and wait, up to some maximum interval, for the process to complete.

The second step is to force KwikNet to shut down all networks and their device drivers and release all memory and operating system resources. This process is initiated with a call to *kn_exit()*. Procedure *kn_exit()* does not return to the caller until KwikNet has fully stopped. Then, and only then, can KwikNet be restarted.

If your application does not call *kn_godown()* to shut down KwikNet before calling *kn_exit()*, KwikNet will automatically attempt a shutdown, waiting up to two minutes for the process to complete before finally stopping. In this case, if the shutdown fails, KwikNet will initiate a fatal exit.

In a multitasking system, procedures *kn_godown()* and *kn_exit()* can only be called from an application task executing at lower priority than the KwikNet Task. In a single threaded system, the functions must be called from your App-Task.

4.6 KwikNet IP and UDP Library Services

The KwikNet Libraries provide a full set of network services from which the real-time system designer can choose. Many of the services are optional and, if not used or configured into your KwikNet Libraries, will not even be present in your final KwikNet system.

The following list summarizes the KwikNet IP and UDP service procedures which are accessible to the user. These procedures are all present in the KwikNet IP Library. They are grouped functionally for easy reference.

<i>kn_enter</i>	Launch the KwikNet TCP/IP Stack
<i>kn_exit</i>	Terminate the KwikNet TCP/IP Stack
<i>kn_godown</i>	Initiate a shutdown of the KwikNet TCP/IP Stack
<i>kn_state</i>	Sense the operating state of the KwikNet TCP/IP Stack
<i>kn_panic</i>	Generate a KwikNet fatal error
<i>kn_yield</i>	Yield to the KwikNet Task (single threaded use only)
<i>kn_addserver</i>	Install (add) a server function (single threaded use only)
<i>kn_fmt</i>	Format a text string
<i>kn_dprintf</i>	Format and log a text message
<i>kn_logbuffree</i>	Free a KwikNet log buffer
<i>kn_netstats</i>	Log KwikNet network statistics
<i>kn_inet_addr</i>	Convert a dotted decimal IP address to numeric form
<i>kn_inet_ntoa</i>	Convert a numeric IP address to dotted decimal string form
<i>kn_inet_ifindex</i>	Find the index number for a specific network interface
<i>kn_inet_ifstate</i>	Query and/or modify the state of a network interface
<i>kn_inet_local</i>	Get the IP address of a local network interface
<i>kn_cksum</i>	Compute an IP checksum
<i>kn_udpopen</i>	Open a UDP channel to send/receive UDP datagrams on a network
<i>kn_udpclose</i>	Close a UDP channel
<i>kn_udpbind</i>	Bind a local IP address to a UDP channel
<i>kn_udpsend</i>	Send a UDP datagram on a network
<i>kn_udpfree</i>	Free a received UDP message packet
<i>kn_dns_query</i>	Make a DNS query for a particular domain name
<i>kn_dns_lookup</i>	Query the local DNS name cache for a particular domain name
<i>kn_dns_srvadd</i>	Add a DNS server to the DNS client's server list
<i>kn_dns_srvdel</i>	Delete a DNS server from the DNS client's server list
<i>kn_gethostbyname</i>	Get the IP address of a host with a specific domain name
<i>kn_gethostprep</i>	Prepare a <i>hostent</i> structure for first use
<i>kn_pingsend</i>	Ping a foreign host on a network
<i>kn_pinguserfn</i>	Register a ping callback function to process ping replies

...more

The following BSD-like services are also available in the KwikNet IP Library.

<i>gethostbyname</i>	Get the IP address of a host with a specific domain name
<i>netlong</i> = <i>htonl(hostlong)</i>	Convert <i>long</i> from host to network endian form
<i>netshort</i> = <i>htons(hostshort)</i>	Convert <i>short</i> from host to network endian form
<i>hostlong</i> = <i>ntohl(netlong)</i>	Convert <i>long</i> from network to host endian form
<i>hostshort</i> = <i>ntohs(netshort)</i>	Convert <i>short</i> from network to host endian form

KwikNet Procedure Descriptions

A description of all KwikNet IP and UDP service procedures is provided in this chapter. The descriptions are ordered alphabetically for easy reference. All of the KwikNet procedures are described using the C programming language.

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 filenames, constants and error codes. All KwikNet procedure, structure and constant names can be readily identified according to the nomenclature introduced in Chapter 1.3.

A consistent style has been adopted for the description of the KwikNet procedures presented in Chapters 4.6 and 5.4. The procedure name is presented at the extreme top right and left as in a dictionary. This method of presentation has been chosen to make it easy to find procedures since they are ordered alphabetically.

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

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

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

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

...more

KwikNet Procedure Descriptions (continued)

Used by

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

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

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

Setup

The prototype of the KwikNet procedure is shown.
The KwikNet header file in which the prototype is located is identified.
Include KwikNet header file *KN_LIB.H* or *KN_SOCK.H* for compilation.

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

File *KN_SOCK.H* is the KwikNet include file which you must include if your application uses the TCP/IP sockets API. This file is located in KwikNet installation directory *TCP*. File *KN_SOCK.H* automatically includes file *KN_LIB.H* if it has not already been included.

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. Additional TCP/IP socket error information is also available via KwikNet procedure *kn_errno()*.

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

Note Special notes, suggestions or warnings are offered where necessary. The following paragraph is an example of such a note.

All KwikNet procedures assume that an integer or unsigned integer is a 16 or 32-bit value dependent only upon the basic register width of the target processor.

Example In many cases, a simple example is provided. The examples are kept simple and are intended only to illustrate the correct calling sequence.

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

Purpose **Get the IP Address of a Host with a Specific Domain Name****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
struct hostent *gethostbyname(const char *name);
```

Description *Name* is a pointer to a string which specifies the domain name for which an IP address is required. Domain names are usually recognized as strings of the form "www.kad战略.com". This function will also accept a dotted decimal name of the form "192.168.0.3" and return the equivalent IP address.**Returns** If successful, a pointer to a private KwikNet *hostent* structure is returned. The structure is defined in KwikNet header file *KN_API.H* as follows:

```
struct hostent {
    const char *h_name;           /* Official name of host */
    char **h_aliases;            /* Alias list */
    int h_addrtype;              /* Host address type */
    int h_length;                /* Length of address */
    char **h_addr_list;          /* List of IP address pntrs */
};
```

If *hostp* is the pointer to structure *hostent*, **hostp* is filled as follows:

```
hostp->h_name = name;           Domain name string used in the query
*hostp->h_aliases = NULL;        Aliases are not supported
hostp->h_addrtype = 2;           IP address family type (AF_INET)
                                Length of an IP address
hostp->h_length = sizeof(struct in_addr);
hostp->h_addr_list =             Pointer to a list of IP address pointers
```

Hostp->h_addr_list references an array of pointers to *in_addr* structures containing the IP addresses. Each IP address is presented in net endian form in field *s_addr* of the *in_addr* structure. The pointer array is terminated by a *NULL* pointer. The number of IP addresses in the list will never exceed *KN_DNSMAXADDRS*, the maximum number of IP addresses which your DNS client has been configured to accept from a DNS server.

If the domain name cannot be resolved within the timeout interval specified by your DNS client configuration or because of an error condition, a *NULL* pointer is returned.

Example The example in the description of *kn_gethostbyname()* illustrates the proper interpretation of the results returned in structure *hostent*.**See Also** *kn_gethostbyname()*

Purpose **Convert Between Host and Network Endian Forms****Used by** ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure**Setup** The macro definitions are in file *KN_API.H*.
C dependent, in-line assembly language expansions are in file *KNZZZCC.H*.
*#include "KN_LIB.H"***Convert 32-bit values***netlong* = *htonl(hostlong)*
hostlong = *ntohl(netlong)***Convert 16-bit values***netshort* = *htons(hostshort)*
hostshort = *ntohs(netshort)***Description** *Hostlong* is any 32-bit value in host endian form.
Netlong is any 32-bit value in net endian form.
Hostshort is any 16-bit value in host endian form.
Netshort is any 16-bit value in net endian form.

If the KwikNet Library has been configured for big endian operation, these macros do nothing since the input values require no conversion.

If the KwikNet Library has been configured for little endian operation, these macros may expand to a function call, an in-line function expansion or a series of C statements, depending upon which C compiler is being used.

The goal is always to ensure the fastest possible execution of these frequently encountered macros. When possible, these macros have been implemented using in-line assembly language statements generated by the C compiler. In some cases, the macros generate calls to assembly language functions of a form supported by the C compiler. As a last resort, the macros expand to a series of in-line C statements.

Returns The input value converted to opposite endian form.**Restriction** These macros can introduce side effects. Therefore, the macro parameters must not use expressions which include operators such as *--* or *++* since they always produce side effects. You must also avoid using expressions which include function calls to fetch parameters if the functions can introduce side effects.**Example** See examples in the descriptions of *kn_cksum()*, *kn_dprintf()* and *kn_fmt()*.

Purpose **Install (Add) a Server Function****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_API.H*.
 #include "KN_LIB.H"
 *void kn_addserver(struct knx_svbblock *servercbp,*
 unsigned long period,
 *int (*serverfnp)(void *), void *param);*

Description *Servercbp* is a pointer to a KwikNet server control block which KwikNet can use to control the periodic execution of this server function. Each server function requires its own unique server control block.

You can create a server control block by declaring it as a unique structure variable or by including it as a *knx_svbblock* structure within some other structure variable. The variable must reside outside all functions in the source module. The call to *kn_addserver()* must provide a pointer to the instance of the structure.

Parameter *period* specifies the interval at which the server function is to be executed by the KwikNet Task. The interval, measured in milliseconds, is converted to a non-zero, integral multiple of KwikNet ticks. The server function is therefore executed at the resulting period, measured in equivalent KwikNet ticks.

If parameter *period* is 0, the server function will be executed by the KwikNet Task whenever a KwikNet clock tick or significant event is serviced. Hence, at a minimum, the server function will execute at the KwikNet clock frequency. However, it will also execute if, at the time your App-Task yields to the KwikNet Task, other stack related services are pending.

Parameter *serverfnp* is a pointer to the server function to be executed by the KwikNet Task. The function is called with a single parameter, a copy of parameter *param* presented in the call to *kn_addserver()*.

The server function must return the value 0 in order to remain on the active server list, ready to be executed at its specified period. If the server function returns a non-zero value, the server will be removed from the KwikNet server list.

Returns Nothing

Restriction This function must only be used in a single threaded system. It can be called while executing in either the user or KwikNet domain.

See Also *kn_yield()*

Purpose **Compute an IP Checksum****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
unsigned short kn_cksum(void *p, unsigned int n);
```

Description *P* is a pointer to a 16-bit aligned region of memory containing an array of 16-bit *unsigned short* integers to be checksummed.*N* is the number of 16-bit *unsigned short* integers in the memory array referenced by pointer *p*.**Returns** The complement of the 16-bit, unsigned IP checksum of the *n* unsigned short integers in the memory array referenced by pointer *p*.

Each 16-bit, unsigned short integer is added to the 16-bit checksum using twos complement arithmetic. Any overflow (carry) from the 16-bit checksum is repetitively added to the checksum until no further overflow occurs.

The algorithms used by KwikNet to implement this procedure are both processor and compiler dependent. The procedure has been coded for fastest possible execution. If the C compiler supports the use of assembly language within C, the procedure is coded in C using assembly language statements of the form supported by the C compiler. Otherwise, the function is coded reasonably efficiently using only C language statements.

Note The checksum algorithm is impervious to the processor's endianness. Hence the 16-bit IP checksum can be stored into and read from the IP packet header without conversion between net and host endianness as illustrated in the example.**Example**

```
#include "kn_lib.h"

unsigned short cksum; /* Computed checksum */
unsigned short pktsum; /* Packet checksum */
struct knx_ip *p; /* IP header pointer */

pktsum = p->xip_cksum; /* Save IP header checksum */
p->xip_cksum = 0; /* Checksum = 0 in packet */

/* Compute IP header checksum */
cksum = ~kn_cksum(p, sizeof(*p) >> 1);

if (cksum != pktsum) {
    kn_dprintf(0, "Received packet has checksum error.\n");
    kn_dprintf(0, "Received %4X; expected %4X.\n",
               ntohs(pktsum), ntohs(cksum));
}
```

Purpose **Query the Local DNS Name Cache for a Particular Domain Name**

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_dns_lookup(const char *namep, struct in_addr *inaddrp);
```

Description *Namep* pointer is a pointer to a domain name string. The domain name must consist of a sequence of dot separated labels of the form "www.kadak.com". The string must be terminated by a '\0' character.

Inaddrp is a pointer to storage for the IP address which corresponds to the domain name given by *namep*. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Returns If the domain name matches an entry in the local DNS name cache, a value of 0 is returned and the corresponding IP address will be stored, in net endian form, at *ipaddr->s_addr*.

If the domain name does not match any of the entries in the local DNS name cache, a value of -1 is returned and the IP address at *ipaddr->s_addr* will be undefined.

See Also *kn_dns_query()*

...more

...continued

Example Also see the example in the description of *kn_dns_query()*.

```
#include "kn_lib.h"

/* Application task which polls after domain name query */
void myPolltask(void)
{
    char          *namep; /* Domain name pointer */
    struct in_addr ipaddr; /* IP address of interest */
    int           status;
    int           delay;

    namep = "www.kadak.com";
    status = kn_dns_query(namep, &ipaddr, NULL, NULL);

    delay = 0;
    while ((status > 0) && (delay++ < 60))
    {
        /* Delay 1 sec */
        cjtkwaitm(cjtmconvert(1000L));

        if (kn_dns_lookup(namep, &ipaddr) == 0)
            break;
    }

    if (status == 0)
        kn_dprintf(0, "Got an instant response.\n");
    else if (status < 0)
        kn_dprintf(0, "Cannot initiate a DNS query.\n");
    else if (delay < 60)
        kn_dprintf(0, "Got a delayed DNS response.\n");
    else kn_dprintf(0, "Did not get a DNS response.\n");
}
```


Purpose **Make a DNS Query for a Particular Domain Name**

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_dns_query(const char *namep, struct in_addr *inaddrp,
                void (*userfn)(struct in_addr *answer,
                              int status, void *userp),
                void *userp);
```

Description *Namep* pointer is a pointer to a domain name string. The domain name must consist of a sequence of dot separated labels of the form "www.kadak.com". The string must be terminated by a '\0' character.

Inaddrp is a pointer to storage for the IP address which corresponds to the domain name given by *namep*. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Userfn is a pointer to your DNS callback function which will be called by KwikNet when the DNS query response is received. This function must be coded as described in Chapter 4.3. Note that the third parameter which it receives, *userp*, is a copy of the pointer variable *userp* presented as a parameter in this *kn_dns_query()* procedure call. If you do not wish to provide such a function, set this parameter to *NULL*.

Userp is any pointer variable which your *userfn()* function might require. *NULL* is an acceptable value.

Returns If successful, a value of 0 is returned and the IP address will be stored, in net endian form, at *ipaddr->s_addr*.

If a DNS query is initiated to determine the IP address, a value of 1 is returned and the IP address at *ipaddr->s_addr* will be undefined. Your DNS callback function will be called when the name becomes available. If you did not provide a callback function, you can poll using *kn_dns_lookup()* until the IP address becomes available.

If the call fails, a value of -1 is returned and the IP address at *ipaddr->s_addr* will be undefined.

See Also *kn_dns_lookup()*

...more

...continued

Example Also see the example in the description of `kn_dns_lookup()`.

```
#include "kn_lib.h"

CJ_ID   dnstaskid;           /* DNS Task id           */
struct in_addr ipaddr;       /* IP address of interest */
volatile int dnsresult;       /* Result passed to task  */

/* Application DNS callback function */

void myDNSfn(struct in_addr *answer, int status,
              void *userp)
{
    if ((long)userp != 0xDEADDEAFL)
        return;             /* Not my DNS query      */

    if (status == 0)
        ipaddr.s_addr = answer->s_addr;
    else ipaddr.s_addr = 0;
    dnsresult = 0;           /* Timeout or got reply   */
    cjtkwake(dnstaskid);     /* Let DNS Task resume    */
}

/* Application task which issues the domain name query */

void myDNStask(void)
{
    int      status;

    dnstaskid = cjtkid();    /* Provide my task id     */

    dnsresult = -1;
    status = kn_dns_query("www.kadak.com", &ipaddr,
                          myDNSfn, (void *)0xDEADDEAFL);

    if (status == 0)
        kn_dprintf(0, "Got an instant response.\n");

    else if (status > 0)
    {
        if (dnsresult == -1) /* Reply may be here already */
            cjtkwait();      /* Wait for reply           */
        if (ipaddr.s_addr != 0)
            kn_dprintf(0, "Got a reply to a DNS query.\n");
        else
            kn_dprintf(0, "Bad datagram or no reply.\n");
    }

    else kn_dprintf(0, "Cannot initiate DNS query.\n");
}
```

Purpose **Add a DNS Server to the DNS Client's Server List**

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

Setup Prototype is in file *KN_API.H*.
 #include "KN_LIB.H"
 *int kn_dns_srvadd(struct in_addr *inaddrp, unsigned long id);*

Description *Inaddrp* is a pointer to a structure containing the DNS server IP address in net endian form. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Id is an arbitrary value provided by the caller to uniquely identify the particular DNS server being added to the DNS server list. This identifier will be required to remove the DNS server from the list.

Returns If successful, the value *0* is returned.

If the DNS server list is full or if the memory needed to add another DNS server is not available, the error status *-1* is returned.

See Also *kn_dns_srvdel()*

Purpose **Remove (Delete) a DNS Server from the DNS Client's Server List**

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_dns_srvdel(struct in_addr *inaddrp, unsigned long id);
```

Description *Inaddrp* is a pointer to a structure containing the DNS server IP address in net endian form. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Id is the unique identifier assigned to the DNS server of interest when that server was added to the DNS server list.

Returns If successful, the value 0 is returned.

If a DNS server with the specified IP address and matching identifier is not present in the DNS client's server list, the error status -1 is returned.

See Also *kn_dns_srvadd()*

Purpose **Format and Log a Text Message****Used by** ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_dprintf(int attrib, const char *fmt, ...);
```

Description *Attrib* is a parameter which defines the message attributes. Applications should use the value of 0 for *attrib*. Valid attributes are described in Chapter 1.6

Fmt is a pointer to a format specification string similar to that expected by the C library procedure *printf()*. Allowable format specifications are summarized in the description of procedure *kn_fmt()*.

The format string is followed by zero or more parameters of types specified by the format string.

Returns Nothing**Example**

```
#include "kn_lib.h"

char      *bufp;                /* Input buffer pointer      */
struct in_addr ipaddr;          /* IP address (numeric)      */
char      ipstring[40];         /* IP addr (dotted decimal) */

bufp = "192. 168. 5";          /* An unusual input string  */

if (kn_inet_addr(bufp, &ipaddr) != 1)
    kn_dprintf(0, "Conversion of '%s' to 0xC0A80500" \
               "(192.168.5.0) failed.\n", bufp);

else if (kn_inet_ntoa(&ipaddr, ipstring) !=
        strlen("192.168.5.0"))
    kn_dprintf(0, "Conversion to '192.168.5.0' failed.\n");

else {
    kn_dprintf(0, "Converted '%s' to 0x%08lx to '%s'.\n",
               bufp, ntohl(ipaddr.s_addr), ipstring);

    /* The previous message should read:
    /* "Converted '192. 168. 5' to 0xc0a80500
    /*                               to '192.168.5.0'."
    */
}
```

See Also *kn_fmt()*, *kn_netstats()*

kn_enter
kn_exit

kn_enter
kn_exit

Purpose **Launch or Terminate the KwikNet TCP/IP Stack**

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_enter(void);
void kn_exit(void);
```

Description Procedure *kn_enter()* must be called to launch (start) the KwikNet TCP/IP Stack.

 Procedure *kn_exit()* must be called to terminate (stop) the KwikNet TCP/IP Stack.

 In a multitasking system, procedure *kn_enter()* must be called before any task can use KwikNet services. Procedure *kn_exit()* must not be called until all tasks, including KwikNet client and server tasks, have stopped using KwikNet services.

 In a single threaded system, procedure *kn_enter()* must be called by the App-Task. Procedure *kn_exit()* must not be called until the App-Task and all active clients and servers, including KwikNet clients and servers, have stopped using KwikNet services.

Returns Nothing

AMX Note Procedure *kn_enter()* can be treated as if it is an AMX Restart Procedure. Alternatively, it can be called from a Restart Procedure or from an application task.

 Procedure *kn_exit()* can be treated as if it is an AMX Exit Procedure. Alternatively, it can be called from an Exit Procedure or from an application task which is executing on behalf of an AMX Exit Procedure.

See Also *kn_godown()*, *kn_panic()*

Purpose **Format a Text String****Used by** ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_fmt(char *bufp, const char *fmt, ...);
```

Description *Bufp* is a pointer to storage for the formatted string.

Fmtp is a pointer to a format specification string similar to that expected by the C library procedure *printf()*. Allowable format specifications are summarized on the next page.

The format string is followed by zero or more parameters of types specified by the format string.

Returns The formatted string is stored at **bufp* and the length of that string is returned. The length is a positive value. The string is terminated with a '\0' character.**Example** Other examples are provided in the descriptions of *kn_cksum()* and *kn_dprintf()*.

```
#include "kn_lib.h"

struct in_addr ipaddr;          /* IP address (numeric)      */
char          buf[80];          /* String buffer             */

/* IP address = 192.168.5.21 */
ipaddr.s_addr = htonl(0xC0A80516);

if (kn_fmt(buf, "IP address 0x%08lX is '%03a'.\n",
           ntohl(ipaddr.s_addr), ipaddr.s_addr) <= 0)
    kn_dprintf(0, "Cannot convert IP address to string.\n");
else {
    kn_dprintf(0, "%s", buf);

    /* The previous message should read:
    /* "IP address 0xC0A80516 is '192.168.005.016'."
    */
}
```

See Also *kn_dprintf()*

...more

Formats ...continued

Allowable format specification strings must be of the form "...%-0##z?..."

where:

% = format specification leadin character; use %% for % in output string

- = left justify in field

0 = zero fill

= field width as decimal value

z = l if variable is *long* (? is one of d, u, x, X)

z = h if variable is *short int* (? is one of d, u, x, X)

z = h for an IP address in hex format (? is one of a, A)

? = one of following field variable specifications

c = character value

d = decimal integer value

u = unsigned decimal integer value

x = unsigned hex integer value (use a .. f)

X = unsigned hex integer value (use A .. F)

f = fill with next char to field width

p = pointer value

s = string value

a = internet IP address as "1.26.3.127"

"%03a" yields "001.026.003.127"

ha = internet IP address as "01.1a.03.ff"

"%hA" yields "01.1A.03.FF"

"%0##a" yields "001.002.003.004"

for any non-zero value "0##".

"%0##ha" yields "0a.0b.0c.0d"

for any non-zero value "0##".

S = copy characters from format string into the output field

until ' ' is encountered

"%20Sabc`" right justifies "abc" in a 20 character field.

"%-20Sabc`" left justifies "abc" in a 20 character field.

Numeric fields that overflow the field width are formatted as xxxxxx to the full width of the field.

Purpose **Get the IP Address of a Host with a Specific Domain Name**

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

Setup Prototype is in file *KN_API.H*.
`#include "KN_LIB.H"`
`int kn_gethostbyname(const char *name,`
`struct hostent *hostp, int timeout);`

Description *Name* is a pointer to a string which specifies the domain name for which an IP address is required. Domain names are usually recognized as strings of the form "www.kadak.com". This function will also accept a dotted decimal name of the form "192.168.0.3" and return the equivalent IP address.

Hostp is a pointer to a *hostent* structure initialized by a previous call to function *kn_gethostprep()*. Structure *hostent* is defined in KwikNet header file *KN_API.H* as follows:

```
struct hostent {
    const char *h_name;           /* Official name of host */
    char **h_aliases;            /* Alias list */
    int h_addrtype;              /* Host address type */
    int h_length;                /* Length of address */
    char **h_addr_list;          /* List of IP address pntrs */
};
```

Timeout is the maximum interval, measured in seconds, which the caller is willing to wait for the DNS client to resolve the domain name.

Returns If successful, a value of 0 is returned and the *hostent* structure at **hostp* is filled with:

<code>hostp->h_name = name;</code>	Domain name string used in the query
<code>*hostp->h_aliases = NULL;</code>	Aliases are not supported
<code>hostp->h_addrtype = 2;</code>	IP address family type (<i>AF_INET</i>)
	Length of an IP address
<code>hostp->h_length = sizeof(struct in_addr);</code>	
<code>hostp->h_addr_list =</code>	Pointer to a list of IP address pointers

Hostp->h_addr_list references an array of pointers to *in_addr* structures containing the IP addresses. Each IP address is presented in net endian form in field *s_addr* of the *in_addr* structure. The pointer array is terminated by a *NULL* pointer. The number of IP addresses in the list will never exceed *KN_DNSMAXADDRES*, the maximum number of IP addresses which your DNS client has been configured to accept from a DNS server.

...more

Returns ...continued

If the domain name cannot be resolved because of an error condition other than a timeout, the error status *-1* is returned.

If the domain name cannot be resolved within *timeout* seconds, the warning value *1* is returned.

Example

```
#include "kn_lib.h"
#define NADDR_MAX 4          /* Max number of IP addresses*/

int getkadakIP(struct in_addr *ipaddr)
{
    struct in_addr *ipp;      /* IP address pointer          */
    struct hostent hent;      /* Query structure      */
                                /* Result buffer        */
    char    resbuf[KN_DNS_HOSTBUF_SIZE(NADDR_MAX)];
    int     i, error;

    /* Prepare for query                                          */
    kn_gethostprep(&hent, resbuf, NADDR_MAX);

    /* Wait up to two minutes (120 seconds) for results          */
    error = kn_gethostbyname("www.kadak.com", &hent, 120);

    if (error)
        return (error);

    /* Pass first available IP address to caller                  */
    ipp = (struct in_addr *) (hent.h_addr_list[0]);
    *ipaddr = *ipp;

    /* Show results and all IP addresses                          */
    kn_dprintf(0, "Name: %s (type %d, length %d)\n",
               hent.h_name, hent.h_addrtype, hent.h_length);

    for (i = 0; i < NADDR_MAX; i++) {
        ipp = (struct in_addr *) (hent.h_addr_list[i]);
        if (!ipp)
            break;

        kn_dprintf(0, "IP address %d is '%03a'.\n",
                   i + 1, ipp->s_addr);
    }

    return (0);
}
```

See Also `gethostbyname()`, `kn_gethostprep()`

Purpose Prepare a Hostent Structure Prior to First Use

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_gethostprep(struct hostent *hostp,
                    void *buf, int naddrs);
```

Description *Hostp* is a pointer to a *hostent* structure which must be initialized by this function before it can be used in calls to *kn_gethostbyname()*.

Buf is a pointer to storage for the results of subsequent domain name queries made with calls to *kn_gethostbyname()*. The storage buffer must be large enough to hold the number of IP addresses you are willing to accept in your query. If you will accept only *n* IP addresses, then you must provide at least *NB* bytes of storage where *NB* is computed using the macro *KN_DNS_HOSTBUF_SIZE(n)* from KwikNet header file *KN_API.H*.

Naddrs is the maximum number of IP addresses that you will accept in response to a subsequent domain name query. If your storage buffer of *NB* bytes will hold *n* IP addresses, *naddrs* must be $\leq n$. The maximum number of IP addresses which can be returned from any domain name query is *KN_DNSMAXADDRES*, the limit specified by your definition of the DNS client in your KwikNet IP Library configuration.

Returns Nothing

See Also *kn_gethostbyname()*

Purpose	Initiate a Shutdown of the KwikNet TCP/IP Stack
Used by	■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure
Setup	Prototype is in file <i>KN_API.H</i> . <pre>#include "KN_LIB.H" int kn_godown(int timeout);</pre>
Description	<p><i>Timeout</i> is the maximum interval, measured in seconds, which the caller is willing to wait for the KwikNet shutdown to complete. If <i>timeout</i> is 0, the caller will wait forever or until an error condition is detected.</p> <p>In a multitasking system, procedure <i>kn_godown()</i> must not be called until all tasks, including KwikNet client and server tasks, have stopped using KwikNet services.</p> <p>In a single threaded system, procedure <i>kn_godown()</i> must be called by the App-Task. Procedure <i>kn_godown()</i> must not be called until the App-Task and all active clients and servers, including KwikNet clients and servers, have stopped using KwikNet services.</p>
Returns	If successful, a value of 0 is returned. On failure, the error status 1 is returned. If a shutdown has already been initiated, a value of -1 is returned.
Restriction	Procedure <i>kn_godown()</i> must be called to shut down the KwikNet TCP/IP Stack prior to calling <i>kn_exit()</i> to terminate operation of the stack. If this restriction is not met, KwikNet will automatically call this function with a 2 minute (120 second) timeout interval when <i>kn_exit()</i> is called.
AMX Note	Procedure <i>kn_godown()</i> can be called from an Exit Procedure or from an application task which is executing on behalf of an AMX Exit Procedure.
See Also	<i>kn_enter()</i> , <i>kn_exit()</i> , <i>kn_panic()</i>

Purpose Convert a Dotted Decimal IP Address to Numeric Form

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_inet_addr(const char *sp, struct in_addr *inaddrp);
```

Description *Sp* is a pointer to a string containing an IP address in dotted decimal form. The string does not have to be terminated by '\0'. Leading whitespace before the numbers in the IP address will be ignored. The separating dot must be the first character after each number.

Hence, "254. 76.2. 1abc" is an acceptable input string.

Inaddrp is a pointer to a structure into which the numeric IP address in net endian form will be stored. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Returns 0 if the string contained a full IP address such as "75.4. 34.12abc".

1, 2 or 3 if the string contained a partial IP address such as "75.4.34", "75.4" or "75" respectively. In each case, the missing fields are assumed to be 0. For example "75.4" is assumed to be "75.4.0.0".

The resulting IP address in numeric, net endian form is stored in the IP address structure at *inaddrp->s_addr*.

Limited error checking is performed. Parsing stops at the first character which is not acceptable within an IP address or as soon as four dot separated values are found.

A value greater than 3 is returned if no numeric values are present. The value -1 is returned if any of the decimal values encountered are outside the range 0 to 255.

Note Unlike its BSD counterpart *inet_addr()*, this KwikNet procedure uses structure *in_addr* to hold the IP address so that future changes in IP address definition can be accommodated without affecting your application.

Example See example in the description of *kn_dprintf()*.

See Also *kn_inet_ntoa()*

Purpose Find the Index Number for a Specific Network Interface

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

Setup Prototype is in file *KN_API.H*.
`#include "KN_LIB.H"`
`int kn_inet_ifindex(const char *nettag);`

Description *Nettag* is a pointer to a string which specifies the 4-character network tag of the network interface of interest.

Returns If successful, the network interface index number *N* is returned. *N* is an integer in the range 0 to *numnet*-1 where *numnet* is the total number network interfaces.

If a network with the specified network tag does not exist, the value -1 is returned.

Example

```
#include "kn_lib.h"

int get_net_pnet(void)
{
    int    n;

    n = kn_inet_ifindex("PNET");
    if (n < 0)
        kn_panic("get_net_pnet: Network PNET is lost");

    return (n);
}
```

See Also `kn_inet_ifstate()`

Purpose **Query and/or Modify the State of a Network Interface****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_inet_ifstate(int n, int newstate, int *prevstate);
```

Description *N* is a network index number used to identify the network interface of interest. *N* must be in the range 0 to *numnet-1* where *numnet* is the total number network interfaces.

If you wish to change the state of the network, parameter *newstate* identifies the required state. *Newstate* must be one of the following values:

<i>KN_NIFS_DOWN</i>	Shut down the network.
<i>KN_NIFS_UP</i>	Start up the network.
0	Leave the state of the network unchanged.

Parameter *prevstate* is a pointer to storage for the state of the network at the time this procedure was called. If *prevstate* is *NULL*, the parameter will be ignored by KwikNet.

Returns If successful, a value of 0 is returned.

If the requested network state change has been initiated, but is not yet complete, the following status code is returned:

<i>KN_WRPENDING</i>	The requested operation is now in progress.
---------------------	---------------------------------------------

On failure, one of the following error status codes is returned:

<i>KN_ERIFACE</i>	The network index does not refer to a valid network.
<i>KN_ERPARAM</i>	The state specified by parameter <i>newstate</i> is invalid or the requested state change cannot be granted.

If parameter *prevstate* is not *NULL*, then upon return **prevstate* will be set to one of the following values:

<i>KN_NIFS_DOWN</i>	The network was idle.
<i>KN_NIFS_TRANSIT</i>	The network was in the process of going up or down.
<i>KN_NIFS_UP</i>	The network was operational.

Restrictions You cannot change the state of an Ethernet network.
 You cannot start a network that is in the process of shutting down.
 You cannot stop a network that is in the process of starting up.
 In these cases, the error status *KN_ERPARAM* will be returned to the caller.

See Also *kn_inet_ifindex()*, *kn_netstats()*, *kn_state()*

Purpose **Get the IP Address of a Local Network Interface****Used by** ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_inet_local(int n, struct in_addr *inaddrp);
```

Description *N* is a network index number used to identify the network interface of interest. Set *n* to 0 to indicate the first network interface defined in the KwikNet Network Configuration Module. Increment *n* by one to reference the next sequential network interface. *N* must be in the range 0 to *numnet-1* where *numnet* is the total number network interfaces.*Inaddrp* is a pointer to a structure into which the IP address of the local network interface will be stored in net endian form. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Returns The value *n+1* is returned if the network interface exists. The network IP address in net endian form is stored in the IP address structure at *inaddrp->s_addr*.The value -1 is returned if there is no network interface corresponding to network index *n*.**Example**

```
#include "kn_lib.h"

/* Return the interface number (1 to numnets) of the      */
/* first net for which a valid IP address can be found.      */
/* The IP address is stored at inaddrp->s_addr.              */
/* Return -1 if no nets have a valid IP address.            */
int get_local_IP(struct in_addr *inaddrp)
{
    int      n = 0;

    while ( (n = kn_inet_local(n, inaddrp)) != -1) {
        if (inaddrp->s_addr != INADDR_ANY)
            break;
    }
    return (n);
}
```

See Also *kn_inet_ifindex()*

Purpose	Convert a Numeric IP Address to Dotted Decimal String Form
Used by	■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure
Setup	Prototype is in file <i>KN_API.H</i> . <pre>#include "KN_LIB.H" int kn_inet_ntoa(struct in_addr *inaddrp, char *sp);</pre>
Description	<p><i>Inaddrp</i> is a pointer to a structure containing an IP address in net endian form. The BSD structure <i>in_addr</i> is defined in file <i>KN_API.H</i> as follows:</p> <pre>struct in_addr { unsigned long s_addr; /* IP address (net endian) */ };</pre> <p><i>Sp</i> is a pointer to storage for the string showing the IP address in dotted decimal form.</p>
Returns	<p>The formatted string is stored at <i>*sp</i> and the length of that string is returned. The length is a positive value. The string is terminated with a '\0' character.</p> <p>The IP address <i>0x7F000017</i> will produce the string "127.0.0.23".</p>
Note	Unlike its BSD counterpart <i>inet_ntoa()</i> , this KwikNet procedure is reentrant. It also uses structure <i>in_addr</i> to hold the IP address so that future changes in IP address definition can be accommodated without affecting your application.
Example	See example in the description of <i>kn_dprintf()</i> .
See Also	<i>kn_inet_addr()</i>

Purpose **Free a KwikNet Log Buffer**

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

Setup Prototype is in file *KN_API.H*.
 #include "KN_LIB.H"
 *void kn_logbuffree(char *bufp);*

Description *Bufp* is a pointer to the KwikNet log buffer which is to be freed. This is a buffer allocated by KwikNet from its private pool of log buffers in response to a request to log a message.

 Your application log function must call *kn_logbuffree()* to free each log buffer that it accepts.

Returns Nothing

Purpose **Log KwikNet Network Statistics**

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

Setup Prototype is in file *KN_API.H*.
 #include "KN_LIB.H"
 void kn_netstats(unsigned long statmask);

Description *Statmask* is a bit mask identifying the subset of network statistics maintained by KwikNet which you wish to log. *Statmask* can be created by ORing one or more of the bit mask constants *KN_NS_XXXX* to select the particular statistics to be logged. The constants *KN_NS_XXXX* are defined in file *KN_API.H*. To log all statistics, set *statmask* to *KN_NS_ALL*.

KwikNet will log network statistics which it has been gathering. The information is formatted into log buffers and presented a line at a time to your data logging function (see Chapter 1.6).

Returns Nothing

See Also *kn_dprintf()*, *kn_inet_ifstate()*

kn_panic

kn_panic

Purpose **Generate a KwikNet Fatal Error**

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_panic(const char *msgp);
```

Description *msgp* is a pointer to a message string to be logged by KwikNet before it initiates a shutdown of the underlying RT/OS.

Returns Never

See Also *kn_exit()*

Purpose Ping a Foreign Host on a Network**Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_pingsend(struct in_addr *inaddrp,
                const char *datap, int ndata,
                int sequence, int timeout);
```

Description *Inaddrp* is a pointer to a structure containing the IP address, in net endian form, of the specific foreign host which you wish to ping. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Datap is a pointer to a data block which you wish to send within the ping message. If you do not wish to define the ping data, set this parameter to *NULL*.*Ndata* is the total number of bytes which will be sent in the data region of the ping message. If you provide a non *NULL* data pointer, then the ping message will include *ndata* bytes of ping from the data block referenced by *datap*.If your data pointer *datap* is *NULL*, KwikNet will try to use a private data string identifying KwikNet as the source of the ping. If *ndata* is greater than the length of the string, the data string will be included in the ping message. Otherwise, it will not. If necessary, the data block will be padded to *ndata* bytes with an incrementing binary pattern.*Sequence* is a sequence number which is sent to the foreign host in the ping message. Although the sequence number is echoed in the ping response, it is ignored by KwikNet.*Timeout* is the number of seconds which you are prepared to allow for the foreign host to respond to the ping message. This parameter will be ignored by KwikNet if you have not called procedure *kn_pinguserfn()* to provide a ping callback function.**Returns** If successful, a value of 0 is returned.
On failure, the error status -1 is returned.**Example** See example in the description of *kn_pinguserfn()*.**See Also** *kn_pinguserfn()*

Purpose	Register a Ping Callback Function to Process Ping Replies
Used by	<input checked="" type="checkbox"/> Task <input type="checkbox"/> ISP <input type="checkbox"/> Timer Procedure <input type="checkbox"/> Restart Procedure <input checked="" type="checkbox"/> Exit Procedure
Setup	Prototype is in file <i>KN_API.H</i> . <pre>#include "KN_LIB.H" void kn_pinguserfn(void (*userfn)(char *, int));</pre>
Description	<p><i>Userfn</i> is a pointer to your ping callback function which will be called by KwikNet whenever a ping reply is received. This function must be coded as described in Chapter 4.4.</p> <p>To cancel the use of your callback function, call <i>kn_pinguserfn()</i> with parameter <i>userfn</i> set to <i>NULL</i>.</p>
Returns	Nothing
Restriction	This procedure will not be present in the KwikNet IP Library unless your KwikNet Library Parameter File indicates that the application intends to handle ping replies. In this case, procedure <i>kn_pinguserfn()</i> will be unresolved when you link your application. To enable this feature, use the KwikNet Configuration Builder to edit your KwikNet Library Parameter File and check the required option box on the IP property page. Then rebuild your KwikNet Libraries.
Note	Once installed, your ping callback will be called to process every ping reply received by KwikNet, including spurious replies, if any, received from interconnected networks. It will continue to be called until you cancel it.
See Also	<i>kn_pingsend()</i> ...more

...continued

Example

```
#include "kn_lib.h"

CJ_ID      pingtaskid;      /* Ping Task id      */
char       *infop;          /* Ping info        */
volatile int pingresult;     /* Result passed to task */

/* Application ping callback function */

void mypingfn(char *datap, int length)
{
    if (length == strlen(infop) + 1)
    {
        if (strcmp(datap, infop) == 0)
            pingresult = length;
        else pingresult--;
    }
    else pingresult = 0;      /* Timeout or bad reply */
    cjtkwake(pingtaskid);    /* Let Ping Task resume */
    kn_pinguserfn(NULL);     /* Cancel my ping function */
}

/* Application task which issues the ping request */

void myPINGtask(void)
{
    struct in_addr ipaddr;    /* IP address */

    pingtaskid = cjtkid();    /* Provide my task id */

    kn_inet_addr("192.168.5.12", &ipaddr);
    infop = "This is my test ping data.";

    kn_pinguserfn(mypingfn); /* Install my ping function */
    pingresult = -1;

    /* Ping host 192.168.5.12 with my message. */
    /* The message must include the '\0' string terminator. */
    /* Use sequence number 1. */
    /* Timeout after 30 seconds. */

    if (kn_pingsend(&ipaddr, infop,
                    strlen(infop) + 1, 1, 30) == 0)
    {
        if (pingresult == -1) /* Reply may be here already */
            cjtkwait();      /* Wait for reply */
        if (pingresult > 0)
            kn_dprintf(0, "Got expected ping reply.\n");
        else if (pingresult < 0)
            kn_dprintf(0, "Got an invalid reply.\n");
        else
            kn_dprintf(0, "Bad datagram or no reply.\n");
    }
}
```

Purpose Sense the Operating State of the KwikNet TCP/IP Stack

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
unsigned int kn_state(unsigned int wstate, unsigned long mswait);
```

Description This function can be used to determine the current operating state of KwikNet or to wait until KwikNet reaches a particular state.

Parameter *wstate* identifies the particular KwikNet state or states of interest. If *wstate* is 0, the current KwikNet state will be returned to the caller without delay. In this case, parameter *mswait* will be ignored.

If parameter *wstate* is not 0, the caller will be blocked until the KwikNet operating state matches one or more of the specified state masks.

wstate must be the logical OR of one or more of following masks:

<i>KN_TS_IDLE</i>	KwikNet is idle (not in use at all).
<i>KN_TS_START</i>	KwikNet is starting up.
<i>KN_TS_RUN</i>	KwikNet is fully operational.
<i>KN_TS_GODOWN</i>	KwikNet shutdown is in progress.
<i>KN_TS_EXIT</i>	KwikNet exit is in progress.
<i>KN_TS_STOPPED</i>	KwikNet has stopped. This state is fleeting. KwikNet will immediately enter the idle state.

Mswait is the number of milliseconds which the caller is prepared to wait for KwikNet to reach any of the specified states. A value of 0 indicates that the caller will wait forever.

Returns If *wstate* is 0, the current KwikNet state is returned.
 The value will be the logical OR of one or more of the above state masks.

If *wstate* is not 0, the KwikNet state at the time of the match is returned.
 If a timeout occurs or the caller cannot be blocked, the value 0 is returned.

Note States *KN_TS_IDLE*, *KN_TS_START*, *KN_TS_RUN* and *KN_TS_STOPPED* are mutually exclusive. State masks *KN_TS_GODOWN* and *KN_TS_EXIT* can be set in conjunction with other state masks.

To detect that KwikNet has fully stopped and is no longer in use, you must wait with state mask *wstate* = *KN_TS_IDLE*.

Restrictions This function will not alter the state of KwikNet. This procedure must not be called by any function executing in the context of the KwikNet Task.

See Also *kn_inet_ifstate()*

Purpose **Bind a Local IP Address to a UDP Channel**

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

Setup Prototype is in file *KN_API.H*.
 #include "KN_LIB.H"
 int kn_udpbind(unsigned long udphandle,
 *struct in_addr *inaddrp);*

Description *Udphandle* is a UDP handle acquired with a previous call to *kn_udpopen()*.

Inaddrp is a pointer to a structure containing the IP address, in net endian form, of the local network interface which you wish to bind to the UDP channel. All UDP datagrams sent on the UDP channel will be transmitted on that network interface.

Use IP address 0.0.0.0 to remove the binding, thereby restoring the UDP channel to the state which existed when the channel was opened.

The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Returns If successful, a value of 0 is returned.

If a local network with the specified IP address does not exist, the error status -1 is returned.

See Also *kn_udpopen()*

Purpose **Close a UDP Channel**

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_udpclose(unsigned long udphandle);
```

Description *Udphandle* is a UDP handle acquired with a previous call to *kn_udpopen()*. This handle must not be used after the UDP channel which it represents has been closed.

Returns If successful, a value of 0 is returned.
 On failure, the error status -1 is returned.

Example See example in the description of *kn_udpopen()*.

See Also *kn_udpopen()*

Purpose **Free a Received UDP Message Packet****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_udpfree(struct knx_udpmsg *msgp);
```

Description *msgp* is a pointer to the UDP message descriptor given to your application's UDP callback function upon receipt of a UDP datagram.**Returns** Nothing**Example** See example in the description of *kn_udpopen()*.**See Also** *kn_udpopen()*

Purpose **Open a UDP Channel to Send/Receive UDP Datagrams on a Network**

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

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
unsigned long kn_udpopen(struct in_addr *inaddrp,
                        int fport, int lport,
                        int (*udprcv)(struct knx_udpmsg *msgp, void *userp),
                        void *userp);
```

Description *Inaddrp* is a pointer to a structure containing the IP address, in net endian form, of the foreign host with whom you wish to communicate using UDP datagrams. Use IP address 0.0.0.0 to identify any host. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Fport is the foreign port number for the host with whom you intend to communicate. If *fport* is 0, then any UDP datagram from the foreign host will be accepted on the UDP channel.

Lport is the local port number to be used for communication.

Udprcv is the name of your application callback function which will be called to process received UDP datagrams. Your function will receive parameter *msgp*, a pointer to a UDP message descriptor. The second parameter, *userp*, is a copy of the pointer variable *userp* received as a parameter in this *kn_udpopen()* procedure call.

Your UDP callback function must be coded as described in Chapter 4.1. It must return 0 if it accepts the UDP message descriptor. In this case, the function must free the UDP message descriptor by calling procedure *kn_udpfree()* when it has finished processing the datagram. The callback function must return -1 if it cannot accept the message descriptor.

Userp is any pointer variable which your *udprcv()* function might require. *NULL* is an acceptable value.

Returns If successful, a non-zero UDP handle is returned.
 On failure, a UDP handle of 0L is returned.

Note This procedure uses structure *in_addr* to hold the IP address so that future changes in IP address definition can be accommodated without affecting your application.

...more

...continued

Example

```
#include "kn_lib.h"

CJ_ID      udptaskid;      /* UDP Task id */
volatile int udpresult;    /* Result passed to task */
struct knx_udpmsg *udpmsgp; /* Saved UDP message pointer */

/* Application UDP callback function */

int myUDPFn(struct knx_udpmsg *msgp, void *userp)
{
    if ((long)userp != 0xFEEDFACEL)
        return (-1);      /* Not my UDP packet */

    udpmsgp = msgp;        /* Save the message pointer */
    udpresult = 0;          /* Got a response */
    cjtkwake(udptaskid);    /* Let UDP Task resume */
    return (0);             /* Accept the message */
}

/* Application task which sends and receives UDP datagrams*/

void myUDPTask(void)
{
    struct in_addr ipaddr;  /* IP address */
    unsigned long handle;   /* UDP channel handle */
    char *dp;               /* Data pointer */

    udptaskid = cjtkid();    /* Provide my task id */
    kn_inet_addr("192.168.5.12", &ipaddr); /* Destination */
    if ( (handle = kn_udpopen(&ipaddr,
                             45, /* Foreign port */
                             43, /* Local port */
                             myUDPFn, (void *)0xFEEDFACEL)) == 0)
        return;             /* Cannot open UDP channel */

    udpresult = -1;
    dp = "KwikNet is asking for a UDP response.\n";
    if (kn_udpsend(handle, &ipaddr, 45, dp, strlen(dp)) == 0)
    {
        /* Wait 60 seconds for response */
        if (udpresult == -1) /* Reply may be here already */
            cjtkwaitm(cjtmconvert(60000L));
        if (udpresult == 0)
        {
            kn_dprintf(0, "Got UDP reply: %s.\n",
                       udpmsgp->xudpm->datap);
            kn_udpfree(udpmsgp); /* Free the message */
        }
    }
    kn_udpclose(handle);     /* Close the UDP channel */
}
```

See Also

`kn_udpclose()`, `kn_udpfree()`

Purpose **Send a UDP Datagram on a Network****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_udpsend(unsigned long udpchannel,
               struct in_addr *inaddrp, int fport,
               char *bufp, int length);
```

Description *Udpchannel* is a UDP handle acquired with a previous call to *kn_udpopen()*.*Inaddrp* is a pointer to a structure containing the IP address, in net endian form, of the foreign host to whom you wish to send a UDP datagram. The BSD structure *in_addr* is defined in file *KN_API.H* as follows:

```
struct in_addr {
    unsigned long s_addr;          /* IP address (net endian) */
};
```

Fport is the foreign port number to which you are sending the UDP datagram.*Bufp* is a pointer to a character buffer containing the data bytes which you wish to send in the UDP datagram.*Length* is the number of bytes in the character buffer referenced by *bufp*.**Returns** If successful, a value of 0 is returned.
On failure, the error status -1 is returned.

The most likely reasons for failure are:

Invalid channel id

Destination is not accessible via any local network interface

No packet buffers are available

Length exceeds the maximum supported UDP/IP data segment size.**Note** This procedure uses structure *in_addr* to hold the IP address so that future changes in IP address definition can be accommodated without affecting your application.**Example** See example in the description of *kn_udpopen()*.**See Also** *kn_udpopen()*

Purpose	Yield to the KwikNet Task
Used by	<input checked="" type="checkbox"/> Task <input type="checkbox"/> ISP <input type="checkbox"/> Timer Procedure <input type="checkbox"/> Restart Procedure <input type="checkbox"/> Exit Procedure
Setup	Prototype is in file <i>KN_API.H</i> . <pre>#include "KN_LIB.H" int kn_yield(void);</pre>
Description	Your single threaded application must regularly yield to the KwikNet Task. Failure to yield at least at the defined KwikNet clock frequency may result in poor performance of the TCP/IP stack.
Returns	Error status is returned. The value 1 is returned if the KwikNet Task executes successfully. The value 0 is returned if the request to execute the KwikNet Task fails. The return value of 0 indicates that the KwikNet Task cannot be executed for some reason. For example, if an application ping callback function calls <i>kn_yield()</i> when it is executed by the KwikNet Task, the call will fail because the KwikNet Task cannot execute recursively.
Restriction	Must only be called in a single threaded system by the App-Task while executing in the user domain.
See Also	<i>kn_addserver()</i>

Purpose **Convert Between Network and Host Endian Forms**

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

Setup The macro definitions are in file *KN_API.H*.
C dependent, in-line assembly language expansions are in file *KNZZZCC.H*.
#include "KN_LIB.H"

Convert 32-bit values

netlong = *htonl(hostlong)*
hostlong = *ntohl(netlong)*

Convert 16-bit values

netshort = *htons(hostshort)*
hostshort = *ntohs(netshort)*

Description *Hostlong* is any 32-bit value in host endian form.
Netlong is any 32-bit value in net endian form.
Hostshort is any 16-bit value in host endian form.
Netshort is any 16-bit value in net endian form.

If the KwikNet Library has been configured for big endian operation, these macros do nothing since the input values require no conversion.

If the KwikNet Library has been configured for little endian operation, these macros may expand to a function call, an in-line function expansion or a series of C statements, depending upon which C compiler is being used.

The goal is always to ensure the fastest possible execution of these frequently encountered macros. When possible, these macros have been implemented using in-line assembly language statements generated by the C compiler. In some cases, the macros generate calls to assembly language functions of a form supported by the C compiler. As a last resort, the macros expand to a series of in-line C statements.

Returns The input value converted to opposite endian form.

Restriction These macros can introduce side effects. Therefore, the macro parameters must not use expressions which include operators such as *--* or *++* since they always produce side effects. You must also avoid using expressions which include function calls to fetch parameters if the functions can introduce side effects.

Example See examples in the descriptions of *kn_cksum()*, *kn_dprintf()* and *kn_fmt()*.

5 KwikNet TCP/IP Sockets

5.1 Introduction to KwikNet Sockets

Sockets is an application programming interface (API) which was developed for UNIX during the early 1980s at University of California, Berkeley. It is primarily used today for TCP programming. Dozens of books and tutorials are available for sockets programming, one of the compelling arguments for the use of sockets.

Programmers new to sockets may observe that the sockets API seems unnecessarily complex. The reasons are historical. Sockets were initially developed to allow interprocess communication via streaming devices in UNIX environments. One process would write to a connection socket and another process would read from a socket at the other end of the connection. Sockets were meant to be a general solution for all types of data transfer. On many UNIX systems you can actually pass a socket to the file *read()* and *write()* calls in place of a file descriptor.

When the Berkeley researchers wanted to extend the endpoints of the socket beyond the host system so that processes on two separate systems could talk, they implemented TCP (as well as other protocols) under the sockets API. The sockets API had to be extended to indicate the type of service to be provided by the socket. The *PF_INET* (as opposed to *PF_UNIX*) parameter in the *socket()* call is a vestige of this legacy.

This use of TCP as a carrier for sockets was TCP's first major popular application outside of the DARPA projects where it was developed. So in a very real sense, TCP owes its widespread popularity today to Berkeley UNIX and sockets.

Over the years, many simpler, cleaner TCP APIs have been proposed. However, by the time TCP became popular on non-UNIX platforms, it was too late. Programmers had become accustomed to the API and sockets had become the de-facto standard for TCP programming.

KwikNet Procedure Descriptions

Each procedure in the KwikNet TCP/IP sockets API is described in Chapter 5.4. The format of each description is identical to that presented in Chapter 4.6 and used to describe procedures in the KwikNet IP Library.

However, since the KwikNet Sample Program provides a complete working illustration of how to use each of the KwikNet TCP/IP procedures, examples are not repeated in the TCP/IP sockets API descriptions presented in Chapter 5.4.

KwikNet Sockets API

The KwikNet TCP sockets API is a subset of that available on UNIX systems. Examples of networking code from other sockets-based systems can be expected to port easily. Furthermore, most of the reference material in books and tutorials will also apply to the KwikNet API.

The KwikNet API has been designed for best use in embedded systems where execution speed, code size and ease of use are of paramount importance. For these reasons, the KwikNet sockets API differs from the Berkeley sockets API. However, a standard sockets API is provided with KwikNet and can be used if code compatibility is of utmost concern.

The API differences are minor. All of the procedure names in the KwikNet TCP/IP Stack are of the form *kn_XXXXX()*. For example, Berkeley procedure *socket()* is KwikNet procedure *kn_socket()*. This naming convention has been adopted by KADAK to avoid any conflicts with symbols in your application or your C run-time libraries.

The standard sockets API uses the procedure *close()* to close a socket. A KwikNet procedure with this name would conflict with procedure *close()* in the standard C library. Consequently, the KwikNet procedure *kn_close()* must be used to close a socket, even when using the KwikNet standard sockets API.

To use the KwikNet standard sockets API, include the following statement in your source modules:

```
#include "KN_SOCK.H"
```

To use the Berkeley standard sockets API, include the following statements in your source modules:

```
#define KN_BSD_SOCKAPI
#include "KN_SOCK.H"
```

Socket Addresses

The endpoint of a socket connection is identified using a socket address. The specification of a socket address is dependent upon the protocol used for communication. The sockets API uses a generalized *sockaddr* structure to specify a socket address. When using the TCP or UDP protocols, it is convenient to cast each *sockaddr* pointer to reference the following Internet specific socket address structure:

```
struct sockaddr_in {
    unsigned short  sin_family;      /* Address family = AF_INET          */
    unsigned short  sin_port;        /* TCP: protocol port                */
    struct in_addr  sin_addr;        /* TCP: IP address                   */
    char            sin_zero[8];     /* TCP: unused (0)                   */
};
```

Member *sin_family* always specifies the Internet Protocol address family (*AF_INET*), stored in host endian form. The protocol port number is stored in member *sin_port* in net endian form. The protocol IP address is stored in member *sin_addr->s_addr* in net endian form. The array *sin_zero[]* is unused and is ignored by KwikNet.

Non-Blocking Sockets

When operations are performed using a socket, the caller requesting the action is usually forced to wait until the operation completes. The socket is said to be in blocking mode.

KwikNet offers an alternative mode of operation in which the socket action is allowed to proceed without blocking the caller. In this case, the socket is said to be in non-blocking mode.

When a socket is created it is placed in blocking mode. Thereafter, your application can call the sockets procedure `kn_setsockopt()` to alter the operating mode of the socket. KwikNet socket option `SO_NONBLOCK` can be used to place the socket in non-blocking mode. The same option can be used to restore the socket to blocking mode. Since `SO_NONBLOCK` is a unique KwikNet option, it is flagged as non-standard in the summary of sockets options presented in Chapter 5.3.

KwikNet Error Codes

KwikNet socket procedures return a positive value or 0 if the call is successful or -1 if the call fails. Procedures which return a socket descriptor return a positive non-zero value if the call is successful.

Standard sockets procedures also return additional error information in the UNIX global variable `errno`. This is the same variable which many standard C libraries use to save error results. However, KwikNet must coexist with these libraries. Furthermore, since KwikNet must operate in multitasking environments, it cannot use `errno` since the results of operations by different tasks would be indeterminate.

For these reasons, global variable `errno` is not altered by KwikNet. Instead, the completion status of each socket operation is recorded within the socket descriptor upon completion of the operation. This error status is then made available to the caller through the `kn_errno()` procedure. Procedure `kn_errno()` must be used to retrieve KwikNet error results, even when using the KwikNet standard sockets API.

Unfortunately, procedures `kn_socket()` and `kn_select()` have no socket descriptor in which to save the error code. Hence, you cannot use `kn_errno()` to retrieve error information after calls to these procedures.

Error codes returned by `kn_errno()` are integer values which are a subset of the standard Berkeley error codes. These error codes are summarized in Appendix B.

5.2 Socket Types

Procedure *kn_socket()* is used to create a socket. The KwikNet TCP/IP Stack only supports sockets for use in the communication domain which uses the protocol family known as the ARPA Internet Protocol, identified as *PF_INET*.

Two socket types are supported: type *SOCK_STREAM* for use with the TCP protocol and type *SOCK_DGRAM* for use with the UDP protocol.

Stream Socket (for TCP)

A socket of type *SOCK_STREAM* provides a sequenced, reliable, full duplex connection between two end points. The local end point is identified using a *kn_bind()* call. A stream socket must be in a connected state before any data can be sent or received using it. A connection is created with a *kn_connect()* call which identifies the remote end of the connection. Once connected, data may be transferred using procedures *kn_send()* and *kn_recv()*. Out-of-band data can be transferred using the *MSG_OOB* option in either of these calls. Finally, when a session has been completed and the local socket is no longer required, procedure *kn_close()* can be used to delete the socket.

The communications protocols used to implement a *SOCK_STREAM* ensure that data is not lost or duplicated. If a piece of data accepted by the protocol stack cannot be successfully delivered within a reasonable length of time, then the connection is considered broken. The *kn_send()* or *kn_recv()* procedures will fail and the socket will report an error code when subsequently interrogated with a *kn_errno()* call. Some protocols include options to keep sockets *warm* by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period of time.

Functions *kn_sendto()*, *kn_sendmsg()*, *kn_recvfrom()* and *kn_recvmsg()* can also be used to send and receive data once a connection has been established. Since the end points of a connected socket are known, there is no need to provide the destination address or storage for the source address when using these functions.

Datagram Socket (for UDP)

A socket of type *SOCK_DGRAM* provides a connectionless, unreliable method for delivering messages of a fixed, usually small, maximum length. The messages are called datagrams. Although the local end point can be identified using a *kn_bind()* call, it is not necessary to bind the socket before using it. A *SOCK_DGRAM* socket allows a datagram to be sent to a correspondent named in the *kn_sendto()* call. Datagrams are received using the *kn_recvfrom()* procedure which identifies the address from which the data is received. Finally, when a session has been completed and the local socket is no longer required, procedure *kn_close()* can be used to delete the socket.

Although a socket of type *SOCK_DGRAM* is connectionless, the *kn_connect()* procedure can still be used to identify the specific peer with whom a conversation is to be held. The connection defines the address to which datagrams are to be sent and the only address from which datagrams are to be received. Furthermore, *kn_connect()* can be called at any time to change the connection address. You can also disconnect by calling *kn_connect()* with a null (all zeroes) address.

Functions *kn_sendmsg()* and *kn_recvmsg()* can also be used to send and receive data as long as your message structure provides the destination address or storage for the source address. If you have provided a destination address with a *kn_connect()* call, you can use function *kn_send()* to send data to that destination. And you can always use function *kn_recv()* to receive data, as long as you do not need to identify the source.

The maximum size of a UDP datagram that KwikNet can send or receive, excluding the UDP and IP headers, is 1472 bytes. When sending, data is not buffered in the socket. All of the data presented to the socket for transmission is sent in one UDP datagram. KwikNet will buffer received UDP datagrams but restricts the total received data to the specified maximum. When receiving, your application must accept the maximum sized datagram or risk loss of data.

Using UDP Sockets

Special care must be taken when using a socket of type *SOCK_DGRAM* with a connectionless protocol such as UDP. When a UDP socket is created, it is considered bound to local IP address 0 and port 0. The newly created socket is idle, unable to receive data from any foreign host.

The UDP socket remains idle, bound to local IP address 0 and port 0, until you explicitly bind it otherwise using *kn_bind()* or until you send data. When data is sent using a UDP socket whose local port is still 0, a unique non-zero port number is automatically bound to the socket to identify the source of the datagram. The local non-zero port number identifies the source in all subsequent transmissions. Once the port number is known, the socket can receive datagrams directed to that port.

Reception is governed by the local IP address and port to which the UDP socket is bound. As long as the socket is bound to IP address 0, the socket can receive datagrams arriving on any of the local interfaces. Once bound to a specific IP address, the socket can only receive datagrams directed to that address. In either case, the datagrams will not be delivered to the socket unless they are destined for the port to which the socket is bound.

Transmission is governed by the destination to which the UDP socket is connected. A socket can be connected to a particular foreign address and port using function *kn_connect()*. Once connected, any of the socket send functions, including *kn_send()*, can be used to send a datagram to the connected host without having to explicitly identify the destination. However, a socket does not have to be connected prior to sending a datagram. Whether connected or not, functions *kn_sendto()* and *kn_sendmsg()* can always be used to send a datagram to a specific foreign host.

The UDP socket connection also restricts the datagrams which the socket can receive. Once the socket has been connected to a specific foreign host, the socket will only receive datagrams directed to it from that foreign host. The socket can be disconnected by "connecting" it to foreign IP address 0 and port 0.

Unless you specifically bind a UDP socket to a particular port, a unique port number will be automatically assigned to the socket the first time the socket is connected or a datagram is sent via the socket. Normally, KwikNet will not allow you to bind a socket to a particular port number if that port number is already used by another UDP channel. To overcome this restriction, you can set the socket *SO_REUSEADDR* option to allow the port number to be reused.

UDP Sockets Examples

The following examples illustrate how a UDP socket can be used in a variety of circumstances for different purposes.

Example 1

To use a UDP socket to communicate on a specific network with one and only one foreign host, proceed as follows. Create a UDP socket and use *kn_bind()* to bind it to a specific local IP address and port. Then call *kn_connect()* to establish a logical connection with the foreign host's IP address and port. Use *kn_recv()* and *kn_send()* to communicate using the socket.

Example 2

A variation of Example 1 will illustrate how logical connections can be manipulated. Until a connection is established, function *kn_send()* cannot be used to send a datagram. However, either *kn_sendto()* or *kn_sendmsg()* can be used at any time to send a datagram to any foreign host without affecting the logical connection. It should also be noted that function *kn_connect()* can also be called to change the connection. The connected foreign host is always used as the destination if a foreign address is not explicitly provided in a request to send a datagram.

Example 3

To accept datagrams from only the foreign hosts to whom you send a datagram, proceed as follows. Create a UDP socket and send a datagram to a foreign host using *kn_sendto()* or *kn_sendmsg()*. The socket is immediately given a unique port number which will be known only to the foreign hosts with which you communicate. Your socket will receive datagrams from any local network interface provided they are directed to your particular port. Note that you cannot use function *kn_send()* to send datagrams on this socket since the socket is not logically connected.

Note

It is recommended that you use the low level UDP interface described in Chapter 4.1 to avoid the memory and execution overhead introduced by the use of sockets. However, if the TCP protocol and sockets interface is also required by your application, there is no compelling reason not to use UDP sockets.

5.3 Socket Options

The operation of sockets is controlled by socket level options. Options are always present at the socket level identified as *SOL_SOCKET*. KwikNet also supports TCP options at protocol level *IPPROTO_TCP*. There are no UDP protocol options. The options are defined in the file *KN_SOCK.H*. Procedures *kn_getsockopt()* and *kn_setsockopt()* are used to access and modify these options.

The following options are recognized by KwikNet. Options marked R can be read using *kn_getsockopt()*. Options marked W can be modified using *kn_setsockopt()*. Unmarked options will return an error indication if referenced. The options marked with > are non-standard extensions offered only by KwikNet.

Option	type	UDP	TCP	Purpose
<i>SO_REUSEADDR</i>	<i>bool</i>	RW	RW	Local address reuse
<i>SO_ACCEPTCONN</i>	<i>bool</i>		R	Check if socket is a listening socket
<i>SO_KEEPAIVE</i>	<i>bool</i>		RW	Keep connections alive
<i>SO_DONTROUTE</i>	<i>bool</i>			Routing bypass for outgoing messages
<i>SO_BROADCAST</i>	<i>bool</i>	RW		Permission to transmit broadcast messages
<i>SO_OOBINLINE</i>	<i>bool</i>		RW	Allow out-of-band data in band
<i>SO_LINGER</i>	<i>struct</i>		RW	Linger on close if data present
<i>SO_SNDBUF</i>	<i>int</i>		RW	Buffer size for send
<i>SO_RCVBUF</i>	<i>int</i>		RW	Buffer size for receive
<i>SO_SNDLOWAT</i>	<i>int</i>		R	Buffer low limit for send
<i>SO_RCVLOWAT</i>	<i>int</i>		R	Buffer low limit for receive
<i>SO_SNDTIMEO</i>	<i>struct</i>		R	Timeout limit for send
<i>SO_RCVTIMEO</i>	<i>struct</i>		R	Timeout limit for receive
<i>SO_TYPE</i>	<i>int</i>	R	R	Get type of socket
<i>SO_ERROR</i>	<i>int</i>	R	R	Get and clear error on a socket
<i>SO_NONBLOCK</i>	<i>bool</i>	RW	RW	> Socket operates in non-blocking mode
<i>SO_RXDATA</i>	<i>int</i>		R	> Get received byte count
<i>SO_TXDATA</i>	<i>int</i>		R	> Get untransmitted byte count
<i>SO_MAXMSG</i>	<i>int</i>	R	R	> Get maximum message segment size
<i>TCP_NODELAY</i>	<i>bool</i>		RW	Do not delay data send to coalesce data
<i>TCP_NOOPT</i>	<i>bool</i>		RW	Do not send TCP options
<i>TCP_MAXSEG</i>	<i>int</i>		R	Get maximum segment size (MSS)

Option *SO_REUSEADDR* indicates that the rules used in validating addresses supplied in a *kn_bind()* call should allow reuse of local addresses.

Option *SO_ACCEPTCONN* can be used to determine if a socket is a listening socket, a socket which accepts requests for connection.

Option *SO_KEEPAIVE* enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken. A process attempting to read from or write to the socket receives an error indication.

Option *SO_DONTROUTE* indicates that outgoing messages should bypass the standard routing facilities. KwikNet does not support this option.

Option *SO_BROADCAST* requests permission to send broadcast datagrams on the socket.

Option *SO_OOBINLINE*, used with protocols that support out-of-band data, requests that out-of-band data be placed in the normal data queue as it is encountered. The data will then be accessible with the *kn_recv()* call without the need to specify the *MSG_OOB* flag.

Option *SO_LINGER* controls the action taken when unsent messages remain queued on a socket at the time a *kn_close()* request to delete the socket is made. If the socket promises reliable delivery of data and *SO_LINGER* is set, KwikNet will block the caller on the *kn_close()* attempt until it is able to successfully transmit the data or until it decides it is unable to do so. A timeout period, termed the linger interval, is specified in the *kn_setsockopt()* call at the time option *SO_LINGER* is enabled. If *SO_LINGER* is disabled at the time that a *kn_close()* request is issued, KwikNet will process the close in a manner that allows the caller to resume as quickly as possible.

Options *SO_SNDBUF* and *SO_RCVBUF* are used to adjust the normal buffer sizes allocated for send and receive buffers respectively. The buffer size may be increased for high volume connections or may be decreased to limit the possible backlog of incoming data. These values reflect the total amount of data which can be buffered at the socket. The receive buffer size determines the largest TCP window size which the socket will advertise. The send buffer size determines the maximum number of bytes which can be held in the socket pending acknowledgment of receipt by the peer. When a socket is created, KwikNet sets both buffering limits to 8 Kbytes. KwikNet places an absolute limit of 16 Kbytes on these values.

Options *SO_SNDLOWAT* and *SO_RCVLOWAT* set the minimum data count for send and receive operations respectively. KwikNet sets each of these values to 1. The values can be read if required.

Options *SO_SNDTIMEO* and *SO_RCVTIMEO* set a timeout value for send and receive operations respectively. Since KwikNet does not support timeouts for send or receive, it reports a timeout value of 0 when either value is read.

Option *SO_TYPE* can be used to determine the type of the socket: *SOCK_STREAM* or *SOCK_DGRAM*. This option is useful for servers that inherit sockets created by other processes.

Option *SO_ERROR* can be used to fetch the most recent error code recorded in the socket. The socket's error code is then reset (cleared). This option is useful for checking for asynchronously occurring socket errors.

Non-Standard Socket Options

The non-standard option *SO_NONBLOCK* can be used to set a socket into non-blocking mode so that the socket user will not be forced to wait if a requested operation cannot be completed at the time of the request. It can also be used to restore a socket to blocking mode so that the socket user will be forced to wait until the requested operation completes. This option's integer parameter is non-zero for non-blocking mode or zero for blocking mode.

Non-standard option *SO_RXDATA* can be used to determine the number of bytes present in the socket's receive buffer, ready to be read.

Non-standard option *SO_TXDATA* can be used to determine the number of bytes still in the socket's transmit buffer, waiting to be sent when conditions permit. Note that bytes which have been sent to the peer but not yet acknowledged are not included in this count.

Non-standard option *SO_MAXMSG* can be used to determine the maximum segment size (MSS) in bytes, excluding protocol headers, which can be sent via the network using the socket. When a TCP or UDP socket is created, the MSS is set according to the largest possible datagram that can be sent on any of the locally available networks.

Once a TCP connection is initiated, the MSS is downwards adjusted to match the largest datagram which can be sent using the connection's local network interface. The MSS is further adjusted downwards if necessary to prevent exceeding the peer's MSS.

The MSS of a UDP socket can be read but cannot be adjusted. If a maximum size UDP datagram must be sent out a local network interface which has a smaller maximum datagram size (smaller MTU), the datagram will be fragmented at the IP layer. If IP fragmentation is not enabled, the UDP datagram will be lost, with no error indication.

TCP Protocol Options

Option *TCP_NODELAY* can be used to adjust the way the TCP protocol sends data. Normally, data is allowed to collect in the socket until a reasonable packet of data can be delivered to the peer. The data threshold is determined by the receive window size announced by the peer. This throttling mechanism assures reasonable delivery times without large numbers of small packets. The *TCP_NODELAY* option can be used to override this mechanism and force data to be sent whenever it is available.

Option *TCP_NOOPT* can be used to prevent the TCP protocol from sending TCP options when a connection is established. If TCP options are inhibited using this option, the local maximum segment size (MSS) will not be announced to the peer.

Option *TCP_MAXSEG* can be used to determine the socket's maximum segment size (MSS) for transmission. This option is the TCP protocol equivalent of the non-standard *SO_MAXMSG* option which is available at the socket level.

This page left blank intentionally.

5.4 KwikNet Socket Services

KwikNet Socket Service Summary

The following list summarizes all of the KwikNet sockets procedures which are accessible to the user. They are grouped functionally for easy reference.

<i>kn_socket</i>	Create a socket (an endpoint for communication)
<i>kn_bind</i>	Bind a local address to a socket
<i>kn_connect</i>	Connect a socket to a specific address
<i>kn_listen</i>	Request a socket to listen for connection requests
<i>kn_accept</i>	Accept a connection request and establish a new socket
<i>kn_close</i>	Close a socket
<i>kn_recv</i>	Receive data from a connected socket
<i>kn_recvfrom</i>	Receive data from a socket (gets sender's address)
<i>kn_recvmsg</i>	Receive scattered data from a socket (gets sender's address)
<i>kn_send</i>	Send data to a socket
<i>kn_sendto</i>	Send data to a socket (with destination address)
<i>kn_sendmsg</i>	Send scattered data to a socket (with destination address)
<i>kn_errno</i>	Fetch most recent status result (error) recorded for a socket
<i>kn_shutdown</i>	Shutdown all or part of a full duplex socket connection
<i>kn_select</i>	Select sockets ready to receive or send data
<i>kn_getpeername</i>	Get the address of the remote end of a connected socket
<i>kn_getsockname</i>	Get the local address of a socket
<i>kn_getsockopt</i>	Get a particular socket option
<i>kn_setsockopt</i>	Set a particular socket option

The following BSD-like services, available from the KwikNet IP Library, are also of use when programming applications which use TCP as their protocol.

<i>netlong</i>	= <i>htonl(hostlong)</i>	Convert <i>long</i> from host to network endian form
<i>netshort</i>	= <i>htons(hostshort)</i>	Convert <i>short</i> from host to network endian form
<i>hostlong</i>	= <i>ntohl(netlong)</i>	Convert <i>long</i> from network to host endian form
<i>hostshort</i>	= <i>ntohs(netshort)</i>	Convert <i>short</i> from network to host endian form

This page left blank intentionally.

Purpose **Accept a Connection Request****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_accept(int s, struct sockaddr *addr, int *addrlen);
```

Description *s* is a socket descriptor identifying the socket on which to wait. The socket *s* must have been created with a call to *kn_socket()*, bound to an address with a call to *kn_bind()* and then made ready to receive connection requests with a call to *kn_listen()*.

Addr is a pointer to storage for the address of the client making the request for connection. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.

Addrlen is a pointer to storage for the length of the client address. On entry, the integer at **addrlen* must define the maximum storage available within the structure referenced by *addr*.

If a request for connection is pending at socket *s*, *kn_accept()* creates a new socket with the same properties as socket *s* and returns the socket descriptor of the new socket to the caller. The new socket must be used for data transfers to or from the client to which the new socket is connected. The new socket cannot be used to accept more connections.

In most cases, the *kn_accept()* caller will be forced to wait for a connection request if none is pending at the time of the call. However, if the socket *s* was marked as non-blocking, *kn_accept()* will return immediately to the caller with an error indication if an outstanding connection request is not present. Socket *s* remains open listening for connection requests.

Returns If successful, a positive, non-zero socket descriptor is returned.
The structure at **addr* contains the client's address.
The length of the address (in bytes) is stored at **addrlen*.

...more

Returns ...continued

On failure, the error status `-1` is returned. The storage at `*addr` and `*addrlen` is unaltered.

The error indicator for socket `s` is set to define the reason for failure. Use `kn_errno()` to retrieve the error code.

<code>EBADF</code>	The socket descriptor <code>s</code> is invalid.
<code>EINVAL</code>	The socket is no longer accepting connections or parameter <code>*addrlen</code> specifies a length that is less than that required to accommodate a valid address.
<code>ECONNABORTED</code>	The connection was aborted.
<code>EOPNOTSUPP</code>	The referenced socket is not of type <code>SOCK_STREAM</code> .
<code>EWouldBlock</code>	The socket is marked non-blocking and no requests for connection are present to be accepted.
<code>ENOMEM</code>	Memory needed to service the request is unavailable.

See Also `kn_bind()`, `kn_listen()`, `kn_socket()`

Purpose Bind a Local Address to a Socket**Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_bind(int s, struct sockaddr *localaddr, int addrlen);
```

Description *s* is a socket descriptor identifying the socket to be bound.*Localaddr* is a pointer to a structure containing the local address to which the socket must be bound. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.If *localaddr* is *NULL*, a null (all zeroes) address in the *AF_INET* address family will be used and parameter *addrlen* will be ignored.*Addrlen* is the length of the address at **localaddr*, measured in bytes.**Returns** If successful, a value of 0 is returned.
On failure, the error status -1 is returned.The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EADDRNOTAVAIL</i>	The specified address is not available from the local machine.
<i>EADDRINUSE</i>	The specified address is already in use.
<i>EINVAL</i>	The socket is already bound to an address or parameter <i>addrlen</i> is invalid.
<i>ENOMEM</i>	Memory needed to service the request is unavailable.

Note When a socket of type *SOCK_DGRAM* is bound to the address identified by *localaddr*, the socket is immediately primed to receive datagrams directed to that address from any foreign source.**See Also** *kn_accept()*, *kn_getsockname()*, *kn_listen()*, *kn_socket()*

Purpose **Close a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN SOCK.H*.

```
#include "KN SOCK.H"
int kn_close(int s);
```

Description *s* is a socket descriptor identifying the socket to be closed.

When a socket is closed, associated naming information and queued data are discarded.

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

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

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

EBADF The socket descriptor *s* is invalid.

See Also *kn_socket()*

Purpose **Connect a Socket to a Specific Address****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_connect(int s, struct sockaddr *destaddr, int addrlen);
```

Description *s* is a socket descriptor identifying the socket to which the specified address is to be bound.

Destaddr is a pointer to a structure containing the specific address to which the socket must be bound. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.

Addrlen is the length of the address at **destaddr*, measured in bytes.

If socket *s* is of type *SOCK_STREAM*, then an attempt will be made to establish a connection with the address specified in the call. Stream sockets may successfully connect only once.

If socket *s* is of type *SOCK_DGRAM*, then **destaddr* specifies the address of the peer with which the socket is to be associated. This is the address to which datagrams are to be sent and is the only address from which datagrams are to be received. For datagram sockets, you can call *kn_connect()* at any time to change the connection address. To disconnect a datagram socket, call *kn_connect()* passing it a null (all zeroes) address.

Returns If successful, a value of 0 is returned.
On failure, the error status -1 is returned.

...more

Returns

...continued

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EADDRNOTAVAIL</i>	The specified address is not available from the local machine or a broadcast address is not allowed as a destination for this socket.
<i>EADDRINUSE</i>	The specified address is already in use.
<i>EAFNOSUPPORT</i>	Addresses in the address family specified by <i>*destaddr</i> cannot be used with this socket.
<i>EINVAL</i>	The socket is already bound to an address or parameter <i>addrlen</i> is invalid.
<i>EISCONN</i>	The socket is already connected.
<i>ETIMEDOUT</i>	Timed out before connection could be established.
<i>ECONNREFUSED</i>	The attempt to connect was forcefully rejected.
<i>EINPROGRESS</i>	The socket is non-blocking and the connection cannot be established immediately.
<i>EALREADY</i>	The socket is non-blocking and a previous connection attempt is in progress but is not yet complete.
<i>EOPNOTSUPP</i>	The operation is not supported by this socket.
<i>ENOMEM</i>	Memory needed to service the request is unavailable.

See Also

kn_bind(), *kn_getsockname()*, *kn_select()*, *kn_socket()*

Purpose **Get Error Code from Recent Socket Operation****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_errno(int s);
```

Description *s* is a socket descriptor identifying the socket from which error information is to be retrieved.**Returns** If successful, a positive error code is returned. These error codes are summarized in Appendix B. The error code identifies the reason for the failure, if any, of the most recent operation attempted using socket *s*.

An error status of *EBADF* is returned if the socket descriptor *s* is invalid, precluding the interrogation of the socket.

Usually the error code retrieved from socket *s* corresponds to the result of the most recent sockets call made by your application. However, since KwikNet always records any relevant socket error which it detects, your call to *kn_errno()* may actually retrieve an error code recorded by KwikNet at some time after your socket operation. You can use this feature to advantage to interrogate the status of a listening socket while it continues to operate.

The following error codes, if observed, indicate that the corresponding error was detected by KwikNet asynchronously to your application.

<i>EPIPE</i>	Cannot send any more data.
<i>ESHUTDOWN</i>	The connection has been shutdown.
<i>EHAVEOOB</i>	Have received out-of-band data.
<i>ECONNRESET</i>	The connection has been reset.

Note The error code associated with socket *s* remains unaltered. To read the error code and reset the error code to 0, call *kn_getsockopt()* with option *SO_ERROR*.**See Also** *kn_getsockopt()*

Purpose **Get the Address (Name) of the Remote End of a Connected Socket**

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

Setup Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_getpeername(int s, struct sockaddr *destaddr,
                   int *addrlen);
```

Description *s* is a socket descriptor identifying the socket for which the remote end (peer) address is desired.

Destaddr is a pointer to storage for the address of the remote end of the current connection on socket *s*. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.

Addrlen is a pointer to storage for the length of the remote address. On entry, the integer at **addrlen* must define the maximum storage available within the structure referenced by *destaddr*.

Returns If successful, a value of 0 is returned.
 The structure at **destaddr* contains the remote end address.
 The length of the address (in bytes) is stored at **addrlen*.

On failure, the error status -1 is returned. The storage at **destaddr* and **addrlen* is unaltered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOMEM</i>	Memory needed to service the request is unavailable.
<i>ENOTCONN</i>	The socket is not connected.
<i>EINVAL</i>	Parameter <i>*addrlen</i> specifies a length that is less than that required to accommodate a valid address.

See Also *kn_connect()*, *kn_getsockname()*, *kn_socket()*

Purpose **Get the Local Address (Name) of a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_getsockname(int s, struct sockaddr *localaddr,
                  int *addrlen);
```

Description *s* is a socket descriptor identifying the socket for which the local address is desired.

Localaddr is a pointer to storage for the local address assigned to socket *s*. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.

Addrlen is a pointer to storage for the length of the local address. On entry, the integer at **addrlen* must define the maximum storage available within the structure referenced by *localaddr*.

Returns If successful, a value of 0 is returned.
The structure at **localaddr* contains the local address.
The length of the address (in bytes) is stored at **addrlen*.

On failure, the error status -1 is returned. The storage at **localaddr* and **addrlen* is unaltered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOMEM</i>	Memory needed to service the request is unavailable.
<i>EINVAL</i>	Parameter <i>*addrlen</i> specifies a length that is less than that required to accommodate a valid address.

See Also *kn_bind()*, *kn_getpeername()*, *kn_socket()*

Purpose **Get a Particular Socket Option****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_getsockopt(int s, int level, int optionid,
                  void *optionval, int *optionlen);
```

Description *s* is a socket descriptor identifying the socket for which the particular socket information is desired.

Level is an identifier indicating the socket or protocol level for which information is required. Use *SOL_SOCKET* for the highest, socket level information. Use *IPPROTO_TCP* for TCP protocol level information. Options for other protocol levels are not supported.

Optionid identifies the option of interest to the caller. The following option identifiers can be used to determine the state of the option or to retrieve its associated parameter. These options are described in Chapter 5.3. Options marked > are non-standard KADAK extensions. Only one option can be specified in each call.

<i>SO_REUSEADDR</i>	<i>bool</i>	UDP	TCP	Local address reuse
<i>SO_ACCEPTCONN</i>	<i>bool</i>		TCP	Check for listening socket
<i>SO_KEEPAIVE</i>	<i>bool</i>		TCP	Keep connections alive
<i>SO_BROADCAST</i>	<i>bool</i>	UDP		Permission to broadcast messages
<i>SO_OOBINLINE</i>	<i>bool</i>		TCP	Allow out-of-band data in band
<i>SO_LINGER</i>	<i>struct</i>		TCP	Linger on close if data present
<i>SO_SNDBUF</i>	<i>int</i>		TCP	Buffer size for send
<i>SO_RCVBUF</i>	<i>int</i>		TCP	Buffer size for receive
<i>SO_SNDLOWAT</i>	<i>int</i>		TCP	Buffer low limit for send
<i>SO_RCVLOWAT</i>	<i>int</i>		TCP	Buffer low limit for receive
<i>SO_SNDTIMEO</i>	<i>struct</i>		TCP	Timeout limit for send
<i>SO_RCVTIMEO</i>	<i>struct</i>		TCP	Timeout limit for receive
<i>SO_TYPE</i>	<i>int</i>	UDP	TCP	Get type of socket
<i>SO_ERROR</i>	<i>int</i>	UDP	TCP	Get and clear error on a socket
<i>SO_NONBLOCK</i>	<i>bool</i>	UDP	TCP	> Socket is non-blocking mode
<i>SO_RXDATA</i>	<i>int</i>		TCP	> Get received byte count
<i>SO_TXDATA</i>	<i>int</i>		TCP	> Get untransmitted byte count
<i>SO_MAXMSG</i>	<i>int</i>	UDP	TCP	> Get maximum message segment size
<i>TCP_NODELAY</i>	<i>bool</i>		TCP	Do not delay send to coalesce data
<i>TCP_NOOPT</i>	<i>bool</i>		TCP	Do not send TCP options
<i>TCP_MAXSEG</i>	<i>int</i>		TCP	Get maximum segment size (MSS)

...more

Description ...continued

Optionval is a pointer to storage for the option value. The size of each option is indicated in the option list. Note that *bool* is an *int* result that is non-zero if the option is enabled or 0 if the option is disabled. The structure *linger* required by option *SO_LINGER* is defined in header file *KN_SOCK.H*. The structure *timeval* required by options *SO_SNDTIMEO* and *SO_RCVTIMEO* is defined in header file *KN_SOCK.H*.

Optionlen is a pointer to storage for the length of the returned option value. On entry, the integer at **optionlen* must define the maximum storage available at the location referenced by *optionval*.

Returns

If successful, a value of 0 is returned.

The storage at **optionval* contains the option value.

The length of the option value (in bytes) is stored at **optionlen*.

On failure, the error status -1 is returned. The storage at **optionval* and **optionlen* is unaltered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOPROTOOPT</i>	The option is unknown at the level indicated.
<i>EINVAL</i>	Parameter <i>*optionlen</i> specifies a length that is less than that required to accommodate the result.

See Also

kn_setsockopt(), *kn_socket()*

Purpose Request a Socket to Listen for Connection Requests**Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_listen(int s, int backlog);
```

Description *s* is a socket descriptor identifying the socket on which requests for connection will be enqueued. The socket *s* must have been created with a call to *kn_socket()* and bound to an address with a call to *kn_bind()*. The socket must be of type *SOCK_STREAM*.*Backlog* is the maximum number of pending connection requests which the socket is permitted to queue. *Backlog* must be greater than or equal to 0 and no greater than the maximum allowed by your KwikNet library configuration.Any request for a connection to the socket will be permitted up to the maximum specified by *backlog*. Requests are kept in a queue in the order received. When the application calls *accept()*, it is given a new socket connected to the client from whom the connection request was received.

If a request for connection is received while the socket queue is full, the request is rejected. Subsequent actions, if any, will be determined by the client whose request was rejected.

Returns If successful, a value of 0 is returned.
On failure, the error status -1 is returned.The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	The connection queue specified by <i>backlog</i> is < 0 or greater than the maximum allowed by your KwikNet library configuration.
<i>EOPNOTSUPP</i>	The operation is not supported by a socket unless the socket is of type <i>SOCK_STREAM</i> .

See Also *kn_accept()*, *kn_bind()*, *kn_socket()*

Purpose **Receive Data from a Connected Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_recv(int s, void *buf, int len, int flags);
```

Description *s* is a socket descriptor identifying the connected socket from which data is to be received.*Buf* is a pointer to storage for the received data.*Len* is the buffer size, measured in bytes.*Flags* is a control variable used to modify the receive process. *Flags* is 0 or the logical OR of any of the following values.

<i>MSG_OOB</i>	Read out-of-band data.
<i>MSG_PEEK</i>	Peek at the received data but do not remove the data from the socket.
<i>MSG_DONTWAIT</i>	Receive in non-blocking fashion.

Returns If successful, the number of bytes of data stored at **buf* is returned.
If the socket is closed by the sender, the value 0 is returned.
On failure, the error status -1 is returned.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOTCONN</i>	The socket is not connected.
<i>ECONNRESET</i>	The connection has been reset.
<i>EINVAL</i>	The buffer length <i>len</i> is declared to be less than 0 or an error was encountered while processing out-of-band data
<i>EWouldBlock</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> and no data is available for reading.

Restrictions If there is no data available at the socket, the caller will be blocked waiting for data to arrive unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a -1 error status and the error code *EWouldBlock* will be stored in the socket descriptor.**See Also** *kn_connect()*, *kn_recvfrom()*, *kn_recvmsg()*,
kn_send(), *kn_sendto()*, *kn_sendmsg()*, *kn_socket()*

Purpose **Receive Data from a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_recvfrom(int s, void *buf, int len, int flags,
                struct sockaddr *from, int *fromlen);
```

Description *s* is a socket descriptor identifying the socket from which data is to be received. The socket can be connected or unconnected.*Buf* is a pointer to storage for the received data.*Len* is the buffer size, measured in bytes.*Flags* is a control variable used to modify the receive process. *Flags* is 0 or the logical OR of any of the following values.

<i>MSG_OOB</i>	Read out-of-band data.
<i>MSG_PEEK</i>	Peek at the received data but do not remove the data from the socket.
<i>MSG_DONTWAIT</i>	Receive in non-blocking fashion.

From is a pointer to storage for the address of the source of the received data. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.

The purpose of this parameter is to allow the sender to be identified when using a connectionless socket such as that used for UDP datagrams on a socket of type *SOCK_DGRAM*. The format of the address is suitable for sending a response using procedure *kn_sendto()*. Set *from* to *NULL* if the address of the message sender is not required.

When using a TCP socket, set parameter *from* to *NULL* and be sure to connect before calling this function. Alternatively use procedure *kn_recv()*.

Fromlen is a pointer to storage for the length of the sender's address. On entry, the integer at **fromlen* must define the maximum storage available within the structure referenced by *from*. If parameter *from* is set to *NULL*, set parameter *fromlen* to *NULL* also.

...more

Returns If successful, the number of bytes of data stored at **buf* is returned.
 If the socket is closed by the sender, the value 0 is returned.
 The structure at **from* contains the sender's address.
 The length of the address (in bytes) is stored at **fromlen*.

On failure, the error status -1 is returned. The storage at **from* and **fromlen* is unaltered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOTCONN</i>	The socket is not connected.
<i>ECONNRESET</i>	The connection has been reset.
<i>EINVAL</i>	The buffer length <i>len</i> is declared to be less than 0 or parameter <i>*fromlen</i> specifies a length that is less than that required to accommodate a valid address or an error was encountered while processing out-of-band data
<i>EWouldBlock</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> and no data is available for reading.

Restrictions If there is no data available at the socket, the caller will be blocked waiting for data to arrive unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a -1 error status and the error code *EWouldBlock* will be stored in the socket descriptor.

See Also *kn_connect()*, *kn_recv()*, *kn_recvmsg()*,
kn_send(), *kn_sendto()*, *kn_sendmsg()*, *kn_socket()*

Purpose **Receive Scattered Data from a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_recvmsg(int s, struct msghdr *msg, int flags);
```

Description *s* is a socket descriptor identifying the socket from which data is to be received. The socket can be connected or unconnected.*Flags* is a control variable used to modify the receive process. *Flags* is 0 or the logical OR of any of the following values.

<i>MSG_OOB</i>	Read out-of-band data.
<i>MSG_PEEK</i>	Peek at the received data but do not remove the data from the socket.
<i>MSG_DONTWAIT</i>	Receive in non-blocking fashion.

Msg is a pointer to a structure defining the manner in which the received data should be processed. Structure *msghdr* is defined in file *KN_SOCK.H* as follows:

```
struct msghdr {
    struct sockaddr *msg_name; /* Socket address          */
    long    msg_namelen;      /* Length of address    */
    struct iovec *msg_iov;    /* Data vector array     */
    long    msg_iovlen;      /* # of entries in msg_iov */
    char    *msg_control;    /* Control data          */
    int     msg_controllen;   /* Length of control data */
    int     msg_flags;        /* Received flags        */
};
```

Msg_name is a pointer to storage for the address of the source of the received data. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.

The purpose of this parameter is to allow the sender to be identified when using a connectionless socket such as that used for UDP datagrams on a socket of type *SOCK_DGRAM*. The format of the address is suitable for sending a response using procedure *kn_sendmsg()*. Set *msg_name* to *NULL* if the address of the message sender is not required.

When using a TCP socket, set *msg_name* to *NULL* and be sure to connect before calling this function.

...more

Description ...continued

Msg_namelen is the length of the sender's address. On entry, *msg_namelen* must define the maximum storage available within the structure referenced by *msg_name*. If parameter *msg_name* is set to *NULL*, set parameter *msg_namelen* to 0.

Msg_iov is a pointer to an array of data vectors describing the locations of storage buffers for the received data. Each data vector is an *iovec* structure which is defined in file *KN_SOCKET.H* as follows:

```
struct iovec {  
    void    *iov_base;           /* Data pointer          */  
    long    iov_len;            /* Length of data        */  
};
```

Iov_base is a pointer to storage for the received data.

Iov_len is the data buffer size, measured in bytes.

Msg_iovlen is an integer defining the number of data vectors (*iovec* structures) which are provided in the array referenced by parameter *msg_iov*.

Msg_control is a pointer to storage for additional control information received along with the message. If control data is not required, set *msg_control* to *NULL*. This parameter is not used by KwikNet since it is not required for any of the supported sockets protocols.

Msg_controllen is the length, in bytes, of the control data storage buffer. On entry, *msg_controllen* must define the maximum storage available within the structure referenced by *msg_control*. If parameter *msg_control* is set to *NULL*, set parameter *msg_controllen* to 0.

Msg_flags is an extra variable received along with the message. This variable is not supported by KwikNet since it is not provided by any of the supported sockets protocols. *Msg_flags* will be undefined on return.

...more

Returns

If successful, the total number of bytes of data received is returned.
If the socket is closed by the sender, the value 0 is returned.
The structure at **msg_name* contains the sender's address.
The length of the address (in bytes) is stored at *msg_namelen*.
The array of *iovec* structures at **msg_iov* is altered as described below.
Fields **msg_control*, *msg_controllen* and *msg_flags* are not altered.

As data is received, the array of data vectors is updated. The data pointer in each data vector is incremented by *n* where *n* is the number of data bytes received using that vector. The corresponding data vector length parameter is decremented by *n*. Hence, upon return, the data vectors will have been updated to reflect the total number of bytes which have been received. Since the data vectors are updated, you cannot use them after the call to examine the received data.

Under error conditions, the data vectors may have been updated before the error was detected and reported. If all of the expected data bytes have not been received, you can issue another call to *kn_rcvmsg()* to receive the remaining bytes of data without altering the data vector array.

On failure, the error status -1 is returned. The storage at **msg_name* and *msg_namelen* is unaltered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOTCONN</i>	The socket is not connected.
<i>ECONNRESET</i>	The connection has been reset.
<i>EINVAL</i>	The buffer length in a data vector was declared to be less than 0 or parameter <i>msg_namelen</i> specifies a length that is less than that required to accommodate a valid address or an error was encountered while processing out-of-band data.
<i>EWouldBLOCK</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> and no data is available for reading.

Restrictions

If there is no data available at the socket, the caller will be blocked waiting for data to arrive unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a -1 error status and the error code *EWouldBLOCK* will be stored in the socket descriptor.

On 16-bit processors, the amount of data which can be received is restricted to 32767 bytes because the value returned by *kn_rcvmsg()* is a signed integer.

See Also

kn_connect(), *kn_rcv()*, *kn_rcvfrom()*,
kn_send(), *kn_sendto()*, *kn_sendmsg()*, *kn_socket()*

Purpose **Select Sockets Ready for Receive or Send****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN SOCK.H*.

```
#include "KN SOCK.H"
int kn_select(int nfd, fd_set *readfds,
              fd_set *writefds,
              fd_set *exceptfds,
              struct timeval *timeout);
```

Description *nfd* indicates the number of descriptors which are to be examined in each descriptor set. For example, if your descriptor sets identify sockets 2, 3 and 5, parameter *nfd* should be 5. For convenience, if *nfd* is -1, KwikNet will examine all sockets identified in each descriptor set.

Readfds is a pointer to a descriptor set which, on entry, identifies the sockets to be interrogated. Upon return, the descriptor set at **readfds* will be updated to identify which of the interrogated sockets have received data ready to be read. A socket will also be declared ready to read if an error other than *EINPROGRESS* or *EWOULDBLOCK* is recorded in the socket while the socket is selected. Set this parameter to *NULL* to ignore sockets which are ready to be read.

Writefds is a pointer to a descriptor set which, on entry, identifies the sockets to be interrogated. Upon return, the descriptor set at **writefds* will be updated to identify which of the interrogated sockets have no data remaining to be sent. Such sockets are ready for sending more data. A socket will also be declared ready for sending if an error other than *EINPROGRESS* is recorded in the socket while the socket is selected. Set this parameter to *NULL* to ignore sockets which are ready for sending.

Exceptfds is a pointer to a descriptor set which, on entry, identifies the sockets to be interrogated. Upon return, the descriptor set at **exceptfds* will be updated to identify which of the interrogated sockets have outstanding exceptions present. KwikNet treats recorded errors other than *EINPROGRESS* and *EWOULDBLOCK* as exceptions. KwikNet also reports an exception if out-of-band data has been received. Set *exceptfds* to *NULL* to ignore sockets with outstanding exceptions.

Timeout is a pointer to a structure which defines the interval for which the caller is prepared to wait for at least one socket to meet the selected criteria. Set *timeout* to *NULL* to wait forever. Set the interval value to 0 to return immediately if no sockets are ready.

...more

Description ...continued

Structure *timeval* is defined in file *KN_SOCKET.H* as follows:

```
struct timeval {
    unsigned long tv_sec;      /* Number of seconds      */
    unsigned long tv_usec;    /* Number of microseconds */
};
```

Type *fd_set* is defined by a *typedef* in file *KN_SOCKET.H*. Variables of type *fd_set* can be manipulated using the following macros which are defined in file *KN_SOCKET.H*. In the descriptions which follow, *fdset* is a variable of type *fd_set* and *s* is a valid socket descriptor.

<i>FD_SET(s, &fdset)</i>	Set socket <i>s</i> identifier in variable <i>fdset</i> .
<i>FD_CLR(s, &fdset)</i>	Clear socket <i>s</i> identifier in variable <i>fdset</i> .
<i>FD_ISSET(s, &fdset)</i>	Test socket <i>s</i> identifier in variable <i>fdset</i> .
<i>FD_ZERO(&fdset)</i>	Clear all socket identifiers in variable <i>fdset</i> .

Use *FD_ZERO* to reset (clear) all socket identifiers in a descriptor set. Use *FD_SET* to identify the specific sockets to be interrogated. *FD_ISSET* returns a non-zero value if the identifier for socket *s* is set or zero if the identifier for socket *s* is clear.

The behaviour of these macros is undefined if the socket descriptor *s* is invalid.

Note that the structure and parameter names are derived from UNIX for which this procedure interrogates both files and sockets.

Returns If successful, this procedure returns the total number of ready sockets identified in the descriptor sets. The variables **readfds*, **writefds* and **exceptfds* are updated to identify the subset of the sockets specified by the caller which match their respective criteria.

On failure, the error status *-1* is returned. Unfortunately, the error indicator defining the reason for failure cannot be recorded. You cannot use *kn_errno()* to retrieve the error code since you do not have a unique socket descriptor to interrogate. The following error codes, although not available for testing, still define the possible reasons for failure.

<i>EBADF</i>	A socket descriptor in one of the sets is invalid.
<i>EINVAL</i>	The <i>timeout</i> parameter is invalid.

See Also *kn_socket()*

Purpose **Send Data to a Connected Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_send(int s, void *buf, int len, int flags);
```

Description *s* is a socket descriptor identifying the connected socket to which data is to be sent.*Buf* is a pointer to the buffer of data to be sent.*Len* is the buffer size, measured in bytes.*Flags* is a control variable used to modify the send process. *Flags* is 0 or the logical OR of any of the following values.

<i>MSG_OOB</i>	Allow sending of out-of-band data.
<i>MSG_DONTWAIT</i>	Send message in non-blocking fashion.

Returns If successful, the number of bytes of data sent from **buf* is returned.
On failure, the error status *-1* is returned.The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOTCONN</i>	The socket is not connected.
<i>ECONNRESET</i>	The connection has been reset.
<i>EPIPE</i>	Cannot send any more out socket <i>s</i> .
<i>EINVAL</i>	The buffer length <i>len</i> is declared to be less than 0 or is invalid for the socket's protocol.
<i>EWouldBLOCK</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> but there is a need to block the caller to complete the operation.
<i>EMSGSIZE</i>	The message is larger than the maximum which can be sent atomically as required by the socket protocol.
<i>ENOBUFS</i>	Memory is not available to complete the request.
<i>EDESTADDRREQ</i>	A destination is required but is not available.

Restrictions If the message data cannot be delivered in its entirety to the socket, the caller will be blocked unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a *-1* error status and the error code *EWouldBLOCK* will be stored in the socket descriptor.**See Also** *kn_connect()*, *kn_recv()*, *kn_recvfrom()*, *kn_recvmsg()*,
kn_sendto(), *kn_sendmsg()*, *kn_socket()*

Purpose **Send Scattered Data to a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_sendmsg(int s, struct msghdr *msg, int flags);
```

Description *s* is a socket descriptor identifying the socket to which data is to be sent. The socket can be connected or unconnected.*Flags* is a control variable used to modify the send process. *Flags* is 0 or the logical OR of any of the following values.

<i>MSG_OOB</i>	Allow sending of out-of-band data.
<i>MSG_DONTWAIT</i>	Send message in non-blocking fashion.

Msg is a pointer to a structure defining the data to be sent and how it should be processed. Structure *msghdr* is defined in file *KN_SOCK.H* as follows:

```
struct msghdr {
    struct sockaddr *msg_name; /* Socket address          */
    long  msg_namelen; /* Length of address      */
    struct iovec *msg_iov; /* Data vector array      */
    long  msg_iovlen; /* # of entries in msg_iov */
    char  *msg_control; /* Control data           */
    int  msg_controllen; /* Length of control data */
    int  msg_flags; /* Received flags         */
};
```

Msg_name is a pointer to the address of the destination to which the data is to be sent. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.

This parameter is required to identify the destination address when using a connectionless socket such as that used for UDP datagrams on a socket of type *SOCK_DGRAM*. The format of the address is compatible with that received by procedure *kn_recvmsg()*. Set *msg_name* to *NULL* if the address of the message destination is not required.

When using a TCP socket, set *msg_name* to *NULL* and be sure to connect before calling this function.

Msg_namelen is the length of the destination address. If parameter *msg_name* is set to *NULL*, set parameter *msg_namelen* to 0.

...more

Description ...continued

Msg_iov is a pointer to an array of data vectors describing the locations of the data to be sent. Each data vector is an *iovec* structure which is defined in file *KN_SOCKET.H* as follows:

```
struct iovec {
    void    *iov_base;           /* Data pointer          */
    long    iov_len;            /* Length of data        */
};
```

IoV_base is a pointer to data to be sent.

IoV_len is the data buffer size, measured in bytes.

Msg_iovlen is an integer defining the number of data vectors (*iovec* structures) which are provided in the array referenced by parameter *msg_iov*.

Msg_control is a pointer to additional control information to be sent along with the message. If control data is not required, set *msg_control* to *NULL*. This parameter is not used by KwikNet since it is not required for any of the supported sockets protocols.

Msg_controllen is the length, in bytes, of the control data buffer. If parameter *msg_control* is set to *NULL*, set parameter *msg_controllen* to 0.

Msg_flags is an additional variable to be sent along with the message. This variable is not supported by KwikNet since it is not required for any of the supported sockets protocols.

Returns If successful, the total number of bytes of data sent is returned.
No fields in structure **msg* are altered.
The array of *iovec* structures at **msg_iov* is altered as follows.

As data is sent, the array of data vectors is updated. The data pointer in each data vector is incremented by *n* where *n* is the number of data bytes sent from that vector. The corresponding data vector length parameter is decremented by *n*. Hence, upon return, the data vectors will have been updated to reflect the total number of bytes which have been sent.

...more

Returns ...continued

Under error conditions, the data vectors may have been updated before the error was detected and reported. If all data bytes have not been sent, you can issue another call to `kn_sendmsg()` to send the remaining bytes of data without altering the data vector array.

On failure, the error status `-1` is returned. The error indicator for socket `s` is set to define the reason for failure. Use `kn_errno()` to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOTCONN</i>	The socket is not connected.
<i>ECONNRESET</i>	The connection has been reset.
<i>EPIPE</i>	Cannot send any more out socket <i>s</i> .
<i>EINVAL</i>	The buffer length in a data vector was declared to be less than 0 or is invalid for the socket's protocol or parameter <i>msg_name_len</i> specifies a length that is less than the length of a valid address.
<i>EWouldBlock</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> but there is a need to block the caller to complete the operation.
<i>EMSGSIZE</i>	The message is larger than the maximum which can be sent atomically as required by the socket protocol.
<i>ENOBUFS</i>	Memory is not available to complete the request.
<i>EDESTADDRREQ</i>	A destination is required but is not available.
<i>EADDRNOTAVAIL</i>	The specified address is not available from the local machine or a broadcast address is not allowed as a destination for this socket.

Restrictions If the message data cannot be delivered in its entirety to the socket, the caller will be blocked unless the socket `s` is marked as non-blocking. In the latter case, the caller will resume with a `-1` error status and the error code `EWouldBlock` will be stored in the socket descriptor.

On 16-bit processors, the amount of data which can be sent is restricted to 32767 bytes because the value returned by `kn_sendmsg()` is a signed integer.

See Also `kn_connect()`, `kn_recv()`, `kn_recvfrom()`, `kn_recvmsg()`, `kn_send()`, `kn_sendto()`, `kn_socket()`

Purpose **Send Data to a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_sendto(int s, void *buf, int len, int flags,
              struct sockaddr *to, int tolen);
```

Description *s* is a socket descriptor identifying the socket to which data is to be sent.
The socket can be connected or unconnected.*Buf* is a pointer to the buffer of data to be sent.*Len* is the buffer size, measured in bytes.*Flags* is a control variable used to modify the send process. *Flags* is 0 or the logical OR of any of the following values.*MSG_OOB* Allow sending of out-of-band data.*MSG_DONTWAIT* Send message in non-blocking fashion.*To* is a pointer to the address of the destination to which the data is to be sent. The format of the IP address in structure *sockaddr* is defined in header file *KN_SOCK.H*.

This parameter is required to identify the destination address when using a connectionless socket such as that used for UDP datagrams on a socket of type *SOCK_DGRAM*. The format of the address is compatible with that received by procedure *kn_recvfrom()*. Set parameter *to* to *NULL* if the address of the destination is not required.

When using a TCP socket, set parameter *to* to *NULL* and be sure to connect before calling this function. Alternatively use procedure *kn_send()*.

Tolen is the length of the destination address. If parameter *to* is set to *NULL*, set parameter *tolen* to 0.

...more

Returns If successful, the number of bytes of data sent from **buf* is returned. On failure, the error status *-1* is returned.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOTCONN</i>	The socket is not connected.
<i>ECONNRESET</i>	The connection has been reset.
<i>EPIPE</i>	Cannot send any more out socket <i>s</i> .
<i>EINVAL</i>	The buffer length <i>len</i> is declared to be less than 0 or is invalid for the socket's protocol or parameter <i>tolen</i> specifies a length that is less than the length of a valid address.
<i>EWouldBLOCK</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> but there is a need to block the caller to complete the operation.
<i>EMSGSIZE</i>	The message is larger than the maximum which can be sent atomically as required by the socket protocol.
<i>ENOBUFS</i>	Memory is not available to complete the request.
<i>EDESTADDRREQ</i>	A destination is required but is not available.
<i>EADDRNOTAVAIL</i>	The specified address is not available from the local machine or a broadcast address is not allowed as a destination for this socket.

Restrictions If the message data cannot be delivered in its entirety to the socket, the caller will be blocked unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a *-1* error status and the error code *EWouldBLOCK* will be stored in the socket descriptor.

See Also *kn_connect()*, *kn_recv()*, *kn_recvfrom()*, *kn_recvmsg()*, *kn_send()*, *kn_sendmsg()*, *kn_socket()*

Purpose **Set a Particular Socket Option****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_setsockopt(int s, int level, int optionid,
                  void *optionval, int optionlen);
```

Description *s* is a socket descriptor identifying the socket for which the particular socket option is to be modified.

Level is an identifier indicating the socket or protocol level for which an option must be modified. Use *SOL_SOCKET* for the highest, socket level options. Use *IPPROTO_TCP* for TCP protocol level options. KwikNet does not support modification of options for other protocol levels.

Optionid identifies the option which is to be modified. The following option identifiers can be used to set the state of the option or to modify its associated parameter. These options are described in Chapter 5.3. Options marked > are non-standard KADAK extensions. Only one option can be specified in each call.

<i>SO_BROADCAST</i>	<i>bool</i>	UDP		Permission to broadcast messages
<i>SO_REUSEADDR</i>	<i>bool</i>	UDP	TCP	Local address reuse
<i>SO_KEEPALIVE</i>	<i>bool</i>		TCP	Keep connections alive
<i>SO_OOBINLINE</i>	<i>bool</i>		TCP	Allow out-of-band data in band
<i>SO_LINGER</i>	<i>struct</i>		TCP	Linger on close if data present
<i>SO_SNDBUF</i>	<i>int</i>		TCP	Buffer size for send
<i>SO_RCVBUF</i>	<i>int</i>		TCP	Buffer size for receive
<i>SO_NONBLOCK</i>	<i>bool</i>	UDP	TCP	> Socket is non-blocking mode
<i>TCP_NODELAY</i>	<i>bool</i>		TCP	Do not delay send to coalesce data
<i>TCP_NOOPT</i>	<i>bool</i>		TCP	Do not send TCP options

...more

Description ...continued

Optionval is a pointer to the option value. The size of each option is indicated in the option list. Note that *bool* is an *int* value that is non-zero if the option is to be enabled or 0 if the option is to be disabled. The structure *linger* required by option *SO_LINGER* is defined in header file *KN_SOCK.H*. The structure *timeval* required by options *SO_SNDTIMEO* and *SO_RCVTIMEO* is defined in header file *KN_SOCK.H*.

Optionlen is the length of the option value at the location referenced by *optionval*.

Note Option *SO_NONBLOCK* is a unique KwikNet option which conditions a socket such that subsequent socket operations proceed with or without blocking the caller. Note that, although a socket is always a blocking socket when first created, this option permits the mode of operation to be altered by your application. The option value in the call must be non-zero to set non-blocking mode or zero to restore the socket to blocking mode.

Returns If successful, a value of 0 is returned.
On failure, the error status -1 is returned.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOPROTOOPT</i>	The option is unknown at the level indicated.
<i>EINVAL</i>	One or more parameters are invalid or parameter <i>optionlen</i> specifies a length that is less than that required to specify the option.

See Also *kn_getsockopt()*, *kn_socket()*

Purpose **Shutdown All or Part of a Full Duplex Socket Connection****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_shutdown(int s, int how);
```

Description *s* is a socket descriptor identifying the socket with the connection which is to be shutdown.

How is an integer which defines how the connection is to be adjusted.

- 0 if no further receives are allowed
- 1 if no further sends are allowed
- 2 if no further receives or sends are allowed

Returns If successful, a value of 0 is returned.
On failure, the error status -1 is returned.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOTCONN</i>	The socket is not connected.

See Also *kn_accept()*, *kn_connect()*, *kn_socket()*

Purpose **Create a Socket (an Endpoint for Communication)**

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

Setup Prototype is in file *KN_SOCK.H*.
 #include "KN_SOCK.H"
 int kn_socket(int domain, int type, int protocol);

Description *Domain* specifies the communications domain within which communication will occur. The domain identifies the protocol family which should be used. The protocol family generally matches the address family for the addresses supplied in subsequent socket operations. The only protocol family support by KwikNet is the ARPA Internet Protocol, identified as *PF_INET* . The corresponding address family is *AF_INET*.

Type defines the semantics of communication supported by the socket. *Type* must be *SOCK_STREAM* for use with the TCP protocol or *SOCK_DGRAM* for use with the UDP protocol.

The *SOCK_STREAM* type of socket provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism can be supported.

The *SOCK_DGRAM* type of socket supports datagrams: connectionless, unreliable messages of a fixed (typically small) maximum length.

Protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified.

The protocol parameter is ignored by KwikNet since the protocols are dictated by the domain and socket type. Streaming sockets use TCP over IP. Datagram sockets use UDP over IP.

...more

Returns If successful, a positive, non-zero socket descriptor is returned.
On failure, the error status *-1* is returned.

If a socket cannot be created, the error indicator defining the reason for failure cannot be recorded. You cannot use *kn_errno()* to retrieve the error code since you have no socket descriptor to interrogate. The following error codes, although not available for testing, still define the possible reasons for failure.

<i>EPROTONOSUPPORT</i>	The socket <i>type</i> or the specified <i>protocol</i> is not supported within the <i>domain</i> .
<i>ENOBUFFS</i>	Resources needed to create a socket are unavailable.

Restrictions Sockets of type *SOCK_DGRAM* cannot be created unless you have configured your KwikNet libraries to allow UDP to be used with sockets.

See Also *kn_accept()*, *kn_bind()*, *kn_connect()*

This page left blank intentionally.

A. Reference Materials and Glossary

A.1 Reference Materials

The following reference books and documents are recommended by KADAK's technical staff as good sources of information about TCP/IP and related protocols. Comer's text provides interesting historical background and a good general introduction to the topics of interest. Siyan's massive document is an excellent TCP/IP handbook.

Books

CARLSON, James [1998], *PPP Design and Debugging*, Addison Wesley Longman, Inc., Reading, Massachusetts.

COMER, Douglas E. [1991], *Internetworking with TCP/IP Volume I, Principles, Protocols and Architectures*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

MUSCIANO, Chuck and KENNEDY, Bill [1997], *HTML: The Definitive Guide, Second Edition*, O'Reilly & Associates, Inc., Sebastopol, California.

PERKINS, David and McGINNIS, Evan [1996], *Understanding SNMP MIBS*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

SIYAN, Karanjit S. [1997], *Inside TCP/IP, Third Edition*, New Riders Publishing, Indianapolis, Indiana.

STALLINGS, William [1999], *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*, Addison Wesley Longman, Inc., Reading, Massachusetts.

Internet Sources

<code>www.rfc-editor.org</code>	RFCs and related documents via web server
<code>comp.protocols.tcp-ip</code>	News group for ongoing TCP/IP discussion
<code>comp.protocols.snmp</code>	News group for ongoing SNMP discussion

This page left blank intentionally.

A.2 KwikNet Glossary

API	An application programming interface defines the method by which a software program can access software components such as procedures in the KwikNet Libraries.
App-Task	The name given to the body of application code which uses KwikNet services in a single threaded system. The App-Task is any application function (except those executed by an interrupt service routine) which calls KwikNet to perform some operation.
ARP	Address Resolution Protocol: the TCP/IP protocol used to resolve the correlation between an IP address and a physical hardware address such as an Ethernet address.
BOOTP	Boot Protocol: an older protocol used to derive a network IP address during the startup (boot) initialization of an IP stack.
BSD	The University of California, Berkeley refers to its TCP/IP software release as the Berkely Software Distribution.
Clock Handler	The name given to the procedure which is called by the ISR or ISP root which services the hardware clock interrupt.
Clock Tick	The interrupt generated by a hardware clock.
Conforming ISP	An AMX Interrupt Service Procedure consisting of an ISP root which calls an Interrupt Handler which has the right to make calls to a subset of the KwikNet service procedures.
DHCP	Dynamic Host Configuration Protocol: a protocol used by a DHCP client to derive a network IP address during the startup (boot) initialization of an IP stack. The client uses services provided by a DHCP server elsewhere on the network.
DNS	Domain Name System: the database distributed across all interconnected networks used to map each text-like machine name to its equivalent IP address.
Error Code	A series of signed integers used by KwikNet to indicate error or warning conditions detected by KwikNet service procedures.
Exit Procedure	An AMX or application procedure executed by AMX during the exit phase when an AMX system is shut down.
Fatal Error	A condition detected by KwikNet which is considered so abnormal that to proceed might risk catastrophic consequences.
FIFO	First in, first out. Usually used to refer to the ordering of elements in a queue or linked list.
FTP	File Transfer Protocol: a TCP/IP protocol used for reliably transferring files between two points on a network.

Handle	An identifier assigned by AMX or KwikNet for use by your application to reference a private AMX or KwikNet data item.
ICMP	Internet Control Message Protocol: a component of the IP protocol which handles error and control messages.
Interrupt Handler	An application procedure called from an ISR or ISP root to service an interrupting device.
Interrupt Service Procedure (ISP)	A procedure (in an AMX application) which is executed in response to an external device interrupt request.
Interrupt Service Routine (ISR)	The procedure in any application which is executed in response to an external device interrupt request. In an AMX application, such a procedure is called an ISP.
IP	Internet Protocol: the protocol that defines the delivery of IP datagrams across an internet in a connectionless, best effort fashion.
IP address	A 32-bit number (address) used to identify the interconnection of a host computer to a physical network. IP addresses are easily recognized when expressed using dotted decimal notation in which IP address <code>0x7F000005</code> is written as "127.0.0.5".
ISP	See Interrupt Service Procedure
ISP root	The ISP code fragment (produced by the AMX Configuration Generator) which informs AMX that an interrupt has occurred and calls an application Interrupt Handler.
ISR	See Interrupt Service Routine
KwikNet Task	The private KwikNet procedure which is responsible for all timing control and event sequencing in a KwikNet application.
Library Configuration Module	A C header file produced during the KwikNet library construction process and used to compile all KwikNet source modules.
Library Parameter File	A text file which can be edited by the KwikNet Configuration Builder to describe a particular KwikNet library configuration.

MAC	Media Access Control: a general term used to define the method by which access to a physical network is controlled. This term is sometimes used when referencing Ethernet cards since Ethernet is a very common MAC protocol.
Memory Block	A portion of a memory pool that has been allocated for use by one or more tasks.
Memory Pool	A collection of memory sections whose use is controlled by the AMX Memory Manager.
Memory Pool Id	The handle assigned to a memory pool by AMX for use as a unique memory pool identifier.
Memory Section	A contiguous region of memory assigned to the AMX Memory Manager for allocation to application tasks.
MIB	Management Information Base: the set of variables which constitute a database maintained by a network host which supports SNMP.
Multi-homed	A host computer with interconnections to multiple physical networks is said to be multi-homed.
Multitasking	A method of program execution in which an operating system makes it appear as though several procedures (called tasks) are running concurrently.
Network Configuration Module	A C source file, produced by the KwikNet Configuration Builder, which defines the network and device driver characteristics of a particular KwikNet application.
Network Parameter File	A text file which can be edited by the KwikNet Configuration Builder to describe the networks and device drivers required for a particular KwikNet application.
Network Tick	A multiple of the system tick from which the fundamental KwikNet unit of time is derived. All KwikNet time intervals in the system are measured in multiples of the network tick.
OS	A short form for the two words operating system. When the term OS is used alone, no assumption about the operational characteristics of the OS can be made.
PING	Packet InterNet Groper: the name given to the process of sending an ICMP echo request in order to learn if a destination is reachable.
PPP	Point to Point Protocol: a network protocol used to control the delivery of IP datagrams between two hosts interconnected by a serial link.

RAM	Alterable memory used for data storage and stacks.
Restart Procedure	An AMX or application procedure executed by AMX during the initialization phase when an AMX system is started.
ROM	Read only memory of all types including PROMs, EPROMs and EAROMs.
RTOS	A short form for referencing a real-time operating system, usually one with multitasking capability and reasonably good response to rapidly occurring external events.
RT/OS	The KwikNet syntax used to specify a general purpose operating system. The term RT/OS is used to encompass both multitasking and single threaded operating systems when the distinction is irrelevant.
Semaphore	A data structure which can be used by an RTOS to provide an event signaling mechanism or mutually exclusive access by tasks to specific resources.
Single threaded	A method of program execution in which all procedures execute sequentially. This method of operation is sometimes called single tasking. Although interrupts can cause a brief digression in the program's sequential operation, execution after an interrupt has been serviced always resumes in the procedure which was preempted by the interrupt.
SLIP	Serial Line Internet Protocol: a simple network protocol used to control the delivery of IP datagrams between two hosts interconnected by a serial link.
SMTP	Simple Mail Transfer Protocol: a TCP/IP protocol used to transfer mail messages from one machine to another.
SNMP	Simple Network Management Protocol: a protocol used to monitor and manage the operation of a host computer and the networks to which it is attached. Operations depend on the ability to access and modify variables in the host's Management Information Base (MIB).

System Configuration Module

A software module, produced by the AMX Configuration Builder, which defines the characteristics of a particular AMX application.

System Tick

A multiple of the hardware clock tick from which the fundamental unit of time for an RT/OS is derived. All time intervals in the system are measured in multiples of the system tick.

Tag

A 4-character name that can be assigned to any AMX system data structure when it is created. A tag can be used to find the identifier of a task, timer, semaphore, event group, mailbox, message exchange, memory pool or buffer pool with a particular name.

Tags are also used to identify KwikNet networks and their associated device drivers.

Tailoring file

A special make file included by the make file which is used to build a library or application. The tailoring file provides macro definitions and implicit rules which specify how the make utility can use a specific set of software development tools (compiler, assembler, librarian, linker/locator).

Target Configuration Module

A software module, produced by the AMX Configuration Builder, which defines the characteristics of your target hardware as used in a particular AMX application.

Task

An application procedure which is executed by an RTOS in a way which makes it look as though all such procedures are executing at once.

Task Id

The handle assigned to a task by KwikNet for use as a unique task identifier.

Task Priority

The priority at which a task executes.

TCP

Transport Control Protocol: the protocol used to provide reliable, full-duplex delivery of data streams across a logical connection established between two end points.

Timer

A facility provided by AMX to permit precise interval measurement in AMX applications.

Timer Id

The handle assigned to a timer by AMX for use as a unique timer identifier.

Timer Procedure

An application procedure which is executed by AMX whenever the corresponding timer interval expires.

UDP

User Datagram Protocol: a protocol which permits applications to send and receive datagrams using only the underlying IP network services. UDP datagrams use a port number, in addition to the IP address, to identify the source and destination of each datagram.

This page left blank intentionally.

B. KwikNet Error Codes

TCP/IP Socket Error Codes

TCP/IP socket error codes are signed integers. An error code of *-1* indicates that a socket error has occurred. Codes greater than zero describe the reason for the error. To assist you during testing, the hexadecimal value of the least significant 16-bits of the error code is listed as it might appear in a register or memory dump.

Mnemonic	Value (dec)	Value (hex)	Meaning
	0	0x0000	Socket call successful
<i>KN_SOCKERR</i>	-1	0xFFFF	Socket call failed (Use <i>kn_errno()</i> to fetch reason)
<i>ENOBUFS</i>	1	0x0001	No memory buffers are available
<i>ETIMEDOUT</i>	2	0x0002	Operation timed out
<i>EISCONN</i>	3	0x0003	The socket is already connected
<i>EOPNOTSUPP</i>	4	0x0004	Operation not supported
<i>ECONNABORTED</i>	5	0x0005	The connection was aborted
<i>EWOULDBLOCK</i>	6	0x0006	Caller would block
<i>ECONNREFUSED</i>	7	0x0007	The connection was refused
<i>ECONNRESET</i>	8	0x0008	The connection has been reset
<i>ENOTCONN</i>	9	0x0009	The socket is not connected
<i>EALREADY</i>	10	0x000A	Operation is already in progress
<i>EINVAL</i>	11	0x000B	Invalid parameter
<i>EMSGSIZE</i>	12	0x000C	Invalid message size
<i>EPIPE</i>	13	0x000D	Cannot send any more
<i>EDESTADDRREQ</i>	14	0x000E	Destination address is missing
<i>ESHUTDOWN</i>	15	0x000F	Connection has been shut down
<i>ENOPROTOOPT</i>	16	0x0010	The option is unknown for this protocol
<i>EHAVEOOB</i>	17	0x0011	Have received out-of-band data
<i>ENOMEM</i>	18	0x0012	No memory available
<i>EADDRNOTAVAIL</i>	19	0x0013	The specified address is not available
<i>EADDRINUSE</i>	20	0x0014	The specified address is already in use
<i>EAFNOSUPPORT</i>	21	0x0015	Address family is not supported
<i>EINPROGRESS</i>	22	0x0016	Operation is in progress
<i>ELOWER</i>	23	0x0017	Unused
<i>EBADF</i>	24	0x0018	The socket descriptor <i>s</i> is invalid (Use <i>KN_EBADF</i> if <i>EBADF</i> conflicts with C)

KwikNet Error Codes

KwikNet error codes are signed integers. Codes less than zero are error codes. Codes greater than zero are warning codes. To assist you during testing, the hexadecimal value of the least significant 16-bits of the error code is listed as it might appear in a register or memory dump.

Mnemonic	Value (dec)	Value (hex)	Meaning
<i>KN_EROK</i>	0	0	Call successful
Warnings			
<i>KN_WRPENDING</i>	1	0x0001	Packet queued for send (waiting for an ARP reply)
<i>KN_WRREJECT</i>	2	0x0002	Received packet not of interest
Device Driver errors			
<i>KN_DER_BADID</i>	-1	0xFFFF	Invalid device id
<i>KN_DER_BADPARAM</i>	-2	0xFFFE	Invalid parameter
<i>KN_DER_BADMODE</i>	-3	0xFFFFD	Invalid in current operating mode
<i>KN_DER_NOSUPPORT</i>	-4	0xFFFFC	Ioctl command is not supported
<i>KN_DER_DEVICE</i>	-5	0xFFFFB	Device specific error occurred
<i>KN_DER_NOTAVAIL</i>	-6	0xFFFFA	Requested data not yet available
Programming errors			
<i>KN_ERPARAM</i>	-10	0xFFFF6	Invalid parameter
<i>KN_ERLOGIC</i>	-11	0xFFFF5	Unexpected sequence of events
System errors			
<i>KN_ERNOMEM</i>	-20	0xFFFE4	Memory not available for allocation
<i>KN_ERNOBUFFER</i>	-21	0xFFFE3	Packet buffer not available
<i>KN_ERQUEUE</i>	-22	0xFFFE2	Queuing resource not available
<i>KN_ERTCPSTATE</i>	-23	0xFFFE1	TCP layer state transition error
<i>KN_ERTIMEOUT</i>	-24	0xFFFE0	TCP layer timeout
<i>KN_ERNOFILE</i>	-25	0xFFFE7	Expected file was missing
<i>KN_ERFILEIO</i>	-26	0xFFFE6	File I/O error
Net errors			
<i>KN_ERSEND</i>	-30	0xFFFE2	Send to net failed at low layer
<i>KN_ERNOARP</i>	-31	0xFFFE1	No ARP response for a given host
<i>KN_ERHEADER</i>	-32	0xFFFE0	Bad upper layer header
<i>KN_ERROUTE</i>	-33	0xFFDF	Cannot find a reasonable next IP hop
<i>KN_ERIFACE</i>	-34	0xFFDE	Cannot find a device interface
<i>KN_ERHARDWARE</i>	-35	0xFFDD	Hardware failure

KwikNet Error Codes (continued)

Mnemonic	Value (dec)	Value (hex)	Meaning
FTP errors			
<i>KN_ERFTPD</i>	-200	0xFF38	Invalid FTP descriptor
<i>KN_ERFTP CMD</i>	-201	0xFF37	FTP command rejected by server
<i>KN_ERFTP SRC</i>	-202	0xFF36	Source file error
<i>KN_ERFTP DST</i>	-203	0xFF35	Destination file error
<i>KN_ERFTP XFER</i>	-204	0xFF34	Transfer failed
<i>KN_ERFTP DIR</i>	-205	0xFF33	Directory error
<i>KN_ERFTP LOGIN</i>	-206	0xFF32	FTP login failed
<i>KN_ERFTP SOCKC</i>	-207	0xFF31	FTP command socket fault
<i>KN_ERFTP SOCKD</i>	-208	0xFF30	FTP data socket fault
<i>KN_ERFTP TERM</i>	-209	0xFF2F	FTP server terminated the session
<i>KN_ERFTP NOUSER</i>	-210	0xFF2E	No user logged in yet
HTTP errors			
<i>KN_ERWEBS D</i>	-220	0xFF24	Invalid web server descriptor
<i>KN_ERWEB B D</i>	-221	0xFF23	Invalid web browser descriptor
<i>KN_ERWEB CGI P</i>	-222	0xFF22	Invalid CGI parameter string
<i>KN_ERWEB NOOP</i>	-223	0xFF21	Operation not allowed
Telnet errors			
<i>KN_ERTELND</i>	-240	0xFF10	Invalid Telnet descriptor
<i>KN_ERTEL SOCK</i>	-241	0xFF0F	Telnet socket fault
<i>KN_ERTEL NOCONN</i>	-242	0xFF0E	No connection with Telnet peer
<i>KN_ERTEL NOCBF</i>	-243	0xFF0D	No callback function provided
<i>KN_ERTEL REJECT</i>	-244	0xFF0C	Invalid operation for Telnet entity
<i>KN_ERTEL INCBF</i>	-245	0xFF0B	Invalid request by callback function
<i>KN_ERTEL NOSESS</i>	-246	0xFF0A	Operation requires a Telnet session
<i>KN_ERTEL INSESS</i>	-247	0xFF09	Operation invalid in Telnet session
<i>KN_ERTEL NOSPC</i>	-248	0xFF08	No buffer space available
<i>KN_ERTEL BUSY</i>	-249	0xFF07	Busy: command send in progress
<i>KN_ERTEL NOLOG</i>	-250	0xFF06	Logging not enabled
<i>KN_ERTEL ACTIVE</i>	-251	0xFF05	Server has active sessions
<i>KN_ERTEL MAXNC</i>	-252	0xFF04	Server has max number of connections

KwikNet Fatal Error Codes

Mnemonic	Value (dec)	Value (hex)	Meaning
<i>KN_FERNOTASK</i>	100	0x0064	Cannot find KwikNet Task
<i>KN_FERNOTMR</i>	101	0x0065	Cannot create network timer
	102	0x0066	reserved
	103	0x0067	reserved
<i>KN_FERFN</i>	104	0x0068	Cannot send fn msg to KwikNet Task
<i>KN_FEREVWAIT</i>	105	0x0069	Cannot send wait msg to KwikNet Task
<i>KN_FEREVSIG</i>	106	0x006A	Cannot send event msg to KwikNet Task
<i>KN_FERSUSPEND</i>	107	0x006B	Cannot suspend a task to wait for event
<i>KN_FERRESUME</i>	108	0x006C	Cannot resume a task after event occurs
<i>KN_FERTIMER</i>	109	0x006D	Cannot start/stop a KwikNet timer
<i>KN_FERNOSEM4</i>	110	0x006E	No semaphores available for use
<i>KN_FERNRESID</i>	111	0x006F	Invalid network resource identifier
<i>KN_FERLOCK</i>	112	0x0070	Resource lock failed
<i>KN_FERUNLOCK</i>	113	0x0071	Resource unlock failed
<i>KN_FERNOMEM</i>	114	0x0072	No memory for allocation
<i>KN_FERBADMEM</i>	115	0x0073	Free memory that was never allocated
<i>KN_FEREXIT</i>	116	0x0074	Cannot send exit to KwikNet Task
<i>KN_FERSTART</i>	117	0x0075	Cannot start KwikNet Stack operation
<i>KN_FERNOSLEEP</i>	118	0x0076	KwikNet Task cannot sleep
<i>KN_FERPORT</i>	119	0x0077	Custom port panic
<i>KN_FERPANIC</i>	120	0x0078	KwikNet TCP/IP Stack panic

C. KwikNet Universal File System Interface

C.1 Introduction

The KwikNet TCP/IP Stack does not require a file system for normal use. However, several of the optional KwikNet components, such as the FTP client and server and the HTTP Web Server, do require file services. For these options, KwikNet provides its own Universal File System (UFS) interface to the file services provided by the file system operating on the host computer.

The Universal File System interface provides access to one of four file systems:

- AMX/FS File System for use with the AMX Multitasking Kernel
- Standard C using the MS-DOS[®] file system
- Standard C using a UNIX-like file system
- Custom user defined file system

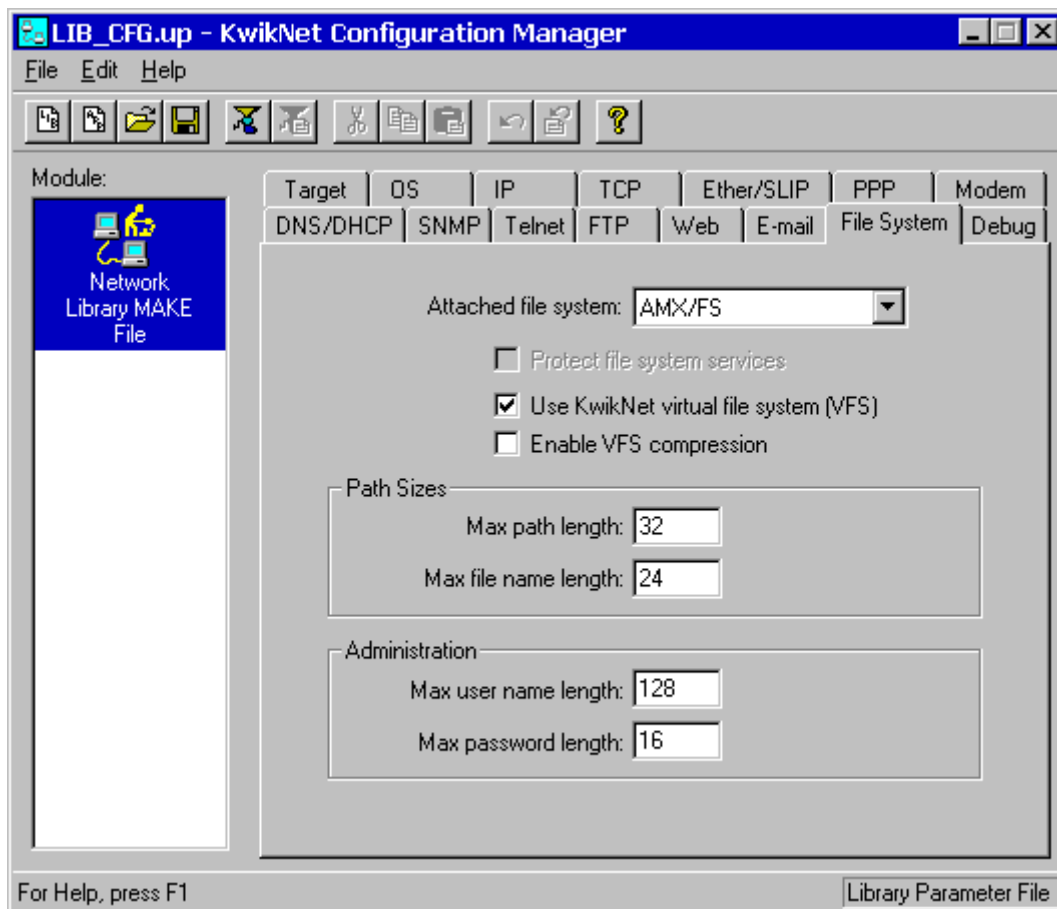
When used with KADAK's AMX Multitasking Kernel, KwikNet supports only three of the four file systems. AMX/FS can be used with AMX and KwikNet on all of the supported target processors. Access to the MS-DOS file system is only available when using AMX 86 or AMX 386/EP on a PC compatible platform. A custom file system can also be adapted for use with AMX and KwikNet.

The Universal File System interface for a UNIX-like file system is provided with an alternate KwikNet product which is suitable for porting to other operating environments.

There should be no need to become familiar with the internal operation of the Universal File System unless you must adapt it for use with your own custom file system. This customization procedure is described in Appendix C.5.

C.2 KwikNet File System Parameters

The KwikNet Universal File System (UFS) interface forms part of the KwikNet IP Library. To include the UFS interface in the library, you must use the KwikNet Configuration Builder to edit your KwikNet Library Parameter File. The Universal File System parameters are edited using the File System property page. The layout of the window is shown below.



Attached File System

From the pull down list, choose the type of underlying file system to which the Universal File System must connect. If you are not using any of the optional KwikNet components which require a file system, select option None.

If you are using a KwikNet option such as FTP or HTTP which does require a file system, choose the file system you will be using with that KwikNet option.

Universal File System Parameters (continued)

Protect File System Services

When operating in a multitasking environment, the file system services must be thread-safe. If the file system you have chosen to use is safe, leave this box unchecked. Otherwise, check this box and KwikNet will use its file system locking mechanism to protect access to the unsafe file system services. In a single threaded environment, file system services are inherently thread-safe. Hence, leave this box unchecked.

Use KwikNet Virtual File System

For embedded systems which do not include a file system, KwikNet offers a very simple Virtual File System (VFS) which can provide access to a limited set of read-only files built into the application. The Virtual File System can be used with or without a file system. It will forward file requests which it is not equipped to handle, to the underlying file system, if one exists.

Check this box if you intend to use the KwikNet Virtual File System. Otherwise, leave this box unchecked.

Enable VFS Compression

Using the KwikNet VFS Generator, you can create compressed HTML files for use with the KwikNet HTTP Web Server. These files must be decompressed by the KwikNet Virtual File System prior to use by the HTTP Web Server. Check this box if any of the read-only files which you have built into your application have been compressed in this manner. Otherwise, leave this box unchecked.

Maximum Path and File Name Lengths

Specify the maximum number of characters which can appear in a path name string used to reference a directory location. The path length must include room for a terminating '\0' character. The path length excludes any file name.

Specify the maximum number of characters which can appear in a file name string used to identify any file. The file name includes the base name and extension(s) and any separating characters. The file name length must include room for a terminating '\0' character. The file name length excludes any path information.

There is no reason to set the path or file name lengths any greater than the maximum allowed by the selected file system. For the KwikNet Virtual File System, a path length of 32 and a file name length of 16 are reasonable. For AMX/FS and MS-DOS file systems, a path length of 128 and a file name length of 16 will be adequate.

To minimize memory waste in embedded applications in which short paths are the norm, the maximum path length can be reduced. However, be aware that KwikNet FTP and HTTP servers may be forced to reject requests for service if the paths and file names which they receive exceed the limits defined by these configuration parameters.

Administration Parameters (see Appendix D)

C.3 Using the AMX/FS File System

The KwikNet Universal File System (UFS) interface supports the AMX/FS File System. No customization is required. Any file devices, including RAM drives, hard drives, floppy drives and custom devices, which have been attached to the AMX/FS File System are accessible through the UFS interface.

To use the AMX/FS File System with KwikNet, you must first install AMX and AMX/FS and test them in your intended target environment. The AMX/FS Sample Program offers a good starting point. Once you have it operating with your target hardware, you are ready to merge it with KwikNet.

AMX System Configuration

The first step is to merge the AMX configuration information required for both AMX/FS and KwikNet into a new AMX configuration suitable for use with both products. You may have to increase the maximum number of tasks, timers and semaphores to meet the expanding requirements. Adjust the parameters in your AMX User Parameter File accordingly.

If you are using options such as the KwikNet FTP client or server or the KwikNet HTTP Web Server, be sure to account for the total number of client and server tasks which you intend to employ.

Since each server or client task must have file access, you may have to increase the maximum number of tasks which are permitted to concurrently use AMX/FS. You may also have to increase the maximum number of files which can be concurrently in use. Adjust the parameters in your AMX User Parameter File accordingly.

Your AMX configuration must include the device drivers for both the KwikNet network interfaces and the AMX/FS file devices. You may have to adjust your choices of interrupt assignments to prevent conflicts among these devices.

Once you have an integrated AMX configuration, try using it to confirm that AMX/FS works well in the presence of KwikNet but without KwikNet actually in use. Then use the configuration to test that KwikNet will operate on your network without any file operations. Then you are ready to try KwikNet and AMX/FS together.

AMX System Startup

Special care must be taken when launching an AMX system which includes both KwikNet and the AMX/FS File System. Initialize AMX/FS before starting KwikNet. This implies that execution of AMX/FS Restart Procedure *fj_restart()* must precede the call to KwikNet function *kn_enter()*.

AMX/FS requires that a logical drive be mounted before it can be accessed. This operation is not supported by the Universal File System interface. Hence, before starting any KwikNet clients or servers which require file support, your application must call AMX/FS procedure *fjdrvopen()* to mount each of the logical drives which these KwikNet components are permitted to access.

There is no strict rule governing when logical drives should be mounted. The simplest solution is to have an application task unconditionally mount all available logical drives and then start KwikNet. In this way, all KwikNet components which require file support will have access to all logical drives whenever required.

Alternate solutions may better meet your needs. For example, suppose you intend to have one FTP server task which only requires access to logical drive *D:*. Your FTP server task can mount drive *D:* and then call KwikNet FTP procedure *knfs_start()* to begin operating as an FTP server. Note that other tasks are not precluded from accessing drive *D:*. The FTP server task is also not prevented from accessing another logical drive mounted by some other task.

AMX System Shutdown

When shutting down your AMX application, you must stop KwikNet before terminating AMX/FS. Hence KwikNet function *kn_exit()* must reach completion prior to the execution of AMX/FS Exit Procedure *fj_exit()*.

End of Line Indication

The use of CR ('*\r*', ASCII *0x0D*), LF ('*\n*', ASCII *0x0A*) or CRLF (CR followed by LF) as an end of line indicator in text files depends on the interpretation (translation) of strings by file streaming functions such as *fread()* and *fwrite()*.

Although the AMX/FS File System is MS-DOS[®] file format compatible, it does not provide a streaming level API. For example, the description of *fjopen()* states that all files are read and written in binary mode only. Hence AMX/FS does no translation of the data, even if the file is opened in text mode.

Unfortunately, protocols such as FTP demand that CRLF be used as the end of line indicator for files sent in text mode. To avoid the overhead of data translation in the FTP client and server, KwikNet assumes that the underlying file system stores text files with CRLF as the end of line indicator. Such files can then be read and written as binary files.

Since MS-DOS stores text files with CRLF terminators, the KwikNet interface is compatible with the most prevalent file system with which it must operate. Furthermore, since AMX/FS has no streaming support, CRLF has been adopted as the end of line indicator to be used by applications when recording text files with AMX/FS.

C.4 Using the MS-DOS File System

When used with AMX 86 or AMX 386/EP, the KwikNet Universal File System (UFS) interface supports the MS-DOS[®] file system. No customization is required. All file devices, including RAM drives, hard drives and floppy drives, are accessible through the UFS interface.

Both AMX 86 and AMX 386/EP include a component called the PC Supervisor which permits these versions of AMX to be used with MS-DOS on PC platforms. Special care must be taken when using the PC Supervisor with AMX and KwikNet as described in Chapter 3.7.1.

The PC Supervisor Task must be of lower priority than tasks which use it to access MS-DOS. This requirement leads to the following recommended task priority order when using KwikNet components such as an FTP client or server or the HTTP Web Server.

PC Supervisor Clock Tick Task	Highest priority
PC Supervisor Keyboard Task	
KwikNet Task	
FTP server task	
HTTP Web Server task	
FTP client task	
PC Supervisor Task	Lowest priority

Note that the order of priority of servers and clients may have to be adjusted to reflect the relative importance of each of these services in your application.

C.5 Using a Custom File System

The KwikNet Universal File System (UFS) interface can be adapted to support a custom file system. To do so, you need only edit file *KNFSUSER.H* to meet the requirements of your custom file system. No other customization is required.

Once file *KNFSUSER.H* is ready, simply edit the File System parameters in your KwikNet Library Parameter File to reference your custom file system. Then build your KwikNet Libraries and link them with your application. The KwikNet Universal File System interface will then use your custom file system for all file operations.

File *KNFSUSER.H* serves two purposes. As its name implies, it is a header file which maps all KwikNet file access functions to those in your file system. However, it is also a code generating module which can provide a custom version of any file access function which is not available in your file system. The code generated by this module will actually reside in KwikNet module *KN_FILES.C* which is located in the KwikNet *INET* installation directory.

The following minimal file system services must be provided by your file system.

- Open a file for read, write or append in text or binary mode
- Close a file
- Write *n* elements of size *m* to a file
- Read *n* elements of size *m* from a file
- Seek within a file
- Tell location of file pointer within a file
- Remove (delete) a file

The following file system services, if provided by your file system, will permit the KwikNet FTP server to offer directory services.

- Make a directory
- Remove (delete) a directory
- Verify that a path name references a directory
- Generate a directory listing into a file

The following file system services, if provided by your file system, will permit the KwikNet HTTP Web Server to use your file system.

- Get a character from a file
- Get a string from a file

This page left blank intentionally.

D. KwikNet Administration Interface

D.1 Introduction

Many network protocols, such as FTP, were originally developed when large mainframe computers were shared by many users. The computers accommodated multiple users, each with a password and all administered by a higher authority. Network protocols were developed to support user names and passwords.

Unfortunately, user name and password administration services are rarely provided by the operating systems found in desktop computers or embedded devices. This is true even of KADAK's AMX Multitasking Kernel.

The KwikNet TCP/IP Stack does not require a user administration system for normal use. However, to accommodate protocols such as FTP, KwikNet provides its own user name and password administrative services.

User Definitions

All KwikNet users are defined in array *kn_users[]* in module *KN_ADMIN.C* located in the KwikNet *INET* installation directory. Each user definition is a *knx_userinfo* structure which is defined in header file *KN_ADMIN.H*.

```
struct knx_userinfo {                                /* User info structure */
    int    xu_access;                                /* User access rights   */
    int    xu_rsv1;                                  /* Fill for alignment   */
    void    *xu_app;                                  /* Reserved for application */
    char    xu_username[KN_FS_LUSER];                /* User name           */
    char    xu_password[KN_FS_LPASS];                /* User password       */
    char    xu_basedir[KN_FS_LPATH+4];              /* User base directory  */
};
```

To add, modify or delete users, you must edit file *KN_ADMIN.C*. Each definition includes a user name, an unencrypted password, a base directory path and a definition of the user's access rights. User names and passwords are character arrays. The base directory defines the path to a file directory considered to be the user's base (home) directory if a file system is employed. A pointer variable in the definition is reserved for the private use of your application.

User Access Rights

User access rights are formed from the logical OR of the following bit masks which are defined in header file *KN_ADMIN.H*.

<i>KN_ADM_ACC_READ</i>	Allow file read
<i>KN_ADM_ACC_WRITE</i>	Allow file write
<i>KN_ADM_ACC_REMOVE</i>	Allow file remove (delete)
<i>KN_ADM_ACC_DIRSEL</i>	Allow directory traversal (selection)
<i>KN_ADM_ACC_DIRLIST</i>	Allow directory listing
<i>KN_ADM_ACC_DIRMK</i>	Allow directory make
<i>KN_ADM_ACC_DIRRM</i>	Allow directory remove (delete)
<i>KN_ADM_ACC_VISIBLE</i>	Allow file visibility
 <i>KN_ADM_ACC_FULL</i>	 Allow full access (all of the above)

These access rights are used by optional KwikNet components, such as the FTP server, to restrict a user's access to directories and files. Access right *KN_ADM_ACC_DIRSEL* is required to be able to change directories. Most of the other access rights are self explanatory.

The visibility right is special. If a user has access right *KN_ADM_ACC_VISIBLE*, then the user will have full view of all files and directories. Without this access right, the user will not be able to view files or traverse directories which are above the user's base directory.

Customizing Administration Services

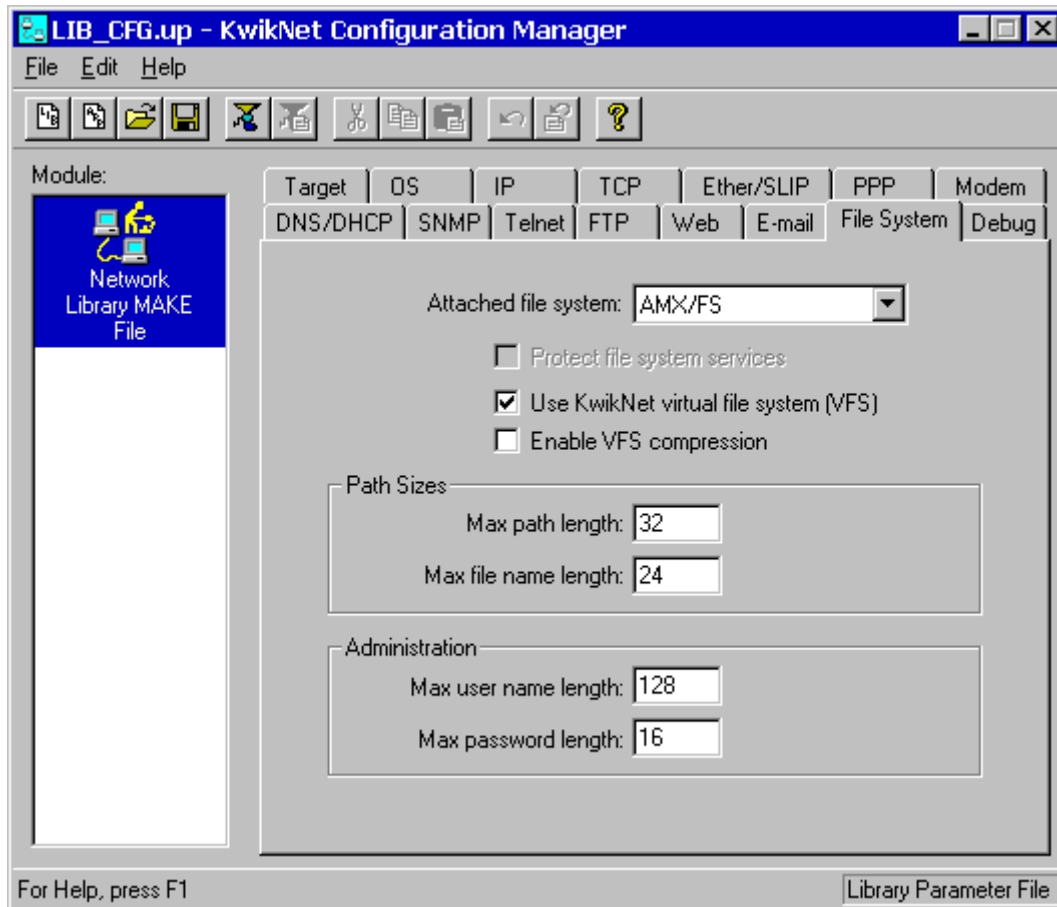
File *KN_ADMIN.C* defines two users. User *anonymous* with password *guest* has read only access to files and directories beginning at a base directory determined by the KwikNet server making use of the user definitions. User *KADAK* with password *KwikNet* has full access to all files from the root directory. You are free to alter these definitions to suit your needs.

File *KN_ADMIN.C* also includes a set of functions which are used by servers, such as the KwikNet FTP server, to validate user names, passwords and access rights. Other functions in the module validate directory path information if a file system is used. Strings of the form ">>>>" identify functions in the file which may require modification. You can alter these validation functions, if necessary, to meet the needs of your application.

After editing the file *KN_ADMIN.C*, you must build your KwikNet libraries to incorporate the revised module. Before building the KwikNet libraries, be sure to adjust the maximum user name and password lengths (see Appendix D.2) to meet or exceed the lengths of the user names and passwords in your user definitions.

D.2 KwikNet Administration Parameters

The KwikNet administration interface forms part of the KwikNet IP Library. To adapt the administration interface parameters for your use, you must use the KwikNet Configuration Builder to edit your KwikNet Library Parameter File. The administration parameters are edited using the File System property page. The layout of the window is shown below.



Administration Parameters (continued)

Maximum User Name and Password Lengths

Specify the maximum number of characters which can appear in a user name string. The user name length must include room for a terminating ' \0 ' character.

Specify the maximum number of characters which can appear in a user password string. The password length must include room for a terminating ' \0 ' character.

For most applications, a user name length of 32 and a password length of 24 will be adequate.

To minimize memory waste in embedded applications in which short user names and passwords might be expected, the maximum lengths can be reduced. However, be aware that the KwikNet FTP server may be forced to reject requests for service if the user names and passwords which it receives exceed the limits defined by these configuration parameters.

File System Parameters (see Appendix C)

E. KwikNet Sample Program Architecture

The manner in which the KwikNet TCP/IP Sample Program starts and operates is completely dependent upon the underlying operating system with which KwikNet is being used. Operation can be either multitasking or single threaded. All sample programs provided with KwikNet and its optional components share the common implementation methodology about to be described.

All KwikNet sample programs are built upon a common framework. The building blocks are a set of files located in toolset directory *TOOLXXX\SAM_COMN* where *XXX* is KADAK's mnemonic for the particular set of software development tools which you are using. The common files and the procedures which they contain are listed in Figure E-1.

A quick review of the common sample program files will indicate that most of the implementation is devoted to the man/machine interface and to the startup process. These two topics always seem to account for the bulk of any networking example, no matter how simple the actual network operations may be.

Console Interface

The KwikNet data logging service, message recording service and console driver have been described in detail in Chapters 1.6 to 1.7 of the KwikNet TCP/IP Stack User's Guide. If you review that material again, you will note that the procedures referenced in the description are all present in the common sample program modules listed in Figure E-1.

The sample programs use KwikNet procedure *kn_dprintf()* to record messages using the KwikNet data logging service. KwikNet passes each such message to the log function *sam_record()* in the Application OS Interface module *KNSAMOS.C*. This common log function is specified on the Application property page of the Network Parameter File provided with each KwikNet sample program.

The sample programs use console driver procedures *knconin()* and *knconins()* for console input and *knconout()* and *knconouts()* for console output.

The sample programs also use a command line parsing service provided by console driver procedure *kncon_parse()*. This procedure parses a command string into its various tokens according to directions provided by the caller.

The sample programs use a common error message generation service provided by console driver procedure *kncon_error()*. This procedure generates an error message on the console device. The error message is derived from a KwikNet error code using a message list provided by the sample program.

Many of the interactive KwikNet sample programs implement a dump command to display recorded messages. These applications call console driver procedure *kncon_logdump()* to extract all of the message strings from the recording array using recording procedure *kn_loggets()*. The extracted messages are displayed on the console device. Once all recorded strings have been displayed, the recording list is reset with a call to recording procedure *kn_logini()*.

<i>KNSAMOS.C</i>	Application OS Interface	
	<i>main()</i>	Sample program main entry point
	<i>sam_osshutdown()</i>	OS shutdown on exit
	<i>sam_ostkprep()</i>	Task create/prepare
	<i>sam_ostkstart()</i>	Task start
	<i>sam_record()</i>	Log function used by <i>kn_dprintf()</i>
	<i>print_task()</i>	Print task
	<i>backg_task()</i>	Background task (AMX only)
	<i>rrproc()</i>	Restart Procedure (AMX only)
	<i>exproc()</i>	Exit Procedure (AMX only)
<i>KNRECORD.C</i>	Message recording services	
	<i>kn_loginit()</i>	Initialize data recording services
	<i>kn_logputc()</i>	Record one character
	<i>kn_logmsg()</i>	Record a message
	<i>kn_loggets()</i>	Get one recorded message from the recorded list
<i>KNCONSOL.C</i>	Console driver	
	<i>knconinit()</i>	Initialize console device for use
	<i>knconexit()</i>	Close console device upon exit
	<i>knconin()</i>	Get a character from the console device
	<i>knconins()</i>	Get a string from the console device
	<i>knconout()</i>	Send a character to the console device
	<i>knconouts()</i>	Send a string to the console device
	<i>kncon_logprep()</i>	Prepare to use console for data logging
	<i>kncon_logputc()</i>	Log a character to the console device
	<i>kncon_logdump()</i>	Log all recorded strings on the console device
	<i>kncon_parse()</i>	Parse a console command line
	<i>kncon_error()</i>	Generate an error message on the console device
<i>KN8250S.C</i>	INS8250 (NS16550) UART driver	
	<i>kn_iouart()</i>	All serial I/O device operations

Figure E-1 KwikNet Sample Program Procedures

KwikNet Sample Program Operation with AMX

When KwikNet is used with AMX, the KwikNet sample programs operate as follows. Once your board level initialization is complete and the C startup code has been executed, the sample program begins execution at *main()* in the Application OS Interface module *KNSAMOS.C*.

The **main program** makes a series of calls to initialize the various components which make up the sample program. Your AMX board support function *chbrdinit()* is called to set up the hardware environment for AMX use. KwikNet board driver procedure *kn_brdrreset()* is called to initialize its interrupt support for all KwikNet device drivers.

The KwikNet message recording interface is initialized with a call to *kn_loginit()*. If the console driver has been configured for use as the recording device, procedure *kn_loginit()* calls console driver procedure *kncon_logprep()* to prepare it accordingly.

Next, the *main()* procedure calls *kn_osprep()* in the KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet IP Library) to initialize the RTOS interface. Since this procedure initializes the KwikNet data logging service, KwikNet procedure *kn_dprintf()* can be used by the sample program even before KwikNet is started.

Finally, the *main()* procedure launches AMX to start the multitasking sample program.

Once AMX is ready, it calls the KwikNet **Restart Procedure** *kn_osready()* in the KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet IP Library) to initialize the AMX resources required by KwikNet and to prepare the memory allocation subsystem for use by KwikNet and your application.

AMX then calls the application **Restart Procedure** *rrproc()* in the Application OS Interface module *KNSAMOS.C* to start the KwikNet sample program as an AMX application. The AMX clock driver is initialized with a call to procedure *chclockinit()*. Task services in the Application OS Interface module are then used to create and start a low priority background task (procedure *backg_task()*) which provides a simulated software clock in case a real hardware clock is unavailable and an AMX clock driver has not been linked with the sample program.

Next, Restart Procedure *rrproc()* starts the sample program's print task, a task used by some sample programs to log messages on the console device. Finally, procedure *app_prep()* in the sample program module is called to prepare all application level components needed by the sample program.

Every KwikNet sample program provides function *app_prep()* as its advance preparation entry point. This procedure creates and starts one or more application tasks which collectively make up the sample program. One of these tasks, usually called the client task, is the task in charge of the sequence of operations performed by the sample program. For example, the client task often provides a user command line console interface which allows you to interactively control sample program activities.

The **sample program begins** operation at task level once AMX completes its startup processing. The client task executes and calls function *app_enter()*, the entry point to the main body of the sample program.

The client task starts KwikNet with a call to KwikNet procedure *kn_enter()*. KwikNet initializes all of its private resources and starts the KwikNet Task which prepares all network interfaces and their associated device drivers for use.

If the sample program requires AMX/FS file services, procedure *sam_osfsprep()* in the Application OS Interface module *KNSAMOS.C* is called to prepare the AMX/FS File System for use. If the client task provides an interactive user interface, the console driver is initialized with a call to procedure *knconinit()*. Thereafter, the client task orchestrates the sequence of network operations it is designed to illustrate.

The **termination process** is handled by the client task. If the console driver was in use, it is closed with a call to *knconexit()*. KwikNet is stopped with a call to *kn_exit()*. After a brief pause to allow KwikNet to stop, the RTOS shutdown is initiated with a call to *sam_osshutdown()* which simply requests AMX to exit in its usual orderly fashion.

AMX executes the sample program Exit Procedure *exproc()* which calls *chclockexit()* to disable the AMX clock driver. Once all Exit Procedures have been called, AMX ceases operation and returns to the *main()* function from which AMX was launched. One final call to procedure *kn_osfinish()* in the KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet IP Library) breaks the connection between KwikNet and its RTOS interface.

KwikNet Porting Kit Sample Program - Multitasking Operation

When the KwikNet Porting Kit is used with a multitasking RTOS, the KwikNet sample programs operate as follows. Once your board level initialization is complete and the C startup code has been executed, the sample program begins execution at *main()* in the Application OS Interface module *KNSAMOS.C*.

The **main program** makes a series of calls to initialize the various components which make up the sample program. Your KwikNet board driver procedure *kn_brdreset()* is called to initialize its interrupt support for all KwikNet device drivers.

The KwikNet message recording interface is initialized with a call to *kn_loginit()*. If the console driver has been configured for use as the recording device, procedure *kn_loginit()* calls console driver procedure *kncon_logprep()* to prepare it accordingly.

Next, the *main()* procedure calls *kn_osprep()* in your KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet IP Library) to initialize your RTOS interface. Since this procedure initializes the KwikNet data logging service, KwikNet procedure *kn_dprintf()* can be used by the sample program even before KwikNet is started. In many cases, procedure *kn_osprep()* will also start your KwikNet clock driver with a call to its initialization procedure *kn_uclockinit()*.

Finally, the *main()* procedure starts your RTOS to run the multitasking sample program. In the example provided with the KwikNet Porting Kit, the RTOS creates a startup task which is executed by the RTOS as it begins operation. The startup task is located at entry point *sam_osmain()* in the Application OS Interface module *KNSAMOS.C*.

Once the RTOS is ready, it executes the startup task procedure *sam_osmain()*. Task services in the Application OS Interface module are used to create and start the sample program's print task, a task used by some sample programs to log messages on the console device. Finally, procedure *app_prep()* in the sample program module is called to prepare all application level components needed by the sample program.

Every KwikNet sample program provides function *app_prep()* as its advance preparation entry point. This procedure creates and starts one or more application tasks which collectively make up the sample program. One of these tasks, usually called the client task, is the task in charge of the sequence of operations performed by the sample program. For example, the client task often provides a user command line console interface which allows you to interactively control sample program activities.

The **sample program begins** operation at task level once the high priority startup task terminates. The client task executes and calls function *app_enter()*, the entry point to the main body of the sample program.

The client task starts KwikNet with a call to KwikNet procedure *kn_enter()*. KwikNet initializes all of its private resources and starts the KwikNet Task which prepares all network interfaces and their associated device drivers for use.

If the client task provides an interactive user interface, the console driver is initialized with a call to procedure *knconinit()*. Thereafter, the client task orchestrates the sequence of network operations it is designed to illustrate.

The **termination process** is handled by the client task. If the console driver was in use, it is closed with a call to *knconexit()*. KwikNet is stopped with a call to *kn_exit()*. After a brief pause to allow KwikNet to stop, the RTOS shutdown is initiated with a call to *sam_osshutdown()* which, if possible, forces the RTOS to terminate execution in an orderly fashion.

If the RTOS ceases operation, it returns to the *main()* function from which it was launched. One final call to procedure *kn_osfinish()* in your KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet IP Library) breaks the connection between KwikNet and your RTOS. In many cases, procedure *kn_osfinish()* will also stop your KwikNet clock driver with a call to its termination procedure *kn_uclockexit()*.

KwikNet Porting Kit Sample Program - Single Threaded Operation

When the KwikNet Porting Kit is used with a single threaded operating system (OS), the KwikNet sample programs operate as follows. Once your board level initialization is complete and the C startup code has been executed, the sample program begins execution at *main()* in the Application OS Interface module *KNSAMOS.C*.

The **main program** makes a series of calls to initialize the various components which make up the sample program. Your KwikNet board driver procedure *kn_brdreset()* is called to initialize its interrupt support for all KwikNet device drivers.

The KwikNet message recording interface is initialized with a call to *kn_loginit()*. If the console driver has been configured for use as the recording device, procedure *kn_loginit()* calls console driver procedure *kncon_logprep()* to prepare it accordingly.

Next, the *main()* procedure calls *kn_osprep()* in your KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet IP Library) to initialize your OS interface. Since this procedure initializes the KwikNet data logging service, KwikNet procedure *kn_dprintf()* can be used by the sample program even before KwikNet is started. In many cases, procedure *kn_osprep()* will also start your KwikNet clock driver with a call to its initialization procedure *kn_uclockinit()*.

Finally, the *main()* procedure assumes its **App-Task role** and calls function *app_enter()*, the entry point to the main body of the sample program.

The App-Task starts KwikNet with a call to KwikNet procedure *kn_enter()*. KwikNet initializes all of its private resources and starts the KwikNet Task which prepares all network interfaces and their associated device drivers for use.

If the App-Task provides an interactive user interface, the console driver is initialized with a call to procedure *knconinit()*. Thereafter, the App-Task orchestrates the sequence of network operations it is designed to illustrate.

The **termination process** is handled by the App-Task. If the console driver was in use, it is closed with a call to *knconexit()*. KwikNet is stopped with a call to *kn_exit()*. After a brief pause to allow KwikNet to stop, the OS is forced to shut down with a call to *sam_osshutdown()*. Rarely is any termination processing performed by this function.

The App-Task procedure *app_enter()* returns to the *main()* function from which it was called. One final call to procedure *kn_osfinish()* in your KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet IP Library) breaks the connection between KwikNet and your OS. In many cases, procedure *kn_osfinish()* will also stop your KwikNet clock driver with a call to its termination procedure *kn_uclockexit()*.

This page left blank intentionally.