KwikNet®

Porting Kit

User's Guide

First Printing: July 15, 1999
Last Printing: September 15, 2002

Manual Order Number: PN713-9

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

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.

KwikNet Porting Kit User's Guide
Table of Contents

Page

KwikNet Porting Kit User's Guide
Table of Contents (continued)

KwikNet Porting Kit User's Guide
Table of Figures

This page left blank intentionally.

1. KwikNet Porting Kit 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. KwikNet is best used with a real-time operating system (RTOS) such as KADAK's AMX™ Real-Time Multitasking Kernel. However, KwikNet can be used in single threaded systems without an RTOS. Designing and implementing an application which requires a TCP/IP stack will always be easier if you start with some form of underlying operating system, even if it is of the crudest form.

This manual describes how to port KwikNet to the operating environment of your choice. You pick the target processor, the software development tools and the multitasking RTOS or single threaded OS.

Although porting KwikNet to your environment is a fairly straight-forward process, it is still not trivial. KADAK has used its extensive knowledge of target processors and software development tools and their quirks to simplify the steps which must be followed for a successful port. Since KwikNet is already available for use with KADAK's AMX kernel, you know that KwikNet has been tested on many target processors with a number of different compilers.

It is assumed that you are familiar with the architecture of the target processor and its interrupt structure. It is further assumed that you are familiar with the rudiments of microprocessor programming including the concepts of code, data and stack separation. Of course, you must also have an intimate knowledge of your multitasking RTOS or single threaded OS. Finally, it is assumed that you have a detailed knowledge of your software development tools, including C compiler, assembler (if needed), object librarian, linker/locator and program loader or debugger.

KwikNet is provided in C source format to ensure that regardless of your development environment, your ability to use and support KwikNet is uninhibited. As will be explained, the source code can easily be adapted to include code fragments programmed in the assembly language of the target processor to improve execution speed.

This manual will not tell you how the KwikNet TCP/IP Stack and its options operate or how to use KwikNet in your application. That information is provided in the KwikNet TCP/IP Stack User's Guide and in the manuals provided with each optional KwikNet component. Before starting the porting process, you should read these manuals to become familiar with KwikNet and the terminology used in this guide.

> 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.

The KwikNet Porting Interface

The KwikNet TCP/IP Stack and your application operate together as illustrated in Figure 1.1-1. The shaded blocks indicate modules which require modification to adapt KwikNet for use with your application. As you can see, very few modules require adaptation.

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 described in the KwikNet Network Configuration Module.

Your custom KwikNet Libraries and the KwikNet Network Configuration Module are derived from parameter files generated by the KwikNet Configuration Builder as described in Chapter 2 of the KwikNet TCP/IP Stack User's Guide. The actual modules are constructed using your software development tools as described in Chapter 4 of this manual.

KwikNet is connected to your RT/OS by an OS Interface Module, a C file containing procedures which provide access to the services of your particular RT/OS. This module is incorporated into the KwikNet IP Library so that it is always available for use by your application. A separate board driver connects KwikNet, its device drivers and your OS Interface Module to your target hardware in an RT/OS independent manner. You must edit these modules to meet the requirements of your particular RT/OS and target hardware.

Figure 1.1-1 also shows an application OS interface, a C module used by KADAK to provide a standard interface between your 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 edit this module to adapt it for use with your RT/OS. You will probably find that portions of the code in this module can, with very little adaptation, be used by your own application.

Finally, your RT/OS must provide a timing source. Although the RT/OS clock driver is shown as a separate component, it is often implemented as an interrupt service routine which resides in the OS Interface Module or in the application OS interface.

> Porting Tip
>
> The separation of the portable KwikNet components into the OS Interface Module, the application OS interface and the board driver will meet most porting needs. However, you are free to adapt these interfaces to meet your RT/OS needs and to accommodate the constraints imposed by your software development tools. As long as the functional requirements are met, the services can be provided in any module of your choice.

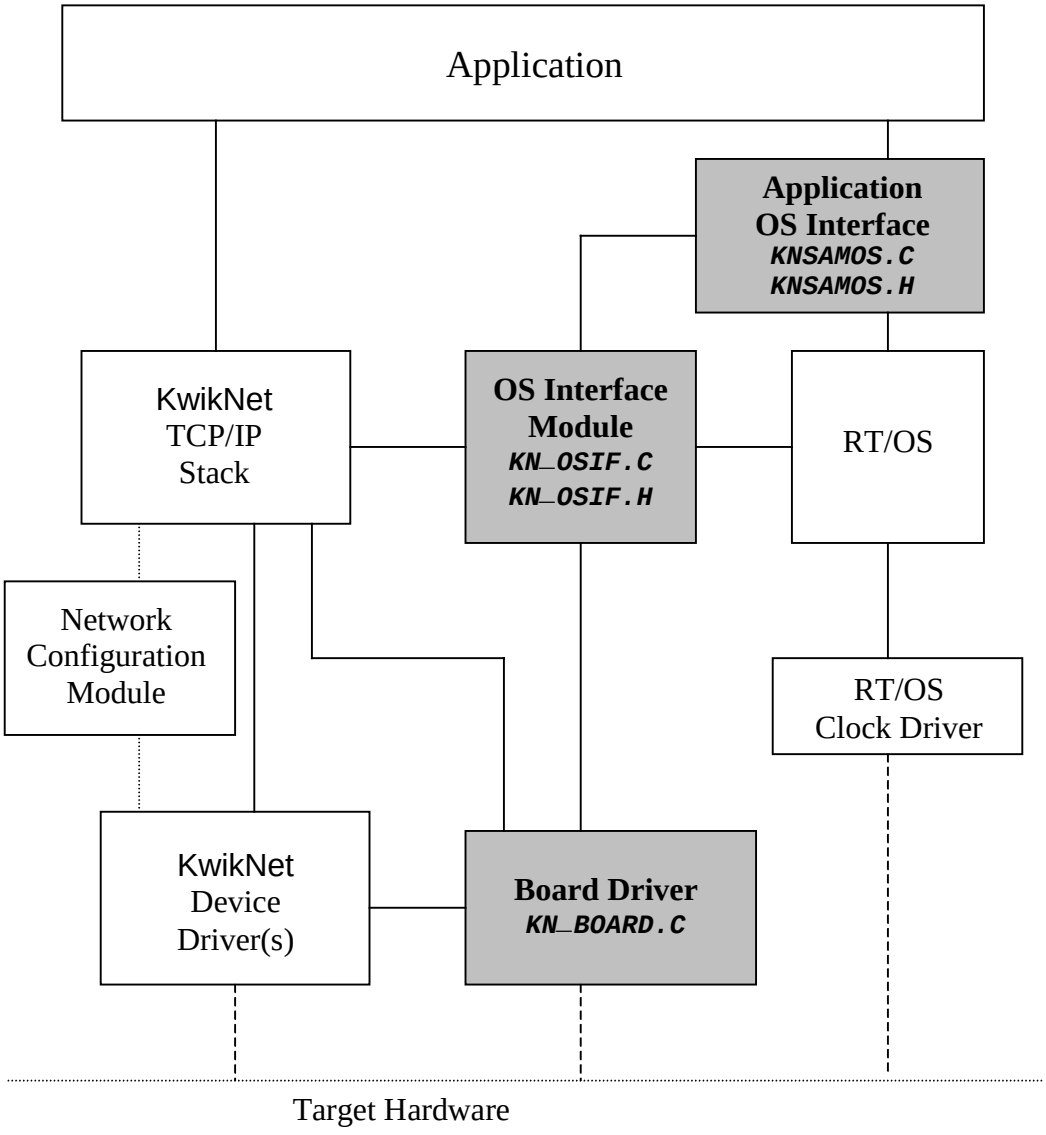

Figure 1.1-1 KwikNet Application Block Diagram

Adapting KwikNet to Your Software Tools

To adapt KwikNet for your use, you will need a make utility capable of running your C compiler, object librarian (archiver) and link/locate utility. The KwikNet library construction process is illustrated in Figure 1.1-2. The shaded blocks indicate modules which require modification to adapt the make process to accommodate your software development tools. As you can see, very few modules require adaptation.

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 of the KwikNet TCP/IP Stack User's Guide.

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 your OS Interface Module.

There are several custom adaptations which must be made for the construction process to succeed. All KwikNet C files include a KwikNet compiler configuration header file *KNZZZCC.H*. This file must be edited to identify 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. Details are provided in Chapter 3. Fortunately, a number of variants of this module are provided with KwikNet ready for use with popular compilers on a variety of target processors.

Your custom OS Interface Module is included in the KwikNet IP Library. This is the module (see Figure 1.1-1) which connects KwikNet to your RT/OS. You must specify the make dependencies and rules which control the compilation of its source file *KN_OSIF.C*. These make specifications are provided in the OS Interface Make File *KN_OSIF.INC* which the make process automatically includes. You must edit this file as described in Chapter 2 to meet your requirements.

As you would probably expect, the make file does not know how to run your C compiler and object librarian. You must provide this information 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.

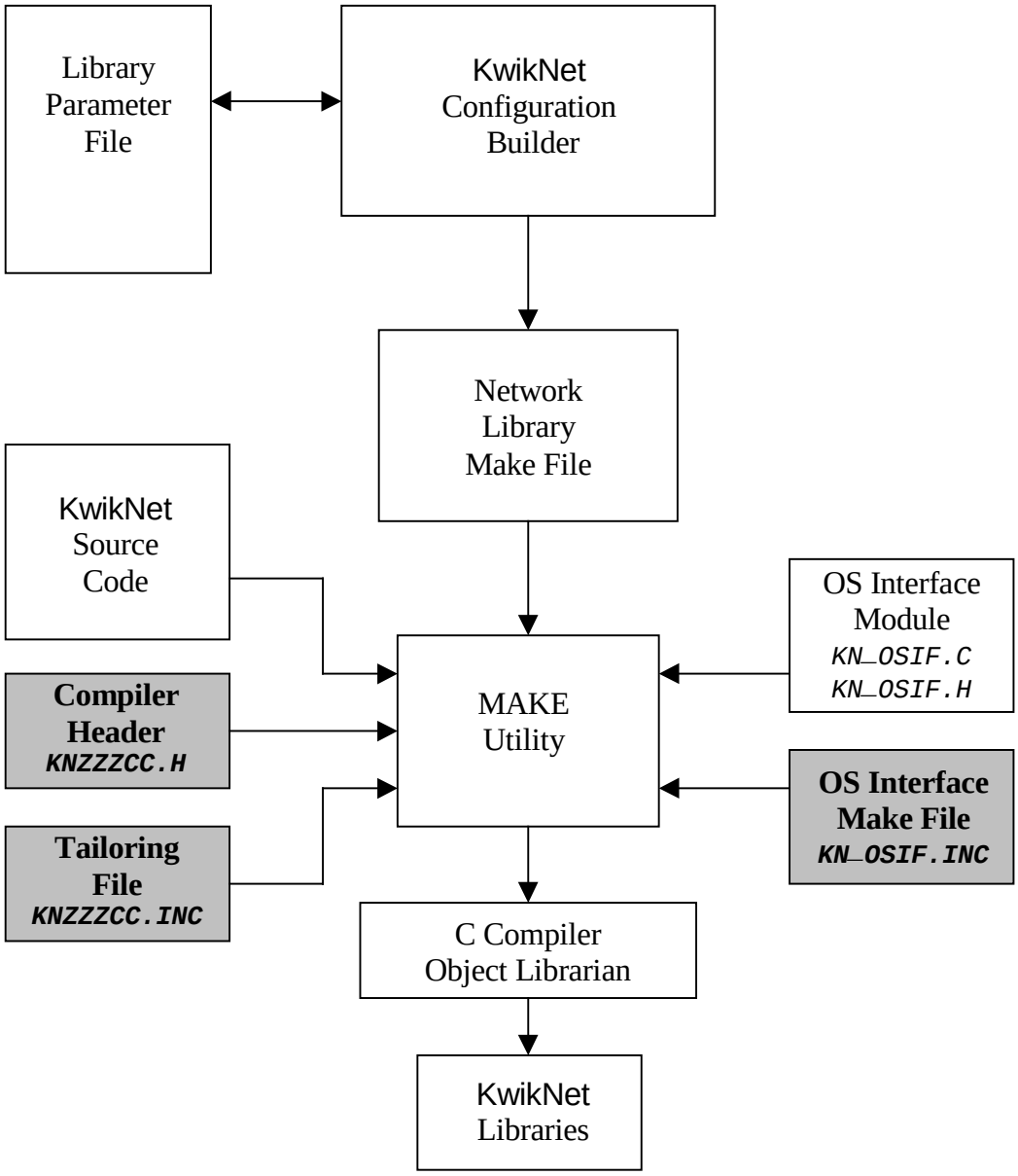

Figure 1.1-2 KwikNet Library Construction

1.2 Getting Started

Installing KwikNet

The KwikNet Porting Kit is installed as described in the Installation Guide which is packaged with the KwikNet product disks. You will observe a number of directories, many of which will contain the source modules for the KwikNet libraries. File *MANIFEST.TXT* in the root of the installation directory is the product manifest containing a list of all KwikNet installed files. You can use this text file to find the location of any of the installed KwikNet files.

Fortunately, there are few KwikNet files which will require modification. The files of interest are located in directory *EXAMPLES*. The following sets of files are provided, each set offering a complete KwikNet port using a particular RT/OS, target processor and software toolset. The file sets are located in the following subdirectories within directory *EXAMPLES*.

MSDOS	MS-DOS, PC hardware, Microsoft tools
DOS4GW	DOS/4GW, PC hardware, WATCOM tools
XRTOS	Custom RTOS, 68xxx hardware, Mentor Graphics (Microtec) tools
XOS	Custom single threaded OS, 68xxx hardware, Mentor Graphics (Microtec) tools

The MS-DOS example illustrates the use of KwikNet with stand-alone MS-DOS operating in real mode on PC compatible hardware. This single threaded example is ready to use with MS-DOS with very little change.

The DOS/4GW example illustrates the use of KwikNet with stand-alone MS-DOS operating in protected mode on PC compatible hardware with the Tenberry DOS/4GW DOS Extender. This single threaded example is ready to use with MS-DOS and DOS/4GW with very little change.

The last two examples are for use with custom KwikNet ports.

If you are using a commercial multitasking RTOS (not KADAK's AMX kernel) or your own in-house RTOS, start with the files from directory *XRTOS*. This example assumes that your RTOS includes the task, semaphore and timing features required by KwikNet for multitasking operation.

If you are using your own single threaded OS or if you are operating without an OS of any kind, start with the files from directory *XOS*.

A Trial Run

Once you have installed the KwikNet Porting Kit, you can build and test a single threaded version of KwikNet for use with MS-DOS without modifying any of the installed files. To do so, you will need Microsoft 16-bit software development tools.

As installed, KwikNet is ready for use with MS-DOS using Microsoft tools and the *NMAKE* make utility. A copy of the MS-DOS porting example from directory *EXAMPLES\MSDOS* is installed in the KwikNet toolset directory *TOOLUU*, ready for use.

If you would rather use the Borland *MAKE* utility, you will have to replace file *KNZZZCC.INC* in the *TOOLUU* directory. Copy tailoring file *B__86MC.15* from directory *EXAMPLES\TF_BORLD* to directory *TOOLUU*, renaming it *KNZZZCC.INC*.

To build the KwikNet Libraries for the TCP/IP Sample Program, skip to Chapter 4.4 and follow the directions. Build the libraries using the Library Parameter File *KNSAMLIB.UP* from directory *TOOLUU\SAM_TCP*. Just replace references to *NETLIB* with *KNSAMLIB*.

With the KwikNet Libraries in place, you are ready to build the TCP/IP Sample Program executable file *KNSAMPLE.EXE* which you will be able to load and run under MS-DOS. Skip to Chapter 5.4 and follow the directions. There is no need to gather the files; they are already in place. Simply run the Microsoft make utility as instructed.

> Porting Tip
>
> When the KwikNet Porting Kit is installed, it is ready to build the MS-DOS porting example. To build any of the other examples, go to the KwikNet installation directory *KNT713* and run batch file *TOOLUU.BAT* without parameters. Follow the instructions which it presents to copy the porting example of interest to toolset directory *TOOLUU*.
>
> You will need the software tools listed on the previous page to build the particular example which you selected.

> Porting Tip
>
> If you use the Borland *MAKE* utility, you will have to use the Borland tailoring file for the example of interest. Copy the Borland tailoring file for the compiler and target processor from directory *EXAMPLES\TF_BORLD* to directory *TOOLUU* and rename it *KNZZZCC.INC* as described in Chapter 4.3.

Files to be Edited

Once you have selected the KwikNet porting example which most closely matches your application requirements, copy the entire subdirectory from the *EXAMPLES* directory to a working directory in which the files can be edited. The following files will require modification as described in the chapter indicated.

Module	Chapter	Purpose
KN_OSIF.C	2	OS Interface Module for your RT/OS
KN_OSIF.H	2	OS Interface Header File for your RT/OS
KN_OSIF.INC	2	OS Interface Make Specification for your RT/OS
KN_BOARD.C		Board driver for your target hardware
KNZZZCC.H	3	Compiler Configuration Header File
KNZZZCC.INC	4	Tailoring File (for use with your make utility)
KN713IP.LBM	4	KwikNet IP Library Specification File
KN713TCP.LBM	4	KwikNet TCP Library Specification File
KN713.LBM*	4	Library Specification Files for optional KwikNet Libraries
KNSAMOS.C	5	Sample Program OS Interface for your RT/OS
KNSAMOS.H	5	Sample Program OS Interface Header File for your RT/OS

The number *713* in some of the filenames is the KADAK part number used to identify the KwikNet Porting Kit.

The KwikNet board driver *KN_BOARD.C* is described in Chapter 3 of the KwikNet Device Driver Technical Reference Manual.

The compiler configuration header file *KNZZZCC.H* and the tailoring file *KNZZZCC.INC* provided with each example are ready for use with Microsoft *NMAKE* and one particular software toolset. Other files are available for use with other tools.

Installation directory *EXAMPLES\CC_H* contains a number of compiler configuration header files ready for use with different compilers and target processors. All of these files have been derived from the equivalent files used by KADAK with AMX. Pick the file which you think most closely matches your C compiler's characteristics and copy that file to your working directory, renaming it *KNZZZCC.H*.

Installation directory *EXAMPLES* also contains a number of tailoring files ready for use with different compilers. Directory *EXAMPLES\TF_BORLD* contains tailoring files ready for use with the Borland *MAKE* utility. Directory *EXAMPLES\TF_MSOFT* contains equivalent tailoring files ready for use with Microsoft *NMAKE*. All of these tailoring files have been derived from the equivalent files used by KADAK with AMX. Pick the tailoring file which you think most closely matches the requirements of your make utility and software tools and copy that file to your working directory, renaming it *KNZZZCC.INC*.

2. KwikNet RT/OS Interface

2.1 Introduction

The KwikNet TCP/IP Stack requires access to services provided by your multitasking RTOS or single threaded OS. All such access is done through a collection of procedures in your OS Interface Module *KN_OSIF.C*. It is the purpose of this chapter to define the OS interface and provide detailed descriptions of each of the procedures which you must provide.

Start by selecting the OS interface files from one of the examples provided in installation directory *EXAMPLES*. The following files make up the OS interface.

KN_OSIF.C	OS Interface Module
KN_OSIF.H	OS Interface Header File
KN_OSIF.INC	OS Interface Make File

The OS Interface Header File *KN_OSIF.H* is ready for use. It identifies the particular OS interface example which you have chosen and specifies whether it is a multitasking RTOS or a single threaded OS. In general, there should be no need to edit this file. However, should you decide to add your own RT/OS specific definitions to the file, follow the edit instructions provided in the file.

Your OS Interface Module *KN_OSIF.C* must be compiled and installed in the KwikNet IP Library as described in Chapter 4. The OS Interface Make File *KN_OSIF.INC*, described in Chapter 2.4, provides your make utility with the information necessary to compile module *KN_OSIF.C*.

Using a Multitasking RTOS

If you are using a multitasking RTOS, pick your OS interface files from the example in directory *XRTOS*. The OS Interface Module *KN_OSIF.C* will contain all of the interface procedures which you require. Follow the directions in Chapter 2.2 and skip Chapter 2.3.

Using a Single Threaded OS

If you are using MS-DOS in real mode, pick your OS interface files from the example in directory *MSDOS*. If you plan to use MS-DOS in protected mode, pick files from example directory *DOS4GW*. Otherwise, pick files from example directory *XOS*. The OS Interface Module *KN_OSIF.C* will contain all of the interface procedures which you require. Skip Chapter 2.2 and follow the directions in Chapter 2.3.

2.2 The Multitasking RTOS Interface

The general operation of KwikNet is described in the KwikNet TCP/IP Stack User's Guide. When used with a multitasking RTOS, KwikNet makes use of services provided by the RTOS to enhance its operational characteristics.

The KwikNet OS Interface includes all of the interface procedures necessary to use KwikNet with your RTOS. It is simply a question of adapting the examples for use with your RTOS. ***Most of the procedures require only a few lines of code.*** Although you may choose to wade in and start editing, you should first take a few moments to read this chapter for an overview of the requirements and the recommended methods of implementation.

Your RTOS interface must provide a task, called the KwikNet Task, which operates at a priority above all other tasks wishing to make use of KwikNet and its network services. This task controls the KwikNet startup process. Once started, the task operates asynchronously, servicing the KwikNet events for which it is responsible. If your application chooses to stop KwikNet, the KwikNet Task supervises the orderly shut down and then ceases to operate.

KwikNet must be able to dynamically allocate and free blocks of memory as it executes. These memory services must be thread-safe so that the integrity of KwikNet is not compromised by the effects of task switching by your RTOS. If your RTOS provides its own memory allocation services, you should adapt the KwikNet OS interface to make use of them. Otherwise, you must provide a semaphore which KwikNet can use to protect its access to your custom memory allocation services or to those available in the standard C library.

KwikNet also needs a timing source, a periodic tickle at the frequency specified by your KwikNet Library Parameter File. KwikNet also expects to be able to initiate a delay, measured in milliseconds, during which the task using some KwikNet service will be forced to relinquish control of the processor in favour of lower priority tasks.

The KwikNet Task provides its own event message queue, thereby eliminating any dependence on the queueing services which your RTOS might provide. However, the KwikNet Task must be able to block itself (sleep) and resume execution (wakeup) at will. Your RTOS interface must provide these services in the most efficient manner possible.

From time to time, KwikNet will have to block the currently executing task pending a particular KwikNet event. The OS interface procedures which provide this blocking and unblocking service are critical to the successful operation of KwikNet.

In order to protect some of its network data structures, KwikNet uses a resource lock to prevent concurrent access by multiple tasks. The resource lock must be provided in the OS interface, usually using a resource semaphore which permits nested ownership of the resource.

KwikNet device drivers must be able to hook their interrupt handlers into the interrupt system of the target processor. The manner in which this is accomplished is both target processor and RTOS dependent. The KwikNet board driver *KN_BOARD.C* resolves the target issues. The OS interface must resolve the RTOS issues.

Summary of RTOS Interface Services

The KwikNet TCP/IP Stack gains access to your RTOS services via the procedures in your OS Interface Module *KN_OSIF.C*. These procedures are summarized below. Detailed specifications are provided in Chapter 2.5.

kn_osprep	Prepare for use of the RTOS
kn_osready	Declare the RTOS ready for use
kn_osfinish	Finished using the RTOS
kn_osenter	Entering KwikNet; setup RTOS resources accordingly
kn_osexit	Leaving KwikNet; relinquish RTOS resources accordingly
kn_osfatal	Handle a fatal error condition
kn_ostaskid	Get the task identifier of the currently executing task
kn_osmeminit	Initialize memory allocator for use by KwikNet
kn_osmemget	Get a block of memory
kn_osmemrls	Release (free) a block of memory
kn_osclkinit	Create/start a periodic timer (clock) for KwikNet use
kn_osclkexit	Stop the KwikNet timer
kn_osdelay	Block the current task for an interval measured in milliseconds
kn_osflagwait	Block the KwikNet Task until the signal flag is raised
kn_osflagup	Unblock the KwikNet Task by raising the signal flag
kn_osblock	Block the current task until a particular event of interest occurs
kn_osunblock	Unblock a task waiting for an event which just occurred
kn_oslocknet	Lock the network resource for exclusive use by the caller
kn_osunlocknet	Unlock the network resource
kn_oslockmem	Lock memory allocation services for use by the caller
kn_osunlockmem	Unlock memory allocation services
kn_oslockfs	Lock file access services for use by the caller
kn_osunlockfs	Unlock file access services
kn_osvinstall	Install an interrupt service routine
kn_osvaccess	Read from and/or write to a processor exception vector

RTOS and KwikNet Startup

The KwikNet OS interface must be initialized with a call to procedure *kn_osprep()* before your RTOS begins operation. Note that this procedure must be called by your application; it is not called by KwikNet. It is recommended that the call be made from your *main()* function.

Procedure *kn_osprep()* must call KwikNet procedure *kn_logbufinit()* to prepare the KwikNet data logger so that KwikNet procedure *kn_dprintf()* can be used for data recording even when KwikNet is not running.

Procedure *kn_osprep()* can then initialize all variables, if any, associated with the RTOS interface. Although rarely necessary, any non-RTOS resources upon which your OS interface depends should also be allocated and initialized.

Once the OS interface is ready for use, your RTOS can be started. Thereafter, the progression of execution will depend upon the way your RTOS works. Your RTOS may automatically create one or more tasks which it then executes. Others may require that you initialize the RTOS, create a task and then start the RTOS to execute that task.

Regardless of how it is done, the RTOS will finally execute some procedure which is part of your application. Your application must call procedure *kn_osready()* in the OS interface to declare the RTOS ready for use. This procedure will usually afford the first opportunity to use the RTOS to create things like tasks, timers and semaphores. It is recommended that procedure *kn_osready()* allocate the RTOS resources which your OS interface must provide for use by KwikNet. Finally, procedure *kn_osready()* must call KwikNet procedure *kn_memprep()* to prepare the KwikNet memory allocation system.

At some point during this startup process, your application must start KwikNet with a call to *kn_enter()*. This call is usually made from a task, be it an RTOS startup task or one of your own application tasks. KwikNet immediately calls your OS interface procedure *kn_osenter()*. If you have not already done so, you must allocate all of the RTOS resources which your OS interface must provide for use by KwikNet. These resource requirements will be described shortly.

Finally, the OS interface procedure *kn_osenter()* must create and start the KwikNet Task. However the task comes into existence, the KwikNet Task must begin execution in response to this start request. Starting the KwikNet Task must be the last action performed by procedure *kn_osenter()*.

Once the KwikNet Task has been started, the execution sequence will depend upon several factors. If your call to *kn_enter()* to start KwikNet was made in some kind of RTOS startup procedure, the KwikNet Task will not begin execution until your RTOS permits. If an application task called *kn_enter()* and that task is of higher priority than your KwikNet Task, then the KwikNet Task will not execute until your other higher priority tasks block for some reason. If an application task called *kn_enter()* and that task is of lower priority than your KwikNet Task, then the KwikNet Task may execute as soon as it is started from within procedure *kn_osenter()*. In other cases, your RTOS may not allow the KwikNet Task to execute, even though it is of higher priority, until a time slice tick or other task rescheduling signal occurs.

All subsequent KwikNet startup processing occurs in the context of the KwikNet Task.

KwikNet Task Operation

Your OS interface must provide an application task which will act as the KwikNet Task. The task is created and started as the final action of OS interface procedure *kn_osenter()*.

If your RTOS does not allow the dynamic creation of a task, you will have to ensure that the KwikNet Task exists before your RTOS is started. Even if tasks can be created dynamically, you may still prefer to let your RTOS automatically create your KwikNet Task from a description which you provide as part of your RTOS configuration. However the task comes into existence, the KwikNet Task must begin execution in response to the trigger (start request) from procedure *kn_osenter()*.

The KwikNet Task must meet your RTOS specifications. It is recommended that 1024 bytes of stack be allocated for use on most target processors. More stack may be needed on complex RISC processors or to satisfy your RTOS demands.

> Note
>
> The KwikNet Task must execute at a priority above that of all application tasks which make use of KwikNet services.

Special consideration may be required if your RTOS does not allow tasks to be dynamically created and/or started. If your RTOS automatically creates and starts a task when the RTOS begins, then your KwikNet Task will have to block (wait) until KwikNet is allowed to actually start. How you do this will be RTOS dependent. For example, your KwikNet Task could wait on a semaphore or event flag until signalled from OS interface procedure *kn_osenter()*.

Once your task is permitted to actually perform as the KwikNet Task, it must call KwikNet procedure *kn_task()*. There will be no return from this procedure until KwikNet is ordered by your application to shut down.

The KwikNet Task calls OS interface procedure *kn_osmeminit()* to initialize your memory allocator for use by KwikNet. Once your memory allocation services are available, KwikNet can use OS interface procedures *kn_osmemget()* and *kn_osmemrls()* to acquire and release variable sized blocks of memory.

After the KwikNet Task has initialized its network interfaces, it calls OS interface procedure *kn_osclkinit()* to create and/or start a periodic software timer for KwikNet use. Operation of the KwikNet timer will be described shortly.

Once the KwikNet Task completes its initialization, it calls OS interface procedure *kn_osflagwait()* to wait for a message to arrive on its private message queue. Messages are generated by KwikNet services invoked by your application tasks, by KwikNet timer ticks and by KwikNet device drivers. All use KwikNet's message posting service which calls OS interface procedure *kn_osflagup()* to force the KwikNet Task to resume servicing its message queue.

The KwikNet Task will continue to execute until ordered to shut down by your application's call, if any, to KwikNet procedure *kn_exit()*.

KwikNet Timer Operation

The KwikNet TCP/IP Stack operates at the clock frequency defined in your KwikNet Library Parameter File which you created using the KwikNet Configuration Builder. The frequency is provided as a parameter on the Target property page. All KwikNet timing intervals are based upon this frequency.

The minimum frequency is 2 Hz. A frequency of 10 or 20 Hz is recommended. Any frequency above 50 Hz will simply introduce unnecessary execution overhead with little noticeable improvement in network throughput.

The KwikNet timer procedure *kn_timer()* must be called by your OS interface at the defined frequency. For example, if the KwikNet frequency is declared to be 20 Hz, you must ensure that procedure *kn_timer()* is executed once every 50 ms. This procedure can be called from an interrupt service routine (ISR), provided the ISR preserves all processor registers which your C compiler considers to be alterable.

Your RTOS will probably give you a mechanism for implementing a software timer to meet this requirement. If software timers are resources created by your RTOS, then you should create the KwikNet timer at the same time as all other RTOS related resources are allocated, usually in OS interface procedure *kn_osenter()*. Alternatively, you can defer creating the timer until the KwikNet Task calls procedure *kn_osclkinit()* to start it.

If your RTOS does not provide timers, then you will have to create such a software timer and hook it to your fundamental hardware clock. In most cases, you will already have such a hook in place for use by your RTOS. In rare cases, you may have to create a task which repetitively delays for the required interval before calling *kn_timer()*.

No matter how the software timer is created or initialized, the timer must not call *kn_timer()* until the KwikNet Task calls OS interface procedure *kn_osclkinit()* to start KwikNet timing. The examples provided with the KwikNet Porting Kit illustrate this point.

If your software timer executes at a frequency greater than the KwikNet frequency, you will have to use a software counter to derive the slower KwikNet tick. When the KwikNet Task calls procedure *kn_osclkinit()* to start the KwikNet timer, it provides the timer period, measured in milliseconds, as a parameter. You can use this parameter to derive the number of actual timer ticks which constitute a KwikNet tick. For example, if the KwikNet clock frequency is 20 Hz (period of 50 ms) but your software timer operates at 100 Hz (period of 10 ms), your timer procedure must only call *kn_timer()* once every 5 ticks.

When KwikNet shuts down the TCP/IP stack, it calls OS interface procedure *kn_osclkexit()* to stop the KwikNet timer. This procedure must ensure that your software timer stops calling the KwikNet timer procedure *kn_timer()*.

If procedure *kn_clkinit()* created and started your timer procedure, then procedure *kn_clkexit()* should delete it. If your timer procedure was created by procedure *kn_osenter()*, then it should be deleted by procedure *kn_osexit()*.

Task Delays

KwikNet calls OS interface procedure *kn_osdelay()* to force the currently executing task to delay (block) for an interval measured in milliseconds. The KwikNet Task never calls this procedure. It is only called by KwikNet services which are executing in the context of an application task. Some of the server and client tasks provided with optional KwikNet components also call *kn_osdelay()* to delay briefly.

Task Identification

KwikNet calls OS interface procedure *kn_ostaskid()* to fetch the identity of the currently executing task. The procedure must return the task identifier as a 32-bit unsigned long value. The value 0 is reserved as an invalid task identifier.

KwikNet RTOS Resources

Your OS interface must provide the following RTOS resources for use by KwikNet. The KwikNet Task and the KwikNet timer requirements have already been described. The resource semaphores must be created by your OS interface procedure *kn_osready()* or *kn_osenter()* unless they already exist by the time the latter procedure is executed.

RTOS resources which are created or initialized by procedure *kn_osenter()* must be released or destroyed by procedure *kn_osexit()*. This requirement must be met if KwikNet can be shut down and restarted by your application.

RTOS resources which are created or initialized by OS interface procedure *kn_osready()* may have to be released or destroyed by your application before the RTOS is shut down. This requirement will be dictated by your RTOS. In some cases, you may be able to relinquish these RTOS resources in procedure *kn_osfinish()* which is called after the RTOS has returned to *main()*.

- A task to operate as the KwikNet Task
- A periodic timing tick to activate the KwikNet timer
- A resource semaphore to guard access to KwikNet networks
- A resource semaphore to guard memory allocation services
  (Only required if C symbol *KN_MEMLOCK* has a non-zero value.)
- A resource semaphore to guard file system access
  (Only required if C symbol *KN_FS_LOCK* has a non-zero value.)

Resource Locking

KwikNet uses a lock to guard access to its network resources. Without the lock, application tasks making use of KwikNet services could generate serious conflicts between themselves and the higher priority KwikNet Task.

A similar lock can be used to guard access to memory allocation services and file system services which are otherwise not thread-safe. These particular locks are optional and are only required if needed to support your memory allocator or file system.

KwikNet calls OS interface procedures *kn_oslockXXX()* and *kn_osunlockXXX()* to reserve and release network (*XXX* is *net*), memory allocation (*XXX* is *mem*) and file system (*XXX* is *fs*) services. In most cases, an RTOS resource semaphore is used to implement the lock. A resource semaphore is a semaphore with a task ownership attribute.

Procedure *kn_oslockXXX()* must ensure that the task making the request is blocked until ownership of the resource has been granted to that task. If the calling task already owns the resource, the caller is not blocked and retains nested ownership of the resource.

Procedure *kn_osunlockXXX()* relinquishes the resource, provided it is owned by the calling task. The task owning a resource must call *kn_osunlockXXX()* once for every call to *kn_oslockXXX()* which it made to reserve the resource. Only when the task's ownership is no longer nested is the resource released and granted to another task, if any, waiting for its use.

The resource semaphore required by KwikNet is sometimes called a mutex. If your RTOS provides a mutex, be sure that it supports the concept of task ownership and allows nesting of a task's ownership.

> Porting Tip
>
> The *XRTOS* example in the KwikNet Porting Kit can be configured to use a resource semaphore or to derive a resource semaphore from a binary semaphore. You can use the latter implementation if your RTOS does not offer a resource semaphore with nested task ownership.

KwikNet Message Queueing

KwikNet does not depend on the message passing capabilities of your RTOS. Instead, it provides its own message queue and depends only on an RTOS signalling service which most reasonable RTOS implementations offer.

The KwikNet Task calls OS interface procedure *kn_osflagwait()* to wait for a message to arrive on its private message queue. Messages are generated by KwikNet services invoked by your application tasks, by KwikNet timer ticks and by KwikNet device drivers. All use KwikNet's message posting service which calls OS interface procedure *kn_osflagup()* to force the KwikNet Task to resume servicing its message queue.

The signalling method relies on an RTOS flag of some kind. Initially the flag is down. The KwikNet Task calls *kn_osflagwait()* to wait for the flag to go up. If the flag is up at the time of the call, the flag is dropped and the KwikNet Task is allowed to continue executing. If the flag is down at the time of the call, the KwikNet Task is forced to block (wait) in procedure *kn_osflagwait()*.

Eventually some task, timer or device driver will force KwikNet to call *kn_osflagup()* to raise the flag. If the flag is up at the time of the call, it stays up and no further action is required. If the flag is down at the time of the call, the flag is raised and the KwikNet Task is forced to resume excution if it was blocked waiting for the flag. When the KwikNet Task resumes execution in procedure *kn_osflagwait()*, the flag is dropped.

In some cases, the flag may be an inherent part of some RTOS services such as *sleep()* or *wake()*. However, be careful. A *wake()* call must generate a pending wake (raise the flag) so that a subsequent *sleep()* request does not inadvertently block the KwikNet Task.

Your RTOS may provide a message queuing service which can be used to implement this feature. Whether called a queue, mailbox or exchange, create a queue which can hold only one element. Procedure *kn_osflagwait()* waits at the queue. Procedure *kn_osflagup()* adds an arbitrary element to the queue (flag goes up). If the queue already has an element in it, an error may be reported but the error can safely be ignored. When the element is retrieved from the queue by procedure *kn_osflagwait()*, the queue goes empty (flag is down).

Another common RTOS feature that can be used effectively is the mailbox which can hold a single non-zero numeric message. Procedure *kn_osflagwait()* waits at the mailbox. Procedure *kn_osflagup()* adds an arbitrary non-zero message to the mailbox (flag goes up). If the mailbox already has a message in it, an error may be reported but the error can safely be ignored. When the message is retrieved from the mailbox by procedure *kn_osflagwait()*, the mailbox is zeroed (flag is down).

Memory Allocation Services

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.

The KwikNet Task calls OS interface procedure *kn_osmeminit()* to initialize your memory allocator. Thereafter, KwikNet calls *kn_osmemget()* to get a block of memory. Some time later, it calls *kn_osmemrls()* to free the block. In many cases, the block will not be freed until KwikNet shuts down.

The KwikNet Library can be configured to support several different memory allocation strategies. The strategy is defined by the parameters in your Library Parameter File. The choices, summarized below, are made on the OS property page using the KwikNet Configuration Builder.

- Use standard C memory allocation functions
- Use RTOS memory allocator
- Use RTOS allocation services to manage allocation from a fixed region of memory provided by the application from one of the following sources:
  (1) a static array in the KwikNet Configuration Module,
  (2) an absolute address in memory or
  (3) a memory region provided by your custom *kn_memacquire()* procedure.

The KwikNet library configuration file *KN_LIB.H* specifies the strategy which you selected. C symbol *KN_MEMSRC* will be defined to have value *KN_MS_STDC* if standard C is to be used. Otherwise, an RTOS dependent allocation method is to be used.

The examples provided with the KwikNet Porting Kit support the use of standard C. If standard C is selected, the examples conditionally compile code to use standard C functions *calloc()* and *free()* to allocate and free memory. No specific memory initialization actions are required since standard C is assumed to manage its own heap.

If you use an RTOS memory management scheme, you must edit OS interface procedures *kn_osmeminit()*, *kn_osmemget()* and *kn_osmemrls()* to use your RTOS services.

Procedure *kn_osmeminit()* receives two parameters: a pointer to a fixed region of memory to be used for allocation and the size of that region. If your RTOS has its own memory allocator, procedure *kn_osmeminit()* can safely ignore these parameters since the RTOS is assumed to have its own memory resources. However, if you chose to use RTOS memory management services to control allocation from a fixed memory region, then procedure *kn_osmeminit()* must give control of the memory region to your RTOS.

> Porting Tip
>
> If your standard C or RTOS memory allocation services are not thread-safe, configure your KwikNet Library to enable KwikNet locking for memory allocation. Your OS interface can then use the memory resource semaphore to protect memory access as shown in the porting examples.

Application Blocking Services

From time to time, a task will use a KwikNet service which will force KwikNet to block (suspend) the task until a particular event occurs. KwikNet depends on two procedures in the OS interface to meet this requirement: *kn_osblock()* and *ks_osunblock()*.

KwikNet calls OS interface procedure *kn_osblock()* to block the currently executing task. The procedure is given a pointer to a callback function and a parameter which must be passed to that function. Procedure *kn_osblock()* must inhibit task preemption, execute the callback function (passing it its parameter) and then block the currently executing task. KwikNet gives the procedure a 32 byte block of storage which can be used to preserve information needed by your RTOS interface to meet the blocking requirements.

When the event of interest occurs, KwikNet calls procedure *kn_osunblock()* to unblock the task waiting for the event. KwikNet passes to the procedure a pointer to the same 32 byte block of storage which procedure *kn_osblock()* used when it blocked the task.

Of particular importance is the need to temporarily inhibit task preemption. Procedure *kn_osblock()* must ensure that the calling task cannot be preempted while it executes the callback function and blocks itself from further execution. Task switching must be disabled until the task is blocked, at which point task switching must again be enabled.

If your RTOS permits task preemption to be enabled and disabled for a specific task, implementing procedure *kn_osblock()* will be simple. Be careful if your RTOS only allows the unconditional enabling or disabling of task switching. For such an RTOS, if you disable task switching and then block (wait, sleep, etc), you may find that task switching remains disabled forever, thereby totally crippling your system.

The following technique can be used if your RTOS supports the dynamic alteration of a task's execution priority. The private storage block can be used to save the task's current execution priority. The task can then raise itself to a priority above that of the KwikNet Task. It then calls the callback function and finally blocks itself. When the event of interest occurs, procedure *kn_osunblock()* unblocks the task which then restores its original execution priority.

Another technique requiring an additional task and an associated message queue can be used. The extra task, called a sleep task, and its message queue must be created when your RTOS resources are first allocated, usually in OS interface procedure *kn_osenter()*. The sleep task must execute at a priority above the KwikNet Task. It starts and waits on its message queue. Procedure *kn_osblock()* saves the callback function pointer, its parameter and the calling task's identifier in the storage block provided by KwikNet. The pointer to the storage block is then added to the sleep task's message queue, causing the higher priority sleep task to preempt the task being blocked. The sleep task calls the callback function and finally suspends the task which sent it the message. When the event of interest occurs, procedure *kn_osunblock()* removes the suspension, allowing the blocked task to resume execution.

The latter technique has a nasty side effect. Since the callback function is executed by the sleep task, it cannot manipulate resources owned by the task being blocked. Hence, the KwikNet callback function cannot unlock the network resource owned by the task being blocked. To overcome this constraint, procedure *kn_osunlocknet()* must allow the sleep task to release network resources owned by other tasks.

Interrupt Vector Manipulation

KwikNet device drivers must be able to connect their interrupt handlers to the underlying interrupt architecture of the target processor. Unfortunately, there are almost as many architectures as there are processors.

The simplest interrupt systems are found in processors like the Motorola MC68000 and the Intel 80x86 families, in which a single, linear vector table is used to dispatch both processor exceptions and device interrupts to an appropriate software handler. Entries in the vector table are simply pointers to the supporting software function.

Protected mode Intel 80x86 systems still use a linear table, but the table entries are anything but simple. Instead, each entry is an 8 byte descriptor which leads through an exception specific gate to an appropriate software handler.

RISC processors such as the PowerPC, ARM and MIPS can be even more difficult to use. In most cases, the *R* in RISC means that most of the interrupt source decoding has been *R*emoved. Frequently all device interrupts funnel through one or two entries in a processor exception table and a software procedure must identify the interrupt source and branch to the appropriate device interrupt handler. For such processors, it will help if your RTOS, like AMX, provides a linear interrupt vector table through which all device interrupts can be dispatched.

Entries in the processor or RTOS vector table are identified by a vector number which, for simple architectures, is usually just the processor vector number. For complex architectures, the vector number will be an interrupt source identifier dictated by your RTOS or your own low level RTOS interface.

The KwikNet OS interface procedure *`kn_osvaccess()`* helps shield KwikNet from the complexity of the processor's interrupt architecture. This procedure is used to access the processor or RTOS vector table in order to attach software handlers to specific exceptions and interrupts.

Procedure *`kn_osvaccess()`* is used to read and/or write a specific entry in your processor or RTOS vector table. The entry is identified by its vector number. The procedure can be used to read and save the current vector content and then install new content, all in one operation. The procedure must ensure that device interrupts are inhibited while the vector content is being manipulated.

> Porting Tip
>
> The KwikNet IRQ identifier assigned to a KwikNet device driver should be the vector number for the actual processor or RTOS vector through which the device interrupt is serviced.

Device Driver Support

KwikNet device drivers are implemented as described in the KwikNet Device Driver Technical Reference Manual. Most network device interfaces use the processor's interrupt facility to enhance operation of the network connected to the interface. The device driver is then responsible for handling one or more interrupts generated by the device interface.

Unfortunately, several factors complicate interrupt handling. To begin with, the target processor dictates how it responds to interrupts generated by internal and external devices. In some cases, extra hardware is added to prioritize the interrupt sources. The interrupt prioritization logic may be internal to the processor or external in the form of an interrupt controller. Finally, your RTOS will superimpose its own rules for the handling of interrupts by your software.

To meet such diverse requirements, KwikNet separates board level interrupt support and RTOS interrupt support. KwikNet device drivers install interrupt handlers and manipulate interrupts using services provided in the KwikNet board driver *KN_BOARD.C*. The board driver uses services provided in the OS Interface Module to ensure compliance with your RTOS and its support for the underlying processor interrupt structure.

The KwikNet board driver *KN_BOARD.C* is described in Chapter 1.8 of the KwikNet Device Driver Technical Reference Manual. It provides support for up to *KN_INTSRCMAX* unique interrupt sources. Unless altered by you, *KN_INTSRCMAX* is defined to be 4. A KwikNet device driver provides an interrupt handler for each of the interrupts which the associated device can generate. Do not lose sight of the fact that most network devices can only generate a single interrupt, albeit for a great many different reasons.

The device driver calls board driver procedure *kn_brdintsvc()* to install a device interrupt handler into a specific interrupt vector. The vector is determined by the KwikNet IRQ identifier which you assigned via the device driver parameters entered in your KwikNet Network Parameter File.

Procedure *kn_brdintsvc()* assigns a handle to the device. The handle is a number from 1 to *KN_INTSRCMAX* which KwikNet uses to identify the interrupt handler. Procedure *kn_brdintsvc()* then calls the OS interface procedure *kn_osvinstall()* passing it the device handle and vector number.

Procedure *kn_osvinstall()* must provide an RTOS compatible interrupt service routine (ISR) for the device. The ISR must call KwikNet procedure *kn_isphandler()* passing it the device handle for the device being serviced. The examples provided with the KwikNet Porting Kit implement one such ISR for each of the *KN_INTSRCMAX* KwikNet interrupt sources.

Procedure *kn_osvinstall()* must modify the specified vector in the processor or RTOS vector table so that subsequent interrupts from the device in question are handled by its ISR. It can do so using OS interface procedure *kn_osvaccess()*.

KwikNet and RTOS Shutdown

Once KwikNet is started, it executes forever unless requested by your application to shut down. To stop KwikNet, an application shutdown task of lower priority than KwikNet must call KwikNet procedure *kn_exit()*. Most of the initial termination processing by KwikNet will occur in the context of the shutdown task. Once most stack operations have ceased, OS interface procedure *kn_osclkexit()* will be called to stop the KwikNet timer.

The shutdown task is then blocked until the KwikNet Task can complete the shutdown. The shutdown task resumes periodically to monitor progress as it waits for the shutdown process to complete.

Once the KwikNet Task has been stopped, procedure *kn_exit()*, executing in the context of the shutdown task, calls OS interface procedure *kn_osexit()* to relinquish all of the RTOS resources previously allocated by procedure *kn_osenter()*.

Most applications which shutdown KwikNet do so in preparation for a final termination. However, once *kn_exit()* returns to your shutdown task, your application can start KwikNet again with a call to procedure *kn_enter()*.

When your application is done and your RTOS has stopped, control returns to your *main()* function. At this point, procedure *kn_osfinish()* must be called to relinquish all C, C++ or hardware resources, if any, allocated by procedure *kn_osprep()* when your *main()* function started. As its last operation, *kn_osfinish()* must call KwikNet procedure *kn_memquit()* to close down the KwikNet memory allocation system.

If your KwikNet application can never be stopped, procedures *kn_osexit()* and *kn_osfinish()* can be empty stubs.

Error Handling

Whenever KwikNet detects an error condition from which recovery is not possible, it calls its fatal error handler *kn_fatal()* with one of the KwikNet fatal error codes *KN_FERxxxxx*. It is recommended that your OS interface procedures adopt this same strategy.

The KwikNet fatal error handler calls your OS interface procedure *kn_osfatal()* giving you one last chance to take whatever abortive action your RTOS may allow.

If your fatal handler *kn_osfatal()* returns, KwikNet will enter a software loop, forever calling its debug breakpoint procedure *kn_bphit()*.

2.3 The Single Threaded OS Interface

The general operation of KwikNet is described in the KwikNet TCP/IP Stack User's Guide. KwikNet is ready for use with single threaded operating systems including MS-DOS or your own stand-alone application. In the latter case, although a formal operating system may not be present, some piece of your application code can still be loosely referred to as your OS, even if it is just that endless software loop that keeps the application humming.

The KwikNet OS Interface includes all of the interface procedures necessary to use KwikNet with your OS. It is simply a question of adapting the examples for use with your OS. ***Few of the procedures will require any modification.*** Although you may choose to wade in and start editing, you should first take a few moments to read this chapter for an overview of the requirements and the recommended methods of implementation.

The main part of KwikNet is a body of code which, for lack of a better term, is called the KwikNet Task through which your application gains access to KwikNet and its network services. This task controls the KwikNet startup process. If your application chooses to stop KwikNet, the task supervises the orderly shut down and then ceases to operate.

Once KwikNet has been started, the KwikNet Task can only execute when given the opportunity by your application. The KwikNet Task executes when you call *`kn_yield()`* giving KwikNet control of the processor. The KwikNet Task will then execute as long as it has work to do. You must call *`kn_yield()`* frequently enough to ensure that the KwikNet Task can meet its fundamental timing obligations.

KwikNet must be able to dynamically allocate and free blocks of memory as it executes. These memory services are inherently thread-safe in a single threaded system. Usually standard C memory allocation services will be adequate. However, if you have your own memory allocator, you should adapt the KwikNet OS interface to use it.

KwikNet also needs a timing source, a periodic tickle at the frequency specified by your KwikNet Library Parameter File. KwikNet also expects to be able to initiate a delay, measured in milliseconds, during which the KwikNet Task will execute but your application will be blocked.

From time to time, KwikNet will have to block your application pending a particular KwikNet event. The OS interface procedures which provide this blocking and unblocking service are critical to the successful operation of KwikNet.

KwikNet device drivers must be able to hook their interrupt handlers into the interrupt system of the target processor. The manner in which this is accomplished is both target processor and OS dependent. The KwikNet board driver *`KN_BOARD.C`* resolves the target issues. The OS interface must resolve the OS issues.

Summary of OS Interface Services

The KwikNet TCP/IP Stack gains access to your OS services via the procedures in your OS Interface Module *KN_OSIF.C*. These procedures are summarized below. Detailed specifications are provided in Chapter 2.5.

kn_osprep	Prepare for use of the OS
kn_osready	Declare the OS ready for use
kn_osfinish	Finished using the OS
kn_osenter	Entering KwikNet; setup OS resources accordingly
kn_osexit	Leaving KwikNet; relinquish OS resources accordingly
kn_osfatal	Handle a fatal error condition
kn_osmeminit	Initialize memory allocator for use by KwikNet
kn_osmemget	Get a block of memory
kn_osmemrls	Release (free) a block of memory
kn_osclkinit	Create/start a periodic timer (clock) for KwikNet use
kn_osclkexit	Stop the KwikNet timer
kn_osvinstall	Install an interrupt service routine
kn_osvaccess	Read from and/or write to a processor exception vector

KwikNet Startup

The KwikNet OS interface must be initialized with a call to procedure *kn_osprep()* when your application first begins. Note that this procedure must be called by your application; it is not called by KwikNet. It is recommended that the call be made from your *main()* function.

Procedure *kn_osprep()* must call KwikNet procedure *kn_logbufinit()* to prepare the KwikNet data logger so that KwikNet procedure *kn_dprintf()* can be used for data recording even when KwikNet is not running.

Procedure *kn_osprep()* can then initialize all variables, if any, associated with the OS interface. Although rarely necessary, any resources upon which your OS interface depends should also be allocated and initialized. As a last step, *kn_osprep()* must call OS interface procedure *kn_osready()* to declare the OS ready for use. As the final step, procedure *kn_osready()* must call KwikNet procedure *kn_memprep()* to prepare the KwikNet memory allocation system.

Once the OS interface is ready for use, your application can be started. At some point, your application must start KwikNet with a call to *kn_enter()*. KwikNet immediately calls your OS interface procedure *kn_osenter()*. You must allocate the OS resources, if any, which your OS interface needs to support its operation with KwikNet.

Finally, the OS interface procedure *kn_osenter()* must call *kn_yield()* to allow the KwikNet Task to start and complete the initialization sequence. Starting the KwikNet Task must be the last action performed by procedure *kn_osenter()*.

The KwikNet Task calls OS interface procedure *kn_osmeminit()* to initialize your memory allocator for use by KwikNet. Once your memory allocation services are available, KwikNet can use OS interface procedures *kn_osmemget()* and *kn_osmemrls()* to acquire and release variable sized blocks of memory.

After the KwikNet Task has initialized its network interfaces, it calls OS interface procedure *kn_osclkinit()* to start a periodic software timer for KwikNet use. Operation of the KwikNet timer will be described shortly.

Once the KwikNet Task completes its initialization, execution resumes following the *kn_yield()* call in procedure *kn_osenter()*. KwikNet is ready for use and your application resumes execution following its call to *kn_enter()*.

KwikNet Task Operation

Your OS interface must periodically call *kn_yield()* to let the KwikNet Task execute. The task is initially started as the final action of OS interface procedure *kn_osenter()*.

The KwikNet Task will resume execution whenever you call *kn_yield()*. It checks for the arrival of a message on its private message queue. Messages are generated by KwikNet services invoked by your application, by KwikNet timer ticks and by KwikNet device drivers. Each message is decoded and serviced as required. When the message queue is empty the KwikNet Task returns to your application and awaits your next call.

The KwikNet Task will continue to execute in this fashion until ordered to shut down by your application's call, if any, to KwikNet procedure *kn_exit()*.

KwikNet Timer Operation

The KwikNet TCP/IP Stack operates at the clock frequency defined in your KwikNet Library Parameter File which you created using the KwikNet Configuration Builder. The frequency is provided as a parameter on the Target property page. All KwikNet timing intervals are based upon this frequency.

The minimum frequency is 2 Hz. A frequency of 10 or 20 Hz is recommended. Any frequency above 50 Hz will simply introduce unnecessary execution overhead with little noticeable improvement in network throughput.

The KwikNet timer procedure *kn_timer()* must be called by your OS interface at the defined frequency. For example, if the KwikNet frequency is declared to be 20 Hz, you must ensure that procedure *kn_timer()* is executed once every 50 ms. This procedure can be called from an interrupt service routine (ISR), provided the ISR preserves all processor registers which your C compiler considers to be alterable.

If your OS does not provide timer services, then you will have to create such a software timer and hook it to your fundamental hardware clock. You may already have such a hook in place for use by your application.

The software timer can be initialized in procedure *kn_osprep()* or *kn_osenter()*. Alternatively, you can defer installation of the timer until the KwikNet Task calls procedure *kn_osclkinit()* to start it.

No matter when the software timer is created or initialized, the timer must not call *kn_timer()* until the KwikNet Task calls OS interface procedure *kn_osclkinit()* to start KwikNet timing. The examples provided with the KwikNet Porting Kit illustrate this point.

If your software timer executes at a frequency greater than the KwikNet frequency, you will have to use a software counter to derive the slower KwikNet tick. When the KwikNet Task calls procedure *kn_osclkinit()* to start the KwikNet timer, it provides the timer period, measured in milliseconds, as a parameter. You can use this parameter to derive the number of actual timer ticks which constitute a KwikNet tick. For example, if the KwikNet clock frequency is 20 Hz (period of 50 ms) but your software timer operates at 100 Hz (period of 10 ms), your timer procedure must only call *kn_timer()* once every 5 ticks.

When KwikNet shuts down the TCP/IP stack, it calls OS interface procedure *kn_osclkexit()* to stop the KwikNet timer. This procedure must ensure that your software timer stops calling the KwikNet timer procedure *kn_timer()*.

If procedure *kn_clkinit()* created and started your software timer, then procedure *kn_clkexit()* should remove it. If your software timer was initialized by procedure *kn_osenter()*, then it should be removed by procedure *kn_osexit()*. If your software timer was initialized by procedure *kn_osprep()*, then it should be removed by procedure *kn_osfinish()*.

Memory Allocation Services

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.

The KwikNet Task calls OS interface procedure *kn_osmeminit()* to initialize your memory allocator. Thereafter, KwikNet calls *kn_osmemget()* to get a block of memory. Some time later, it calls *kn_osmemrls()* to free the block. In many cases, the block will not be freed until KwikNet shuts down.

The KwikNet Library can be configured to support several different memory allocation strategies. The strategy is defined by the parameters in your Library Parameter File. The choices, summarized below, are made on the OS property page using the KwikNet Configuration Builder.

- Use standard C memory allocation functions
- Use OS memory allocator
- Use custom allocation services to manage allocation from a fixed region of memory provided by the application from one of the following sources:
  (1) a static array in the KwikNet Configuration Module,
  (2) an absolute address in memory or
  (3) a memory region provided by your custom *kn_memacquire()* procedure.

The KwikNet library configuration file *KN_LIB.H* specifies the strategy which you selected. C symbol *KN_MEMSRC* will be defined to have value *KN_MS_STDC* if standard C is to be used. Otherwise, an OS allocator or custom allocation method is to be used.

The examples provided with the KwikNet Porting Kit support the use of standard C. If standard C is selected, the examples conditionally compile code to use standard C functions *calloc()* and *free()* to allocate and free memory. No specific memory initialization actions are required since standard C is assumed to manage its own heap.

If you use an OS allocator or a custom memory management scheme, you must edit OS interface procedures *kn_osmeminit()*, *kn_osmemget()* and *kn_osmemrls()*.

Procedure *kn_osmeminit()* receives two parameters: a pointer to a fixed region of memory to be used for allocation and the size of that region. If your OS has its own memory allocator, procedure *kn_osmeminit()* can safely ignore these parameters since your OS is assumed to have its own memory resources.

However, if you choose to provide custom services to control allocation from a fixed memory region, then procedure *kn_osmeminit()* must accept the specified memory region and prepare it for use. Procedure *kn_osmemget()* must allocate memory from this region. Procedure *kn_osmemrls()* must free memory previously allocated from the region.

Interrupt Vector Manipulation

KwikNet device drivers must be able to connect their interrupt handlers to the underlying interrupt architecture of the target processor. Unfortunately, there are almost as many architectures as there are processors.

The simplest interrupt systems are found in processors like the Motorola MC68000 and the Intel 80x86 families, in which a single, linear vector table is used to dispatch both processor exceptions and device interrupts to an appropriate software handler. Entries in the vector table are simply pointers to the supporting software function.

Protected mode Intel 80x86 systems still use a linear table, but the table entries are anything but simple. Instead, each entry is an 8 byte descriptor which leads through an exception specific gate to an appropriate software handler.

RISC processors such as the PowerPC, ARM and MIPS can be even more difficult to use. In most cases, the *R* in RISC means that most of the interrupt source decoding has been *R*emoved. Frequently all device interrupts funnel through one or two entries in a processor exception table and a software procedure must identify the interrupt source and branch to the appropriate device interrupt handler. For such processors, it will help if your OS provides a linear interrupt vector table through which all device interrupts can be dispatched.

Entries in the processor or OS vector table are identified by a vector number which, for simple architectures, is usually just the processor vector number. For complex architectures, the vector number will be an interrupt source identifier dictated by your OS or your own low level interrupt interface.

The KwikNet OS interface procedure *`kn_osvaccess()`* helps shield KwikNet from the complexity of the processor's interrupt architecture. This procedure is used to access the processor or OS vector table in order to attach software handlers to specific exceptions and interrupts.

Procedure *`kn_osvaccess()`* is used to read and/or write a specific entry in your processor or OS vector table. The entry is identified by its vector number. The procedure can be used to read and save the current vector content and then install new content, all in one operation. The procedure must ensure that device interrupts are inhibited while the vector content is being manipulated.

> Porting Tip
>
> The KwikNet IRQ identifier assigned to a KwikNet device driver should be the vector number for the actual processor or OS vector through which the device interrupt is serviced.

Device Driver Support

KwikNet device drivers are implemented as described in the KwikNet Device Driver Technical Reference Manual. Most network device interfaces use the processor's interrupt facility to enhance operation of the network connected to the interface. The device driver is then responsible for handling one or more interrupts generated by the device interface.

Unfortunately, several factors complicate interrupt handling. To begin with, the target processor dictates how it responds to interrupts generated by internal and external devices. In some cases, extra hardware is added to prioritize the interrupt sources. The interrupt prioritization logic may be internal to the processor or external in the form of an interrupt controller. Finally, your OS may superimpose its own rules for the handling of interrupts by your software.

To meet such diverse requirements, KwikNet separates board level interrupt support and OS interrupt support. KwikNet device drivers install interrupt handlers and manipulate interrupts using services provided in the KwikNet board driver *KN_BOARD.C*. The board driver uses services provided in the OS Interface Module to ensure compliance with your OS and its support for the underlying processor interrupt structure.

The KwikNet board driver *KN_BOARD.C* is described in Chapter 1.8 of the KwikNet Device Driver Technical Reference Manual. It provides support for up to *KN_INTSRCMAX* unique interrupt sources. Unless altered by you, *KN_INTSRCMAX* is defined to be 4. A KwikNet device driver provides an interrupt handler for each of the interrupts which the associated device can generate. Do not lose sight of the fact that most network devices can only generate a single interrupt, albeit for a great many different reasons.

The device driver calls board driver procedure *kn_brdintsvc()* to install a device interrupt handler into a specific interrupt vector. The vector is determined by the KwikNet IRQ identifier which you assigned via the device driver parameters entered in your KwikNet Network Parameter File.

Procedure *kn_brdintsvc()* assigns a handle to the device. The handle is a number from 1 to *KN_INTSRCMAX* which KwikNet uses to identify the interrupt handler. Procedure *kn_brdintsvc()* then calls the OS interface procedure *kn_osvinstall()* passing it the device handle and vector number.

Procedure *kn_osvinstall()* must provide an interrupt service routine (ISR) for the device. The ISR must call KwikNet procedure *kn_isphandler()* passing it the device handle for the device being serviced. The examples provided with the KwikNet Porting Kit implement one such ISR for each of the *KN_INTSRCMAX* KwikNet interrupt sources.

Procedure *kn_osvinstall()* must modify the specified vector in the processor or OS vector table so that subsequent interrupts from the device in question are handled by its ISR. It can do so using OS interface procedure *kn_osvaccess()*.

KwikNet and OS Shutdown

Once KwikNet is started, it executes forever unless requested by your application to shut down. To stop KwikNet, your application must call KwikNet procedure *kn_exit()*. The function in which the call is made will be referred to as the shutdown function. All of the termination processing by KwikNet will occur in the context of the shutdown function. Once most stack operations have ceased, OS interface procedure *kn_osclkexit()* will be called to stop the KwikNet timer.

The KwikNet Task is then executed to complete the shutdown process. Then OS interface procedure *kn_osexit()* is called to relinquish all of the OS resources previously allocated by procedure *kn_osenter()*.

Most applications which shutdown KwikNet do so in preparation for a final termination. However, once *kn_exit()* returns to your shutdown function, your application can start KwikNet again with a call to procedure *kn_enter()*.

When your application is finished, control returns to your *main()* function. At this point, procedure *kn_osfinish()* must be called to relinquish all C, C++ or hardware resources, if any, allocated by procedure *kn_osprep()* when your *main()* function started. As its last operation, *kn_osfinish()* must call KwikNet procedure *kn_memquit()* to close down the KwikNet memory allocation system.

If your KwikNet application can never be stopped, procedures *kn_osexit()* and *kn_osfinish()* can be empty stubs.

Error Handling

Whenever KwikNet detects an error condition from which recovery is not possible, it calls its fatal error handler *kn_fatal()* with one of the KwikNet fatal error codes *KN_FERxxxxx*. It is recommended that your OS interface procedures adopt this same strategy.

The KwikNet fatal error handler calls your OS interface procedure *kn_osfatal()* giving you one last chance to take whatever abortive action your OS may allow.

If your fatal handler *kn_osfatal()* returns, KwikNet will enter a software loop, forever calling its debug breakpoint procedure *kn_bphit()*.

2.4 RT/OS Interface Make File

The OS Interface Make File *KN_OSIF.INC* contains all of the make specifications which you must edit to define how your OS Interface Module is to be compiled. This module also provides access to the RT/OS object modules and/or libraries which must be linked with the KwikNet sample programs.

Figure 2.4-1 shows a listing of a typical OS Interface Make File *KN_OSIF.INC*. This example is taken directly from the *EXAMPLES\XRTOS* directory. You should examine the other examples as well to see how they differ in ways that might apply to your RT/OS.

The specification begins with a set of macro definitions which are only used within the OS Interface Make File itself. In this example, macros *OSHDRP* and *OSLIBP* have been defined as the paths to the RTOS header and library files. Note that global macro *$(OSPATH)* is the path to your RT/OS installation directory. It is defined on the command line used to invoke your make utility when building the KwikNet Libraries.

Macro *HOSIF* must be defined, even if it is an empty (blank) macro. It must identify all of the RT/OS header files upon which file *KN_OSIF.C* and its header file *KN_OSIF.H* depend. Note that the header file *KN_OSIF.H* will not include any other header files unless you have explicitly edited it to do so.

If you port the KwikNet Sample Program, then your application OS interface module *KNSAMOS.C* and its header file *KNSAMOS.H* may depend on RT/OS header files not already identified by macro *HOSIF*. If so, be sure to add those RT/OS header files to the *HOSIF* definition.

Target *OSHDR* must be defined. It must list the file names of all header files identified by macro *HOSIF*. The file names must not include path information.

For each file listed in target *OSHDR*, there must be a rule provided which copies the header file from its source directory to the current directory.

If you port the KwikNet Sample Program, you must identify the RT/OS object modules and/or library files with which the samples must be linked. These modules must be identified in a dependency list defined by macro *OSLIB*. An empty list is permissible.

Target *OSLIBGET* must be defined without any dependencies. Its purpose is to copy all of the files identified by macro *OSLIB* from their source directory to the directory identified by global macro *$(ULINK)*. An empty command list is permissible.

Target *OSLIBDEL* must also be defined with no dependencies. Its purpose is to delete all of the files copied when target *OSLIBGET* was resolved. It must delete all of the files identified by macro *OSLIB* from the destination directory identified by *$(ULINK)*. An empty command list is permissible.

> Porting Tip
>
> For portability of the make process, it is recommended that you use the global macros *$(CMDCOPY)* and *$(CMDDEL)* to invoke the copy and delete (erase) commands of the operating system which is executing your make utility.


```
# ----------------------------------
# The following macros are used only in this include file.

# OSHDRP = full path to the directory containing the RTOS header file(s)
# OSLIBP = full path to the directory containing the RTOS library file(s)

OSHDRP     = $(OSPATH)\INCLUDE
OSLIBP     = $(OSPATH)\LIB


# ----------------------------------
# Define HOSIF to identify the OS dependent header files that
# are included by your custom KwikNet OS Interface modules
# KN_OSIF.C and KN_OSIF.H and by the KwikNet Sample Program
# OS interface modules KNSAMOS.C and KNSAMOS.H.

HOSIF =    $(OSHDRP)\U_RTOS.H \
           $(OSHDRP)\U_CFG.H


# ----------------------------------
# Define rules to copy the OS header file(s) to the MAKE directory
# in order to compile the KwikNet OS Interface module KN_OSIF.C
# and the KwikNet sample programs.  Be sure to copy all of the
# OS header files identified by macro HOSIF.

OSHDR: \
           U_RTOS.H \
           U_CFG.H
# no commands required


U_RTOS.H:                     $(OSHDRP)\U_RTOS.H
           $(CMDCOPY) $(OSHDRP)\U_RTOS.H U_RTOS.H

U_CFG.H:                      $(OSHDRP)\U_CFG.H
           $(CMDCOPY) $(OSHDRP)\U_CFG.H U_CFG.H


# ----------------------------------
# Define OSLIB to identify the OS library and/or object files that must
# be linked with the KwikNet sample programs to satisfy the needs of
# your KwikNet OS interface.

OSLIB =    $(OSLIBP)\U_RTOS.LIB \
           $(OSLIBP)\U_CFG.OBJ


# ----------------------------------
# Define a target to copy the OS library and/or object file(s) to the
# $(ULINK) directory in order to link the KwikNet sample programs.
# Be sure to copy all of the files identified by macro OSLIB.

OSLIBGET:
           $(CMDCOPY) $(OSLIBP)\U_RTOS.LIB $(ULINK)\U_RTOS.LIB
           $(CMDCOPY) $(OSLIBP)\U_CFG.OBJ $(ULINK)\U_CFG.OBJ


# Define another target to delete the OS library and/or object file(s)
# copied by OSLIBGET.

OSLIBDEL:
           $(CMDDEL) $(ULINK)\U_RTOS.LIB
           $(CMDDEL) $(ULINK)\U_CFG.OBJ

# ------------------- End of INCLUDE file -------------------------------
```

Figure 2.4-1 KwikNet OS Interface Make File

2.5 RT/OS Interface Procedures

The KwikNet TCP/IP Stack gains access to your RT/OS services via the procedures in your OS Interface Module *KN_OSIF.C*. A detailed specification for each procedure is provided in this chapter. The specifications are presented in the order in which they appear in the following summary.

Multitasking RTOS or Single Threaded OS Interface:

kn_osprep	Prepare for use of the RT/OS
kn_osready	Declare the RT/OS ready for use
kn_osfinish	Finished using the RT/OS
kn_osenter	Entering KwikNet; setup RT/OS resources accordingly
kn_osexit	Leaving KwikNet; relinquish RT/OS resources accordingly
kn_osfatal	Handle a fatal error condition
kn_osmeminit	Initialize memory allocator for use by KwikNet
kn_osmemget	Get a block of memory
kn_osmemrls	Release (free) a block of memory
kn_osclkinit	Create/start a periodic timer (clock) for KwikNet use
kn_osclkexit	Stop the KwikNet timer
kn_osvinstall	Install an interrupt service routine
kn_osvaccess	Read from and/or write to a processor exception vector

Multitasking RTOS Interface Only:

kn_ostaskid	Get the task identifier of the currently executing task
kn_osflagwait	Block the KwikNet Task until the signal flag is raised
kn_osflagup	Unblock the KwikNet Task by raising the signal flag
kn_osdelay	Block the current task for an interval measured in milliseconds
kn_osblock	Block the current task until a particular event of interest occurs
kn_osunblock	Unblock a task waiting for an event which just occurred
kn_oslocknet	Lock the network resource for exclusive use by the caller
kn_osunlocknet	Unlock the network resource
kn_oslockmem	Lock memory allocation services for use by the caller
kn_osunlockmem	Unlock memory allocation services
kn_oslockfs	Lock file access services for use by the caller
kn_osunlockfs	Unlock file access services

kn_osprep
kn_osready
kn_osfinish

kn_osprep
kn_osready
kn_osfinish

Purpose **Begin and End Use of the RT/OS**

Caller Procedures *kn_osprep()* and *kn_osfinish()* should be called from your *main()* function as illustrated in the KwikNet Sample Program OS interface module *KNSAMOS.C*. Procedure *kn_osready()* must be called by your application once the RT/OS is ready for use.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osprep(void);
void kn_osready(void);
void kn_osfinish(void);
```

Description Procedure *kn_osprep()* must be called prior to first use of your RT/OS by your application. It must call KwikNet procedure *kn_logbufinit()* to prepare the KwikNet data logger so that KwikNet procedure *kn_dprintf()* can be used for data recording even if KwikNet is not running.

Procedure *kn_osprep()* must initialize all variables, if any, associated with the RT/OS interface. It can then allocate the C, C++ or hardware resources, if any, which your OS interface requires for its operation.

Procedure *kn_osready()* must be called by your application once the RT/OS is ready for use. It must conclude with a call to KwikNet procedure *kn_memprep()* to prepare the KwikNet memory allocation system.

Procedure *kn_osfinish()* must be called after your application has shut down and no longer requires the use of your RT/OS. Its purpose is to relinquish all C, C++ or hardware resources, if any, allocated by procedure *kn_osprep()*. It must conclude with a call to KwikNet procedure *kn_memquit()* to shut down the KwikNet memory allocation system. If the RT/OS cannot be shut down once started, procedure *kn_osfinish()* can be an empty stub which will never be executed.

Returns Nothing

Multitasking Operation

Procedure *kn_osprep()* must be called before the RTOS is started. It cannot use any RTOS services. Procedure *kn_osready()* must be called as soon as the RTOS is ready for use. Procedure *kn_osfinish()* must not be called until the RTOS has stopped operating. It cannot use any RTOS services.

Single Threaded Operation

Except for the mandatory calls to KwikNet procedures *kn_logbufinit()*, *kn_memprep()* and *kn_memquit()*, these procedures rarely have much to do in a single threaded OS.

Purpose **Allocate/Release RT/OS Resources on Entry to/Exit from KwikNet**

Caller These procedures are called by KwikNet as it starts up and shuts down the KwikNet TCP/IP Stack.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osenter(void);
void kn_osexit(void);
```

Description Procedure *kn_osenter()* is the first OS interface procedure called by KwikNet after your application starts KwikNet with a call to *kn_enter()*. Its purpose is to allocate all of the RT/OS resources which your OS interface must provide for the proper operation of KwikNet.

Procedure *kn_osexit()* is the last OS interface procedure called by KwikNet as it shuts down in response to your application's call to *kn_exit()*. Its purpose is to relinquish all RT/OS resources, if any, allocated by procedure *kn_osenter()*. If KwikNet will not be shut down once started, procedure *kn_osexit()* can be an empty stub which will never be executed.

Returns Nothing

Multitasking Operation

Procedure *kn_osenter()* must create and/or initialize the RTOS resources required by KwikNet. If the task(s), timer(s) and resource semaphore(s) which you must provide do not already exist, they must be created. In particular, the KwikNet Task and the network locking semaphore must exist. You may choose to leave creation of the KwikNet timer to OS interface procedure *kn_osclkinit()*. The final operation of this procedure must be to start the KwikNet Task.

KwikNet is usually only shut down as a prelude to stopping your RTOS. Procedure *kn_osexit()* must release all of the RTOS resources allocated by *kn_osenter()*. This requirement must be met if KwikNet is to be subsequently restarted by your application. However, if your application is terminating and your RTOS can be shut down with resources left allocated but no longer in use, then procedure *kn_osexit()* will not have to release the allocated resources.

Single Threaded Operation

KwikNet makes no demands on a single threaded OS. Consequently, these procedures rarely have any OS dependent operations to perform. The final operation of procedure *kn_osenter()* must be to call *kn_yield()* to start the KwikNet Task. Procedure *kn_osexit()* must release any OS resources allocated by *kn_osenter()*.

Purpose **Handle a Fatal Error Condition**

Caller Called from the KwikNet fatal error procedure *kn_fatal()* upon detection of an unrecoverable error condition.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osfatal(int fatalerr);
```

Parameters Parameter *fatalerr* is one of the KwikNet fatal error codes listed in Appendix B of the KwikNet TCP/IP Stack User's Guide. Fatal error code *KN_FERPORT* can be used to identify fatal conditions detected by your OS interface.

Description Procedure *kn_osfatal()* is given the opportunity to handle a fatal error in an RT/OS dependent fashion.

The examples provided with the KwikNet Porting Kit call data logging procedure *kn_dprintf()* to record an error message and then generate a debug trap by calling the KwikNet breakpoint procedure *kn_bphit()*.

Returns Nothing

Procedure *kn_osfatal()* need not return to KwikNet. However, if it does, KwikNet will enter a software loop, repetitively calling its breakpoint procedure *kn_bphit()*.

Multitasking Operation

If your RTOS allows an application to abort or restart in some manner, you may be able to initiate that action within procedure *kn_osfatal()*.

If the procedure returns to KwikNet, the task which was executing when the fatal error condition was detected will become compute bound, preventing all lower priority tasks from executing.

Single Threaded Operation

If your OS allows an application to abort or restart in some manner, you may be able to initiate that action within procedure *kn_osfatal()*.

If the procedure returns to KwikNet, your application will appear to hang.

Purpose **Initialize the RT/OS Memory Allocator**

Caller Called by the KwikNet Task as it starts up.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osmeminit(char *memp, unsigned long memsz);
```

Parameters *Memp* is a pointer to a region of memory which your OS interface can manage for memory allocation purposes. *Memp* references the memory region specified by you in your KwikNet Network Parameter File.

If your KwikNet network configuration indicates that you are using standard C for memory allocation or that your RT/OS memory allocator does not require a fixed block of memory for its use, *memp* will be *NULL*.

Memsz is the size, in bytes, of the memory region referenced by *memp*. If *memp* is *NULL*, the value of *memsz* will be undefined.

Description If your KwikNet network configuration indicates that you are using standard C for memory allocation or that your RT/OS memory allocator does not require a fixed block of memory for its use, then procedure *kn_osmeminit()* does nothing. Otherwise it must prepare the memory region for subsequent use by memory allocation procedures *kn_osmemget()* and *kn_osmemrls()*.

Returns Nothing

Multitasking Operation

If standard C memory allocation is used or if your RTOS provides its own memory (heap) for allocation, leave procedure *kn_osmeminit()* unaltered. Otherwise, edit this procedure to match your RTOS requirements.

Single Threaded Operation

If standard C memory allocation is used, leave procedure *kn_osmeminit()* unaltered. Otherwise, edit this procedure only if you wish to use or implement your own memory allocation services.

Purpose **Get a Zero Filled Block of Memory / Release a Block of Memory**

Caller Called by KwikNet procedures executing in the context of the KwikNet Task or the currently executing application or task.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void *kn_osmemget(unsigned int memsize);
void kn_osmemrls(void *blockp);
```

Parameters *Memsize* is the number of bytes of memory required by the caller.

Blockp is a pointer to a block of memory previously allocated by procedure *kn_osmemget()*.

Description If standard C memory allocation is used, leave procedure *kn_osmemget()* unaltered. Otherwise, edit the procedure to use your RT/OS memory allocation services to acquire a block of *memsize* bytes of memory. If necessary, zero fill the block of memory which it provides.

If standard C memory allocation is used, leave procedure *kn_osmemrls()* unaltered. Otherwise, edit the procedure to use your RT/OS memory allocation services to release the block of memory referenced by *blockp*.

Returns *Kn_osmemrls()* returns nothing.
Kn_osmemget() returns a pointer to a zero filled region of *memsize* bytes of memory. If enough memory is not available, *NULL* is returned.

Multitasking Operation

If your RTOS has its own memory allocator, use it to get and free memory. Alternatively, if your RTOS has a general purpose heap manager, you can use it to control access to the memory region (heap) presented to procedure *kn_osmeminit()*. Otherwise, edit procedure *kn_osmemget()* to allocate memory from the memory region.

If your standard C or RTOS memory allocation services are not thread-safe, be sure to configure KwikNet to provide memory access locking and implement OS interface procedures *kn_oslockmem()* and *kn_osunlockmem()*. Your allocation operations must be bracketed by calls to these locking procedures as illustrated in the examples provided with the KwikNet Porting Kit.

Single Threaded Operation

The memory allocation procedures provided with the examples in the KwikNet Porting Kit are ready to use with standard C. There is no need to edit these procedures unless you intend to use the memory management services provided by your OS or prefer to implement your own.

Purpose **Start and Stop the RT/OS Timer Providing the KwikNet Clock Source**

Caller Procedure *kn_osclkinit()* is called by the KwikNet Task as it prepares the TCP/IP stack for use. Procedure *kn_osclkexit()* is called in the context of the shutdown task/function which called *kn_exit()* to stop KwikNet.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osclkinit(unsigned long ms);
void kn_osclkexit(void);
```

Parameters Parameter *ms* is the period of the KwikNet clock measured in milliseconds. For example, if the KwikNet clock frequency is configured to be 20 Hz, *ms* will have the value *50* specifying a 50 ms clock period.

Description Procedure *kn_osclkinit()* must start an OS interface software timer which forces KwikNet timer procedure *kn_timer()* to be called once every *ms* milliseconds. Note that it is acceptable to call procedure *kn_timer()* from within an interrupt service routine.

Procedure *kn_osclkexit()* must stop the OS interface software timer. At the very least it must inhibit the software timer from making subsequent calls to procedure *kn_timer()*.

Returns Nothing

Multitasking Operation

If your RTOS allows a software timer to be dynamically created and/or started, create such a timer and start it. Otherwise, add such a software timer as a hook within your RTOS timer support module or your hardware clock interrupt service routine.

You may prefer to create your software timer within your OS interface procedure *kn_osenter()* when all other RTOS resources required by KwikNet are allocated. If you do so, be sure that your software timer inhibits calls to *kn_timer()* until procedure *kn_osclkinit()* is called by the KwikNet Task to start the KwikNet timer.

Single Threaded Operation

Add your software timer as a hook within your OS timer support module or your hardware clock interrupt service routine.

If your clock hook is installed by OS interface procedure *kn_osprep()* or *kn_osenter()*, be sure that it inhibits calls to *kn_timer()* until procedure *kn_osclkinit()* is called by the KwikNet Task to start the KwikNet timer.

Purpose **Make and/or Install an RT/OS Compatible Interrupt Service Routine**

Caller Called by KwikNet device drivers to add their interrupt handler to the processor or RT/OS interrupt system.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osvinstall(int vector, int handle);
```

Parameters Parameter *vector* identifies an entry in the processor or RTOS exception table or interrupt vector table. For most simple interrupt architectures, *vector* is the interrupt vector number. For complex architectures, the vector number may be dictated by your RT/OS or by your custom exception handling software.

See the topics "Interrupt Vector Manipulation" and "Device Driver Support" in Chapter 2.2 or 2.3 for a full discussion of interrupt vector identification.

Parameter *handle* is a number ranging in value from 1 to *KN_INTSRCMAX*. This parameter identifies the particular KwikNet device driver which services interrupts generated via the specified interrupt vector. Constant *KN_INTSRCMAX*, defined in the KwikNet board driver *KN_BOARD.C*, specifies the maximum number of KwikNet device driver interrupt sources that this module can support.

Description This procedure must install an RT/OS compatible interrupt service routine (ISR) into the processor or RT/OS interrupt vector table. It can do so using OS interface procedure *kn_osvaccess()*.

The ISR must call KwikNet procedure *kn_isphandler()*, passing it the device handle for the device being serviced. The examples provided with the KwikNet Porting Kit implement one such ISR for each of the *KN_INTSRCMAX* KwikNet interrupt sources. Procedure *kn_isphandler()* is prototyped in the OS Interface header file *KN_OSIF.H* as follows:

```
void kn_isphandler(int handle);
```

Returns Nothing

...more

...continued

Multitasking Operation

The *XRTOS* example provided with the KwikNet Porting Kit uses the following technique for implementing procedure *kn_osvinstall()*.

This simple approach can be adopted if your RTOS and C compiler permit an interrupt service routine (ISR) to be coded in C. Suppose that the following ISR (with modification) can service interrupts from the device with a KwikNet device handle of 1. When procedure *kn_osvinstall()* is called with a *handle* value of *1*, it simply modifies the entry in the interrupt vector table (per parameter *vector*) to force procedure *os_ispsrc1()* to be used for interrupt service.

```
void _interrupt os_ispsrc1(void)
{
      rtos_isrenter();
      kn_isphandler(1);
      rtos_isrleave();
      }
```

Single Threaded Operation

The examples provided with the KwikNet Porting Kit assume that your C compiler permits an interrupt service routine (ISR) to be coded in C. Suppose that the following ISR (with modification) can service interrupts from the device with a KwikNet device handle of 2. When procedure *kn_osvinstall()* is called with a *handle* value of *2*, it simply modifies the entry in the interrupt vector table (per parameter *vector*) to force procedure *os_ispsrc2()* to be used for interrupt service.

```
void _interrupt os_ispsrc2(void)
{
      kn_isphandler(2);
      }
```

Purpose **Read from/Write to the Processor or RT/OS Interrupt Vector Table**

Caller Called by procedures in the KwikNet board driver *KN_BOARD.C*, the sample program OS interface module *KNSAMOS.C* and the OS Interface Module *KN_OSIF.C*.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osvaccess(int vector, void *vectp, void *oldvectp);
```

Parameters Parameter *vector* identifies an entry in the processor or RTOS exception table or interrupt vector table. For most simple interrupt architectures, *vector* is the interrupt vector number. For complex architectures, the vector number may be dictated by your RT/OS or by your custom exception handling software.

See the topics "Interrupt Vector Manipulation" and "Device Driver Support" in Chapter 2.2 or 2.3 for a full discussion of interrupt vector identification.

Parameter *vectp* is a pointer to storage containing the description of the interrupt service routine (ISR) to be installed. If *vectp* is *NULL*, the vector table must not be altered.

Parameter *oldvectp* is a pointer to storage for a description of the interrupt service routine (ISR) in use at the time this procedure was called. If *oldvectp* is *NULL*, the parameter is ignored.

By default, an ISR description is just a pointer to the actual ISR. Hence, *vectp* and *oldvectp* are actually pointers to an ISR pointer. This ISR description is used by all of the examples provided with the KwikNet Porting Kit. However, for more complex interrupt architectures, you may have to change the ISR description to match your RT/OS or processor requirements. If the ISR description is changed, be sure to modify all source modules which reference procedure *kn_osvaccess()*.

Description If parameter *oldvectp* is not *NULL*, procedure *kn_osvaccess()* must extract the ISR description from the entry in the processor or RT/OS interrupt vector table specified by parameter *vector* and store it in memory at **oldvectp*.

Then, if parameter *vectp* is not *NULL*, the procedure must modify the entry in the interrupt vector table to match the ISR description found in memory at **vectp*.

Device interrupts must be disabled while the vector table entry is being read and modified to ensure that read/write operations are indivisible.

...more

...continued

Returns Nothing

Multitasking Operation

If your RTOS provides services for modifying the processor or RTOS interrupt vector table, modify procedure *kn_osvaccess()* to make use of them. Otherwise, adapt the *XRTOS* example provided with the KwikNet Porting Kit for use with your target processor and RTOS.

Single Threaded Operation

The MS-DOS example provided with the KwikNet Porting Kit is ready for use, without modification, with MS-DOS operating in real mode on an 80x86 processor.

The DOS/4GW example provided with the KwikNet Porting Kit is ready for use with MS-DOS and the Tenberry DOS/4GW DOS Extender operating in protected mode on an 80386, 80486 or Pentium processor. However, without modification, it is restricted for use with interrupt vectors 8 to 15 (IRQ0 to IRQ7).

If your OS provides services for modifying the processor's interrupt vector table, modify procedure *kn_osvaccess()* from the *XOS* example to make use of them. Otherwise, adapt one of the *MSDOS*, *DOS4GW* or *XOS* examples provided with the KwikNet Porting Kit for use with your target processor.

Purpose **Get the Task Identifier of the Currently Executing Task**

Caller Called by the KwikNet Task.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
KNP_OS_ID kn_ostaskid(void);
```

Description KwikNet uses a 32-bit task identifier declared as type *KNP_OS_ID*. The value *(KNP_OS_ID)0* is reserved as an invalid task identifier.

If the task identifier used by your RTOS can be represented in 32 bits or less, then procedure *kn_ostaskid()* can fetch the task identifier for the currently executing task from your RTOS and cast it to be a KwikNet task identifier.

Otherwise, this procedure must convert your RTOS task identifiers into a form that can be represented in 32 bits.

Returns The RTOS identifier for the currently executing task, cast to be a KwikNet task identifier of type *KNP_OS_ID*.

If this function is called while not executing in the context of a task, it should return *(KNP_OS_ID)0*, the KwikNet representation for an invalid task identifier.

Multitasking Operation

This procedure must be implemented as described.

Single Threaded Operation

This procedure is not present in the single threaded OS Interface Module.

Purpose **Block/Unblock the KwikNet Task using a Signal Flag**

Caller The KwikNet Task calls procedure *kn_osflagwait()* to wait for the next occurrence of a significant KwikNet event. The KwikNet message posting service calls procedure *kn_osflagup()* to signal such an event.

Calls to *kn_osflagup()* can originate in the KwikNet Task, in the KwikNet timer procedure or in an application task or device interrupt handler making use of KwikNet services.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osflagwait(void);
void kn_osflagup(void);
```

Description A detailed description of the operation of these two procedures is provided in Chapter 2.2 under the topic "KwikNet Message Queueing". Several different RTOS strategies for implementing these procedures are explored.

Returns Nothing

Multitasking Operation

These procedures must be implemented as described in Chapter 2.2 under the topic "KwikNet Message Queueing".

Single Threaded Operation

These procedures are not present in the single threaded OS Interface Module.

Purpose **Suspend (Delay) the Currently Executing Task for a Timed Interval**

Caller Called by any task which has requested a KwikNet operation which requires that the task pause briefly to allow lower priority tasks an opportunity to execute. Most KwikNet server and client tasks will call this procedure.

The KwikNet Task never calls this procedure.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osdelay(unsigned long ms);
```

Parameters Parameter *ms* specifies the required delay interval, measured in milliseconds.

Description The currently executing task must be forced to pause for *ms* milliseconds.

If the minimum timing resolution provided by your RTOS is greater than *ms* milliseconds, the task must pause, at the very least, until the next RTOS tick. The examples provided with the KwikNet Porting Kit meet this requirement.

Returns Nothing

Multitasking Operation

This procedure must be implemented as described.

Single Threaded Operation

Although this procedure must be present in the single threaded OS Interface Module, you will observe that the examples provided with the KwikNet Porting Kit require no modification.

The procedure is included in the OS interface so that you can adapt it to perform other essential, application dependent operations which must continue, even while KwikNet is spinning waiting for the delay interval to expire.

Purpose **Block/Unblock a Task Waiting for an Event**

Caller KwikNet calls procedure *kn_osblock()* to block the currently executing task pending a particular KwikNet event. When KwikNet detects the event, it calls *kn_osunblock()* to unblock the task waiting for the event.

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_osblock(void (*func)(void *),
                  void *param, void *infop);
void kn_osunblock(void *infop);
```

Parameters Parameter *func* is a pointer to a KwikNet function which must be called by procedure *kn_osblock()*.

Param is a pointer which must be passed to function *func* as a parameter.

Infop is a pointer to a block of 32 bytes of memory storage which procedure *kn_osblock()* can use, if necessary, to save information for subsequent use by procedure *kn_osunblock()*.

Description Procedure *kn_osblock()* must inhibit task preemption, call function *func* passing it the parameter *param* and then block the currently executing task to await some KwikNet event.

A detailed description of the operation of these two procedures is provided in Chapter 2.2 under the topic "Application Blocking Services". Several different RTOS strategies for implementing these procedures are explored.

Returns Nothing

Multitasking Operation

These procedures must be implemented as described in Chapter 2.2 under the topic "Application Blocking Services".

Single Threaded Operation

Although these procedures must be present in the single threaded OS Interface Module, you will observe that the examples provided with the KwikNet Porting Kit require no modification. Procedure *kn_osblock()* spins in the KwikNet Task waiting for a software flag to be set by *kn_osblock()*, indicating that the event of interest has occurred.

The procedures are included in the OS interface so that you can adapt procedure *kn_osblock()* to perform other essential, application dependent operations which must continue, even while KwikNet is spinning waiting for the event.

Purpose

kn_[un]locknet() **- Lock/Unlock a Network Resource**
kn_[un]lockmem() **- Lock/Unlock Memory Allocation Services**
kn_[un]lockfs() **- Lock/Unlock File System Access**

Caller

These procedures are only called in the context of a task, either by the KwikNet Task or by another task which has invoked some KwikNet service.

KwikNet calls procedure *kn_oslocknet()* to reserve network resources for the exclusive use of the currently executing task while critical network operations are performed. When the operations are complete, KwikNet calls procedure *kn_osunlocknet()* to release the network resources.

Memory allocation procedures *kn_osmemget()* and *kn_osmemrls()* can call procedure *kn_oslockmem()* to ensure exclusive use of these procedures by the currently executing task. When the memory block is allocated or freed, procedure *kn_osunlockmem()* must be called to release the lock.

The KwikNet Universal File System (UFS) file system interface calls procedure *kn_oslockfs()*, when necessary, to ensure exclusive use of a particular file system service by the currently executing task. Note that only file services accessed through the UFS are locked in this manner. When the file system operation is complete, the UFS calls procedure *kn_osunlockfs()* to release the lock.

Setup

Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_oslocknet(void);      void kn_osunlocknet(void);
void kn_oslockmem(void);      void kn_osunlockmem(void);
void kn_oslockfs(void);       void kn_osunlockfs(void);
```

Description

A detailed description of the operation of these resource locking procedures is provided in Chapter 2.2 under the topic "Resource Locking".

Network resource locking must always be supported.

Memory locking is only required if your memory allocation services are not thread-safe. To enable memory locking, use the KwikNet Configuration Builder to edit your KwikNet Library Parameter File and check the box labelled "Protect memory get/free operations" on the OS property page.

Symbol *KN_MEMLOCK* is defined in KwikNet header file *KN_LIB.H* to indicate that memory locking is disabled (*0*) or enabled (non-zero). Memory locking procedures *kn_oslockmem()* and *kn_osunlockmem()* are conditionally compiled so that they exist only if memory locking is required (*KN_MEMLOCK* is non-zero).

...more

Description ...continued

File access locking is only required if you are using a file system with file services that are not thread-safe. To enable file access locking, use the KwikNet Configuration Builder to edit your KwikNet Library Parameter File and check the box labelled "Protect file system services" on the File System property page.

Symbol *KN_FS_LOCK* is defined in KwikNet header file *KN_LIB.H* to indicate that file access locking is disabled (*0*) or enabled (non-zero). File access locking procedures *kn_oslockfs()* and *kn_osunlockfs()* are conditionally compiled so that they exist only if file access locking is required (*KN_FS_LOCK* is non-zero).

Returns Nothing

Multitasking Operation

These procedures must be implemented as described in Chapter 2.2 under the topic "Resource Locking".

Single Threaded Operation

These procedures are not present in the single threaded OS Interface Module.

This page left blank intentionally.

3. Target Processor and Compiler Use

3.1 Introduction

KwikNet C source files include header file *KN_LIB.H*, your KwikNet Library Configuration Module which defines the subset of KwikNet features required by your application. Any of your C source files which access KwikNet services must also include this header file.

KwikNet header file *KN_LIB.H* includes another header file which is both target processor and C compiler dependent. This file is the compiler header file *KNZZZCC.H* first introduced in Chapter 1.1 (see Figure 1.1-2). File *KNZZZCC.H* is referred to as the KwikNet compiler configuration header file.

> Porting Tip
>
> This long chapter provides a complete specification for the content of the KwikNet compiler header file *KNZZZCC.H*.
>
> Pick the file for your target processor and C compiler from the examples in installation directory *EXAMPLES\CC_H*. Little, if any, editing will be required. For other targets or compilers, pick the architecturally closest example as a starting point.

Compiler header file *KNZZZCC.H* serves several purposes. It identifies the specific features which your C compiler supports and, if necessary, provides alternatives to standard C usage. It also determines how critical section protection, device I/O operations, interrupt level manipulations and clock services are to be performed. The file can also include C or assembly language code fragments to optimize the execution speed of certain time-critical or frequently occurring KwikNet operations.

So how can this header file generate code fragments? The answer lies in the fact that the compiler configuration header file *KNZZZCC.H* is included more than once by some KwikNet C source modules. For example, KwikNet module *KN_UTIL.C* includes file *KNZZZCC.H* once (via *KN_LIB.H*) to get the usual kind of definitions which you expect to see in a header file. It then defines a symbol, such as *KN_CCNEED_SWAP*, and includes file *KNZZZCC.H* again. This time, the definitions within the file are ignored since they have already been included once. However, because symbol *KN_CCNEED_SWAP* has been defined, the fragment of code which implements a fast swapping function will be inserted into the compiled copy of module *KN_UTIL.C*. This technique permits the KwikNet fragments requiring customization to be collected into a single file which can be edited by you without a detailed understanding of the KwikNet framework.

The remainder of this chapter will describe how to edit the compiler configuration header file *KNZZZCC.H* to meet your requirements. Chapter 3.2 describes the edits required to specify features supported by your C compiler. Chapter 3.3 summarizes the processor and compiler dependent low level services which KwikNet requires and suggests methods for their implementation. Actual implementation examples are provided in Chapter 3.4.

The KwikNet compiler configuration header files *H_tttXXX.vvv* listed in Figure 3.1-1, although created and tested for use with KwikNet and the AMX Real-Time Multitasking Kernel, are but a few of those provided with the KwikNet Porting Kit as examples. The files will be found in installation directory *EXAMPLES\CC_H*.

The mnemonic *ttt* identifies the target processor family. The mnemonic *XXX* identifies the compiler vendor. The file extension *vvv* indicates the revision of the compiler with which the header file was first tested.

When using any of these files, copy the file and rename it *KNZZZCC.H*.

File	Target	Compiler
H__86MC.15	80x86 (real mode)	Microsoft 16-bit Visual C/C++ v1.5, v1.52
H__86PD.50	80x86 (real mode)	Paradigm 16-bit C/C++ v5.0, v6.0
H__86TC.50	80x86 (real mode)	Borland 16-bit C/C++ v5.0
H__86WC.110	80x86 (real mode)	WATCOM (Sybase) 16-bit C/C++ v11.0
H_386BCB.50	80x86 (protected mode)	Borland 32-bit C/C++ v5.0
H_386MCB.42	80x86 (protected mode)	Microsoft 32-bit Visual C/C++ v4.2 and up
H_386MWB.331	80x86 (protected mode)	MetaWare High C/C++ v3.31 and up
H_386PD.60	80x86 (protected mode)	Paradigm 32-bit C/C++ v6.0
H_386WC.110	80x86 (protected mode)	WATCOM (Sybase) 32-bit C/C++ v11.0
H_68KDA.42	68000	Diab-SDS C/C++ v4.2 and up
H_68KIM.842	68000	TASKING (Intermetrics) C/C++ v8.4.2
H_68KME.20	68000	Metrowerks C/C++ v2.0 and up
H_68KMR.45	68000	Mentor Graphics (Microtec) C/C++ v4.5G
H_CFDA.42	ColdFire	Diab-SDS C/C++ v4.2 and up
H_CFME.25	ColdFire	Metrowerks C/C++ v2.5
H_PPCDA.41	PowerPC	Diab-SDS C/C++ v4.1 and up
H_PPCME.42	PowerPC	Metrowerks C/C++ v4.2 and up
H_PPCMW.41	PowerPC	MetaWare High C/C++ v4.1 and up
H_ARMRM.211	ARM (ARM mode)	ARM Ltd. C/C++ SDK v2.11 and up
H_ARMMW.410	ARM (ARM mode)	MetaWare High C/C++ v4.1 and up
H_ARBRM.211	ARM (Thumb mode)	ARM Ltd. C/C++ SDK v2.11 and up
H_ARBMW.410	ARM (Thumb mode)	MetaWare High C/C++ v4.1 and up
H_M32MW.430	MIPS32	MetaWare High C/C++ v4.3e
H_BFAD.30	Blackfin	Analog Devices C/C++ v3.0

Figure 3.1-1 Compiler Configuration Header File Examples

3.2 C Compiler Adaptation

The KwikNet compiler configuration header file *KNZZZCC.H* must identify the specific features which your C compiler supports. Figure 3.2-1 illustrates such a specification for a typical, ANSI compliant C compiler.

Standard C Header Files

KwikNet assumes that a subset of the following standard C header files will be provided with your C compiler. Unless otherwise specified, KwikNet assumes that all of the standard C definitions needed for argument passing, string and memory manipulation and memory allocation will be available in the subset marked with the * character.

KN_HSTDARG	*stdarg.h*	* Standard arguments
KN_HSTDLIB	*stdlib.h*	* Standard library
KN_HSTRING	*string.h*	* String manipulation
KN_HMEMORY	*memory.h*	Memory manipulation
KN_HMEM	*mem.h*	Memory manipulation
KN_HMALLOC	*malloc.h*	Memory allocation
KN_HSTDIO	*stdio.h*	Standard I/O

You can override KwikNet's choice of C header files by defining symbol *KN_CCHDRTYPE* to identify the specific set of header files to be used. For example, if your C compiler does not provide file *stdarg.h* and puts its memory manipulation definitions in file *mem.h*, you must define *KN_CCHDRTYPE* as follows.

```
#define  KN_CCHDRTYPE   (KN_HSTDLIB | KN_HSTRING | KN_HMEM)
```

Parameter Passing Conventions

Some C compilers use a keyword such as *__cdecl* to define the parameter passing conventions which a C function must follow when interfacing with modules coded in other languages such as assembly language. If your C compiler uses such a keyword, define symbol *KN_CCPP* to be that keyword. Otherwise define *KN_CCPP* to be an empty string as in Figure 3.2-1.

If your C compiler allows variable length argument lists in C function declarations (using "*...*"), define symbol *KN_CCDOTS* to be *1*. Otherwise define *KN_CCDOTS* to be *0*.

Random Number Generator

If your C library includes the random number functions *rand()* and *srand()*, define symbol *KN_CCRAND* to be *1*. Otherwise define *KN_CCRAND* to be *0*. If *KN_CCRAND* is *0*, KwikNet will use its own primitive pseudo-random number generator.


```
#ifndef  KN_CCHDR_H
#define  KN_CCHDR_H     1

/* Compiler has memory functions in string.h                          */
/* Malloc is in stdlib.h.                                             */
/* Stdarg.h is available.                                             */
/* Therefore, use the default definition of available header types    */
/* and DO NOT define KN_CCHDRTYPE.                                    */

#define KN_CCPP                 /* Uses standard C parameter         */
                                /* passing convention                */
#define KN_CCDOTS       1       /* Can use "..." in prototypes       */
#define KN_CCRAND       1       /* rand(), srand(seed) ARE available */


/* Macro to get rid of "unused argument" warnings.                    */
/* With compilers that can suppress these warnings, define as empty.  */
/* #define USE_ARG(x) ((void)(x))                                     */
#define USE_ARG(x)

/* Turn off nonsensical warnings.                                     */
/* Because of the many supported configurations and options           */
/* some warnings cannot be avoided.                                   */
/* --- none ---                                                       */

/* Far and huge keywords not supported                                */
#define KN_CCFAR
#define KN_CCHUGE

#endif   /* KN_CCHDR_H */

/* File I/O Definitions                                               */
#ifdef   KN_CCNEEDFILE
#ifndef  KN_CCFILE_H
#define  KN_CCFILE_H    1
#include <stdio.h>
#endif /* KN_CCFILE_H */
#endif /* KN_CCNEEDFILE */
```

Figure 3.2-1 C Compiler Adaptations

Eliminating Warnings

Many C compilers generate warnings if C code is not absolutely pristine. Others produce warnings because of shortcomings in the compiler. If possible, use your compiler's command line switch to avoid such warnings. Alternatively, use the *#pragma* statement, if any, to eliminate the warning.

For example, the Borland (Inprise) C compiler generates a warning if a structure is referenced before it is defined. Such references are common in structures which are linked to each other. To avoid this warning, the following Borland specific statement can be inserted into the compiler configuration header file *KNZZZCC.H*.

```
#pragma warn -stu
```

The most common warning is the reminder that a parameter passed to a function is not actually used by the function. There are times when such functions cannot be avoided without paying an execution penalty that is otherwise unwarranted. If possible, use your compiler's command line switch to avoid such warnings. Alternatively, insert the *#pragma* statement, if any, which eliminates the warning. In either of these cases, define macro *USE_ARG* to be an empty string. If neither of these solutions is possible, define the macro *USE_ARG* as shown in Figure 3.2-1 to force all arguments to actually be used.

Segmented Memory Access

When using the 16-bit Intel 80x86 processor with its segmented memory architecture, there is difficulty allocating and accessing memory arrays which exceed 64K bytes. Compilers overcome these hurdles by introducing keywords (such as *_far* and *_huge*) to be used as modifiers for pointer references and array allocation.

If you are using KwikNet on the 16-bit Intel 80x86 processor operating in real mode, define symbol *KN_CCFAR* and *KN_CCHUGE* to be the appropriate keyword used by your compiler. Otherwise define both *KN_CCFAR* and *KN_CCHUGE* to be an empty string as shown in Figure 3.2-1.

Interrupt Function Definitions

If your compiler permits an interrupt service routine (ISR) to be coded in C, then you should provide compiler dependent definitions in header file *KNZZZCC.H* similar to those illustrated below for use by your OS Interface Module *KN_OSIF.C*.

```
typedef void _interrupt (*KN_CCINTFNP)(void);
#define KN_CCINTFUNC(isrname) void _interrupt isrname(void)
```

File I/O Definitions

If you are using a KwikNet option such as FTP which requires a file system, you may be using the file services from your C library. If so, KwikNet will define symbol *KN_CCNEEDFILE* and include your compiler configuration header file *KNZZZCC.H* to gain access to your C file I/O definitions, usually located in C header file *stdio.h*. Note that the inclusion of your file I/O definitions must be done outside the header definition region bounded by the *KN_CCHDR_H* definition.

3.3 Low Level Services

The KwikNet compiler configuration header file *KNZZZCC.H* must specify how critical section protection, device I/O operations, interrupt level manipulations and clock services are to be performed. It can also introduce C or assembly language code fragments to optimize the execution speed of certain time-critical or frequently occurring KwikNet operations.

The low level services can be implemented using any of the following techniques. Examples of each technique are provided in Chapter 3.4.

- C macros which use in-line assembly language statements
- C functions coded in C with in-line assembly language statements
- C macros and equivalent C library macros or functions
- C functions coded in C using only C language statements

Figure 3.3-1 illustrates one possible specification of the low level services required by KwikNet. **Control symbols** of the form *KN_FN_xxxx* are defined in the definitions region of the KwikNet compiler configuration header file *KNZZZCC.H* following the C compiler adaptation parameters. Each control symbol definition specifies how a particular low level service or set of services is to be implemented.

If a control symbol is defined to be *KN_AS_KNMACRO*, then the low level service will be implemented for you by KwikNet using a C macro definition.

If a control symbol is defined to be *KN_AS_KNFUNC*, then the low level service will be implemented for you by KwikNet as a C function coded entirely in C.

If a control symbol is defined to be *KN_AS_FNCC|KN_AS_FNPROTO*, then the low level service must be implemented by you as a function (or its equivalent) in header file *KNZZZCC.H* using any of the techniques listed above and described in this chapter. The function prototype will be provided for you by KwikNet.

Some C compilers allow you to declare an in-line assembly language macro which looks like a function, feels like a function but is not actually a function. The definition of such a macro provides the prototype and the function, all in one. When declaring such a function, the control symbol must be defined as *KN_AS_FNCC*. Since the constant *KN_AS_FNPROTO* is omitted from the definition, KwikNet will not generate a function prototype, thereby avoiding any possible conflict with your in-line definition.

If you provide your own implementation of a low level service as a macro or as a function complete with its own prototype, the associated control symbol must be defined to have the value *0*. Example 3.4.1-A illustrates this requirement.

> Note
>
> The definition of the critical section control symbol *KN_FN_CRIT* must be done outside the usual header definition region bounded by the *KN_CCHDR_H* definition. Note that the definition is only required if symbol *KN_CCNEED_CRITDEF* has been defined by KwikNet.

```
#ifndef  KN_CCHDR_H
#define  KN_CCHDR_H     1
/* :                                                                  */
/* : Compiler definitions (see Figure 3.2-1)                          */
/* :                                                                  */


/* ------------------------------------------------------------------ */
/* Low Level Services                                                 */

/* Endian swapping                                                    */
#define KN_FN_SWAP      (KN_AS_FNCC | KN_AS_FNPROTO)

/* IP checksum generation                                             */
#define KN_FN_CKSUM     (KN_AS_FNCC | KN_AS_FNPROTO)

/* Interrupt level manipulation                                       */
#define KN_FN_INTSUPP   (KN_AS_FNCC | KN_AS_FNPROTO)

/* Device I/O operations                                              */
#define KN_FN_MEMIO     KN_AS_KNMACRO
#define KN_FN_MEMREP    KN_AS_KNMACRO

/* Clock read must be atomic                                          */
#define KN_FN_CLKRD     (KN_AS_FNCC | KN_AS_FNPROTO)

#endif   /* KN_CCHDR_H */


/* ------------------------------------------------------------------ */
/* Critical Section Services                                          */

#ifdef   KN_CCNEED_CRITDEF
#ifndef  KN_CCCRITDEF_H
#define  KN_CCCRITDEF_H 1

/* Define critical section protection mechanism                       */
#define KN_FN_CRIT      (KN_AS_FNCC | KN_AS_FNPROTO)

#endif /* KN_CCCRITDEF_H */
#endif /* KN_CCNEED_CRITDEF */
```

Figure 3.3-1 Specifying Low Level Services

The following low level KwikNet services are governed by control symbols *KN_FN_CRIT* and *KN_FN_INTSUPP*. Since these services are target dependent, you must provide them. There are no defaults provided by KwikNet. However, examples are provided for several different target processors and compilers.

KN_FN_CRIT	*kn_csenter()*	Enter into a critical section
KN_FN_CRIT	*kn_csexit()*	Exit from a critical section
KN_FN_INTSUPP	*kn_brdintlvl()*	Change interrupt priority level

The following low level KwikNet services are optional. You only need to provide them if you wish to optimize their performance. They are governed by control symbols *KN_FN_SWAP* and *KN_FN_CKSUM*. The KwikNet defaults for these services can be used.

KN_FN_SWAP	*kn_swap16()*	Reverse byte order in a 16-bit unsigned integer
KN_FN_SWAP	*kn_swap32()*	Reverse byte order in a 32-bit unsigned integer
KN_FN_CKSUM	*kn_cksum()*	Compute IP datagram checksum

The following low level KwikNet device I/O services must be provided. However, you may be able to use the default services provided by KwikNet. The services are governed by control symbols *KN_FN_MEMxxx* and *KN_FN_DVCxxx*.

If you are using a processor which uses memory mapped I/O addressing, you must provide the memory mapped input/output services governed by control symbols *KN_FN_MEMIO* and *KN_FN_MEMREP*. You can leave control symbols *KN_FN_DVCIO* and *KN_FN_DVCREP* undefined or defined as *0*.

KN_FN_MEMIO	*kn_inmNN()*	8, 16 and 32-bit read from device memory
KN_FN_MEMIO	*kn_outmNN()*	8, 16 and 32-bit write to device memory
KN_FN_MEMREP	*kn_inmsNN()*	8, 16 and 32-bit block read from device memory
KN_FN_MEMREP	*kn_outmsNN()*	8, 16 and 32-bit block write to device memory

If you are using a processor like the Intel 80x86 which uses I/O ports for device addressing, you must provide the port input/output services governed by control symbols *KN_FN_DVCIO* and *KN_FN_DVCREP*. It is recommended that these services be implemented as shown in the example files *H__86xxx.vvv* or *H_386xxx.vvv* listed in Figure 3.1-1. You can leave control symbols *KN_FN_MEMIO* and *KN_FN_MEMREP* undefined or defined as *0*.

KN_FN_DVCIO	*kn_inpNN()*	8, 16 and 32-bit read from device I/O port
KN_FN_DVCIO	*kn_outpNN()*	8, 16 and 32-bit write to device I/O port
KN_FN_DVCREP	*kn_inpsNN()*	8, 16 and 32-bit block read from device I/O port
KN_FN_DVCREP	*kn_outpsNN()*	8, 16 and 32-bit block write to device I/O port

The following low level KwikNet clock services may have to be provided to handle the atomic (indivisible) manipulation of 32-bit clock tick counts. They are governed by control symbols *KN_FN_CLKRD* and *KN_FN_CLKDIFF*. The KwikNet defaults for these services can be used on most 32-bit processors.

KN_FN_CLKRD	*kn_tickrd()*	Read a 32-bit clock tick count
KN_FN_CLKDIFF	*kn_tickdiff()*	Compute a 32-bit clock tick difference

Critical Section Protection

From time to time, KwikNet must perform a sequence of operations which must appear to be done as one indivisible operation. Such a sequence is called a critical section. KwikNet invokes the low level service *kn_csenter()* as it enters a critical section. It then invokes *kn_csexit()* as it leaves the region.

These low level services must allow recursion so that KwikNet can enter one critical section from another and remain critical until it finally exits from the first region. On most target processors, this requirement can be most easily met by disabling interrupts on entry to the critical section and restoring interrupts to their previous state upon exit.

These low level services can be implemented as macros or functions. If possible, they should be implemented as in-line assembly language macros for best performance. If implemented as C functions, they would be prototyped as follows:

```
unsigned int kn_csenter(void);
void kn_csexit(unsigned int);
```

Service *kn_csenter()* always returns a parameter which is subsequently passed to *kn_csexit()*. Usually the parameter is the previous state of the interrupt system. However, the parameter and its purpose is up to you.

You must define symbol *KN_FN_CRIT* to specify which of the techniques described in Chapter 3.4 was used to implement these services.

Using RTOS Critical Section Protection

If you are using an RTOS which provides critical section protection, you may be able to use the RTOS services instead of implementing your own.

You may have to implement the KwikNet critical section services as functions which make the appropriate calls to your RTOS. In this case, define symbol *KN_FN_CRIT* as in Figure 3.3-1 and code the critical section services as functions using the technique illustrated in Example 3.4.2-B (see Chapter 3.4).

In other cases, you may be able to map the KwikNet critical section services directly to those of your RTOS as in the following example. Symbol *KN_FN_CRIT* must be defined as *0* since low level service functions are not being implemented and KwikNet prototypes are not required since the services have been defined using macros. The following definitions should replace the definition of symbol *KN_FN_CRIT* in the critical section definition region of header file *KNZZZCC.H*.

```
extern short rtos_critical(short);
#define kn_csenter()          ((unsigned int)rtos_critical(0))
#define kn_csexit(prevstate)  ((void)rtos_critical(prevstate))
#define KN_FN_CRIT 0
```


Interrupt Priority Level Manipulation

From time to time, KwikNet must perform a sequence of operations which must be done as one indivisible operation without the possibility of a specific device interrupt. For example, the KwikNet board driver *KN_BOARD.C* includes a function which must install a pointer to an interrupt service procedure into the processor's interrupt or exception vector table. Such an installation must be done without an interrupt from the device.

KwikNet invokes the low level service *kn_brdintlvl()* as it enters and leaves one of these protected interrupt regions. This service must allow recursion so that KwikNet can enter one such region from another and remain uninterrupted until it finally exits from the first region. On most target processors, this requirement can be most easily met by disabling interrupts on entry to the region and restoring interrupts to their previous state upon exit. On some target processors, such as the Motorola 68000 family, you can gain even better control by inhibiting only device interrupts above a particular priority level.

This low level service can be implemented as a macro or function. If possible, it should be implemented as an in-line assembly language macro for best performance. If implemented as a C function, it would be prototyped as follows:

```
unsigned long kn_brdintlvl(unsigned long p);
```

Service *kn_brdintlvl()* receives a single parameter *p* which specifies the required interrupt state. If *p* is *0*, the service must unconditionally disable interrupts from all sources. If *p* is *~0*, the service must unconditionally enable interrupts from all sources. If *p* is any other value, the service must restore interrupts to the state specified by *p*.

Service *kn_brdintlvl()* always returns a parameter which specifies the state of the interrupt system prior to its invocation. The parameter values *0* and *~0* are reserved for use as described above. Any other values can be used by you to specify the prior state of the interrupt system.

You must define symbol *KN_FN_INTSUPP* to specify which of the techniques described in Chapter 3.4 was used to implement this service.

End-for-End Byte Swapping

KwikNet must adjust 16-bit and 32-bit big endian network values to match the endian characteristics of your target processor. Low level services must be provided to reverse the order of the bytes in a 16-bit or 32-bit unsigned value.

These low level services can be implemented as macros or functions. To use the default C macro versions defined by KwikNet in header file *KN_API.H*, simply define symbol *KN_FN_SWAP* as follows:

```
#define KN_FN_SWAP    KN_AS_KNMACRO
```

If your C compiler cannot handle these simple KwikNet macros, you can use the default C functions defined by KwikNet in header file *KN_API.H* and implemented in file *KN_UTIL.C*. Simply define symbol *KN_FN_SWAP* as follows:

```
#define KN_FN_SWAP    KN_AS_KNFUNC
```

However, for best performance, the low level byte swapping services should be implemented as in-line assembly language macros. If implemented as C functions, these services would be prototyped as follows:

```
unsigned short kn_swap16(unsigned short val);
unsigned long kn_swap32(unsigned long val);
```

Parameter *val* is the 16-bit or 32-bit value which is to be end-for-end byte swapped.

The 16-bit parameter *val* is returned from *kn_swap16()* with *val[0..7]* and *val[8..15]* interchanged.

The 32-bit parameter *val* is returned from *kn_swap32()* with *val[0..7]* interchanged with *val[24..31]* and *val[8..15]* interchanged with *val[16..23]*.

You must define symbol *KN_FN_SWAP* to specify which of the techniques described in Chapter 3.4 was used to implement these services.

IP Checksum Calculation

KwikNet must calculate the checksum for every IP datagram which it manipulates. Optimization of the low level service which performs this operation can greatly enhance performance.

This low level service can be implemented as a macro or function. Rarely is a macro warranted, since the complexity of the algorithm and its frequency of use would adversely affect code size. To use the default C function version defined by KwikNet in header file *KN_API.H* and implemented in file *KN_UTIL.C*, simply define symbol *KN_FN_CKSUM* as follows:

```
#define KN_FN_CKSUM  KN_AS_KNFUNC
```

However, for best performance, this low level IP datagram checksum algorithm should be implemented using in-line assembly language within a C function. See any of the examples provided in the compiler configuration header files *KNtttCC.xxx* listed in Figure 3.1-1. If implemented as a C function, it would be prototyped as follows:

```
unsigned short kn_cksum(void *p, unsigned int n);
```

The checksum service *kn_cksum()* returns the 16-bit checksum of the *n* 16-bit integer values from the memory array referenced by pointer *p*. The checksum is computed by summing the *n* 16-bit integers from the memory array using one's complement arithmetic and then returning the one's complement of the sum.

An interesting characteristic of this algorithm is that it is endian independent. This fact can be used to advantage when implementing this low level service. Examine any of the in-line assembly language implementations in the *KNtttCC.xxx* header files or review the C language implementation in file *KN_UTIL.C*.

You must define symbol *KN_FN_CKSUM* to specify which of the techniques described in Chapter 3.4 was used to implement this service.

Memory Mapped Device I/O

If you are using a processor which uses memory mapped I/O addressing, you must provide memory mapped versions of the low level KwikNet device I/O services. You can leave the I/O device port symbols *KN_FN_DVCIO* and *KN_FN_DVCREP* undefined or defined as *0*.

If your C compiler supports the keyword *volatile*, you can use the default C macro versions defined by KwikNet in header file *KN_DVCIO.H*. Simply define symbols *KN_FN_MEMIO* and *KN_FN_MEMREP* as follows:

```
#define KN_MEMIO      KN_AS_KNMACRO
#define KN_MEMREP     KN_AS_KNMACRO
```

Alternatively, you can use the default C function versions defined by KwikNet in header file *KN_DVCIO.H* and implemented as C functions in the KwikNet board driver *KN_BOARD.C*. Simply define symbols *KN_FN_MEMIO* and *KN_FN_MEMREP* as follows:

```
#define KN_MEMIO      KN_AS_KNFUNC
#define KN_MEMREP     KN_AS_KNFUNC
```

When implemented as C functions, these low level services are prototyped as follows:

```
unsigned char  kn_inm8(unsigned long addr);
unsigned short kn_inm16(unsigned long addr);
unsigned long  kn_inm32(unsigned long addr);
void           kn_outm8(unsigned long addr, unsigned char val);
void           kn_outm16(unsigned long addr, unsigned short val);
void           kn_outm32(unsigned long addr, unsigned long val);

void           kn_inms8(unsigned long addr, void *valp, int n);
void           kn_inms16(unsigned long addr, void *valp, int n);
void           kn_inms32(unsigned long addr, void *valp, int n);
void           kn_outms8(unsigned long addr, void *valp, int n);
void           kn_outms16(unsigned long addr, void *valp, int n);
void           kn_outms32(unsigned long addr, void *valp, int n);
```

Parameter *addr* is the device's linear memory address. Parameter *val* is an unsigned 8, 16 or 32-bit value to be written to the device.

For the block read services, parameter *valp* is a pointer to storage for an array of *n* unsigned 8, 16 or 32-bit values to be read from the device. For the block write services, parameter *valp* is a pointer to an array of *n* unsigned 8, 16 or 32-bit values to be written to the device.

If you wish, you can implement these low level services as macros or functions which use in-line assembly language for best performance. In this case, you must define symbols *KN_FN_MEMIO* and *KN_FN_MEMREP* to specify which of the techniques described in Chapter 3.4 was used to implement the services.

Device Port I/O

If you are using a processor like the Intel 80x86 which uses I/O ports for device addressing, you must provide the low level port input/output services. You can leave memory mapped device I/O symbols *KN_FN_MEMIO* and *KN_FN_MEMREP* undefined or defined as *0*.

These low level services can be implemented as macros or functions. It is recommended that you implement these services as shown in any of the 80x86 example files *H__86xxx.vvv* or *H_386xxx.vvv* listed in Figure 3.1-1.

If implemented as C functions, these low level services are prototyped as follows:

```
unsigned char  kn_inp8(unsigned int port);
unsigned short kn_inp16(unsigned int port);
unsigned long  kn_inp32(unsigned int port);
void           kn_outp8(unsigned int port, unsigned char val);
void           kn_outp16(unsigned int port, unsigned short val);
void           kn_outp32(unsigned int port, unsigned long val);

void           kn_inps8(unsigned int port, void *valp, int n);
void           kn_inps16(unsigned int port, void *valp, int n);
void           kn_inps32(unsigned int port, void *valp, int n);
void           kn_outps8(unsigned int port, void *valp, int n);
void           kn_outps16(unsigned int port, void *valp, int n);
void           kn_outps32(unsigned int port, void *valp, int n);
```

Parameter *port* is the device's I/O port address. Parameter *val* is an unsigned 8, 16 or 32-bit value to be written to the device port.

For the block read services, parameter *valp* is a pointer to storage for an array of *n* unsigned 8, 16 or 32-bit values to be read from the device port. For the block write services, parameter *valp* is a pointer to an array of *n* unsigned 8, 16 or 32-bit values to be written to the device port.

You must define symbols *KN_FN_DVCIO* and *KN_FN_DVCREP* to specify which of the techniques described in Chapter 3.4 was used to implement the services.

Clock Services

KwikNet maintains a clock tick counter which it uses for elapsed time monitoring. The count is a 32-bit unsigned long integer value stored in public variable *kn_ticks*. KwikNet must be able to read this 32-bit value atomically, even on systems with a 16-bit memory system or target processor. KwikNet must be also be able to compare this 32-bit value with another 32-bit value.

Reading Clock Tick Count

If your C compiler **can generate code** to fetch the tick count from variable *kn_ticks* with a single, indivisible memory read, then KwikNet's tick access requirements will be met. This will be the case for most 32-bit target processors. For such a C compiler, simply leave symbol *KN_FN_CLKRD* undefined.

If your C compiler **cannot generate code** to fetch the tick count from variable *kn_ticks* with a single, indivisible memory read, then you must provide a low level service *kn_tickrd()* to perform the operation. This will be the case for most 16-bit target processors. The low level service can be implemented as a macro or function. To use the default C function version defined by KwikNet in header file *KN_API.H* and implemented in file *KN_UTIL.C*, simply define symbol *KN_FN_CLKRD* as follows:

```
#define KN_FN_CLKRD  KN_AS_KNFUNC
```

However, for best performance, the low level tick read service should be implemented as an in-line assembly language macro. If implemented as a C function, the service would be prototyped as follows:

```
unsigned long kn_tickrd(void);
```

The service *kn_tickrd()* must return the 32-bit unsigned long integer from the KwikNet public variable *kn_ticks*. If the service guards the access to *kn_ticks* by disabling interrupts, it must not unconditionally enable interrupts upon exit. Instead, it must restore interrupts to the state which existed prior to entry to the service.

You must define symbol *KN_FN_CLKRD* to specify which of the techniques described in Chapter 3.4 was used to implement this service.

Clock Tick Difference Computation

If your C compiler **can correctly evaluate** the following expression where *x* and *y* are unsigned long values, then KwikNet's tick differencing requirements will be met. This will be the case for most C compilers. For such a C compiler, simply leave symbol *KN_FN_CLKDIFF* undefined.

```
(((long)(x - y)) <= 0L)
```

If your C compiler **cannot correctly evaluate** the above expression, then you must provide a low level service *kn_tickdiff()* to perform the tick differencing operation. The low level service can be implemented as a macro or function. To use the default C function version defined by KwikNet in header file *KN_API.H* and implemented in file *KN_UTIL.C*, simply define symbol *KN_FN_CLKDIFF* as follows:

```
#define KN_FN_CLKDIFF   KN_AS_KNFUNC
```

However, for best performance, the low level tick difference service should be implemented as an in-line assembly language macro. If implemented as a C function, the service would be prototyped as follows:

```
long kn_tickdiff(unsigned long nticks);
```

The service *kn_tickdiff()* must read the 32-bit unsigned long integer from the KwikNet public variable *kn_ticks*, subtract the unsigned long parameter value *nticks* and return the difference as a signed, long integer value. If the service guards the access to *kn_ticks* by disabling interrupts, it must not unconditionally enable interrupts upon exit. Instead, it must restore interrupts to the state which existed prior to entry to the service.

You must define symbol *KN_FN_CLKDIFF* to specify which of the techniques described in Chapter 3.4 was used to implement this service.

3.4 Code Fragment Implementation

The KwikNet compiler configuration header file *KNZZZCC.H* must specify how critical section protection, device I/O operations, interrupt level manipulations and clock services are to be performed. These services and their C function prototypes were introduced in Chapter 3.3.

The low level services can be implemented using any of the following techniques.

- C macros which use in-line assembly language statements
- C functions coded in C with in-line assembly language statements
- C macros and equivalent C library macros or functions
- C functions coded in C using only C language statements

If performance is not an issue, the easiest solution is to use the default C macro or function provided by KwikNet for the service. There are, however, a few services which must be implemented by you. Even for these, you should follow the examples provided in one of the compiler configuration header files listed in Figure 3.1-1.

The implementation technique to be used will be influenced by a number of factors. For best performance, use in-line assembly language and implement the service as a macro or function. Use a macro if the service involves very little in-line code. Use a function for a service such as the IP datagram checksum calculation which, although too long to be suitable for in-line use, still warrants assembly language implementation for performance.

Unfortunately, your choice of implementation technique will ultimately depend on the compilation features provided by your C compiler.

In the remainder of this chapter, the various techniques will be explored. For each technique, one or more of the low level services described in Chapter 3.3 will be implemented using the technique. In each case, the example will be based on a particular C compiler and target processor.

> Note
>
> Each low level service can be implemented using the technique best suited for that service. All services do not have to be implemented using the same technique.

3.4.1 C Macro Using In-Line Assembly Language

If your C compiler supports in-line assembly language code fragments, many of the frequently used, low level services can benefit. This technique can only be used if your C compiler allows in-line functions to receive parameters and return values.

Example 3.4.1-A

The following example illustrates the implementation of the critical section services using Diab-SDS C/C++ on a Motorola M68000 processor. Note that this code fragment resides in its own definition region bounded by the *KN_CCNEED_CRITDEF* definition as previously illustrated in Figure 3.3-1.

The compiler's *asm* directive is used to prototype the in-line function and to allow its definition using assembly language. Because the prototype has been provided and an instance of the function will not actually be generated, the KwikNet symbol *KN_FN_CRIT* is defined to be *0*. KwikNet will not generate a conflicting function prototype and will not try to force an instance of the function.

Every reference to these low level services will yield an in-line expansion of the function. Function *kn_csenter()* returns the previous interrupt state in register *d0* as specified by the Diab-SDS C to assembly language interface. Note that the function *kn_csexit()* has two specifications: one for use if parameter *p* is in a register and one for use if parameter *p* is a memory variable.

```
/* Example: Diab-SDS C compiler for the M68000 processor                */
/* -------------------------------------------------------------------- */
/* Low level critical section services                                  */

#ifdef    KN_CCNEED_CRITDEF
#ifndef   KN_CCCRITDEF_H
#define   KN_CCCRITDEF_H 1

asm unsigned int kn_csenter(void)
{
          move.w    sr,d0
          ori.w     #0x0700,sr
}

asm void kn_csexit(unsigned int p)
{
% reg p;
          move.w    p,sr
% mem p;
          move.l    p,d0
          move.w    d0,sr
}

#define   KN_FN_CRIT 0

#endif    /* KN_CCCRITDEF_H */
#endif    /* KN_CCNEED_CRITDEF */
```

Example 3.4.1-B

The following example illustrates the implementation of the low level services for clock tick manipulation using WATCOM (Sybase) C/C++ on an Intel 80x86 processor operating in real mode. Note that the macro definitions must reside within the definition region of header file *KNZZZCC.H*.

The compiler's *#pragma aux* directive is used to prototype the in-line functions and to allow their definition using assembly language. Because the prototype has been provided and an instance of each function will not actually be generated, the KwikNet symbols *KN_FN_CLKRD* and *KN_FN_CLKDIFF* are defined to be *0*. KwikNet will not generate conflicting function prototypes and will not try to force an instance of the functions.

Every reference to these low level services will yield an in-line expansion of the function. Function *kn_tickrd()* returns the value of variable *kn_ticks* in register pair *DX:AX* as specified by the WATCOM C to assembly language interface. Function *kn_tickdiff()* receives its parameter *nticks* in register pair *CX:BX* and subtracts it from the value of variable *kn_ticks*. The result is returned in register pair *DX:AX*.

```
/* Example: WATCOM C compiler for the real mode 80x86 processor             */
/* ------------------------------------------------------------------       */
#ifndef   KN_CCHDR_H
#define   KN_CCHDR_H 1
/* :                                                                        */
/* : Compiler definitions (see Figure 3.2-1)                                */
/* :                                                                        */
/* ------------------------------------------------------------------       */
/* Low level clock tick manipulation services                               */

extern unsigned long kn_ticks;
unsigned long kn_tickrd(void);
long kn_tickdiff(unsigned long);

#pragma aux kn_tickrd =                                                    \
          "pushf",                                                         \
          "cli",                                                           \
          "mov      ax,word ptr kn_ticks",                                 \
          "mov      dx,word ptr kn_ticks+2",                               \
          "popf";             /* Return result in DX:AX */

#pragma aux kn_tickdiff =                                                  \
          "pushf",                                                         \
          "cli",                                                           \
          "mov      ax,word ptr kn_ticks",                                 \
          "mov      dx,word ptr kn_ticks+2",                               \
          "popf",                                                          \
          "sub      ax,bx",                                                \
          "sbb      dx,cx"                                                 \
          parm [cx bx];       /* Return result in DX:AX */

#define   KN_FN_CLKRD         0
#define   KN_FN_CLKDIFF       0

#endif    /* KN_CCHDR_H */
```


Example 3.4.1-C

The following example illustrates the implementation of the low level service for interrupt priority manipulation using WATCOM (Sybase) C/C++ on an Intel 80x86 processor operating in real mode.

The compiler's *#pragma aux* directive is used to prototype the in-line function and to allow its definition using assembly language. Because the prototype has been provided and an instance of the function will not actually be generated, the KwikNet symbol *KN_FN_INTSUPP* is defined to be *0*. KwikNet will not generate a conflicting function prototype and will not try to force an instance of the function.

Every reference to this low level service will yield an in-line expansion of the function. Function *kn_brdintlvl()* receives its parameter *p* in register pair *DX:AX*. The result is also returned in register pair *DX:AX* as specified by the WATCOM C to assembly language interface.

```
/* Example: WATCOM C compiler for the real mode 80x86 processor               */
/* ------------------------------------------------------------------         */
#ifndef   KN_CCHDR_H
#define   KN_CCHDR_H 1
/* :                                                                          */
/* : Compiler definitions (see Figure 3.2-1)                                  */
/* :                                                                          */
/* ------------------------------------------------------------------         */
/* Low level interrupt priority manipulation service                          */

unsigned long kn_brdintlvl(unsigned long);

#pragma aux kn_brdintlvl =                                                    \
"         sub       dx,dx",                                                   \
"         pushf",                                                             \
"         test      ah,2",           /* Test required IF state */           \
"         pop       ax",             /* DX:AX = previous PSW = result */    \
"         jz        short ldis",                                            \
"         sti",                      /* Enable interrupts */                \
"         jmp       short lexit",                                           \
"ldis:    cli",                      /* Disable interrupts */               \
"lexit:"                                                                      \
          parm [dx ax];              /* Return result in DX:AX */

#define   KN_FN_INTSUPP 0

#endif    /* KN_CCHDR_H */
```

3.4.2 C Functions Coded in Assembly Language

You can use this technique if your C compiler allows you to implement a C function which, although called from C, is actually coded in assembly language. For many compilers, this is the only technique you can use to create an assembly language function which receives parameters and returns a value.

You can use this technique even if your C function will not use assembly language at all. For example, the low level C function may be completely coded in C but make use of non-standard C macros or functions provided with your C compiler to manipulate registers within your target processor.

Example 3.4.2-A

The following example illustrates the implementation of the byte swapping services using Borland (Inprise) C/C++ on an Intel 80x86 processor operating in protected mode.

There are two parts required: the definition of symbol *KN_FN_SWAP* and the assembly language implementation of the C callable functions.

You must define the KwikNet symbol *KN_FN_SWAP* within the definition region of header file *KNZZZCC.H* as follows:

```
/* Example: Borland C compiler for the protected mode 80x86 processor         */
/* ------------------------------------------------------------------         */
#ifndef   KN_CCHDR_H
#define   KN_CCHDR_H 1
/* :                                                                          */
/* : Compiler definitions (see Figure 3.2-1)                                  */
/* :                                                                          */

/* ------------------------------------------------------------------         */
/* Low level endian swapping services                                         */

#define KN_FN_SWAP  (KN_AS_FNCC | KN_AS_FNPROTO)

#endif    /* KN_CCHDR_H */
```

The definition states that the swapping services are to be implemented in header file *KNZZZCC.H* using assembly language code fragments. It also states that KwikNet must provide its standard C function prototypes for these services. If you provide your own custom compiler dependent prototypes for the low level service functions, be sure to omit *KN_AS_FNPROTO* from your definition of *KN_FN_SWAP* so that the corresponding KwikNet prototypes will be omitted, thereby avoiding possible conflicts.

Following the definition region in header file *KNZZZCC.H* you must implement the low level services as C functions coded in assembly language.

Since your definition of *KN_FN_SWAP* indicates that your byte swapping services are available in a code fragment, KwikNet defines symbol *KN_CCNEED_SWAP* and includes header file *KNZZZCC.H* to generate one instance of the code fragment in the appropriate KwikNet source module. In this example, the code fragment from header file *KNZZZCC.H* is actually inserted into the source file *KN_UTIL.C*.

Example 3.4.2-A (continued)

Note that this code fragment must reside in its own definition region bounded by the KwikNet *KN_CCNEED_SWAP* definition as shown in the example below.

The compiler's *asm* directive is used to code the body of the C functions using assembly language. Function *kn_swap16()* returns the 16-bit result in register *AX* as specified by the Borland C to assembly language interface. Function *kn_swap32()* returns the 32-bit result in register *EAX*. Note that the Borland compiler allows these functions to reference parameter *p* from within the assembly language code fragment. Also note that the Borland pragma *warn* has been used to avoid erroneous warnings that these functions fail to return a value.

```
/* Example: Borland C compiler for the protected mode 80x86 processor         */
/* ------------------------------------------------------------------         */
/* Low level endian swapping functions                                        */
/* This code sequence is included once within module KN_UTIL.C                */

#ifdef    KN_CCNEED_SWAP
#ifndef   KN_CCSWAP_H
#define   KN_CCSWAP_H             1

/* Prevent warning about function not returning a value                       */
#pragma warn -rvl


unsigned short kn_swap16(unsigned short p)
{
          asm {
          movzx     eax,p
          xchg      al,ah
          }
}         /* Return result in AX */


unsigned long kn_swap32(unsigned long p)
{
          asm {
          mov       eax,p
          xchg      al,ah
          ror       eax,16
          xchg      al,ah
          }
}         /* Return result in EAX */


/* Restore warning state to its previous condition                            */
#pragma warn .rvl

#endif /* KN_CCSWAP_H */
#endif /* KN_CCNEED_SWAP */
```

Example 3.4.2-B

The following example illustrates the implementation of the critical section services using Microsoft C/C++ on an Intel 80x86 processor operating in real mode.

There are two parts required: the definition of symbol *KN_FN_CRIT* and the assembly language implementation of the C callable functions.

You must define the KwikNet symbol *KN_FN_CRIT* within the header file *KNZZZCC.H* as shown below. Note that the definition must reside in its own definition region bounded by the *KN_CCNEED_CRITDEF* definition as previously illustrated in Figure 3.3-1.

```
/* Example: Microsoft C compiler for the real mode 80x86 processor          */
/* ------------------------------------------------------------------       */
/* Low level critical section services                                      */

#ifdef    KN_CCNEED_CRITDEF
#ifndef   KN_CCCRITDEF_H
#define   KN_CCCRITDEF_H 1

#define   KN_FN_CRIT      (KN_AS_FNCC | KN_AS_FNPROTO)

#endif    /* KN_CCCRITDEF_H */
#endif    /* KN_CCNEED_CRITDEF */
```

The definition states that the critical section services are to be implemented in header file *KNZZZCC.H* using assembly language code fragments. It also states that KwikNet must provide its standard C function prototypes for these services. If you provide your own custom, compiler dependent prototypes for the low level service functions, be sure to omit *KN_AS_FNPROTO* from your definition of *KN_FN_CRIT* so that the corresponding KwikNet prototypes will be omitted, thereby avoiding possible conflicts.

Following the definition region in header file *KNZZZCC.H* you must implement the low level services as C functions coded in assembly language.

Since your definition of *KN_FN_CRIT* indicates that your critical section services are available in a code fragment, KwikNet defines symbol *KN_CCNEED_CRIT* and includes header file *KNZZZCC.H* to generate one instance of the code fragment in the appropriate KwikNet source module. In this example, the code fragment from header file *KNZZZCC.H* is actually inserted into the source file *KN_UTIL.C*.

Example 3.4.2-B (continued)

Note that this code fragment must reside in its own definition region bounded by the KwikNet *KN_CCNEED_CRIT* definition as shown in the example below.

The compiler's *__asm* directive is used to code the body of the C functions using assembly language. Function *kn_csenter()* returns the previous interrupt state in register *AX* as specified by the Microsoft C to assembly language interface. Note that the Microsoft compiler allows function *kn_csexit()* to reference its parameter *p* from within the assembly language code fragment. Also note that the Microsoft pragma *warning* has been used to avoid an erroneous warning that function *kn_csenter()* fails to return a value.

```
/* Example: Microsoft C compiler for the real mode 80x86 processor           */
/* ------------------------------------------------------------------        */
/* Low level critical section functions                                      */
/* This code sequence is included once within module KN_UTIL.C               */

#ifdef    KN_CCNEED_CRIT
#ifndef   KN_CCCRIT_H
#define   KN_CCCRIT_H 1

/* Prevent warning about function not returning a value                      */
#pragma warning( disable : 4035 )

unsigned int kn_csenter(void)
{
          __asm {
          pushf
          pop       ax
          cli
          }
}         /* Return result in AX */


void kn_csexit(unsigned int p)
{
          __asm {
          mov       ax,word ptr p
          push      ax
          popf
          }
}

#endif /* KN_CCCRIT_H */
#endif /* KN_CCNEED_CRIT */
```

3.4.3 Simple C Macros

If you do not want to optimize a particular low level service using assembly language techniques, you can use the KwikNet C macro for the service, if one is available. Alternatively, you may be able to implement the service as a C macro of your own. This technique is especially suitable for mapping KwikNet services to equivalent C macros provided by your C compiler or to functions available in your C runtime library.

Example 3.4.3-A

The following KwikNet low level services are available as KwikNet macros. Note that some of these services are also available as KwikNet functions (see Chapter 3.4.4). The memory mapped device I/O services can only be used if your C compiler supports the *volatile* keyword. The 32-bit memory mapped device I/O services can only be used if your C compiler and target processor support atomic (indivisible) access to 32-bit device addresses.

Symbol	Service	Purpose
KN_FN_SWAP	*kn_swap16()*	Byte reverse 16-bit unsigned value
KN_FN_SWAP	*kn_swap32()*	Byte reverse 32-bit unsigned value
KN_FN_MEMIO	*kn_inmXX()*	8, 16 and 32-bit memory mapped device read
KN_FN_MEMIO	*kn_outmXX()*	8, 16 and 32-bit memory mapped device write
KN_FN_MEMREP	*kn_inmsXX()*	8, 16 and 32-bit memory mapped device block read
KN_FN_MEMREP	*kn_outmsXX()*	8, 16 and 32-bit memory mapped device block write
KN_FN_CLKRD	*kn_tickrd()*	Read a 32-bit clock tick count
KN_FN_CLKDIFF	*kn_tickdiff()*	Compute a 32-bit clock tick difference

To use these services, simply define the associated KwikNet symbol to be *KN_AS_KNMACRO* as illustrated in the example below. Note that the definition must reside within the definition region of header file *KNZZZCC.H*.

```
/* Example: Most C compilers for most processors                        */
/* -------------------------------------------------------------        */
#ifndef   KN_CCHDR_H
#define   KN_CCHDR_H 1
/* :                                                                    */
/* : Compiler definitions (see Figure 3.2-1)                            */
/* :                                                                    */

/* -------------------------------------------------------------        */
/* Low level services implemented by KwikNet as macros                 */

#define KN_FN_SWAP   KN_AS_KNMACRO /* Endian swapping services          */
#define KN_FN_MEMIO  KN_AS_KNMACRO /* Memory mapped device I/O services */
#define KN_FN_MEMREP KN_AS_KNMACRO /* Memory mapped device block I/O services */

#endif    /* KN_CCHDR_H */
```


Example 3.4.3-B

The following example illustrates the implementation of the device I/O services using MetaWare C/C++ on an Intel 80x86 processor operating in protected mode.

The low level device I/O services can be mapped directly to MetaWare services available as macros or library functions. Note that the MetaWare macros or function prototypes must be available in the C header files included by KwikNet (see Chapter 3.2).

Because these services are being provided by MetaWare C, the KwikNet symbols *KN_FN_DVCIO* and *KN_FN_DVCREP* are defined to be *0*. KwikNet will not generate conflicting function prototypes and will not try to force an instance of equivalent functions.

```
/* Example: MetaWare C compiler for the protected mode 80x86 processor        */
/* ------------------------------------------------------------------         */
#ifndef   KN_CCHDR_H
#define   KN_CCHDR_H 1
/* :                                                                          */
/* : Compiler definitions (see Figure 3.2-1)                                  */
/* :                                                                          */

/* ------------------------------------------------------------------         */
/* Low level device I/O services                                              */

/* The following low level services can be mapped directly to                 */
/* MetaWare services available as macros or library functions.                */

#define kn_inp8(p)              _inb((int)(p))
#define kn_inp16(p)             _inw((int)(p))
#define kn_inp32(p)             _ind((int)(p))

#define kn_outp8(p, d)          _outb((int)(p), (int)(d))
#define kn_outp16(p, d)         _outw((int)(p), (int)(d))
#define kn_outp32(p, d)         _outd((int)(p), (int)(d))

#define kn_inps8(p, b, c)       _insb((int)(p), b, c)
#define kn_inps16(p, b, c)      _insw((int)(p), b, c)
#define kn_inps32(p, b, c)      _insd((int)(p), b, c)

#define kn_outps8(p, b, c)      _outsb((int)(p), b, c)
#define kn_outps16(p, b, c)     _outsw((int)(p), b, c)
#define kn_outps32(p, b, c)     _outsd((int)(p), b, c)

#define KN_FN_DVCIO   0
#define KN_FN_DVCREP  0

#endif    /* KN_CCHDR_H */
```

3.4.4 C Functions Coded in C

If you do not want to optimize a particular low level service using assembly language techniques, you can use the KwikNet C function for the service, if one is available. Alternatively, you can implement the service as a C function of your own, although rarely will you have to do so. The low level services that are not provided by KwikNet as either macros or C functions are, by their very nature, target processor dependent and will most likely have to be coded using the techniques described in Chapters 3.4.1 or 3.4.2.

Example 3.4.4-A

The following KwikNet low level services are available as KwikNet C functions. Note that some of these services are also available as KwikNet macros (see Chapter 3.4.3). If possible, use the equivalent macro version. The memory mapped device I/O services should only be used if your C compiler does not support the *volatile* keyword, thereby preventing you from using the equivalent KwikNet macros. Furthermore, the 32-bit memory mapped device I/O services can only be used if your C compiler and target processor support atomic (indivisible) access to 32-bit device addresses.

Symbol	Service	Purpose
KN_FN_SWAP	*kn_swap16()*	Byte reverse 16-bit unsigned value
KN_FN_SWAP	*kn_swap32()*	Byte reverse 32-bit unsigned value
KN_FN_CKSUM	*kn_cksum()*	IP datagram checksum computation
KN_FN_MEMIO	*kn_inmXX()*	8, 16 and 32-bit memory mapped device read
KN_FN_MEMIO	*kn_outmXX()*	8, 16 and 32-bit memory mapped device write
KN_FN_CLKRD	*kn_tickrd()*	Read a 32-bit clock tick count
KN_FN_CLKDIFF	*kn_tickdiff()*	Compute a 32-bit clock tick difference

To use these services, simply define the associated KwikNet symbol as illustrated in the example below. Note that the definition must reside within the definition region of header file *KNZZZCC.H*.

```
/* Example: Most C compilers for most processors                          */
/* ---------------------------------------------------------------        */
#ifndef   KN_CCHDR_H
#define   KN_CCHDR_H 1
/* :                                                                      */
/* : Compiler definitions (see Figure 3.2-1)                              */
/* :                                                                      */

/* ---------------------------------------------------------------        */
/* Low level services implemented as KwikNet functions                   */

#define KN_FN_CKSUM    KN_AS_KNFUNC /* IP datagram checksum service       */

#endif    /* KN_CCHDR_H */
```


This page left blank intentionally.

4. KwikNet Library Construction

4.1 Preparation

KwikNet is provided in source form ready to create customized KwikNet Libraries which meet your particular network needs.

The KwikNet Library construction process, described in Chapter 1, is illustrated in the block diagram of Figure 1.1-2. The components shown in that figure will be referenced throughout this chapter as the construction procedure is unveiled. You should review that material at this time.

In Chapter 1.2, you were advised to select a KwikNet porting example from which to derive your KwikNet implementation. The set of files from that example were to be copied to a working directory and edited to meet the requirements of your RT/OS and software development tools.

In particular, the RT/OS interface files *KN_OSIF.** were to have been edited as described in Chapter 2. Your choice of C compiler header file *KNZZZCC.H* may also have required modification to match your C compiler. Complete specifications were presented in Chapter 3.

The remaining files from the porting example are used only for constructing the KwikNet Libraries. They may have to be edited for use with your software development tools.

If you are not using the object module librarian (archiver) used in one of the porting examples, you will have to revise the Library Specification Files *KN713*.LBM* to operate with your librarian as described in Chapter 4.2.

You may also have to edit your choice of KwikNet tailoring file *KNZZZCC.INC* so that your make utility will be able to run your software development tools. Tailoring files are described in Chapter 4.3.

Once these last few files are ready, you can proceed with the make by following the directions presented in Chapter 4.4.

KwikNet Directories and Files

The make process depends upon the structure of the KwikNet installation directory *KNT713*. When KwikNet is installed, the following subdirectories are created within directory *KNT713*.

INET	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
TCP	TCP protocol
MAKE	KwikNet make directory
CFGBLDW	KwikNet Configuration Builder; template files
ERR	Construction error summary
TOOLUU	Toolset specific files
TOOLUU\LIB	Toolset specific libraries will be built here
TOOLUU\DRIVERS	KwikNet Device Drivers

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

Other directories containing example software and sample programs will also be present but are not involved in the make process.

A single toolset specific directory *TOOLUU* will be present. This directory will be used to house modules which are specific to the software development tools which you are using. KADAK uses a two or three character mnemonic to identify each of the toolset combinations which it supports. The software toolset for the KwikNet Porting Kit has been assigned the mnemonic *UU*. You can use some other mnemonic if you wish.

4.2 Software Development Tools

To construct the KwikNet Libraries you will need a make utility, a C compiler and an object module librarian (archiver). The porting examples provided with the KwikNet Porting Kit can be used with either Microsoft or Borland make utilities. Examples are also provided for using several different compilers for different target processors.

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.

None of the modules provided with KwikNet are coded in assembly language. Hence, you will not need an assembler to build the KwikNet Libraries. However, you will need an assembler if your C compiler requires it for object module generation. You will also require an assembler if your OS Interface Module *KN_OSIF.C* is complemented by one or more assembly language modules.

Make Utility

To construct the KwikNet Libraries, you will require a make utility such as Microsoft *NMAKE* or Borland *MAKE*. The construction process is initiated by executing your make utility from within subdirectory *MAKE* in the KwikNet installation directory.

The make files provided with KwikNet purposely avoid the use of constructs which might not be readily portable. The exception is the use of the *!include*, *!ifdef*, *!ifndef*, *!else* and *!endif* constructs which are supported by both Microsoft and Borland. All other potentially non-portable syntax has been isolated to the KwikNet tailoring file which will be described in Chapter 4.3.

If your make utility rejects any of the KwikNet make files because of the syntax used, you will have to edit that make file to adapt it for your use. Note that the KwikNet Network Library Make File, say *NETLIB.MAK*, is actually generated by the KwikNet Configuration Builder from the Network Library Template File *KN713LIB.MT*. It is the template file which you may have to edit for compatibility with your make utility.

C Compiler

KwikNet is coded entirely in the C language. You must provide a C compiler which can be invoked by your make utility using a command line directive. The command line string used to run your C compiler must be defined in your KwikNet tailoring file (see Chapter 4.3).

If you have not already done so, be sure to edit your KwikNet compiler configuration file *KNZZZCC.H* as described in Chapter 3 to match your compiler's capabilities.

You must be aware of the conditions which will exist when your C compiler is invoked by the make utility. When making KwikNet Libraries, the current directory will always be one directory level below the KwikNet installation directory *KNT713*. For example, when a KwikNet C source file in directory *KNT713\INET* is being compiled, that directory will be the current directory when your C compiler is executed.

Object Module Librarian

To construct the KwikNet Libraries, you will require an object module librarian, sometimes called an archiver. The librarian must be able to combine a set of object modules produced by your C compiler into a single library module. The command used to run your librarian must be defined in your KwikNet tailoring file (see Chapter 4.3).

Object librarians expect you to provide a list of the object modules which are to be collected together to form a library module. The object module list is often specified in a text file which KADAK refers to as a Library Specification File. A separate Library Specification File is required for each of the KwikNet Libraries.

KN713IP.LBM	KwikNet IP Library	
KN713TCP.LBM	KwikNet TCP Library	
KN713PPP.LBM	KwikNet PPP Library	(optional)
KN713FTP.LBM	KwikNet FTP Library	(optional)
KN713WEB.LBM	KwikNet HTTP Web Server Library	(optional)
KN713SNM.LBM	KwikNet SNMP Agent Library	(optional)
KN713TEL.LBM	KwikNet TELNET Library	(optional)
KN713TFT.LBM	KwikNet TFTP Library	(optional)

The following sets of sample Library Specification Files are included with the porting examples in the KwikNet Porting Kit. The file sets are located in the following subdirectories within directory *KNT713\EXAMPLES*.

MSDOS	Microsoft librarian for 16-bit, real mode 80x86
DOS4GW	WATCOM librarian for 32-bit, protected mode 80386 and up
XRTOS	Mentor Graphics (Microtec) 68000 librarian
XOS	Mentor Graphics (Microtec) 68000 librarian

You must be aware of the conditions which will exist when your librarian is invoked by the make utility. When making KwikNet Libraries, the current directory will always be one directory level below the KwikNet installation directory *KNT713*. For example when the KwikNet TCP Library is being built, directory *KNT713\TCP* will be the current directory when your librarian is executed.

4.3 The KwikNet Tailoring File

The KwikNet Libraries are constructed using your make utility, C compiler and object module librarian. The make process is guided by your Network Library Make File, say *NETLIB.MAK*, which is generated by the KwikNet Configuration Builder.

A file which KADAK calls a **tailoring file** is used to tailor the library construction process for the particular C compiler and object librarian which you are using. It is the tailoring file which provides the make commands to compile a C module or to construct a library module. Obviously, the tailoring file must use the make syntax which is acceptable to your make utility.

Two sets of tailoring files are provided with the KwikNet Porting Kit, one for use with Borland *MAKE* and one for use with Microsoft *NMAKE*. These tailoring files will be found in installation directory *KNT713\EXAMPLES*. Borland compatible tailoring files will be found in subdirectory *TF_BORLD*. Those for Microsoft are located in subdirectory *TF_MSOFT*. If you are not using either of these make utilities, pick a tailoring file suited to one of them and edit it to match the syntax required by your particular make utility.

The following tailoring files provided with the KwikNet Porting Kit examples illustrate proper usage with the following make utilities, target processors and C compilers.

Tailoring File	Make Utility	Target Processor	Compiler
	Microsoft		
M__86MC.15	*NMAKE*	80x86 (real mode)	Microsoft 16-bit Visual C/C++ v1.5, v1.52
M_386WC.110	*NMAKE*	80x86 (protected mode)	WATCOM 32-bit C/C++ v11.0
M_68KMR.45	*NMAKE*	68000	Mentor Graphics (Microtec) C/C++ v4.5G
	Borland		
B__86MC.15	*MAKE*	80x86 (real mode)	Microsoft 16-bit Visual C/C++ v1.5, v1.52
B_386WC.110	*MAKE*	80x86 (protected mode)	WATCOM 32-bit C/C++ v11.0
B_68KMR.45	*MAKE*	68000	Mentor Graphics (Microtec) C/C++ v4.5G

Pick the tailoring file which most closely matches your choice of make utility, target processor and C compiler. Copy that file to your working directory and rename it *KNZZZCC.INC*. This is the tailoring file which will be used to create your KwikNet Libraries.

The following growing list of tailoring files, although created and tested for use with KwikNet and the AMX Real-Time Multitasking Kernel, are provided with the KwikNet Porting Kit. Replace *m_* in the filename with *M_* for use with Microsoft *NMAKE* or *B_* for use with Borland *MAKE*.

Tailoring File	Target Processor	Compiler
m__86MC.15	80x86 (real mode)	Microsoft 16-bit Visual C/C++ v1.5, v1.52
m__86TC.50	80x86 (real mode)	Borland 16-bit C/C++ v5.0
m__86PD.50	80x86 (real mode)	Paradigm 16-bit C/C++ v5.0, v6.0
m__86WC.110	80x86 (real mode)	WATCOM 16-bit C/C++ v11.0
m_386BCB.50	80x86 (protected mode)	Borland 32-bit C/C++ v5.0
m_386MCB.10	80x86 (protected mode)	Microsoft 32-bit Visual C/C++ v1.0
m_386MCB.42	80x86 (protected mode)	Microsoft 32-bit Visual C/C++ v4.2
m_386MCB.50	80x86 (protected mode)	Microsoft 32-bit Visual C/C++ v5.0
m_386MWB.331	80x86 (protected mode)	MetaWare High C/C++ v3.31
m_386MWB.360	80x86 (protected mode)	MetaWare High C/C++ v3.60
m_386WCB.110	80x86 (protected mode)	WATCOM 32-bit C/C++ v11.0
m_386PD.60	80x86 (protected mode)	Paradigm 32-bit C/C++ v5.0, v6.0
m_68KDA.42	68000	Diab-SDS C/C++ v4.2, v4.3
m_68KIM.842	68000	TASKING (Intermetrics) C/C++ v8.4.2
m_68KIM.92	68000	TASKING (Intermetrics) C/C++ v9.2r0
m_68KME.20	68000	Metrowerks C/C++ v2.0
m_68KMR.45	68000	Mentor Graphics (Microtec) C/C++ v4.5G
m_68KMR.51	68000	Mentor Graphics (Microtec) C/C++ v5.1
m_CFDA.42	ColdFire	Diab-SDS C/C++ v4.2, v4.3
m_CFME.25	ColdFire	Metrowerks C/C++ v2.5
m_PPCDA.41	PowerPC	Diab-SDS C/C++ v4.1, v4.2, v4.3
m_PPCME.42	PowerPC	Metrowerks C/C++ v4.2
m_PPCME.50	PowerPC	Metrowerks C/C++ v5.0
m_PPCMW.41	PowerPC	MetaWare High C/C++ v4.1, v4.3
m_ARMMW.410	ARM	MetaWare High C/C++ v4.1
m_ARMMW.420	ARM	MetaWare High C/C++ v4.2
m_ARMRM.211	ARM	ARM Ltd. C/C++ SDK v2.11
m_ARMRM.250	ARM	ARM Ltd. C/C++ SDK v2.50
m_ARMRM.10	ARM	ARM Ltd. C/C++ ADS v1.0, v1.1
m_ARBXXX.vvv	Thumb	(see vendors *m_ARMXXX.vvv* shown above)
m_M32MW.430	MIPS32	MetaWare High C/C++ v4.3e
m_BFAD.30	Blackfin	Analog Devices C/C++ v3.0

Editing the KwikNet Tailoring File

You must edit your KwikNet tailoring file *KNZZZCC.INC* so that your make utility will be able to run your software development tools.

Since you may be porting to a compiler and librarian with which KADAK has no experience, it is impossible to specify the command line switches which you will have to use to compile KwikNet C source files or to make library modules.

You should review the command line definitions used in the tailoring files provided with the KwikNet Porting Kit. These examples illustrate proper usage of two different make utilities and several target processors and C compilers.

Figures 4.3-1, 4.3-2 and 4.3-3 show a listing of the tailoring file *M__86MC.15* provided for use with Microsoft *NMAKE*. It illustrates the use of the Microsoft 16-bit Visual C/C++ compiler (*CL*) to compile KwikNet C source files for use on a 16-bit 80x86 target processor. It also shows how a KwikNet library is built using the Microsoft object librarian (*LIB*).

The tailoring file must define the filename extensions which your software development tools use for different types of files. The extensions are defined as macros (see Figure 4.3-1). Other commonly encountered extensions are *.O*, *.S* and *.A*.

The make file must be able to copy a file, erase a file and change the current working directory. Since some make utilities balk if they encounter an empty rule, the make file must also be able to issue a command that does nothing. Macros *CMDCOPY*, *CMDDEL*, *CMDCD* and *CMDNOP* (see Figure 4.3-1) define the operating system dependent commands which the make utility can execute to perform these operations.

Some make utilities require that you specify the filename extensions which are permitted in implicit rules. Microsoft and Borland both allow use of the *.SUFFIXES* directive for this purpose.

Macro *CCCOMPILE* is defined to be the command which the make utility can execute to invoke your C compiler. The macro is used in the subsequent definition of the implicit rule which the make utility must follow to compile a C source file. Several definitions of macro *CCCOMPILE* are provided (see Figure 4.3-2) to allow the C compilation rules to be easily adjusted via the make command line. Since this macro is only used in the implicit rule for running the compiler, you are free to revise this adaptation methodology to best suit your own needs.

Another C compilation macro *CSCOMPILE* (see Figure 4.3-2) is defined for use in the construction of KwikNet sample programs. This macro gives the C compiler access to the header files located in the sample program make directory *SAM_MAKE*. The construction of a KwikNet sample program is described in Chapter 5.

Finally, two implicit rules must be defined in the tailoring file (see Figure 4.3-3). The first implicit rule provides the command(s) to be executed by the make utility to compile a KwikNet C source file to generate an object module. The second implicit rule provides the command(s) to be executed to generate a KwikNet Library module from a collection of the compiled object modules.

The C Compilation Implicit Rule

You must be aware of the conditions which will exist when your C compiler is invoked by the implicit rule to compile a KwikNet module. When making KwikNet Libraries, the current directory will always be one directory level below the KwikNet installation directory *KNT713*. For example, when a KwikNet C source file in directory *KNT713\INET* is being compiled, that directory will be the current directory when your C compiler is executed. In some cases, directory *KNT713\MAKE* may be the current directory.

Your C compiler will require access to C header files from the then current directory and from KwikNet directories *..\INET* and *..\TCP*. It will also require access to the standard C header files required by your C compiler. How you provide such access will depend upon the operating environment in which you are doing your software development. In the tailoring file example shown in Figure 4.3-2, access to the KwikNet directories is provided using the compiler's */I* command line switch.

Since the current directory is always one level below the KwikNet installation directory, you may be able to redirect compiler warnings and error messages to the KwikNet error directory at *..\ERR* as illustrated in the tailoring file example in Figure 4.3-3.

The Library Build Implicit Rule

You must be aware of the conditions which will exist when your librarian is invoked by the implicit rule to create a KwikNet Library module. When making KwikNet Libraries, the current directory will always be one directory level below the KwikNet installation directory *KNT713*. For example when the KwikNet TCP Library is being built, directory *KNT713\TCP* will be the current directory when your librarian is executed.

The object modules to be inserted into the library will be in the directory specified by the make macro expansion *$(KL)*. It is recommended that the object modules be copied to the current directory so that path information can be omitted from the object module list in your Library Specification File. Be sure to delete the object modules from the current directory after the library has been built.

If the library module is created in the current directory, be sure to copy it to the destination directory with the correct library filename. You may also wish to save a copy of the library summary report, if any, produced by the librarian. These operations are illustrated in the example in Figure 4.3-3.

Since the current directory is always one level below the KwikNet installation directory, you may be able to redirect librarian warnings and error messages to the KwikNet error directory at *..\ERR* as illustrated in the tailoring file example in Figure 4.3-3.

```
# ------------------- Make specific INCLUDE file ----------------------
# These make directives require Microsoft NMAKE v1.40 or compatible.
# The make command directives are valid for use under DOS or Windows.

# Define the file extensions required by your tools
# AEXT is the file extension for assembler source files
# OEXT is the file extension for object files
# LEXT is the file extension for library files
# XEXT is the file extension for executable load module files
# LNKS is the file extension for link specification files
# LBMS is the file extension for library specification files

AEXT = ASM
OEXT = OBJ
LEXT = LIB
XEXT = EXE
LNKS = LKS
LBMS = LBM


# Define the commands which the make utility can execute to
# perform the following operations:

# Copy srcfile destfile
CMDCOPY = copy

# Delete file X
CMDDEL  = erase

# Make path X the current directory
CMDCD   = cd

# No operation
CMDNOP  = rem


# To avoid conflicts with implicit rules established by the make utility
# for its predefined list of suffixes, clear the suffixes list with an
# empty .SUFFIXES command.
.SUFFIXES :

# If required by the make utility, define extensions which
# can be used in implicit rules.
.SUFFIXES : .$(LNKS) .c .$(AEXT) .$(LBMS)
```

Figure 4.3-1 KwikNet Tailoring File (Part 1)

```
# Note:  The following make macros can be defined by the user
#        on the make command line when the make utility is invoked
#        to construct KwikNet.
#
#        These macros can also be injected by providing the macro
#        definitions in a file.  The file name must be provided by
#        defining macro "CCCFILE=filename" on the make command line.
#
# Define "CCFLAGS=switches" if you wish to override the default
#        C compile switches provided in this module.
#
# Define DBGINFO if you wish modules to be compiled for debugging.
# Define "CCDEBUG=switches" if you wish to override the default
#        C debug compile switches provided in this module.


# Define C compilation switches and debug switches to be used
# unless overidden by definitions on the make command line or
# in the file specified by macro CCCFILE.

!ifndef DBGINFO
# Compile WITHOUT debug information
!ifndef CCFLAGS
CCCOMPILE = CL /c /G1 /Gs /Gt256 /Alfw /Ze /W3 /I..\INET /I..\TCP
CSCOMPILE = CL /c /G1 /Gs /Gt256 /Alfw /Ze /W3 /I..\SAM_MAKE
!else
CCCOMPILE = CL $(CCFLAGS)
CSCOMPILE = CL $(CCFLAGS) /I..\SAM_MAKE
!endif

!else
# Compile WITH debug information
!ifndef CCDEBUG
CCCOMPILE = CL /c /G1 /Gs /Gt256 /Alfw /Ze /Z7 /W3 /I..\INET /I..\TCP
CSCOMPILE = CL /c /G1 /Gs /Gt256 /Alfw /Ze /Z7 /W3 /I..\SAM_MAKE
!else
CCCOMPILE = CL $(CCDEBUG)
CSCOMPILE = CL $(CCDEBUG) /I..\SAM_MAKE
!endif
!endif
```

Figure 4.3-2 KwikNet Tailoring File (Part 2)

```
# Note:  Microsoft NMAKE defines the following filename macros which
#        are used in the implicit rules defined in this module.
# Macro $(@B) specifies the target filename with no path or extension.
# Macro $@ specifies the target filename including path and extension.

# Create object file from C file in current source directory.
# Move object file to the library directory specified by $(O).
{.}.c{$(O)}.$(OEXT):
         $(CCCOMPILE) /Fo$(@B).$(OEXT) $(@B).C >..\ERR\$(@B).E
         copy $(@B).$(OEXT) $@
         erase *.$(OEXT)


# Create library file from object files and library command file
# in the library directory specified by $(KL).
.$(LBMS).$(LEXT):
         erase $@
         erase $(@B).$(LEXT)
         copy $(KL)\$(@B).$(LBMS)
         copy $(KL)\*.$(OEXT)
         LIB @$(@B).$(LBMS) >..\ERR\$(@B).LBE
         copy $(@B).$(LEXT) $@
         copy $(@B).RPT $(KL)\$(@B).RPT
         erase *.$(LBMS)
         erase *.$(OEXT)
         erase *.$(LEXT)
         erase *.RPT

# ------------------- End of INCLUDE file -----------------------------
```

Figure 4.3-3 KwikNet Tailoring File (Part 3)

4.4 Making the KwikNet Library

KwikNet is provided in source form ready to create customized KwikNet Libraries which meet your particular network needs. The libraries are constructed using your make utility, C compiler and object module librarian (archiver). The make utility takes as input a make file, called the Network Library Make File, which specifies how the libraries are to be built.

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*. This process is described in Chapter 2.1 of the KwikNet TCP/IP Stack User's Guide.

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 1.1-2 summarizes the components which are required to build the KwikNet Libraries. Several of these components are the files which you have edited to port KwikNet to your operating environment.

All of your updated porting files must be copied from your working directory to the appropriate KwikNet installation directories prior to making the KwikNet Libraries. Each of the following files must be moved to the indicated destination directory.

Source File	Destination Directory	File Purpose
NETLIB.UP	*MAKE*	KwikNet Library Parameter File
NETLIB.MAK	*MAKE*	KwikNet Library Make File
KN_OSIF.C	*INET*	OS Interface Module for your RT/OS
KN_OSIF.H	*INET*	OS Interface Header File for your RT/OS
KN_OSIF.INC	*TOOLUU*	OS Interface Make Specification for your RT/OS
KNZZZCC.INC	*TOOLUU*	Tailoring File (for use with your make utility)
KNZZZCC.H	*TOOLUU*	Compiler Configuration Header File
KN713IP.LBM	*TOOLUU\LIB*	KwikNet IP Library Specification File
KN713TCP.LBM	*TOOLUU\LIB*	KwikNet TCP Library Specification File
KN713.LBM*	*TOOLUU\LIB*	Library Specification Files (for optional KwikNet Libraries)
KN_BOARD.C	*TOOLUU\DRIVERS*	Board driver for your target hardware

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. Most can also be used with Windows 2000 and Windows XP.

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 2000 or XP Start menu, choose the Command Prompt from the Programs (or All Programs) folder. The Command Prompt may be located in the Accessories folder. Make the KwikNet installation *MAKE* directory the current directory.

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

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

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

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

In each case, the make symbol *TOOLSET* is defined to be the toolset mnemonic *UU*. 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).

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 *TOOLUU\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 *UU* directory *TOOLUU\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 *TOOLUU\LIB*. The following KwikNet Libraries will be created from these object files and placed in directory *TOOLUU\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.

KN713IP.A	KwikNet IP Library
KN713TCP.A	KwikNet TCP Library
KN713OPT.A	KwikNet Library for optional KwikNet component *OPT* where *OPT* may be one of *PPP*, *FTP*, *WEB*, *SNM*, *TEL* or *TFT*.

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

KN_LIB.UP	KwikNet Library Parameter File
KN_LIB.MAK	KwikNet Network Library Make File
KN_LIB.H	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 *KN713LIB.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 *KN713LIB.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 *TOOLUU\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.

5. KwikNet Application Construction

5.1 Building an Application

Now that you have ported KwikNet to your operating environment and are able to construct the KwikNet Libraries, you are ready to 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 Network Library Make File. Using that file, create your KwikNet Libraries following the procedure described in Chapter 4.4.

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.

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.

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 (hopefully none) or unusual operation of the KwikNet TCP/IP Stack.

5.2 KwikNet Sample Programs

An overview of a KwikNet application was presented in Chapter 1 and illustrated as a block diagram in Figure 1.1-1. You should review that material now.

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

Sample Program Directories and Files

When KwikNet is installed, the following subdirectories on which the sample program construction process depends are created within directory *KNT713*.

CFGBLDW	KwikNet Configuration Builder; template files
ERR	Construction error summary
TOOLUU	Toolset specific files
TOOLUU\DRIVERS	KwikNet Device Drivers
TOOLUU\LIB	Toolset specific libraries will be built here
TOOLUU\SAM_MAKE	Sample program make directory
TOOLUU\SAM_TCP	KwikNet TCP/IP Sample Program directory containing:
KNSAMPLE.MAK	TCP/IP Sample Program make file
KNSAMPLE.C	TCP/IP Sample Program
KNZZZAPP.H	Application Header
KNSAMLIB.UP	Library Parameter File
KNSAMNCF.UP	Network Parameter File
KNSAMPLE.LKS	Link Specification File
TOOLUU\SAM_COMN	Common sample program source files:
KNSAMOS.C	Application OS Interface
KNSAMOS.H	Application OS Interface header file
KNRECORD.C	Message recording services
KNCONSOL.C	Console driver
KNCONSOL.H	Console driver header
	Console driver serial I/O support:
KN8250S.C	INS8250 (NS16550) UART driver

Other sample program subdirectories such as *SAM_FTP*, *SAM_TEL*, *SAM_TFTP* and *SAM_WEB* will also be present within directory *TOOLUU* if you have purchased the corresponding optional KwikNet components.

A single toolset specific directory *TOOLUU* will be present. This directory will be used to house modules which are specific to the software development tools which you are using. KADAK uses a two or three character mnemonic to identify each of the toolset combinations which it supports. The software toolset for the KwikNet Porting Kit has been assigned the mnemonic *UU*. You can use some other mnemonic if you wish.

The Application OS Interface

All KwikNet sample programs share a common implementation strategy. The application interacts directly with KwikNet. However, for portability, the application interacts with your RT/OS through the Application OS Interface. One such module, *KNSAMOS.C*, is provided with each of the porting examples in the KwikNet Porting Kit.

If you port the KwikNet sample program(s) to your operating environment, you will have to edit the Application OS Interface *KNSAMOS.C* and its header file *KNSAMOS.H*. Edit the copy of these files which you transferred to your working directory when you selected the set of files for a particular porting example.

Editing the Application OS Interface

The Application OS Interface module *KNSAMOS.C* includes the *main()* function used by all KwikNet sample programs. This function may have to be altered to properly start up and shut down your RT/OS. The *main()* function calls the various application and RT/OS initialization and termination procedures as recommended in the RT/OS Interface description presented in Chapter 2.

All KwikNet sample programs call procedure *sam_osshutdown()* after KwikNet has been shutdown. The purpose of the call is to terminate execution of your RT/OS so that the application can gracefully return to the *main()* function. For most single threaded applications, procedure *sam_osshutdown()* can be empty.

The KwikNet data logging procedure *sam_record()* is located in the Application OS Interface, giving you full control over the dispatch of messages generated by KwikNet and its sample programs. All of the examples provided with the KwikNet Porting Kit use the data recording service in module *KNRECORD.C* to record such messages. However, the implementation varies according to the way the RT/OS operates as indicated by the explanations provided in each of the sample *KNSAMOS.C* files. One way or another, the message to be logged is passed to procedure *kn_logmsg()* in the data recording module *KNRECORD.C*.

RTOS Services in the Application OS Interface

When used with a multitasking RTOS, the Application OS Interface must provide the following task management service. Procedure *sam_ostkprep()* must create an RTOS compatible instance of a task. Procedure *sam_ostkstart()* must force that task to begin execution at the earliest possible opportunity. The RT/OS independent task definition structure used by the sample programs can be found in the sample specific header file *KNZZZAPP.H*.

If a task automatically begins to execute when it is created, procedure *sam_ostkprep()* should ignore the request and let procedure *sam_ostkstart()* create and start the task. If tasks cannot be dynamically created, you will have to predefine a set of tasks and activate one of these tasks each time the application calls procedure *sam_ostkstart()* to start a task. None of the KwikNet sample programs require more than five such tasks.

Data Recording

A data recording service is provided with the KwikNet sample programs. Procedure *kn_logmsg()* in module *KNRECORD.C* can be used to record messages generated by KwikNet and the application. Procedure *sam_record()* in the Application OS Interface acts as a funnel to deliver each KwikNet message to procedure *kn_logmsg()* which then records the message into its string array *kn_recordlist[]*.

KwikNet messages will only be logged through procedure *sam_record()* if data logging is enabled on the Application property page of your KwikNet Network Parameter File. Be sure to enter *sam_record* as the name of the logging function in the field provided.

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 a character array. Symbol *KN_REC_MEMSIZE* defines the size of that array. Symbol *KN_REC_NUM* defines the maximum number of message strings which can be recorded into the array.

Procedure *kn_loginit()* in module *KNRECORD.C* must be called by the application before the data recording service can be used by KwikNet or the application. For this reason, the *main()* function in the Application OS Interface module *KNSAMOS.C* calls *kn_loginit()* as one of its earliest operations.

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.

> 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.

Console Device Use

The KwikNet sample programs provide support for a simple, interactive console device. The console driver in module *KNCONSOL.C* 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.

5.3 Tailoring File Enhancements

The KwikNet tailoring file is a file used to tailor the KwikNet Library construction process for the particular C compiler and object librarian which you are using. Tailoring files are described in Chapter 4.3. It is the tailoring file which provides the make commands to compile a C module or to construct a library module. Obviously, the tailoring file must use the make syntax which is acceptable to your make utility.

A make file is provided with each KwikNet sample program. The make file can be used with your make utility to generate the sample program load module as described in the next chapter. To use these make files, the KwikNet tailoring file must be adapted to specify the make commands needed to compile the sample program C modules and to link the resulting object modules with various libraries to create the sample program load module.

Editing the KwikNet Tailoring File

You must edit your KwikNet tailoring file *KNZZZCC.INC* so that your make utility will be able to run your software development tools to compile the sample program modules and link them to create a load module.

Tailoring files were described in Chapter 4. Figures 4.3-1, 4.3-2 and 4.3-3 show a listing of the tailoring file *M__86MC.15* provided for use with Microsoft *NMAKE*. It illustrates the use of the Microsoft 16-bit Visual C/C++ compiler (*CL*) to compile C source files for use on a 16-bit 80x86 target processor.

Macro *CSCOMPILE* (see Figure 4.3-2) is defined for use in the construction of KwikNet sample programs. It defines the command which the make utility can execute to invoke your C compiler. This macro gives the C compiler access to the header files located in the sample program make directory *SAM_MAKE*.

Macro *CSCOMPILE* is used in the subsequent definition of the implicit rule which the make utility must follow to compile a sample program C source file. Several definitions of macro *CSCOMPILE* are provided (see Figure 4.3-2) to allow the C compilation rules to be easily adjusted via the make command line.

Finally, two implicit rules must be defined in the tailoring file (see Figure 5.3-1). The first implicit rule provides the command(s) to be executed by the make utility to compile a sample program C source file to generate an object module. The second implicit rule provides the command(s) to be executed to create a load module by linking the collection of the compiled object modules with the KwikNet Libraries and other application libraries.

The C Compilation Implicit Rule

You must be aware of the conditions which will exist when your C compiler is invoked by the implicit rule to compile a sample program module. When making a sample program, the current directory will always be one directory level below the KwikNet toolset directory *KNT713\TOOLUU*. For example, when one of the common C source files in directory *KNT713\TOOLUU\SAM_COMN* is being compiled, that directory will be the current directory when your C compiler is executed.

Your C compiler will require access to C header files from the then current directory and from the KwikNet sample construction make directory *..\SAM_MAKE*. It will also require access to the standard C header files required by your C compiler. How you provide such access will depend upon the operating environment in which you are doing your software development. In the tailoring file example shown in Figure 4.3-2, access to the KwikNet directory *..\SAM_MAKE* is provided using the compiler's */I* command line switch.

Since the current directory is always one level below the KwikNet toolset directory, you may be able to redirect compiler warnings and error messages to the KwikNet error directory at *..\..\ERR* as illustrated in the tailoring file example in Figure 5.3-1.

The Implicit Rule for Assembly

None of the KwikNet sample program source files are coded in assembly language. Hence there is usually no need for the make utility to run an assembler to build sample program object modules. However, an example of such an implicit rule (see Figure 5.3-1) is included with each of the tailoring files provided with the KwikNet Porting Kit. You are free to revise this rule to meet your application requirements.

The Implicit Rule for Linking

You must be aware of the conditions which will exist when your linker is invoked by the implicit rule to create a sample program load module. When making a sample program, the current directory will always be one directory level below the KwikNet toolset directory *KNT713\TOOLUU*. For example, when the KwikNet TCP/IP Sample Program is being built, directory *KNT713\TOOLUU\SAM_TCP* will be the current directory when your linker is executed. The object modules to be linked will be in this directory.

Most program linkers allow you to define a command file which specifies the list of object files and libraries to be linked. KADAK calls such a file a Link Specification File and gives it the file extension *LKS*. Examples are included with each of the porting examples provided with the KwikNet Porting Kit. For example, the Link Specification File for the KwikNet TCP/IP Sample Program is file *KNSAMPLE.LKS*.

The Link Specification File for the sample program must be edited to match the syntax required by your program linker. The sample program object modules listed in the file will not have path information appended because the directory in which they reside is always current at the time your linker is invoked. For the same reason, the KwikNet Libraries can always accessed via the path *..\LIB*. So, for example, the KwikNet IP Library can be identified as file *..\LIB\KN713IP.LIB*. If necessary, be sure to adjust the filename extensions for object modules and libraries in the Link Specification File to match those defined in your tailoring file.

The implicit rule for linking the sample program is illustrated in Figure 5.3-1. The rule is invoked by the dependency on a file with extension matching macro expansion *$(LNKS)*. The rule constructs a load module with extension matching macro expansion *$(XEXT)*. The macros *LNKS* and *XEXT* are defined in your tailoring file as shown in Figure 4.3-1 in Chapter 4.3

Since the current directory is always one level below the KwikNet toolset directory, you may be able to redirect linker warnings and error messages to the KwikNet error directory at *..\..\ERR* as illustrated in the tailoring file example in Figure 5.3-1.

```
# Sample Program Construction Rules

!ifdef ULINK
# Create object file from C file in current source directory.
# Move object file to the sample program directory specified by $(ULINK).
{.}.c{$(ULINK)}.$(OEXT):
        $(CSCOMPILE) /Fo$@ $(@B).C >..\..\ERR\$(@B).E


# Create object file from assembly file in current source directory.
# Move object file to the sample program directory specified by $(ULINK).
{.}.$(AEXT){$(ULINK)}.$(OEXT):
        MASM $(@B).$(AEXT) /ML /N, $@ >..\..\ERR\$(@B).E


# Link an executable file in current directory.
{$(ULINK)}.$(LNKS){$(ULINK)}.$(XEXT):
        LINK $(@B).$(LNKS) >..\..\ERR\$(@B).LKE
!endif

# ------------------- End of INCLUDE file -------------------------------
```

Figure 5.3-1 Sample Program Tailoring File Enhancements

5.4 Making the Sample Program

The KwikNet sample programs are provided ready to be constructed using your make utility, C compiler and link/locate utility. The make utility takes as input a sample program make file which specifies how the program is to be built.

KwikNet Parameter Files

Two KwikNet parameter files are provided with each KwikNet sample program.

The Library Parameter File describes the KwikNet options and features illustrated by the sample program. This file is used to construct the KwikNet Libraries for the sample program.

The Network Parameter File 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 sample program.

Building the KwikNet Libraries

Before you can construct any of the KwikNet sample programs, you must first build the associated KwikNet Libraries.

Use the KwikNet Configuration Builder to edit the sample program Library Parameter File. For example, to build the KwikNet Libraries for the KwikNet TCP/IP Sample Program, edit Library Parameter File *KNSAMLIB.UP*. Use the Configuration Builder to generate the Network Library Make File *KNSAMLIB.MAK*. This process is described in Chapter 2.1 of the KwikNet TCP/IP Stack User's Guide.

Use the Network Library Make File *KNSAMLIB.MAK* to build the KwikNet Libraries. Follow the directions provided in Chapter 4.4.

> Porting Tip
>
> The KwikNet Porting Kit includes a batch file which will help you prepare to construct any of the sample programs provided with KwikNet and its optional components. In the KwikNet installation directory *KNT713*, run batch file *TOOLUU.BAT* without parameters for a description of its usage. Use the batch file to copy your ported files from your working directory into the appropriate directories.

Gathering Files

The block diagram in Figure 1.1-1 summarizes the components which are fundamental to any KwikNet application. Several of these components are the files which you have edited to port KwikNet to your operating environment.

All of your updated porting files must be copied from your working directory to the appropriate KwikNet toolset directories prior to making any of the KwikNet sample programs. Some of the updated files will have already been copied to the appropriate directories in order to create the sample program's KwikNet Libraries.

To build the KwikNet TCP/IP Sample Program using make file *KNSAMPLE.MAK*, each of the following updated files must be present in the indicated destination directory. For other KwikNet sample programs, replace the program name *KNSAMPLE*, the parameter file names *KNSAM*.UP* and references to directory *SAM_TCP* with the appropriate names.

Source File	Destination Directory	File Purpose
KN_OSIF.H	*INET*	OS Interface Header File for your RT/OS
KN_OSIF.INC	*TOOLUU*	OS Interface Make Specification for your RT/OS
KNZZZCC.INC	*TOOLUU*	Tailoring File (for use with your make utility)
KNZZZCC.H	*TOOLUU*	Compiler Configuration Header File
KNSAMOS.C	*TOOLUU\SAM_COMN*	Application OS Interface
KNSAMOS.H	*TOOLUU\SAM_COMN*	Application OS Interface header file
KNSAMPLE.LKS	*TOOLUU\SAM_TCP*	TCP/IP Sample Program Link Specification File
KN_BOARD.C	*TOOLUU\DRIVERS*	Board driver for your target hardware

> Porting Tip
>
> Batch file *TOOLUU.BAT* provided with KwikNet will gather all of these components for you. In the KwikNet installation directory *KNT713*, run batch file *TOOLUU.BAT* without parameters for a description of its usage. Use the batch file to copy your ported files from your working directory into the appropriate directories.

The 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 *TOOLUU\SAM_TCP*.

All of the compilers and librarians used at KADAK were tested under Windows® NT. Most can also be used with Windows 2000 and Windows XP.

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 2000 or XP Start menu, choose the Command Prompt from the Programs (or All Programs) folder. The Command Prompt may be located in the Accessories folder. Make the KwikNet toolset *TOOLUU\SAM_TCP* directory the current directory.

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

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

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

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

In each case, the make symbol *TOOLSET* is defined to be the toolset mnemonic *UU*. The symbol *OSPATH* is defined to be the string *yourospath*, the full path (or the path relative to directory *TOOLUU\SAM_TCP*) to the directory containing your RT/OS components (header files, libraries and/or object modules).

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 *KN713CFG.CT* in directory *CFGBLDW*. The file is left in the sample program directory *TOOLUU\SAM_TCP*.

The KwikNet Sample Program load module *KNSAMPLE.xxx* is created in toolset directory *TOOLUU\SAM_TCP*. The file extension of the load module will match your definition of macro *XEXT* in your tailoring file (see Figure 4.3-1).

> Note
>
> For other KwikNet sample programs, replace the program name *KNSAMPLE*, the Network Configuration Module name *KNSAMNCF* and references to directory *SAM_TCP* with the appropriate names.

5.5 RT/OS Examples

5.5.1 Using a Custom RTOS

KwikNet can be used with any custom in-house or commercial multitasking RTOS. This porting example interfaces to a non-existent RTOS deemed to have the functional capabilities present in most reasonable RTOS implementations. The example has been built using Mentor Graphics (Microtec) C/C++ software development tools targeted for a 68000 processor.

Since the RTOS does not exist, this example has never been executed. However, all of the OS interface procedures have been implemented and will serve as excellent working models for your port.

A hardware clock operating at 1 KHz has been assumed as the fundamental source of timing for the RTOS and KwikNet. The KwikNet clock frequency has been defined to be 20 Hz. It has been assumed that a clock device driver provided by the RTOS will have properly initialized the hardware clock when the RTOS begins execution. Furthermore, the RTOS is assumed to provide a clock hook which will call an application function coded in C whenever a clock interrupt is serviced.

The console driver for the custom RTOS porting example is configured to use a UART serial driver connected to a terminal. File *KN8250S.C* in the common sample program directory *KNT713\TOOLUU\SAM_COMN* is a simple device driver for an INS8250 or NS16550 compatible UART.

Standard C is used for memory allocation. The KwikNet memory locking feature is enabled to permit multiple tasks operating under the RTOS to use the non-reentrant C library memory allocation functions.

Sample programs which require a file system are configured to use a custom, user defined file system. The KwikNet file access locking feature is enabled to permit multiple tasks operating under this RTOS to use the custom file I/O functions which are assumed to be non-reentrant.

Source Files

The source files for the KwikNet custom RTOS porting example are located in KwikNet installation directory *KNT713\EXAMPLES\XRTOS*.

5.5.2 Using MS-DOS

KwikNet has been tested with MS-DOS v6.22 operating in real mode on PC compatible hardware. This single threaded KwikNet porting example was constructed using Microsoft 16-bit software development tools.

The standard PC hardware clock operating at 18.2 Hz was used as the fundamental source of timing for KwikNet and the application. The KwikNet clock frequency has been defined to be 18 Hz. Microsoft C library function *_chain_intr()* is used by the clock interrupt service routine *kn_osclockisr()* in the OS Interface Module *KN_OSIF.C* to chain to the original clock handler.

The console driver for the MS-DOS porting example is configured to use the PC screen and keyboard as a terminal.

Standard C is used for memory allocation. Memory locking is not required for single threaded applications.

Sample programs which require a file system are configured to use standard C file operations. The Microsoft C standard I/O library provides access to the underlying MS-DOS file system. File access locking is not required for single threaded applications.

The KwikNet OS Interface Module *KN_OSIF.C* and Application OS Interface *KNSAMOS.C*. should require little, if any, modification for use with your application.

Note that the Microsoft C library functions *_dos_getvect()* and *_dos_setvect()* are used by procedure *kn_osvaccess()* in the OS Interface Module *KN_OSIF.C* to modify entries in the processor interrupt table. Library function *_chain_intr()* is also used to chain to the original clock handler after servicing the KwikNet clock.

Source Files

The source files for the KwikNet MS-DOS porting example are located in KwikNet installation directory *KNT713\EXAMPLES\MSDOS*.

5.5.3 Using the DOS/4GW DOS Extender with MS-DOS

KwikNet has been tested with the Tenberry DOS/4GW DOS Extender operating with MS-DOS v6.22 in protected mode on PC compatible hardware. This single threaded KwikNet porting example was constructed using WATCOM (Sybase) 32-bit software development tools.

The standard PC hardware clock operating at 18.2 Hz was used as the fundamental source of timing for KwikNet and the application. The KwikNet clock frequency has been defined to be 18 Hz. WATCOM C library function *_chain_intr()* is used by the clock interrupt service routine *kn_osclockisr()* in the OS Interface Module *KN_OSIF.C* to chain to the original clock handler.

The console driver for the DOS/4GW porting example is configured to use the PC screen and keyboard as a terminal.

Standard C is used for memory allocation. Memory locking is not required for single threaded applications.

Sample programs which require a file system are configured to use standard C file operations. The WATCOM C standard I/O library provides access to the underlying MS-DOS file system through the DOS/4GW DOS Extender. File access locking is not required for single threaded applications.

The KwikNet OS Interface Module *KN_OSIF.C* and Application OS Interface *KNSAMOS.C*. should require little, if any, modification for use with your application.

Note that the WATCOM C library functions *_dos_getvect()* and *_dos_setvect()* are used by procedure *kn_osvaccess()* in the OS Interface Module *KN_OSIF.C* to modify entries in the processor interrupt table. When used with the DOS/4GW DOS Extender, these functions only support modification of vector entries 8 through 15 corrsponding to PC interrupt requests IRQ0 to IRQ7.

Source Files

The source files for the KwikNet DOS/4GW porting example are located in KwikNet installation directory *KNT713\EXAMPLES\DOS4GW*.

5.5.4 Using KwikNet Without an OS

KwikNet can be used without any formal operating system (OS). However, even in this case, a KwikNet OS interface must be provided. The interface must simply operate without the benefit of conventional OS services. This porting example has been built using Mentor Graphics (Microtec) C/C++ software development tools targeted for a 68000 processor.

Since the operating environment is unknown, this porting example has never been executed. However, all of the OS interface procedures have been implemented and will serve as excellent working models for your port.

A hardware clock operating at 1 KHz has been assumed as the fundamental source of timing for your application and KwikNet. The KwikNet clock frequency has been defined to be 20 Hz. It has been assumed that you will provide a clock device driver which will properly initialize the hardware clock when your application begins execution. Furthermore, it is assumed that the driver provides a clock hook which will call an application function coded in C whenever a clock interrupt is serviced.

The console driver for this porting example is configured to use a UART serial driver connected to a terminal. File *KN8250S.C* in the common sample program directory *KNT713\TOOLUU\SAM_COMN* is a simple device driver for an INS8250 or NS16550 compatible UART.

Standard C is used for memory allocation. Memory locking is not required for single threaded applications.

Sample programs which require a file system are configured to use a custom, user defined file system. File access locking is not required for single threaded applications.

> Source Files
>
> The source files for the KwikNet custom non-OS porting example are located in KwikNet installation directory *KNT713\EXAMPLES\XOS*.

This page left blank intentionally.