USNet®
User's Manual

Revision 2.52
December 2004

U S SOFTWARE®
EMBEDDED EXCELLENCE

Copyright and Trademark Information

Copyright 1996-2004 Lantronix, Inc. All rights reserved. No part of this publication may be reproduced, translated into another language, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Lantronix, Inc.

Lantronix®, U S Software®, USNet®, USFiles®, USLink®, SuperTask!®, MultiTask!™, NetPeer™, TronTask!®, Soft-Scope®, and GOFAST® are trademarks of Lantronix, Inc. Other brands and names are marked with an asterisk (*) and are the property of their respective owners.

Lantronix, Inc. makes no warranty of any kind with regard to this material, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Lantronix, Inc. assumes no responsibility for any errors that may appear in this document. Lantronix, Inc. makes no commitment to update or to keep current the information contained in this document.

Lantronix, Inc.
15353 Barranca Parkway
Irvine, CA 92618
(949)453-3990
Fax (949) 453-3995

For Support Contact:
Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxinfo.com
www.smxinfo.com

Documentation Conventions

Computer output and code examples: `Courier`, usually in a separate paragraph.

Function names and command names: ***Bold italic***, usually followed by parentheses, as in ***main()*** function.

Variables: Courier italic (*`mt_busy`*).

File names: Times bold (the file **usrclk.asm**), usually in lower case.

Key names: Initial capital, in angle brackets, as in press <Enter>.

Menu names and selections, dialog box names, screen titles, window titles: Times bold, as in **File** menu**.**

Notes: Indicate important information.

Cautions: Indicate potential damage to hardware or data.

Documentation History

Revision Number	Date
2.1	October 1996
2.50	August 1997
2.50.1	May 1998
2.51	June 1999
2.52	December 2004

Contents

1. Introduction 1

Overview 1

What is Supplied 2

USNet Design Considerations 3
- Size 4
- Adaptability 4
- Clarity 4
- External Support 4
- Packaging 4
- Reentrancy 4
- ROM Residence 5
- Device Drivers 5
- Modularity 5

Recommended Reading 5
- Books 5
- On the Internet 6

Your Experience 6

Overview of the Development Process 7
- Analyzing the Design Problem 7
- Obtaining Design Tools and Verifying Your System 8

2. Quick Start 9

Installation 9
- Installing for Windows or DOS 9
- Installing for UNIX 9
- Directory Structure 10
- Version 10
- Documentation 10

Porting 11

Configuration 11

Compiling USNet 11
- Building the Libraries 11

Running the Main Test Programs 12
- Guidelines for Testing 12
- Configuring netconf.c for Testing 13
- Test 1 - LTEST 15
- Test 2 - EMTEST 17
- Test 3 - MTTEST 20

3. Beginning Your Application 23

Developing a Simple Application 23
Include Files 25
Initializing USNet 25
Establishing a Connection 26
Terminating USNet 28
Compiling Your Application 29
Code Listings 30

Developing Your Application 35

4. Configuration 37

Overview 37

Configuring the Makefiles 38
Editing the config.mak File 38
Editing the compiler.mak File 39

Configuring the Network (netconf.c) 40
Host Name 41
Network Name 41
Network Mask 41
IP Address 41
Hardware Address 42
Flags 42
Link Layer 42
Adapter 42
Parameters 43

Configuring the Drivers 45
Standard Drivers 45
NDIS Drivers 46
ODI Drivers 46

Configuring Local Parameters (local.h) 47
NNETS Macro 48
NCONNS Macro 48
NCONFIGS Macro 48
NBUFFS Macro 48
MAXBUF Macro 48
USSBUFALIGN Macro 49
FRAGMENTATION Macro 49
IPOPTIONS Macro 49
USS_IP_MC_LEVEL Macro 49
KEEPALIVETIME Macro 50
MIB2 Macro 50
RELAYING Macro 50
chksum_INASM Macro 50
DHCP Macro 50
DNS Macro 50
TCP_SACK Macro 51

LOCALHOSTNAME Macro.....51
USERID Macro & PASSWD Macro.....51
LOCALSETUP Macro.....51
LOCALSHUTOFF Macro.....51
USS_PROXYARP Macro.....51

Selecting Protocols 52

5. Dynamic Protocol Interface.....53

Overview 53

Blocking Versus Non-Blocking Operation.....53

Include Files.....54

Initialization and Termination.....54

Ninit.....54
Nterm.....55
Portinit.....55
Portterm.....56

Connections.....57

Open, Close, Read, and Write.....57
Nopen.....58
Nclose.....60
Nread.....61
Nwrite.....62
Dynamic Protocol Interface Macros.....63
SOCKET_NOBLOCK.....64
SOCKET_BLOCK.....64
SOCKET_ISOPEN.....64
SOCKET_HASDATA.....64
SOCKET_CANSEND.....65
SOCKET_TESTFIN.....65
SOCKET_MAXDAT.....65
SOCKET_RXTOUT.....65
SOCKET_IPADDR.....65
SOCKET_OWNIPADDR.....66
SOCKET_PUSH.....66
SOCKET_FIN.....66

Multicast API (DPI).....66

ussHostGroupJoin.....67
ussHostGroupLeave.....67

Examples.....68

Broadcasting Examples.....68
TCP File Transfer Example.....69
Non-Blocking Operations Example.....70

6. BSD Socket Interface.....71

About BSD Sockets.....71

Porting from UNIX ... 72
Porting to UNIX ... 72
Writing New Code ... 72

Structures and Definitions ... 73

BSD Socket Interface Functions ... 73
accept ... 76
bind ... 77
closesocket ... 78
connect ... 79
fcntlsocket ... 80
gethostbyname ... 81
gethostbyname_r ... 82
getpeername ... 83
getsockname ... 84
getsockopt, setsockopt ... 85
ioctlsocket ... 87
listen ... 88
readsocket ... 89
recv ... 90
recvfrom ... 92
recvmsg ... 94
selectsocket ... 95
send ... 97
sendmsg ... 98
sendto ... 99
shutdown ... 100
socket ... 101
writesocket ... 102

Multicast API (BSD) ... 102

7. Network Applications and Protocols ... 103

Overview ... 103

RARP ... 103
Get IP Address ... 104

BOOTP ... 104
Get Boot Record ... 104
Open Connection for Booting ... 104
Read Bootload Data ... 105

DHCP ... 105

TFTP and FTP ... 106
Start Server ... 106
Send File ... 106
Receive File ... 107

Telnet ... 107

IGMP / Multicast ... 108

NAT **108**
Configuration 109

8. Test Programs **111**

Overview **111**

BENCH **111**

DHCPTEST **111**

EMTEST **111**

FTTEST **112**

HTTEST **113**

LTEST **113**

MCRXTEST and MCTXTEST **113**

MTTEST **114**

PING **114**

PITEST **115**

RYTEST **115**

SOTEST **115**

TELNET **116**

TNSERV **116**

UXSERV **116**

9. Porting **117**

Overview **117**

Compiler and Processor Support **117**
Processor Supported But Not Compiler 117
Neither Processor Nor Compiler Is Supported 117

Hardware Configuration **118**
Timer Support 118
Display and Keyboard Support 119
Interrupts 119
Low-Level I/O 120

Porting to a New Multitasking RTOS **120**
Multitasking Configuration 121
Creating Tasks 121
Yielding Control 122
Preemption 122
Signaling 122

10. Device Drivers 127

Overview **127**

Data Structures **127**
Messh (MESSH) Structure 128
Net (NET) Structure 129

Support Functions **130**
Clear Interrupt 131
Disable and Enable Interrupts 131
Install Interrupt Vector 131
Restore Interrupt Vector 132
Map I/O Address 132
Adding Messages to a Queue 132
Removing Messages from a Queue 134
Writing/Reading to/from the Controller 135

Configuring Interrupt Table Size **136**

Configuring a New Processor **136**

Error Codes **136**

Writing a Device Driver **137**

Character Drivers **137**
Interrupt Handler 139
Transmit Routine 140
Open Connection 141
Close Connection 141
Configure and Start Up 141
Shut Down 143
Protocol Table 143

Block Drivers **144**
Interrupt Handler 145
Transmit Routine 149
Open Connection 150
Close Connection 151
Configure and Start Up 151
Shut Down 152
Protocol Table 153

Adapters **153**

11. Performance 155

Benchmarks **155**

Elaborate Compiler Options **155**

Special Benchmark Configurations **156**
Lavish Resources 156
Unusual Test Procedures 156

Design Questions **156**

Copying of Data 157
Drivers 157
Protocol Interfaces 157
Function Structure 157

Benchmark Results 158

Benchmark Details 161
AMD 386, ARCNET, 40 Mhz 161
AMD 386, 40 Mhz 161
AMD 386: 115,200 bps 8250, 40 Mhz 161
Fujitsu SPARClite, 40 Mhz 162
Intel 386, 33 Mhz 162
Intel 386SX, 25 Mhz 162
Motorola 68360, 25 Mhz 162
Two-Hop Routing 163

Benchmark Listings 163

12. Technical Background 167

Overview 167

TCP Retransmission 168

Sliding Window 169

TCP Delayed ACK 170

Congestion Control 171

Silly Window Syndrome 171

ARP Caching 172

A. Terminology 173

B. Trace Output 175

Overview 175

Displaying Trace Data 175

C. RTOS-Specific Information 177

MTOS 177

MultiTask! 179

Hitachi HI-SH7 182

VRTX 184

D. Driver-Specific Information 187

3C509 187

DC21040 .... **188**
DC21140 .... **190**
EN360 .... **192**
I82557 .... **193**
I82595 .... **194**
I82596 .... **195**
MB86960 .... **197**
NE1000 .... **198**
NE2000 .... **200**
NE2100 .... **201**
NS8390 .... **202**
SMC91C92 .... **204**
WD8003 .... **205**

E. Dynamic Configuration of the Routing Table 209

Overview **209**

Routing Table Configuration Functions **209**
ConfLock 210
ConfFree 210
ConfFind 210
ConfDel 211
ConfAdd 211
ConfRename 211
ConfDisplay 212

Index **213**

1. Introduction

Overview

Hello, and welcome to USNet®. USNet is a library of software routines that support TCP/IP protocols. USNet supports the TCP/IP protocols shown in Table 1-1.

Table 1-1: USNet-Supported Protocols

Protocol	Description
TCP	Transmission Control Protocol: Transport layer with connections, flow control and error correction
UDP	User Datagram Protocol: Simple connectionless transport layer
IP	Internet Protocol: The network layer
ICMP	Internet Control Message Protocol: Part of IP for practical purposes
ARP	Address Resolution Protocol: Retrieves a host's network controller's hardware address, given the host's Internet address
RARP	Reverse Address Resolution Protocol: Retrieves a host's Internet address, given the host's network controller's hardware address

The logical relationships between the protocols are illustrated in the figure below:

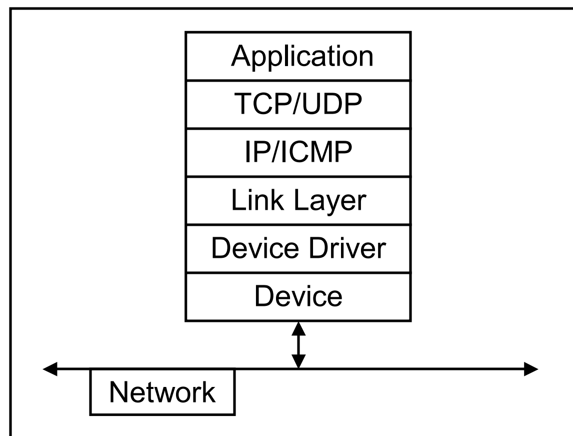


Figure 1-1: Protocol Stack

USNet's TCP/IP protocol suite allows diverse systems to communicate with each other. Typically, USNet software is used in a target embedded system that communicates to a server. The target application interfaces with the outside world, performing some form of data collection. When necessary, the target application opens a connection to the server and transmits the data. USNet takes on the responsibility of providing a reliable connection and reliable data transport when using TCP/IP.

USNet can be used with or without a Real Time Operating System (RTOS). For applications that only require a single connection or multiple connections, an RTOS is not necessary. For applications that manage multiple servers or both clients and servers, an RTOS such as SuperTask!™ or SMX® is required. USNet also provides the flexibility to adapt to any third-party RTOS.

The USNet software library supports a number of different processors, compilers, and RTOS's. Support for one processor plus i8086 real mode is provided in your release. (i8086 support is provided so you can build and run the test programs on PC's.) If your application requires a processor, compiler, or RTOS that is not supported, see Chapter 9, *Porting* for guidance. USNet is supplied with full source code, so you can port it or modify it any way you wish.

USNet offers 2 API's:

1. Dynamic Protocol Interface (DPI) — Simple, proprietary API. See Chapter 5.
2. Berkeley Sockets (BSD) — Standard API. See Chapter 6.

Please refer to Appendix A, *Terminology* for the definition of terms you are unfamiliar with.

What is Supplied

Each USNet delivery CD contains the complete set of source files, makefiles, and device drivers, along with sample timer and display code, interrupt code, and startup code for a DOS or UNIX environment. Files are provided for one target processor plus i8086 real mode.

The delivery CD also contains the source code for sample application protocols and test programs that are useful when building networking into your application. The table below lists the sample applications supplied with USNet.

Table 1-2: Applications Supplied with USNet

Application	Description
FTP	File Transfer Protocol (file transfer using TCP)
TFTP	Trivial File Transfer Protocol (file transfer using UDP)
TELNET	Terminal emulation and remote login
PING	Check if host responds, using ICMP
BOOTP	Bootstrap Protocol
BENCH	Benchmark program to help judge performance of your application
EMTEST	Tests the network connection and the target's ability to support TCP communications
FTTEST	Verifies that your target can communicate with a server
LTEST	Verifies that you can compile, link, load, and execute USNET on your target
MTTEST	Verifies USNET's operation with your target multitasker

USNet Design Considerations

The USNet design considers many of the special requirements of the embedded world, such as:

- Size
- Adaptability
- Clarity
- External support
- Packaging
- Reentrancy
- ROM residence
- Device drivers
- Modularity

Size

The complete TCP/IP protocol, including all needed subroutines but excluding the application level, totals about 25 kilobytes of code on an x86. The protocols can be individually configured, so the minimum system is even smaller than this. The fixed RAM requirement is typically less than 1 kilobyte. Each active connection needs buffer space, which is dynamically allocated with the buffer space requirements depending on the application.

Adaptability

The USNet library is supplied as C Source code, and will operate with any 8-bit, 16-bit or 32-bit processor that has an ANSI C compiler. USNet is written in simple basic ANSI code that will cause trouble only in very unsatisfactory compilers.

NOTE: Look out for long integer and function pointer support in 8-bit compilers.

Clarity

The code does not contain any conditional controls for different compilers or processors. There are absolutely no statements of the form:

```
#ifdef COMPILER_SOSO
do it so-so
#else
do it right
#endif
```

All the support for different byte ordering or word size is invisible to the user.

External Support

The package, as delivered, uses only a few basic ANSI C services, and runs with or without multitasking. Methods are provided to configure a multitasker and to replace the ANSI services.

Packaging

USNet is supplied and configured in source code. The applications are packaged as C subroutines. There are only about 30 external routines, with names not likely to conflict with any other names. The stack frame is kept as small as possible.

Reentrancy

The code is reentrant and can be used with preemptive multitasking and nested interrupts.

ROM Residence

The code is ROMable in a wide sense of the word: All initialized data is type "`const`," and there are no attempts to change code or constants.

Device Drivers

USNet considers drivers as extensions to hardware, and uses a separate data link layer. In other words, the device drivers and link layers are designed as separate modules. This results in short and simple drivers independent of the link layer, and allows new drivers to be added without requiring recoding of the link layer. The link layer processes the link-level protocol such as Ethernet, SLIP, PPP, or ARCnet.

Modularity

In addition to the main stack, USNet offers various add-on modules, such as a web server, NAT support, and SNMP. By separating these from the main stack, you are saved cost and memory space by omitting them if they are unneeded. Most add-ons are documented in separate manuals. Some simpler ones are documented in this manual.

Recommended Reading

This manual documents USNet only. It assumes you are already familiar with TCP/IP. If you are new to TCP/IP, please read one or more of the books listed below. Also, this manual does not go into detail about TCP/IP standards. These are documented fully in the RFC's. See the Internet references below.

Books

TCP/IP Illustrated
Volume 1: The Protocols
W. Richard Stevens
ISBN 0-201-63346-9

TCP/IP Illustrated
Volume 2: The Implementation
Gary R. Wright
W. Richard Stevens
ISBN: 0-201-63354-X

Internetworking with TCP/IP
Volume 1: Principles, Protocols, and Architecture
Douglas E. Comer
Second Edition
ISBN 0-13-468505-9

Internetworking with TCP/IP
Volume 2: Design, Implementation, and Internals

Douglas E. Comer
Second Edition
ISBN 0-13-125527-4

Troubleshooting TCP/IP
Analyzing the Protocols of the Internet
Mark A. Miller P.E.
ISBN 1-55851-268-3

The Simple Book
An Introduction to Internet Management
Second Edition
Marshall T. Rose
ISBN 0-13-177254-6

UNIX Network Programming
W. Richard Stevens
ISBN 0-13-949876-1

On the Internet

RFCs (requests for comment) are a series of documents that represent the TCP/IP standards as they continue to evolve. All RFCs are available over the Internet via anonymous FTP. The most important ones for USNet are:

RFC 768	UDP
RFC 791	IP
RFC 792	ICMP
RFC 793	TCP
RFC 1122	Explanations and clarifications of all the above, also some additions and corrections

Here is an abbreviated example FTPsession:

```
% ftp ftp.rfc-editor.org
.
Name: anonymous
Password: <your email address>
.
ftp> cd in-notes
.
ftp> get rfc1122.txt
.
ftp> quit
```

Your Experience

This manual assumes you are familiar with TCP/IP and related protocols, C programming, make utilities, and your target hardware. For help learning TCP/IP, see the previous section, *Recommended Reading*. It is likely that you will need to become familiar with the assembly language of your target processor. For command-line compilers, makefiles are provided with the source code to make

building the library and your applications easier and more efficient. You should understand how the makefiles work and be familiar with standard utilities that pertain to your compiler/assembler.

If your hardware is not supported, you will need to develop several low-level interface routines. For this reason, you should know how to perform device-level programming for your target hardware, e.g., serial ports, timers, interrupts, etc.

Overview of the Development Process

The following text provides an overview of the typical process used to develop embedded networking applications using USNet. The overview assumes that you have met the hardware, software, and experience requirements given earlier in the manual.

There are approximately ten (10) basic steps in the development process:

1. Analyze the design problem and its constraints.
2. Obtain and install all of the development tools and verify their operation.
3. Install USNet (at first without multitasking).
4. Compile, load, execute, and verify USNet's **LTEST** application on your target.
5. Configure and compile USNet's **EMTEST** application for your target.
6. Verify that the Network Controller hardware, network servers, and network cables are functional.
7. Load, execute, and verify USNet's **EMTEST** application on your target.
8. If you are using multitasking, reconfigure USNet with an RTOS. Compile, load, execute, and verify USNet's **MTTEST** application on your target.
9. Develop and debug your application.
10. Generate your production code. Set the macro **TRACE_DEBUG** in config.mak to 0 (to optimize code space).

Steps 1, 2, and 3 are covered in the remainder of this chapter. Steps 4 through 8 are discussed at length in Chapter 2, *Quick Start*.

Analyzing the Design Problem

Proper configuration of USNet and its dependencies is crucial to the success of your application. For example, you must select a target processor that can handle all of the tasks required by the application. You will need to consider whether or not you will use a multitasking OS. When analyzing the application, you might want to ascertain the minimum network throughput and response time requirements. You should know such things as what ROM/RAM resources are available to the application and whether there is enough room for the target application. It might be necessary to compile USNet to know how much code space it will use, or to do a timing and resource analysis to ensure adequate load and resource headroom. Be sure to allow room for additional protocols or client/server applications that you might decide to use later.

Obtaining Design Tools and Verifying Your System

If possible, compile and load some simple test programs on the target hardware. Verify that you can use your debugger or ICE tools while executing your test program on the target.

2. Quick Start

Installation

USNet is delivered on a CD-ROM. There is no installation utility. Simply copy the files to your hard disk and change the attribute of all files from read-only to read/write. For UNIX, also change CRLF to LF. These steps are detailed in the next two sections.

You must have a recent version of Opus Make installed on your development system (e.g. v6.12). If you do not already have it, please purchase it directly from www.opussoftware.com. All of USNet's makefiles are written to use Opus Make. Note that a few USNet configuration files are automatically generated when you do a make.

Installing for Windows or DOS

To install to Windows or DOS:

1. Copy the USNet, USFILES, or USSW directory to your hard disk. Use Windows Explorer or xcopy. For example:

   ```
   xcopy /s/v d:\usnet\*.* c:\usnet
   ```

2. Change into the directory and type:

   ```
   attrib /s /d –r
   ```

   to clear the read-only flag of all files and directories.
   /s recurses subdirectories; /d changes the flag for the directories too.

3. Manuals are supplied in the \MANUALS directory and at www.smxinfo.com/2. Copy these to your hard disk, if desired.

4. Release notes are supplied in the DOC subdirectory of USNET, USFILES, or USSW. Please take time to review these files and the readme.txt file in the installation root directory.

Installing for UNIX

1. Copy the USNET, USFILES, or USSW directory to your hard disk:

   ```
   mkdir usnet   (or usfiles or ussw)
   cd usnet
   cp -R /cdrom/usnet .
   ```

2. Enable read/write permissions for all directories and files:

```
find . -type d | xargs chmod 755
find . -type f | xargs chmod 644
```

3. Change CRLF to LF in all files:

```
find . -type f | xargs perl -pi -e 's/\r\n/\n/g'
```

4. Manuals are supplied in the \MANUALS directory and at www.smxinfo.com/2. Copy these to your hard disk, if desired.

5. Release notes are supplied in the DOC subdirectory of USNET, USFILES, or USSW. Please take time to review these files and the readme.txt file in the installation root directory.

Directory Structure

USNet is installed into a hierarchial directory structure, as shown:

appsrc	Test programs and applications
config	Tool chain definition
<cpu>	CPU-specific files
<compiler>	Compiler-specific files
doc	Additional documentation
drvsrc	Drivers and CPU support
<cpu>	CPU-specific files
<compiler>	Compiler-specific files
include	USNet header files
lib	USNet libraries
netsrc	Core USNet source code
supsrc	Low-level code common across other products in this family
unsupp	Unsupported USNet utilities

Other directories may be present if you have purchased USNet add-on packages. These packages are delivered on separate disks:

iapsrc	USNet Internet Access Package
pppsrc	USNet PPP support package
snmpsrc	USNet SNMP package

Version

The USNet version number should be marked on the CD-ROM, but if not, check the file netsrc\vsnlog.txt in the delivered code.

Documentation

Most USNet add-on products are documented in separate manuals.

Manuals are supplied in PDF format in the \MANUALS directory of the CD-ROM and at www.smxinfo.com/2. Also see the text files in the DOC directory for important additional information.

Porting

USNet supports many processor, compiler, and RTOS combinations. However, if you need to port to a new one, see the relevant sections in **Chapter 9, *Porting*** for guidance, and then return here. However, before doing the port, you may want to follow the steps in this section using DOS PC's, so you can get familiar with USNet.

USNet provides many device drivers for network controllers. However, if yours is unsupported, please see **Chapter 10, *Device Drivers*** for the information you need to develop your own driver.

Configuration

Before you can build the USNet library and test programs, you must do some basic configuration of USNet. You must specify the processor and compiler you are using and the path to the compiler, assembler, and other tools. Also, you must specify the path to the USNet root directory and select which add-on products you are using. These are configured in the makefiles config.mak (in the USNet root) and compiler.mak (in the directory config\<cpu>\<compiler>).

See **Chapter 4, *Configuration*** and then return here.

Compiling USNet

As mentioned previously, USNet uses the Opus Make utility to build the libraries and sample application programs. If other **make** utilities are installed on the development system, you might need to rename the Opus Make executable (e.g. omake.exe). Another approach is to write a small DOS batch file that calls Opus Make. When taking this approach, keep in mind the **make** utility can take several command-line options.

The makefiles are set up to compile the libraries as a default build. To specify a specific target program, add the name of the program after the **make** command. If the project needs to be completely rebuilt, specify a target of "clean" which will cause the makefiles to delete libraries, object, executable, and map files. More detail on compiling is presented in each section below.

Building the Libraries

Once files **config.mak** and **compiler.mak** have been properly defined, the USNet libraries can be built. This is done by following these steps:

1. Change the directory to the USNet install directory:

   `cd usnet`

2. Execute the **make** command at the prompt by typing:

   `make` `(or omake)`

The **make** command initiates the build process and must be associated with Opus Make. The top-level makefile will call makefiles in the subdirectories which will build each object file. Prior to building an application program, the object files are consolidated into a library called **lib\ussw.lib.** This library is then linked with your application.

Two source code files might require modification in order to run USNet's test programs. The first file, **netconf.c**, located in the **netsrc** directory, defines the hosts that are on the network. Each host is

defined by several parameters such as name, IP address, and Ethernet address. The second file, **local.h**, resides in the **drvsrc\<***cpu***>** directory (where <*cpu*> is the CPU as defined in **config.mak**) and is used to define how USNet is configured for the application. The number of physical connections, buffers, and other TCP/IP options are set here. For testing purposes under the conditions assumed, only **netconf.c** will need to be modified. File **local.h** and its parameters are described in section *Configuring Local Parameters* of Chapter 4, *Configuration*.

Running the Main Test Programs

USNet includes many test programs, which are documented in Chapter 8. Here we show how to configure and run the main test programs. You should run them first on PC's as a confidence test and then on your target hardware. The developers of USNet use these test programs to bring up and verify the porting layer for the supported platforms. Because of this, the procedures in this section are possibly the most effective means of getting USNet running on your target. We recommend that you complete each test before going to the next. The tests outlined below assume you are using USNet-supported hardware and software. If you are not, please refer to Chapter 9, *Porting,* to complete any unresolved interface requirements before starting the tests in this section.

Guidelines for Testing

If USNet supports your processor and compiler, the included makefile, as shipped to you, was tested and found working. Please use it to:

1. Build the USNet library.
2. Build the USNet test programs.

If you have trouble with the makefile, contact us for support. We recommend that you initially **not** attempt any of the following:

- Integrating the USNet makefile and your own makefile.
- Converting the makefile to a different tool, such as the Borland IDE, or the old Microsoft **make**.
- Testing without a makefile.

Here are some guidelines that should help in USNet integration testing:

- Become comfortable with your toolchain. Prepare batch commands, increase download baud rates, talk with your toolchain vendors.
- Test using the USNet trace output. Do not try to save time or trouble by not implementing character display. (See Chapter 9, *Porting*, and Appendix B, *Trace Output*.)
- Do not start with untested hardware. If you don't have any diagnostics available, get a commercial board that is reasonably close to your own and run USNet in that board. Then move to your own hardware.
- As much as possible, make sure that all the network cabling is verified before you start testing.

- Test just one unknown at a time, and proceed step by step. First run LTEST, then EMTEST, then (optionally) MTTEST, then your application.
- Always keep the last test that worked as a fallback position. Whenever a test fails, go back to what works and retry that. (A cable may have become loose!) Then try a different, smaller step.
- Set `TRACE_DEBUG = 3` in config.mak to help report error conditions in the stack. Do a **grep** or **search** on "`Nprintf`" in the stack modules to locate error traps.
- The header file **net.h** contains error return number translations and meanings.
- Use the function ***Nprintf()*** in your test programs as a trace output tool.
- Use a LAN analyzer to capture and troubleshoot your test programs' data traffic during stack communications.

Configuring netconf.c for Testing

Before the test programs are compiled and run, file **netconf.c**, in the **netsrc** directory, must be configured for the test network. For full discussion of netconf.c, see section *Configuring the Network* in Chapter 4, *Configuration*.

Typical configuration for the two machines in the test network can be defined as follows:

Development Machine

Name:	develop
IP address:	192.31.23.20
IP mask:	Class C
Network Card:	NE2000
Interrupt number:	10
Ethernet port:	0x300

Target Machine

Name:	target
IP address:	192.31.23.25
IP mask:	Class C
Network Card:	NE2000
Interrupt number:	11
Ethernet port:	0x320

Place the above information into the `netdata[]` array in file **netconf.c**. USNet uses `netdata[]` as its definition of the network topology. To configure the `netdata[]` array:

1. Edit file **netconf.c** located in directory **netsrc**.
2. Page down to the definition of `netdata[]`.
3. Comment out all entries in the table. These entries are provided as examples for different types of network connections.

4. Add an entry for a loop test, used by **ltest**, by inserting this line:

```
"loop", "ether", C, {127,0,0,1}, EA0, 0, Ethernet, WRAP, 0, 0,
```

5. Add an entry for the development system by inserting this line:

```
"develop", "ether", C, {192,31,23,20}, EA0, 0, Ethernet,
NE2000, 0, "IRNO=10 PORT=0x300",
```

6. Add an entry for the target system by inserting this line:

```
"target", "ether", C, {192,31,23,25}, EA0, 0, Ethernet,
NE2000, 0, "IRNO=11 PORT=0x320",
```

NOTE: In steps 4, 5, and 6, the commas and quotes are necessary.

7. Save and exit **netconf.c.**

Now **netconf.c** is configured to run **ltest.c** and **fttest.c**. These are the fields in each entry, described by position:

Field	Description
first	The machine name, which must match the name returned by the macro ***LOCALHOSTNAME()*** as defined in **local.h**.
second	The network name. A mnemonic to describe the network to which the host is connected.
third	The network mask to be used by the stack.
fourth	The host's IP address.
fifth	The host's network card's Ethernet address. Set this field to zero (`EA0`) if the network card has its address already defined on board.
sixth	A flag used to tell what roles the host is to play on the network, such as a DNS server, router, or dial-up connection. If the host is strictly communicating with other hosts on the network, set the sixth field to 0.
seventh	The link layer to be used in the stack. For the test network defined above, the link layer will be Ethernet.
eighth	The name of the USNet driver to use for the link layer interface board.
ninth	The adapter driver used for PCMCIA connections.
tenth	The initialization information for the driver. It defines the Interrupt number, port address, and other driver-specific information.

One possible difference between the assumed setup and the application is that the application target might have more than one physical network connection. In this case, the host will have more than one entry, one for each connection. Each entry will have the same host name, but a different IP address and network name, possibly a different driver, and a different initialization string to reflect the hardware differences between network interfaces.

Test 1 - LTEST

LTEST Overview

LTEST is highly recommended as the first test to run when verifying USNet on a new system. In order to reduce complexity, simplified functions are used for the network interface and the system clock.

LTEST uses a wrap driver while executing read/write tests on your target. (It can also be run on a PC in DOS or from the Windows 98 or XP command prompt. You may wish to try this first before running on your target, especially if you have to port USNet to it first.) It sets up a TCP connection through a loopback device driver, so that all communication takes place within the unit under test. It exercises a number of features of the TCP layer by forcing unusual but valid behavior in the outgoing TCP segments. These behaviors are introduced by writing directly to internal data structures, which may create some issues for future maintenance, but this method is simple and allows important features to be easily tested.

LTEST displays trace data during execution. The trace port is most commonly configured as an RS232 port on your target. If the test is successful it will display about 290 lines of trace data with "`No errors in LTEST`" at the end of the trace. If you don't have trace capability, you can use your debugger to verify execution results by setting various breakpoints in LTEST.

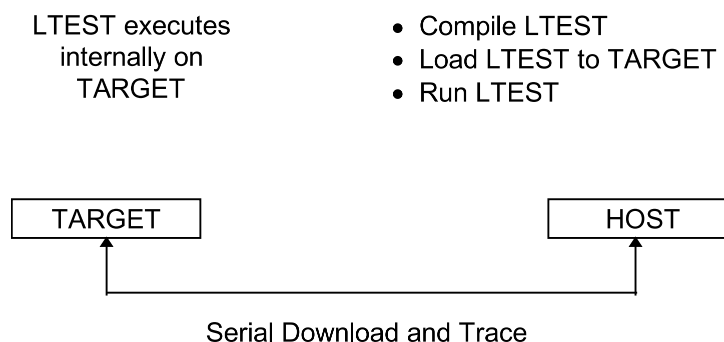


Figure 3-1: LTEST Configuration

LTEST Goals

There are two basic goals of LTEST:

- To verify that you can build LTEST with your selected compiler
- To verify that LTEST loads and runs on the target

LTEST Configuration Requirements

Before you can build LTEST, you must first:

1. Determine whether the USNet-supplied character display routines support your hardware. There are some cases where you might have to replace the ***Nputchr()*** routine with your own version. (See Chapter 9, *Porting*.) This function displays character trace output while LTEST is running.

LTEST Development Requirements

1. To build LTEST on your development platform, type:

   `make ltest`

2. Load LTEST onto your target.
3. Execute and verify that LTEST ran. Make sure that the ***LOCALHOSTNAME()*** function will return the string "loop" when running this test. If your target is a PC, just type "**`set host=loop`**" and then "**`ltest`**" at the command line; otherwise, have your startup routine vector to the LTEST `main` on reset.

LTEST Pass Indicators

LTEST will display the following trace output if the test passed.

This concise listing was created with the `TRACE_DEBUG` constant set to 1 in config.mak.

```
ARP 767676767676 -> 192.9.202.1
ARP 767676767676 -> 192.9.202.1
***SEND AND RECEIVE 20 MESSAGES
—20 MESSAGES OK
***FRAGMENTATION
***FRAGMENTATION WITH RETRANSMIT
reTX1 14900 C1/204 ST1 SQ2669 MS741
—FRAGMENTATION OK
***SEQUENCE NUMBER ROLLOVER
—ROLLOVER OK
***OVERLAPPING MESSAGES
—OVERLAP OK
***OUT OF ORDER MESSAGES
—OUT OF ORDER OK
***DUPLICATE MESSAGES
—DUPLICATE OK
***RETRANSMISSION
reTX1 45399 C1/204 ST2 SQ77c MS298
```

```
—RETRANSMISSION OK
no errors in LTEST
```

Potential Sources of Failure for LTEST

Here are some sample problems that would cause LTEST to fail. Since LTEST doesn't use any target resources other than the CPU, RAM, and ROM, most problems are due to errors in environment initialization.

- Target stack space is too small.
- Target memory RAM/ROM control registers are not set up properly.

You must ensure that LTEST completes successfully before continuing with the next test.

Test 2 - EMTEST

EMTEST Overview

EMTEST should be the second test you run when verifying USNet on a new system.

EMTEST acts as an FTP client and verifies that your target can communicate with an FTP server. A sample file is repeatedly written to and retrieved from an FTP server, and any errors are reported. This test program is well-suited to embedded systems because it uses simulated file i/o (i.e. the file is built in memory and no file system is required). Running this program overnight is useful for demonstrating that there are no long-term problems with the implementation, such as message buffer loss.

EMTEST will require your attention to detail when completing all configuration tasks. You will edit **netconf.c** to configure your test network and **drvconf.h** to define your driver link. You will know that EMTEST is working by observing the trace output.

The FTP server that you run EMTEST against can be a UNIX FTP server or an FTP server utility running on a PC. You can also run EMTEST against the USNet FTTEST program running on another host. See Chapter 8 for information about building and running FTTEST. We recommend that for this quick start you use a UNIX or PC FTP server, such as the following:

1. FileZilla Server: http://sourceforge.net/projects/filezilla/
2. War FTP Daemon: http://www.warftp.org/
3. GuildFTPd
4. Serv-U
5. WS_FTP Pro

EMTEST Goals

There are three basic goals of EMTEST:

- To verify EMTEST builds

- To verify data communication through the driver and network interface controller to the server
- To successfully run EMTEST

EMTEST Configuration Requirements

Before you can build EMTEST, you must:

1. Configure `TRACE_DEBUG =3` in the config.mak (in Chapter 4, *Configuration*, see *Configuring the Makefiles*).
2. Configure the network configuration file to match your test setup (in Chapter 4, *Configuration*, see *Configuring the Network*).
3. Configure the driver configuration file (in Chapter 4, *Configuration*, see *Configuring the Drivers*).
4. Configure your local parameters. In most cases the installed defaults will work. (In Chapter 4, *Configuration*, see *Local Parameters*.)
5. Make sure that your target host name matches the host name in **netconf.c** (in Chapter 4, *Configuration*, see *Local Parameters*, *Hostname*).
6. Set the define: `SERVER="develop"` in **EMTEST.C** to match the server's host name in **netconf.c**.
7. Edit **local.h** if you need to change the login, userid, or password that is used when logging in to the ftp server.
8. As discussed above, we recommend that you run EMTEST against a UNIX or PC FTP server. However, if you wish to run against FTTEST (on a DOS PC), you must:
    a. Install USNet to a new directory and configure the build to use the I8086 processor. This will allow you to build real-mode USNet test programs.
    b. Edit the makefile `PTH` parameter to reference your compiler.
    c. Configure **netconf.c** to specify your network connection to the server.
    d. Build FTTEST.
    e. Use the "`set host=develop`" at the DOS prompt to set the server's host name to the same name as in **netconf.c.**
    f. We recommend starting the server with a debugger; otherwise, you will have to reset your computer every time you want to exit the server. For efficiency the server has no exit command. For example: **`TD FTTEST`** would start the Borland debugger, after which you can start FTTEST by pressing <F9>.

EMTEST Development Requirements

1. Check that your network controller(s) work while using other applications or test programs.
2. Verify that your cables are terminated, and plugged in correctly.
3. Verify that your UNIX or PC FTP server works using another test program or application.
4. If you are using a UNIX FTP server, configure the **/etc/hosts** file with the IP address of the target.

5. If you are using FTTEST as a server, you can verify that it operates by connecting two PC's together: One PC runs FTTEST as a server, the other as a client.

   a. To start the server, at the DOS prompt type:

   ```
   set host=develop
   FTTEST
   ```

   b. Type FTTEST "`server`" at the DOS prompt for the client.

   ```
   set host=target
   FTTEST "server"
   ```

6. Build EMTEST on your development platform.
7. Load EMTEST onto your target.
8. Connect the server and target systems.
9. Make sure the server is running, and then execute EMTEST on the target.
10. Verify that EMTEST ran.

EMTEST Pass Indicators

The trace output will show continuous communication between the target and the server. In this test the trace output has some diagnostic significance. (See Appendix B, *Trace Output*.) On most UNIX servers, the connection resources will eventually be used while the test is running, and the test will subsequently stop.

Potential Sources of Failure for EMTEST

As you have already determined, there any many things that can go wrong with this test. Be certain that all your hardware is working and has been tested outside of this test. Verify that none of the problems listed below exist:

- Driver I/O address is incorrect in **netconf.c**
- Driver interrupt is incorrect in **netconf.c**
- IP address mismatch between server and target (see **netconf.c** for target and PC server or the **/etc/hosts** table for the UNIX server)
- Incorrect driver for Ethernet controller
- Network card I/O address or interrupt conflicts with other hardware resources
- Network cables are not terminated, grounded, or connected
- Network cables have shorts or opens
- Network card hardware is damaged
- The host name in the PC does not match the host name in **netconf.c**

Test 3 - MTTEST

MTTEST Overview

The purpose of MTTEST is to verify USNet's operation with your multitasking Real Time Operating System (RTOS). If you are not using an RTOS you can skip this test. If your RTOS is not supported by USNet, refer to Chapter 9, *Porting*, and complete all USNet RTOS interface porting and configuration tasks before returning to this test. It is especially important to run this test if you have ported to a new RTOS.

The initial part of this test exercises basic OS features used by USNet. The next part sends and receives data using separate tasks. MTTEST is both a server and a client program; it must be run against another copy of MTTEST. To simplify things, the peer copy can be run under DOS using a "no RTOS" version of USNet. Or UXSERV can be run on a UNIX machine, against MTTEST. The test program consists of two tasks that communicate as shown in Figure 8-1.

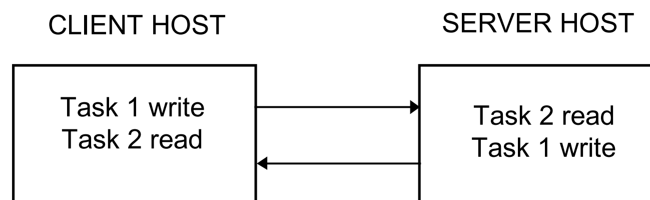


Figure 8-1: MTTEST

The file **mttest.c** contains both the server and the client code. MTTEST uses the symbolic constant **SERVER** and the **LOCALHOST** macro to determine if it should run as a client or a server. SERVER is #defined in **mttest.c**, and **LOCALHOST** is defined in **local.h**. If SERVER is the same string as the value returned by **LOCALHOST**, then the system will run as the server. Otherwise, the system will run as the client.

The protocol used is TCP. The tasks read and write at full speed doing no waiting or checking. (The read tasks do check the data though.) The test will very quickly find out if there is something wrong with the multitasking support. It is also a good test of the TCP flow control, because it overloads just about any data link you can give it.

To start the test, first start the server, and then the client. The test will run until stopped with the <Escape> key, or until there is an error.

Different versions of MTTEST have been written for the operating systems supported by USNet. If you are porting to a new RTOS, you need to make appropriate changes to **mttest.c**, and of course you need to configure the multitasking support in the header **mtmacro.h.**

MTTEST Goals

There are three basic goals of MTTEST:

- To verify MTTEST builds

- To verify that the target RTOS works with USNet
- To successfully run MTTEST

MTTEST Configuration Requirements

Before you can build MTTEST, you must:

1. Install USNet to a new directory for the MTTEST server. The method depends on whether your computer is MS-DOS or UNIX.

   For MS-DOS computers:

   a. Install USNet to a new working directory and configure the build for an I8086 processor and no multitasking. Building for the "i8086" processor will allow you to build real mode USNet test programs.

   b. Edit the makefile `PTH` constant to reference your compiler.

   c. Configure **netconf.c** to specify your network connection to the server.

   d. Build MTTEST.

   e. Enter `"set host=develop"` at the DOS prompt to set the server's host name to the same name that is in the configuration file.

   f. We recommend starting the server with a debugger. For example: "**`TD MTTEST`**" would start the turbo debugger, after which you can start MTTEST by pressing <F9>.

   For UNIX computers:

   a. Copy the file **uxserv.c** to a working directory on the UNIX host.

   b. Compile **uxserv.c** with a native C compiler. All support for this program comes from standard libraries.

   c. Run **uxserv** to start the server side of the test.

2. Install USNet to a new directory for the multitasking target.

NOTE: If you are using a new RTOS you will have to adapt USNet's RTOS macros to your target's RTOS. To do this, see Chapter 9, *Porting*. Once you have completed the interface tasks, you can skip the next bullet. If USNet does support your RTOS you will want to continue to the next bullet.

   a. Edit the makefile `PTH` constant to reference your compiler.

   b. Configure **netconf.c** to specify your network connection to the server (in Chapter 4, *Configuration*, see *Configuring the Network*).

3. Set the define: `SERVER="develop"` in **MTTEST.C** to match the server's host name in **netconf.c**.

4. Configure the driver configuration file (in Chapter 4, *Configuration*, see *Configuring the Drivers*).

5. Configure your local parameters (in Chapter 4, *Configuration*, see *Local Parameters*).

6. Start the multitasking target. For an MS-DOS system, enter "`set host=target`" at the DOS prompt to set the target's host name to the same name that is in the configuration file. Then launch mttest.

MTTEST Development Requirements

Other than the RTOS requirements, MTTEST is similar in requirements to EMTEST.

1. Check that your network controller(s) work while using other applications or test programs.
2. Verify that your cables are terminated and plugged in correctly.
3. Verify that your server works using another test program or application.
4. If possible, execute MTTEST between two PCs before connecting an unknown MTTEST server application to an unknown MTTEST target application.
5. Build MTTEST on your development platform.
6. Load MTTEST onto your target.
7. Start the MTTEST server.
8. Start MTTEST on your target.

MTTEST Pass Indicators

MTTEST will show continuous communication between the target and the server. For a definition of the trace, see Appendix B, *Trace Output*.

Potential Sources of Failure for MTTEST

MTTEST is as complex as EMTEST with the addition of an RTOS interface. During your integration, you may have to develop very simple test programs to flush out the following RTOS interface problems:

- RTOS macros have defects
- Driver I/O address is incorrect in **netconf.c** for either server or target
- Driver interrupts are incorrect in **netconf.c** for either server or target
- IP address mismatch between server and target (see **netconf.c** on target and PC server)
- Incorrect driver for Ethernet controller on server or target
- Network card I/O address or interrupt conflicts with another card in the PC
- Network cables are not terminated, grounded, or connected
- Network cables have shorts or opens
- Network card hardware is damaged
- The host name in the PC server does not match the host name in **netconf.c** for the server

Verify that MTTEST executed on the target without any failures.

3. Beginning Your Application

Developing a Simple Application

Before developing your full application, it is instructive to develop a small simple first application. Many of the problems encountered during development are eliminated by first working through the test programs and creating a simple application. This section describes the rudimentary design of an application consisting of a server program and a client program. The server will wait for the client to establish a connection, then will wait for the client to send a request for data. Once the client has established the connection to the server, it will send a request for some number of bytes of data. The server then begins sending a buffer of data for a predefined number of times, while the client reads the data, checks the data's integrity, and sends a confirmation message. The server and client machines are assumed to be the `"develop"` and `"target"` entries, respectively, in **netconf.c**.

The code presented in this section is intended to illustrate USNet's Dynamic Protocol Interface (DPI) as simply as possible; therefore, some of the code might seem inefficient. Refer to Chapter 5 in this manual for more information on the DPI. If the application requires BSD sockets, also consult this manual for information about USNet's BSD interface.

NOTE: The choice between TCP and UDP must be thought through properly. A common misconception is that data transferred via TCP arrives in packets. Data transferred by TCP should be thought of as a stream. If an application calls the write function three times, each time writing 20 bytes of TCP data, the local stack may be combine this information into a single TCP segment with a 60-byte data payload. The remote side read will then receive one 60-byte data chunk. The application-layer protocol is responsible for parsing the data into useful information.

The first question to answer about a first application is "What is the data to be exchanged?" Most of USNet's test programs send a buffer of sequential numbers that can be easily checked by the remote host.

If the numbers in the received buffer do not match up, an error is generated. This type of data is probably the easiest to generate and check quickly. An application can construct such a buffer of data with this code:

```
#define DATA_SIZE 100      /* Number of bytes in buffer */
   .
   .
   .
unsigned short count;                /* Index counter */
unsigned char junk[DATA_SIZE]        /* Buffer */
   .
   .
   .
```

```
for(count=0;count<DATA_SIZE;count)
  junk[count] = count%256;           /* Number is 0 -> 255 */
  .
  .
  .
```

Once the data has been received, the buffer can be checked by a similar section of code:

```
  .
  .
  .

/*
**  Data received and stored in junk[]
*/
for(count=0;count<DATA_SIZE;count){
  if (junk[count] != count%256 )
  Nprintf(" BAD DATA ");
}
  .
  .
  .
```

The next question that needs to be addressed is "What roles do the server and client play?" Do they exchange data? Does one side control the other? What protocols should be used in the exchange? The server's role in the application outlined above is very basic. It will control the transfer of a buffer as outlined above, to the client via a TCP connection. The client's role is to receive the buffer from the server, then check the data's integrity. This type of transfer could be used to send control information from a server to a factory floor or to a remote sensing station.

Once the crucial design questions have been answered, the server and client need to be defined. File **netconf.c** already should contain the hardware definitions for the server `"develop"` and client `"target"` so no further hardware information needs to be declared for USNet. Since the application is going to be using TCP, port numbers must be assigned to both sides of the connection. Port numbers must be consistent between the server and client. Because the server is going to perform a passive open, it will listen on its port for incoming messages from any remote site's port. The client side must receive and send to the same server port. The following section of code defines the server- and client-specific information:

```
        .
        .
        .
#define SERVER_NAME "develop"  /* Server name in

netconf.c */
#define SERVER_PORT      1500  /* Server port number */
#define DATA_SIZE         200  /* Data buffer size in

bytes */
#define ITERATIONS         10  /* Number of passes */
        .
        .
        .
```

This information must be included in both the server and client programs. For the outlined sample application, this information is stored in file **firstapp.h**. A listing of **firstapp.h** is included at the end of this section. The server name must match the name of the server host and the name in **netconf.c**. Port numbers below 1024 have conventions regarding their use, so for general applications select port numbers greater than 1024.

The server and client programs will be very similar. There will be two differences between the two programs: First, the client will have a complimentary set of ***Nread()*** and ***Nwrite()*** functions to that of

the server. Second, the client will check the integrity of the incoming data. Other than these two differences, the overall design considerations are the same. The design of the server will be presented first, then the client design will be shown but without the detailed explanations.

The server program will have the name **server.c** and the client, **client.c**. Both these files must reside in the **appsrc** directory. Any program using USNet requires four main features:

- Include files
- Initialization
- The establishment of a connection
- Termination

Include Files

Three USNet header files must be included in the proper order at the top of **server.c**. The files in their proper order are:

```
   .
   .
   .
#include "net.h"               /* Prototypes and definitions */
#include "local.h"             /* USNet configuration */
#include "support.h"           /* Support function prototypes */

/*
**  Include application-specific information
*/
#include "firstapp.h"
   .
   .
   .
```

The file **net.h** contains the function prototype information and type definitions. File **local.h** contains USNet's configuration, such as the number of physical connections, buffers, and options. Finally, the file **support.h** contains prototypes of internal support functions. If the application requires the use of BSD sockets, the file **socket.h** replaces **net.h** and **support.h**. Any application-specific information stored in a header file should also be included.

Initializing USNet

Two functions are required to initialize USNet. The first, ***Ninit()***, will zero all data structures, move the `netdata[]` table from ROM to RAM, and initialize the stack. The second, ***Portinit()***, will call the initialization routines for all the drivers belonging to the local host. If there is more than one physical connection defined in `netdata[]`, each will be initialized. This is where the second field in `netdata[]` comes into play. Both of these functions return a negative number if an error has occurred. Inside the ***main()*** function of the server, add this initialization code:

```
int main(void){
      int error_code;
            .
            .
            .
      error_code = Ninit();   /* Initialize USNet */
      if( error_code < 0 ) return (error_code);

      error_code = Portinit("*");
                              /* Intialize all connections */
      if( error_code < 0 ) return (error_code);
            .
            .
            .
```

Function ***Ninit()*** does not take any parameters. Function ***Portinit()*** takes a single parameter defining the physical connection to be initialized. If only the Ethernet connection, `"ether"` from **netconf.c**, on host `"develop"` is to be initialized, substitute `"ether"` for `"*"`. The `"*"` indicates USNet should initialize all physical connections for the local host.

Establishing a Connection

Once the initialization is complete, the server can open a connection via the ***Nopen()*** function. Since the server is going to be doing a passive open, it will remain in the ***Nopen()*** function until the client establishes a connection. If the connection was successfully established, ***Nopen()*** will return a connection number; otherwise, it will return a negative number indicating an error. The connection number is used by the ***Nread()*** and ***Nwrite()*** functions to indicate on which connection the operation is to be performed.

The following code will create a passive open in the server:

```
        .
        .
        .
  /*
  ** Perform a passive open on port SERVER_PORT
  */
  conno = Nopen("*", "TCP/IP", SERVER_PORT, 0, 0);
  if( conno < 0 ) return (conno);
        .
        .
        .
```

Function ***Nopen()*** takes five parameters:

Parameter	Description
first	Specifies the name of the remote host. * indicates the server should accept a connection from any host defined in `netdata[]`. To do an active open to a client, the "*" could be replaced with `"target"`.
second	Tells USNet what protocol will be used in the connection. Other valid options are "UDP/IP" or "ICMP/IP".
third	Tells USNet which port the local host will be using.
fourth	Indicates which port the remote site will be using. Since the server is doing a passive open, the fourth parameter is zero to indicate the server should accept a connection from any port at a remote host.
fifth	A flag that can instruct USNet to do a non-blocking open if set to `S_NOWA`.

If ***Nopen()*** returns with a connection number, the client has established a connection. Now the server will wait for the client's request, then begin transferring the data through the established connection by using the ***Nwrite()*** function to send the data. An ***Nread()*** function receives confirmation from the client if the data was intact. Both functions return the number of bytes written or read if successful; otherwise, they return a negative error number. To write the buffer of sequenced data and check for the client response, add this code to **server.c**:

```
   .
   .
   .
/* Call to Nopen() returns conno here */
   .
   .
   .
/* Build junk[] data here */
   .
   .
   .
/*
**  Loop through data transfer.  ITERATIONS
**   defined previously in code.
*/

for(i = 0; i<ITERATIONS;i++){
   /*
   **  Wait for request for number of bytes to send
   */
   error_code = Nread(conno, data_req, sizeof(data_req);
   if( error_code < 0 ) return (error_code);
         .
         .
         .
   /*
   ** Convert data_req buffer to integer data_requested here
   */
         .
         .
         .
   /*
   **  Write data
   */
   error_code = Nwrite(conno, junk, data_requested);
   if( error_code <= 0 ) return (error_code);


   /*
   **  Read client response
   */
   error_code = Nread(conno, status, sizeof(status));
   if( error_code < 0 ) return (error_code);
}
   .
   .
   .
```

Both the ***Nwrite()*** and ***Nread()*** functions take three parameters. The first is the connection number, which specifies the connection that will be used for the transfer. In the example above, the connection number, `conno`, was returned by the ***Nopen()*** performed earlier. The second parameter for ***Nwrite()*** is the buffer containing the data to send, and for ***Nread()*** the buffer is the storage place to receive the data. The final variable is the maximum length of the buffer for ***Nread()*** and the data length to write for ***Nwrite()***. The length is specified in bytes.

Terminating USNet

After the data exchange is complete, both sides of the application are ready to terminate USNet. Each function in the termination sequence is a reciprocal function to those called to establish a connection. Therefore, the first thing to do is close the connection by calling ***Nclose()***. Finally, USNet is terminated by calling ***Nterm()***, which actually calls the ***Portterm()*** function to shut down the physical connections. Add the following code to **server.c**:

```
        .
        .
        .
   /*
   **  Terminate USNet
   */
   Nclose(conno);                       /* close the connection */
   Nterm();                             /* Terminate USNet */
   return 0;
}                                       /* End of main */
```

Function ***Nclose()*** takes a single parameter, the connection number, `conno`, returned by ***Nopen()***. For every open connection, a call to ***Nclose()*** is required. Function ***Portterm()*** also takes a single parameter, the physical connection that needs to be shut down. In the defined application, ***Portterm()*** could take the parameter "`ether`" since the local host has a single physical connection defined in the `netdata[]` table. The parameter "`*`" indicates all connections should be shut down. Finally, ***Nterm()*** does not take any parameters.

A source code listing of **server.c** is included at the end of this section. The code listed is slightly more complete than the code included above. It also contains comments describing what each section of code is doing.

For **client.c**, the overall structure in the program is the same, with two differences between the USNet calls themselves. The include files, defined constants, and the call to ***Ninit()*** are the same as in **server.c**. The first difference is in the call to ***Nopen()***. Program **client.c** will do an active open to machine "`develop`" so the name of the server and the TCP port on the server must be specified. The following code should be in **client.c**:

```
    .
    .
    .
conno=Nopen(SERVER_NAME, "TCP/IP", Nportno(), SERVER_PORT, 0);
    .
    .
    .
```

When machine "`target`" calls this ***Nopen()***, it will begin to actively establish the connection to the server. In this call to ***Nopen()***, the client does not need a well-defined port number, so a call to ***Nportno()*** is used. Function ***Nportno()*** returns a random port number greater than 1024.

The second difference is in the calls to ***Nwrite()*** and ***Nread()***. Since the client will be doing the complimentary operations of the server, its data collection loop will be:

```
.
.
.
/*
**  Loop through data transfer
*/
   for(i = 0; i<ITERATIONS;i++){
          /*
          **  Generate random number between 1 and DATA_SIZE
          **  then convert to a buffer "char data_req[2]."
          **  Send request to server
          */
          error_code = Nwrite(conno, data_req, sizeof(data_req));
          if( error_code < 0 ) return (error_code);

          /*
          **  Read the data from the server
          */
          error_code = Nread(conno, junk, sizeof(junk));
          if( error_code <= 0 ) return (error_code);
                 .
                 .
                 .
          /*
          **  This is where the data's integrity would
          **  be checked.
          */
                 .
                 .
                 .
          /*
          **  Write out status
          */
          error_code = Nwrite(conno, "All Done", 8);
          if( error_code < 0 ) return (error_code);
   }
```

One can see that these operations are the compliments of the server side. Finally, the termination is the same as in **server.c**.

A source code listing of **client.c**, with comments, is included following the listing of **server.c**.

Compiling Your Application

The makefiles delivered with USNet are designed to handle building an application without major modifications. Make sure files **client.c** and **server.c** are in the application source directory **appsrc**. To compile these programs:

1. Change the directory to the **appsrc** directory:

   ```
   D:\> cd usnet\appsrc
   ```

2. Make program **server.exe**:

   ```
   D:\usnet\appsrc> make server.exe
   ```

   This will initiate the default build of an executable. The makefile determines how to do the build based on the file extension **.exe**.

3. Make program **client.exe**:

   ```
   D:\usnet\appsrc> make client.exe
   ```

Check for compiler errors and warnings. Address any that crop up before running either program. Once both programs are built, they are ready to run by doing the following:

1. Ensure "target" and "develop" are connected via Ethernet.
2. Copy **client.exe** to "target".
3. Execute **server.exe** on "develop".
4. Execute **client.exe** on "target".

The program server will print out a few messages, and then wait until the connection is made. Once the client begins, trace messages should appear on both machines.

Code Listings

This section includes listings of **firstapp.h**, **server.c**, and **client.c**.

Listing of firstapp.h

```
/*
**  Copyright 1997  U S Software Corp.
**
**  firstapp.h – Header file used by server.c and client.c
**    from the USNet quick start guide.
*/

/*
**  Check to see if this has been included previously
*/
#ifndef _FIRSTAPP_H
#define _FIRSTAPP_H


/*
**  Useful constants. These should be included in server.c and client.c.
*/
#define SERVER_NAME "buford"      /* Name of server as defined in netconf.c */
#define SERVER_PORT   1500        /* TCP port that server communicates through */
#define DATA_SIZE     200         /* Size of data buffer to transmit in bytes */
#define ITERATIONS    10          /* Number of times to send the data buffer */

#endif                            /* _FIRSTAPP_H */
```

Listing of server.c

```
/*
**  Copyright 1997  U S Software Corp.
**
**  server.c – Simple server test application.  To be used
**    in conjunction with client.c.
**
*/

/*
**  Include at least the following files for an application
**  using the Dynamic Protocol Interface.
*/
#include "net.h"
#include "local.h"
#include "support.h"
```

```
/*
**  Useful constants.  This is where any application-specific
**  information would be included.
*/
#include “firstapp.h”

/*
**  Server starts here.
*/
void main(void){

int error_code,                   /* Error codes returned by interface */
    conno;                        /* Connection to remote client */
unsigned short count,             /* Count index in junk[] */
               pass,              /* Number of times data sent to client */
               data_request;      /* Number of bytes client requested */
unsigned char junk[DATA_SIZE],    /* Sample junk data */
              data_size[2];       /* Buffer of number of bytes client wants */
char status[10];                  /* Client status */


  /*
  **  Attempt to initialize the stack.  This will zero
  **  all data structures, start the clock, and run the
  **  init() routines of each layer.
  */
  Nprintf(“Server attempting a Ninit\(\) \n”);
  error_code = Ninit();
  if ( error_code < 0 ){
    Nprintf(“ Failed to initialize due to code %d
                          \n”,error_code);
    return;
  }

  /*
  **  Attempt to initialize the physical connections on this
  **  host.
  */
  Nprintf(“Server attempting a Portinit\(\) \n  “);
  error_code = Portinit(“*”);
  if ( error_code < 0 ){
    Nprintf(“ Failed to initialize ports due to code %d
                          \n”,error_code);
    Nterm();
   /* Terminate USNet */
    return;
  }

  /*
  **  Build the data buffer.  The buffer is just numbers
  **  from 0 to 255.
  */
  for(count=0;count<DATA_SIZE;count++)
    junk[count]=count%256;

  /*
  **  Open a server connection.  The server will enter the
  **  LISTEN state and wait for the client to establish the
  **  connection.  Nopen() returns the connection number.
  **  If conno<0 an error occurred.
  */


  Nprintf(“Server doing an Nopen\(\) on %d \n”,SERVER_PORT);
  conno = Nopen(“*”, “TCP/IP”, SERVER_PORT, 0, 0);
  if ( conno < 0 ){
   Nprintf(“ Failed to open connection due to code %d \n”,conno);
   Nterm();                /* Terminate USNet */
   return;
  }

  /*
  **  Connection has been established.  Begin writing buffer
```

```
  **  the number of times specified by ITERATIONS.
  */
  Nprintf("Server writing data to client %d times\n", ITERATIONS);
  for(pass=0;pass<ITERATIONS;pass++){
    /*
    **  Read the client's request for the number of bytes to send.
    */
    data_request = 0;
    error_code = Nread(conno, data_size, sizeof(data_size));
    if( error_code <= 0 ) {
         Nprintf(" Failed on data request due to code %d\n", error_code);
         Nclose(conno);
         Nterm();
         return;
    }
    data_request = (0xff00 & (data_size[0]<<8)) |     /* convert to number */
                    (0x00ff & data_size[1]);
    Nprintf("  Received request for %u \n", data_request);
    /*
    **  Write out the junk data to connection conno.
    */
    error_code = Nwrite(conno, junk, data_request);
    if( error_code < 0 ) {
         Nprintf(" Failed on writing data due to code %d\n", error_code);
         Nclose(conno);
         Nterm();
         return;
    }
    /*
    **  Read status from client to see if it has finished
    **  reading.  In this test we don't care what the client
    **  wrote as long as the reading of the data was OK.
    **  The client will check the integrity of the data.
    **  If the data was received OK, then the client will send
    **  a small packet.  Therefore we do not check status.
    */
    error_code = Nread(conno, status, sizeof(status));
    if( error_code < 0 ) {
         Nprintf(" Failed on reading data due to code %d\n",error_code);
         Nclose(conno);
         Nterm();
         return;
    }
    /*
    **  Got this far?  If so, we had a successful pass.
    */
    Nprintf(" Pass %d complete \n",pass);
  }
  Nprintf("Server program completed successfully \n");
  Nclose(conno);                        /* Close down the connection */
  Nterm();                              /* Terminate USNet */
  return;
}
```

Listing of client.c

```
/*
**  Copyright 1997  U S Software Corp.
**
**  client.c — Simple client test application.  To be used in
**  conjunction with server.c.
**
*/

/*
**  Include at least the following files for an application
```

```
**  using the Dynamic Protocol Interface.
*/
#include "net.h"
#include "local.h"
#include "support.h"

/*
**  Useful constants.  This is where any application-specific
**  information would be included.
*/
#include "firstapp.h"

/*
**  Client starts here.
*/
void main(void){

int error_code,                   /* Error codes from function calls */
    conno;                        /* Physical connection number */
unsigned short count,             /* Count index in junk[] buffer */
               pass,              /* Number of times server sent data */
               client_port,       /* Client-side port number */
               data_request;      /* Number of bytes requested by client */
short data_read;                  /* Number of bytes read by client */


unsigned char junk[DATA_SIZE],    /* junk buffer */
              data_size[2];       /* Request sent to server */
char status[10];                  /* Status buffer */

  /*
  **  Attempt to initialize the stack.  This will zero all
  **  data structures, start the clock, and run the init()
  **  routines of each layer.
  */
  Nprintf("Client attempting a Ninit\(\) \n");
  error_code = Ninit();
  if ( error_code < 0 ){
    Nprintf(" Failed to initialize due to code %d\n", error_code);
    return;
  }

  /*
  **  Attempt to initialize the physical connections on
  **  this host.
  */

  Nprintf("Client attempting a Portinit\(\) \n  ");
  error_code = Portinit("*");
  if ( error_code < 0 ){
    Nprintf(" Failed to initialize ports due to code %d\n",error_code);
    Nterm();                      /* Terminate USNet */
    return;
  }

  /*
  **  Open a client connection.  The client will establish
  **  the connection because the server is in the LISTEN
  **  state.  Nopen() returns the connection number.
  **  If conno<0 an error occurred.
  */

  client_port = Nportno();
  Nprintf("Client doing an Nopen\(\) on %d ",client_port);
  Nprintf("to server port %d \n",SERVER_PORT);
  conno = Nopen(SERVER_NAME, "TCP/IP", client_port, SERVER_PORT, 0);
  if ( conno < 0 ){
   Nprintf(" Failed to open connection due to code %d \n", conno);
   Nterm();                       /* Terminate USNet */
   return;
  }
```

```
/*
**  Connection has been established.  Begin writing buffer
**  the number of times specified by ITERATIONS.
*/
Nprintf("Client reading data from server %d times\n",ITERATIONS);
for(pass=0;pass<ITERATIONS;pass++){

  /*
  **  Zero out the buffer to ensure we do not check the
  **  previously sent data.
  */
  memset(junk, 0, DATA_SIZE);

  /*
  **  Generate a request for data.  Number of bytes range from
  **  1 to DATA_SIZE.  Then send data request to the server.
  */
  data_request = TimeMS()%DATA_SIZE + 1;
                                            /* TimeMS returns ms count */
  data_size[0] = data_request>>8;
                                            /* Store number in buffer */
  data_size[1] = 0x00ff & data_request;
                                            /*   Finish storing number */
  printf("  Sending request for %u bytes \n",data_request);
  error_code = Nwrite(conno, data_size, sizeof(data_size));
  if( error_code < 0 ) {


       Nprintf(" Failed on send data request due to code %d\n",error_code);
       Nclose(conno);
       Nterm();
       return;
  }

  /*
  **  Read the requested number of bytes of junk data
  **  from connection conno.  DATA_SIZE the maximum
  **  buffer size.  Nread() will return the number of
  **  actual bytes read in error_code.
  */
  data_read = Nread(conno, junk, DATA_SIZE);
  if( data_read < 0 ) {
       Nprintf(" Failed on reading data due to code %d\n",error_code);
       Nclose(conno);
       Nterm();
       return;
  }

  /*
  **  Check the integrity of the data.  The buffer
  **  received is supposed to contain numbers from 0
  **  to 255 in order.  This section reads through junk[]
  **  and checks the values against expected values.
  */

  for(count=0; count<data_read; count++){
       if( junk[count] != count%256 ){
            Nprintf("Bad Data Received:\n");
            Nprintf("   Byte number %d ",count);
            Nprintf("is %d ",junk[count]);
            Nprintf("but should be %d \n", count%0x256);
            Nclose(conno);
            Nterm();
            return;
       }
  }
```

```
    /*
    **  Send the status to the server to indicate that the
    **  client successfully read the data.
    */
    Nprintf("  Data was intact.  Read %u bytes \n",data_read);
    error_code = Nwrite(conno, "All Done", 8);
    if( error_code < 0 ) {
         Nprintf(" Failed on writing data due to code %d\n",error_code);
         Nclose(conno);
         Nterm();
         return;
    }

    /*
    **  Got this far?  If so, we had a successful pass.
    */
    Nprintf(" Pass %d complete \n",pass);
   }

   Nprintf("Client program completed successfully \n");
   Nclose(conno);                    /* Close the connection */
   Nterm();                          /* Terminate USNet */
   return;

}
```

Developing Your Application

Congratulations on your success with your integration efforts! Now that you are ready to start developing your application, there are a few points to keep in mind:

- Set `TRACE_DEBUG = 3` in config.mak to help report error conditions in the stack. Do a **grep** or **search** on ***Nprintf*** in the stack modules to locate error traps.
- The header file **net.h** contains error number translation.
- Use ***Nprintf*** in your application as a trace tool.
- Use a LAN analyzer to capture data traffic during stack communications.
- Use an incremental development approach when adding new functionality to your application. Unit test each feature before integrating new features.

When you have finished developing your application, set `TRACE_DEBUG = 0` in config.mak. This will remove the once-useful debug code from your final application build.

4. Configuration

Overview

This section provides an in-depth look at the configuration of USNet.

The following text assumes that your processor, compiler, and RTOS are supported. If you are not sure, refer to the **readme.txt** file in the USNet root directory. If they are not supported, please refer to Chapter 9, *Porting* to complete all interface tasks before returning to this section. If you are developing a new network controller driver see Chapter 10, *Device Drivers*.

One of the primary configuration tasks for USNet is the construction of the network configuration table, netdata. This is the structure that defines the network to USNet. Other important configuration issues are covered in this chapter. Some of the issues here are also addressed in **release.txt** and **readme.txt**.

The following table summarizes the modules that contain configuration parameters. The text below the table briefly describes the purpose of each module.

Table 4-1: Configuration Files

Configuration	File(s)	Location
Makefile	config.mak compiler.mak	<root>\config.mak <root>**config****<cpu>****<compilers>**\compiler.mak
Network	netconf.c	<root>**netsrc**\netconf.c
Local Parameters	local.h	<root>**drvsrc****<cpu>**\local.h
Protocol Selection	local.h	<root>**drvsrc****<cpu>**\local.h

Notes for Table 4-1:

<root> = Install directory

<cpu> = Directory named after specific CPU, such as i8086, i386, or m68k

<compiler> = Directory named after specific compiler, such as msoft or borland

Makefile configuration: **config.mak** and **compiler.mak** contain build instructions for USNet.

Network configuration: **netconf.c** contains a table of all network connections.

Local parameter configuration: **local.h** contains site-dependent and CPU-specific definitions, such as read/write buffer sizes, packet size, and other parameters.

Protocol selection: You can remove the protocols that you will not use in the header file **local.h**.

Auto-generated Configuration Files

These files are automatically generated when you do a make.

<root>\include\config.h
<root>\include\config.inc

Configuring the Makefiles

USNet provides you with these working makefiles (see Table 4-1 for their locations):

config.mak defines the target structure, CPU, compiler, installed add-on packages, and libraries.

compiler.mak defines paths to toolchains and other toolchain-related issues.

Once the software is installed, you must edit these files to compile the USNet library and its test programs on the development system.

Editing the config.mak File

The first file you must change to define the development environment is **config.mak**, located in the USNet install directory.

1. Define where USNet is installed by setting ***USROOTDIR*** to the complete path name to the install directory. For example:

   ```
   USROOTDIR = D:\usnet
   ```

2. Define the target processor by uncommenting the appropriate CPU definition line. If the target CPU is an i8086, remove the initial '#' in the line:

   ```
   #CPU = i8086 # Intel real mode, COMPILERS: borland, msoft
   ```

   It should then read:

   ```
   CPU = i8086  # Intel real mode, COMPILERS: borland, msoft
   ```

   Comment out all other CPU lines.

3. Define the target compiler that will be used for development by uncommenting the appropriate ***COMPILER*** definition line. The CPU line lists which compilers are supported for each CPU. In the current case, Borland was selected as the compiler, so uncomment this line:

   ```
   COMPILER = borland
   ```

   All other ***COMPILER*** lines must be commented out. If an unsupported compiler is selected for the target CPU, the makefile generates an error at compile time.

4. Define which USNet add-on packages are installed by uncommenting the appropriate PRODLIST lines. For example, if the Internet Access Package is installed, uncomment the following line:

```
#PRODLIST += iap
```

By default, USNet includes the common code and drivers:

```
PRODLIST += net
PRODLIST += drv
PRODLIST += sup
```

5. Define which RTOS will be used in conjunction with USNet by defining the ***RTOS*** macro. All RTOSes do not support all target CPUs. For instance, HI-SH77 will only run on an SH3. Keep this in mind when selecting an RTOS.

   If the target RTOS is U S Software's MultiTask!, change the line reading:

   ```
   RTOS = none
   ```

   To:

   ```
   RTOS = MT
   ```

   where MT specifies which **include\rtos** subdirectory to pull support files from.

   USNet supports the following RTOSes:

HI-SH7	Hitachi uITron*; SH1 and SH2 only
HI-SH77	Hitachi uITron; SH3 only
IRMX	Intel iRMX* EMB
MT	U S Software's MultiTask!
MTOS	Industrial Programming
NONE	No RTOS
PPSM	RTOS for DragonBall*
REALOS	Fujitsu mITron*
RX850	NEC uITRON* RTOS
TT	U S Software's TronTask! version 2.0x
TT3	U S Software's TronTask! version 3.0x
VRTX	Microtek VRTX*

   *trademarked by their respective companies

6. Use the USER_INCS macro to define additional include paths.

7. Use the USER_LIBS macro to define other libraries. Specify the full path to each library.

8. Define the trace debug level using the ***TRACE_DEBUG*** macro:

TRACE_DEBUG = 0	removes all ***printf()*** debugging
TRACE_DEBUG = 9	includes verbose debugging
TRACE_DEBUG = 3	is the default

   Set TRACE_DEBUG to the desired debugging trace level. (This is used to define NTRACE, which is used in the code.) The default value is 3. Valid values range from 0 to 9 where 9 has the highest level of detail. A higher value is usually used during development to enhance error reporting, and changed to zero when you are ready for production.

The USROOTDIR, CPU, and COMPILER macros are used to find the directories containing USNet support files.

Editing the compiler.mak File

In the directory **$(*USROOTDIR*)\CONFIG\$(*CPU*)\$(*COMPILER*)**, edit the **compiler.mak** file to specify the toolchain path, where:

$(***USROOTDIR***) = path where USNet was installed

$(***CPU***) = target CPU

$(***COMPILER***) = target compiler

For example, for i8086, Borland C, with USNet installed to d:\usnet, the path would be:

d:\usnet\config\i8086\borland\compiler.mak

1. Change the definition of where the compiler is installed by changing the `PTH` symbol. If the development tools are located in **D:\BORLAND**, then `PTH` should be set to:

   ```
   PTH = d:\borland
   ```

   The compiler, assembler, librarian, and linker paths are defined from this path. Check the following macros to ensure that they point to the proper paths:

   CC Defined to be the command-line compiler

   AS Defined to be the assembler command

   LNK Defined to be the linker command

   LIBR Defined to be the librarian

2. There may be other `PTH`-like symbols, depending on the compiler. Some represent DOS extenders, assemblers, or other utilities. These will also have to be changed.

3. Find the area labeled `user options`. This area defines compile-time options configurable by the user. For an i8086 target under the Borland compiler there are three options: Memory model (***MMODL***), target system (***TRG_ID***), and debug method (***DBG_ID***). Make sure that these options are configured to match your application design.

   Other types of user options might be available depending on the target processor and compiler. Such options include board-level support, starting addresses, base addresses, clock rates, endian, and more. You must review the **compiler.mak** file completely.

4. Most of these tools also require a set of default switches and/or command-line options to compile properly. You might need to redefine these macros to fit the needs of your application:

   CFLAGS Compiler command-line options

   AFLAGS Assembler command-line options

   LFLAGS Linker command-line options

Configuring the Network (netconf.c)

The initial network configuration table `netdata` is in module **netconf.c**. This table provides information that defines network connections.

Each table entry describes a network connection somewhere on the network. In a simple network you would probably have one entry per host. However, you are not limited to one network per host. You are allowed two or more entries for the same host, if for example, one host was connected to two different networks. An example could be a host with an Ethernet connection and a serial connection.

A network structure contains the following fields:

- Host Name
- Network Name
- Network Mask
- IP Address
- Hardware Address
- Flags
- Link Layer
- Driver
- Adapter
- Parameters

Host Name

This is the string host name of the target. The names must be unique for each target. This name is only used internally to USNet as a logical pointer to the target's network parameters.

Network Name

This is a mnemonic string that specifies the network associated with this interface . If a host has more than one network interface, the `netdata` table will contain multiple entries for the host, and each entry will specify a different network name. For point to point connections (networks that have only two hosts), use descriptive names such as "serial1". For networks that have more than two hosts, use a unique name for each network, and use the same name for each connecting port.

If no Internet address is given when opening a connection, the network name is used to identify the network.

Note: This field is actually named pname and the comment indicates "port name". It is clearer to think of this as a network name. In a future release we plan to rename this field.

Network Mask

This defines which part of the Internet address specifies the host (zero bits) and which specifies the network (one bits). This is sometimes called a subnet mask when a network is divided into subnets. For instance, the mask for a class C network is `{0xff,0xff,0xff,0}` which written in dotted notation would be `255.255.255.0`. Generally, IP networks use class A, class B, or class C network masks. The supplied `#define` constants A, B, and C serve as abbreviations for these standard masks.

When two hosts have the same network address (the part of the Internet address for which the address mask is one), USNet assumes that they can talk directly to each other.

IP Address

This is the network (Internet) address of the connection. If this field is initially set to zero, USNet can be configured to obtain the address using RARP, BOOTP, DHCP, or PPP.

Hardware Address

This is the physical address for the network interface. This field can usually be filled with the symbolic constant `EA0`, which is defined as a structure containing zeros. For local network interfaces, USNet will obtain the physical address from the network controller hardware when the hardware is initialized. For remote systems, the physical address can be obtained using ARP. There are three cases when the physical address should be filled in:

- When the system running USNet is to be used as a RARP server, the IP addresses and physical addresses for the remote systems being serviced must be filled in. This provides a mapping between IP addresses and physical addresses.
- When the system running USNet is providing Proxy ARP service, an entry containing the IP addresses and physical addresses for systems being proxied must be filled in.
- For a local network controller for which the hardware does not store the physical address.

Flags

Table 4-2: Configuration Flags (net.h)

Flag	**Meaning**
NOTUSED	The connection becomes invisible. This is useful for configuring out a record in the NETDATA structure without actually deleting it.
TIMESERVER	The host will respond to ICMP time requests.
DNSVER	This entry can act as a DNS server.
ROUTER	This entry can act as a router.
DIAL	This connection will be used to dial another host.
NATLOCAL	This interface connects to a network on the private side of a NAT router.
PROXYARP	The host will provide proxy services for ARP requests for this IP address.

Link Layer

This is the link level protocol, which must be defined in **net.h**. This is the name of the network, such as Ethernet, or the name of a specific low-level protocol, such as SLIP or PPP.

Adapter

The adapter will be initialized before the driver, and shut off after the driver. The following files are included:

PCMCIA1 - PCMCIA using card services

PCMCIA2 - PCMCIA using socket services

You can create your own adapter module if you need special initialization for a driver. Use the value 0 if there is no adapter. The adapter modules are defined in **net.h**.

Parameters

This is a text string giving the hardware parameters for the network controller. The parameter information is only used in the driver and the adapter initialization. A discussion of the parameters used by each device driver, along with additional notes on the device drivers, can be found in Chapter 10, *Device Drivers*. The supplied code uses the following conventions:

`IRNO` = n interrupt number, in some cases vector address (n = a number, such as 10 or 3)

`PORT` = p I/O address base (p = a hex number, such as 0x2F8 or 0x3E8)

`BUFFER` = a memory-mapped I/O base (a = a hex address; 0x0A00FFFF)

`BAUD` = b serial baud rate (b = baud rate, 9600 or 19200)

`EABASE` = b address where the Ethernet address is found

`CLOCK` = c clock rate (c = clock rate, 115200)

To give a number in hexadecimal, use the prefix `0x` as in `0x2ef`.

Examples

Here is an example of an Ethernet configuration entry:

```
"hostX", "enet", C, {192,168,201,4}, EA0, 0, Ethernet,
WD8003, 0, "IRNO=3 PORT=0x280 BUFFER=0xd0000",
```

`HostX` is connected to a network called "`enet`" which uses the class C address mask. The four numbers within wave brackets are the Internet address (for class C, the first 3 numbers identify the network, the fourth the host itself). The hardware address is `EA0`. No flag bits are used. The link layer is Ethernet, the driver is `WD8003` (file **wd8003.c)**. There is no adapter. The parameter string gives the interrupt number, the port address, and the shared buffer address.

A serial port might be as follows:

```
"hostX", "serial", C, {192,168,202,4}, EA0, 0, SLIP, I8250, 0, "IRNO=3
PORT=0x2f8 CLOCK=115200 BAUD=38400",
```

This defines `hostX,` connected to a network named "`serial`". Network class, Internet address, hardware address, and the flags are defined and used exactly as for Ethernet. The link protocol is `SLIP`. The driver is `I8250`. No adapter is used. The parameter string gives the interrupt number, the port address and the baud rate. Be sure these values are correct for your hardware configuration or the connection will not work.

Either example above is all that is needed for a system that has no need to communicate with hosts beyond the local subnet. When the system running USNet connects to other hosts on the same subnet, the routing logic will use the subnet mask to recognize those hosts that can be reached directly.

For systems that need to communicate with hosts beyond the local network, at least one additional entry needs to be made in the table. This entry is for the default router, as shown in this example.

```
"hostX", "enet", C, {192,168,201,4}, EA0, 0, Ethernet,
WD8003, 0, "IRNO=3 PORT=0x280 BUFFER=0xd0000",
```

```
 "gw", "enet", C, {192,168,201,1}, EA0, ROUTER, Ethernet,
0, 0, "0 ",
```

Note that if the router is an independent system, the driver, adapter, and parameter fields can be specified as "0". The router must have an address on the same subnet (4th parameter), and should use the same network name (2nd parameter).

If DHCP is used to automatically configure the network at boot time, then the configuration in the table becomes even simpler, as shown in the following example.

```
"hostX", "enet", C, {0,0,0,0}, EA0, 0, Ethernet,
WD8003, 0, "IRNO=3 PORT=0x280 BUFFER=0xd0000",

 "gw", "enet", C, {0,0,0,0}, EA0, ROUTER, Ethernet,
0, 0, "0 ",
```

Here, the IP address for both the system itself and the default router must be given as {0,0,0,0}, since the DHCP server will provide this information.

If DNS is used in the system, the DHCP server can also be used to automatically retrieve the IP addresses of up to two DNS servers. When the DNS server information is retrieved via DHCP, then the preceding example is appropriate.

If the DNS server information is configured statically, then an entry to the table must be added for each DNS server. The DNS server can be located on another network, so long as a router is available. The following example is for a DNS server entry.

```
"dns1", "xnet", C, {12,3,6,43}, EA0, DNSVER, 0, 0, 0, "0 ",
```

It can be convenient to run USNet on a number of systems in a test network, and the same configuration table can be used for all of the systems. In this case, the logic that determines the host name (1st parameter) can be used to pick out which entry in the table refers to the system itself. Also, the host names of other entries in the table can be used when the application opens a connection. The following examples illustrate tables that use these features.

This example shows a complete table with multiple entries. You can see it is coded as a standard C struct:

```
const struct NETDATA netdata[]={
"test1", "nnet", C, {192,168,201,1}, EA0, 0, Ethernet,
    NE2000, 0, "IRNO=5 PORT=0x300",
"test1", "serial", C, {192,168,202,1}, EA0, 0, SLIP,
    I8250, 0, "IRNO=3 PORT=0x2f8 CLOCK=115200 BAUD=38400",
"test1", "tnet", C, {192,168,203,1}, EA0, 0, Ethernet,
    EXP16, 0, "IRNO=4 PORT=0x340",
"test2", "nnet", C, {192,168,201,2}, EA0, 0, Ethernet,
    WD8003, 0, "IRNO=5 PORT=0x300 BUFFER=0xca000",
"test3", "serial", C, {192,168,202,2}, EA0, 0, SLIP, I8250,
    0, "IRNO=3 PORT=0x2e8 CLOCK=115200 BAUD=38400",
"test4", "tnet", C, {192,168,203,2}, EA0, 0, Ethernet,
    NE2000, 0, "IRNO=10 PORT=0x320",
"sun", "tnet", C, {192,168,203,3}, EA0, 0, 0, 0, 0, 0,
};
```

Host "`test1`" has multiple entries defining different networks: two Ethernet, and one serial, with the Ethernet connections using different controllers (note that each Ethernet record has a different port address and interrupt number). `test1` is connected to three different networks and can communicate with `test2` via "`nnet`", `test3` via "`serial`", and `test4` or the `sun` workstation via "`tnet`". Note that the Internet address is what uniquely defines the network. The `nnet` network is defined by the Internet addresses `192,168,201,`*x* where *x* specifies the host.

This next example shows tables on different hosts required for two hosts to communicate via Ethernet.

Host1:

```
"host1", "nnet", C, {192,168,201,2}, EA0, 0, Ethernet,
    NE2000, 0, "IRNO=5  PORT=0x300",
"host2", "nnet", C, {192,168,201,3}, EA0, 0, Ethernet,
    NE2000, 0, "IRNO=5  PORT=0x300",
```

Host2:

```
"host1", "nnet", C, {192,168,201,2}, EA0, 0, Ethernet,
    NE2000, 0, "IRNO=5  PORT=0x300",
"host2", "nnet", C, {192,168,201,3}, EA0, 0, Ethernet,
    NE2000, 0, "IRNO=5  PORT=0x300",
```

Both tables are identical. The same would be true for a serial connection. **Distinguishing between the local and remote hosts is not a function of the network configuration table.**

Also, in the example above, the interrupt and port numbers are identical in each entry, but they do not have to be. That depends on your hardware configuration.

Configuring the Drivers

There are three (3) types of drivers that may be configured in USNet:

- Standard drivers
- NDIS drivers
- ODI drivers

Standard Drivers

All drivers referenced by the network configuration table in **netconf.c** need to be defined to USNet; however, only those required by the local system need to be compiled and linked into the executable. Therefore, some drivers will be defined to the system but will not be linked in.

Drivers are defined in **net.h**. All drivers referenced in the network configuration table need to be defined here. Drivers used in **netconf.c** that will not be linked in are defined as 0.

An example is shown here. Note that the `WRAP` driver is used to have a host communicate with itself. This is a feature used by the test program **LTEST**.

```
#define WRAP &WRAP_T      /* to talk to self, for testing */
extern PTABLE WRAP_T;

#define WD8003 &WD8003_T /* Western Digital E'net brd */
extern PTABLE WD8003_T;

#define NE2000 &NE2000_T /* Novell 2000 E'net interface */
extern PTABLE NE2000_T;

#define EN360 0           /* Motorola 68360 E'net */

#define I8250 &I8250_T    /* PC serial ports */
extern PTABLE I8250 _T;
```

NDIS Drivers

USNet can also use standard NDIS drivers. NDIS is a network driver standard developed by 3Com and Microsoft for DOS and OS/2. NDIS drivers (in binary format) are often available from the board manufacturer. To configure one of these, proceed as follows:

1. Configure the NDIS driver as instructed by the supplier. (You can get the NDIS documentation and utilities from 3Com.)
2. Specify NDIS as the driver in **netconf.c**.
3. Edit file **protocol.ini** (this is an NDIS system file, not a USNet file) to add the definition:

```
;USNET NDIS resident stub is configured here
;
      [USNETDRV]
      DRIVERNAME=USNET$
      BINDINGS=drivername
```

   where ***drivername*** is the NDIS driver you want to use.

4. Add this line to **config.sys**:

```
      device=[path]\usnet.dos
```

   where **[*path*] = <*root*>\drvsrc\usnet.dos**

5. When the target CPU is I8086, the file **usnet.dos** is automatically built and placed into **<*root*>\drvsrc**.

   If you have a different target compiler, you must build **usnet.dos** with the command:

```
      make usnet.dos
```

   You can do this at any time; configuration operations should not affect the **usnet.dos** driver.

ODI Drivers

USNet can be configured to coexist with a Novell Ethernet connection. In other words, one may run a USNet application and a Novell network simultaneously across the same Ethernet interface. To do this, use an ODI driver as follows:

1. When setting up the network configuration, use "ODI" in the device driver field, as in the following example:

```
"hostX", "enet", C, {192,168,201,4}, EA0, 0, Ethernet, ODI, 0, 0,
```

   No driver arguments are required.

2. In addition, you will need to add the line FRAME ETHERNET_II to the link driver section of your **net.cfg** file. Look for this in your netware client directory (**nwclient**). Add it to the last line of the link driver section as shown here:

```
      Link Driver NE2000
            PORT 300
            INT 10
            FRAME Ethernet_802.2
            MEM D0000
            FRAME ETHERNET_II
```

```
NetWare DOS Requester
     FIRST NETWORK DRIVE = F
```

Configuring Local Parameters (local.h)

USNet is configured mainly by editing file **local.h** in the **drvsrc\<***cpu***>** directory (where <*cpu*> is the CPU as defined in **config.mak**). Other files are also configurable, but do not have the scope of **local.h**. These are the macros in the order they appear in the file. Following this summary is more detailed information for each macro.

NNETS	sets the maximum number of network controllers in one host.
NCONNS	sets the maximum number of open logical connections in one host.
NCONFIGS	sets the maximum entries in the data structure `netconf`, which is a table similar to the `netdata` array. Structure `netdata` is stored in ROM. During initialization `netdata` is copied to `netconf`, which is stored in RAM.
NBUFFS	sets the number of message buffers.
MAXBUF	sets the size of the message buffers.
USSBUFALIGN	sets the alignment boundary for the message buffer array.
FRAGMENTATION	sets whether the code to fragment and reassemble IP packets is included.
IP OPTIONS	is the IP option support.
USS_IP_MC_LEVEL	sets the level of support for IP multicast.
KEEPALIVETIME	is the BSD socket keepalive time.
MIB2	enables the collection of statistics for use with an SNMP agent.
RELAYING	defines whether or not host is to relay.
chksum_INASM	tells USNet that the checksum routine will be performed in assembly so the routine in **support.c** will not be needed. Not all the CPUs supported by USNet have the checksum routine ***Nchksum()*** in assembly.
DHCP	configures support for DHCP client functions.
DNS	configures support for DNS client functions.
TCP_SACK	enables selective ACK for TCP.
LOCALHOSTNAME	obtains USNet's host name.
USERID	identifies a user on an FTP server.
PASSWD	authenticates a user on an FTP server.
LOCALSETUP	performs user defined initialization.
LOCALSHUTOFF	performs user defined clean up.
USS_PROXYARP	enables proxy ARP feature.

NNETS Macro

This is the number of physical network connections associated with a host. If a host has two serial connections and an Ethernet connection, set ***NNETS*** to at least three. These three connections would be configured in the `netdata[]` table in **netconf.c** in a manner similar to:

```
"develop", "ether", C, {192,168,201,1}, EA0, 0, Ethernet,
   NE2000, 0, "IRNO=10 PORT=0x300",

"develop", "serial1", C, {192,168,202,1}, EA0, 0, PPP, I8250,
   0, "IRNO=4 PORT=0x2f8 CLOCK=115200",

"develop", "serial2", C, {192,168,203,1}, EA0, 0, PPP, I8250,
   0, "IRNO=3 PORT=0x2e8 CLOCK=115200",
```

NCONNS Macro

This is the maximum number of open logical connections ("sockets") in one host. When ***Nopen()*** establishes a connection, it returns a value from `0` to (`NCONNS-1`). Enough memory is set aside to handle these connections based on the value set. When estimating your need, consider that a TCP close leaves the connection block reserved for about a minute.

NCONFIGS Macro

This is the total allowable number of hosts USNet can interact with. This value reserves space in the `netconf[]` array which serves as the network configuration table, routing table, and ARP cache. This is the RAM version of the `netdata[]` array. When ***Ninit()*** is called, `netdata[]` is copied from ROM to `netconf[]`. ***NCONFIGS*** must be at least the same size as the number of entries in `netdata[]` as defined in **netconf.c**.

NBUFFS Macro

This is the number of working message buffers available to USNet. When USNet passes packets up and down the stack, it uses these buffers. These buffers are also used for internal purposes. USNet contains a large number of dynamic queues, so there is no exact formula for ***`NBUFFS`***. Too few buffers will hurt performance. The rule of thumb is five buffers per possible active connection.

MAXBUF Macro

Size, in bytes, of each message buffer reserved by NBUFFS. Each link layer may have different requirements. Ethernet requires buffers about 1536 bytes long. Typically:

MAXBUF = 36 + largest packet size.

For the ODI driver:

MAXBUF = 92 + largest packet size.

Table 4-3: *MAXBUF* Sizes

Link Layer	Packet Size	MAXBUF	MAXBUF for ODI
Ethernet	1500	1536	1592
ARCNET	1500	1536	1592
SLIP	576	602	668
PPP	1500	1536	1592

Beware of DMA though! USNet reserves internally an extra 4 bytes after a message buffer, because DMA typically moves the CRC into memory. However, some hardware moves more than this, and you have to increase *MAXBUF* accordingly. See the driver list in **readme.txt** for additional *MAXBUF* requirements.

USSBUFALIGN Macro

This value specifies the alignment boundary for the start of the array of message buffers, and also the alignment for the data area within a message buffer. The setting will depend on the memory access characteristics for the host processor and the network controller. Changes to this setting should be carefully reviewed.

FRAGMENTATION Macro

This value specifies whether or not to support fragmentation at the IP layer. Do not fragment packets if you can avoid it. TCP and UDP can handle much larger data packets than Ethernet can handle, so the IP layer will chop up or assemble large packets depending on this switch:

0 = Do not do any type of fragmentation. Code is removed at compile time.
1 = Reassemble incoming large data packets.
3 = Reassemble incoming large data packets and fragment outgoing large packets.

IPOPTIONS Macro

This macro enables RFC IP option support, chiefly the source routing options. This is required in the standard, but little used and perhaps obsolete. Uses up 90 bytes extra per connection block.

USS_IP_MC_LEVEL Macro

This specifies the level of support to include for IP multicasting. The IP multicast feature allows for efficient communication with a group of hosts.

0 = no support
1 = support sending multicast IP datagrams
2 = support sending and receiving multicast IP datagrams

KEEPALIVETIME Macro

This is the time to keep a BSD socket connection open, in milliseconds. Default is 2 hours but inactive, as required by the standard. To use, uncomment the line and change the value as needed.

MIB2 Macro

This enables the collection of Management Information Base statistics for use with an SNMP agent, such as the USNet SNMP agent. An SNMP agent gives access to these statistics. Undefine this to remove SNMP code, if you will not be running an SNMP agent.

RELAYING Macro

This specifies whether USNet should relay packets. The TCP/IP standard requires relaying to be off by default.

`1` = Relay packets to another host
`2` = Do not relay

chksum_INASM Macro

This specifies whether the checksum routine is written in assembly or not. Define it if `checksum` is in assembly. Some platforms that USNet supports do not have an assembly routine, such as PowerPC, so this should be undefined.

DHCP Macro

This value specifies which DHCP client features to include.

undefined = do not include any DHCP code
1 = include code for DHCP client features
2 = include code and automatically call DHCP functions when initializing a network interface

DNS Macro

This value specifies which DNS client features to include.

undefined = do not include any DNS code
1 = include code for DNS client features
2 = include code and automatically call DNS functions if needed to resolve a host name

TCP_SACK Macro

Define this macro to enable the selective ACK feature for TCP. The selective ACK feature can improve throughput for TCP connections that suffer datagram loss for reasons other than congestion.

LOCALHOSTNAME Macro

USNet must know its own host name, in several places such as PPP when negotiating a CHAP session. The host name is specified with this macro. For DOS and UNIX environments, the macro gets the name using the ANSI C function ***getenv()***. In these cases, the name would be defined with a command-line command such as `set host=`*`myhostname`*.

For embedded targets, the supplied ***LOCALHOSTNAME()*** loads a fixed name. You will want to keep the host names unique within a network, as you would on any network to avoid ambiguities. There is no absolute rule against duplicate names; however, there may be consequences. For instance, host XXX cannot open by name another host called XXX, or if a network had a host YYY and two hosts XXX, YYY would communicate with the XXX listed first in the network configuration table and the second XXX could not be reached in this manner. All XXX hosts, however, could still talk to host YYY. Unless you have some special needs, it is best to keep your hostnames unique.

If you have a network with a large number of identical hosts, you may want to supply your own ***LOCALHOSTNAME()*** macro. This could get the name from an EPROM or a similar source. It could also read an identification off a network controller and match this to a table. This method of course requires that all hosts have an identical hardware configuration.

USERID Macro & PASSWD Macro

These specify the user name USNet should use when connecting to a remote site, or the name USNet expects when someone connects to USNet. These are used in PPP, FTP, and Dial-up connections. They are used for establishing a PPP connection using PAP and/or CHAP. The supplied FTP server does not require a user ID or a password.

LOCALSETUP Macro

This macro is called at the beginning of ***Ninit()*** to perform any local initialization. This is where you might want to place any application initialization. For example, this macro can be used to specify a function that performs proprietary hardware initialization that is needed before accessing a network interface.

LOCALSHUTOFF Macro

This macro is called at the end of ***Nterm()*** to shut down any local options. This is the complement to ***LOCALSETUP***. For example, this macro can be used to specify a function that performs proprietary hardware clean up once network functions are no longer needed.

USS_PROXYARP Macro

Define this macro in order to allow the system running USNet to respond to ARP requests on behalf of other hosts. This can be useful, for example, when the system running USNet should perform

bridge-like functions, relaying network frames to hosts on one network while making it appear that the hosts are part of another network.

Selecting Protocols

File **net.h** defines the protocols (including the link layer, but not the drivers) known to USNet. Any protocols that you do not need you can take out with the `#undef` statement in the local configuration file **local.h**. The following is an example of how this is done:

```
#undef UDP          /* — not needed — */
```

Systems that have only a serial interface and use a protocol such as PPP or SLIP can undefine ARP, RARP and Ethernet.

5. Dynamic Protocol Interface

Overview

This chapter details the usage of USNet's Dynamic Protocol Interface. The Dynamic Protocol Interface provides a simple and efficient interface to the USNet stack. It is an alternative to the BSD Sockets Interface (Chapter 6). oThe following issues are covered:

- Blocking versus non-blocking operation
- Include files
- Initialization and termination
- Connections
- Open, read, write, and close functions
- Macros for setting and obtaining control information on connections
- Multicast API
- Examples

Blocking Versus Non-Blocking Operation

There are two modes of operation that affect how your application deals with network events in a non-multitasking system: Blocking and non-blocking.

Blocking is the default mode. This mode will halt processing while waiting for a network event to complete or timeout. An example of this would be a wait for a return from a TCP open. Blocking mode would halt processing until the open returned a connection number or timed out. This behavior is usually unsatisfactory for most embedded systems.

Non-blocking allows processing to continue while polling the status of the network event. Non-blocking is desirable in a non-multitasking system because it makes efficient use of CPU time while waiting for network events to complete.

In a multitasking system, blocking is the recommended mode of operation because blocking does not actually block processing as it does in a non-multitasking system.

Non-blocking issues are addressed in the appropriate sections in this chapter. An example of non-blocking is also given at the end of this chapter.

Include Files

All programs that call USNet routines need to contain the following include statements in the order shown:

```
#include "net.h"
#include "local.h"
#include "support.h"
```

An application that uses the ICMP protocol directly also needs:

```
#include "icmp.h"
```

Initialization and Termination

Ninit() performs general initialization, such as initialization of tables and buffers. It must be the first network function called and can't be called again unless the function ***Nterm()*** has been called first.

Portinit() and ***Portterm()*** are used to initialize and shut down the system's network interfaces.

Detailed descriptions of these functions follow.

Ninit

Performs general network initialization.

```
int Ninit(void);
```

Ninit() takes no parameters.

See also: ***Nterm, Portinit, Portterm***

Return Value

0	Success.
ENOBUFS	No buffers configured. Check **local.h** variables *NCONFIGS* and *NNETS*.
USER	User-defined error return from ***LOCALSETUP()*** found in **local.h**.

Example

```
main()
{
   /* initialize all connections */
   if (Ninit() < 0)
       /* process error */
}
```

Nterm

Shuts down networking.

```
int Nterm(void);
```

Nterm() takes no parameters. Any open network interfaces will be shut down, so ***Portterm()*** does not need to be called before ***Nterm()***. Network support can be restarted by making a call to ***Ninit()***.

See also: ***Ninit, Portinit, Portterm***

Return Value

`0`	Always returns 0.

Example

```
/* shut down all network connections */
Nterm();
```

Portinit

Initializes one or more network interfaces.

```
int Portinit(char *name);
```

`name`	If "*****", then all network interfaces for this host will be initialized; otherwise, this refers to a specific network interface defined in **netconf.c**.

Portinit() initializes the specified network interfaces. Note that all interfaces can be initialized all at once, or individually. The initialization routine will prepare the device driver to transmit and receive network frames, and will install and enable the interrupt service routine for the network device driver.

See also: ***Ninit(), Nterm(), Portterm()***

Return Value

`NE_PARAM`	Parameter error. The device driver did not accept the initialization string specified in **netconf.c**.
`EHOSTUNREACH`	The specified network name (when "*****" is not used) is not in **netconf.c** for this host. This could also mean that the host name is wrong.
`NE_HWERR`	A hardware error occurred. Generally, this indicates an error with the network controller.

Examples

```
/* initialize all network interfaces */

main()

{
    if (Ninit() < 0)
        /* process error */
    if (Portinit("*") < 0)
        /* process error */
}
```

```
/* Initialize a specific network interface */
main()
{
    if (Ninit() < 0)
        /* process error */
    if (Portinit("serial") < 0)
        /* process error */
}
```

Portterm

Shuts down one or more network interfaces.

```
int Portterm(char *name);
```

name — If "*", then all network interfaces for this host will be shut down; otherwise, this refers to a specific network interface defined in **netconf.c**.

Shuts down the specified network interfaces. Note that all interfaces can be shut down at once, or individually. The shut down routine will put the network controller into an idle state, and restore the interrupt vector associated with the network device driver to its original state. The shutdown is reversible: Just make another call to ***Portinit()***. A call to ***Portterm()*** can be omitted prior to calling ***Nterm()***, because ***Nterm()*** automatically calls ***Portterm()***.

See also: ***Ninit(), Nterm(), Portinit()***

Return Value

`0` — Always returns 0.

Examples

```
/* shut down all network connections */
Portterm("*");

/* shut down a specific network connection */
Portterm("serial");
```

Connections

Connections behave very much like files: You can open and close a connection, you can read data from it, and write data to it. The main difference is that a connection has a user at each end, and a file has only one user. The data you read is the data the other user wrote, and vice versa.

USNet offers the user two basic kinds of connections: TCP and UDP. There are two primary differences:

- TCP performs error correction and flow control, and UDP does not. You can read TCP like a local disk file: You want to check for errors, but they should not occur and if they do you quit. Doing this with UDP would be difficult, and writing applications using UDP is quite cumbersome. It is best to leave UDP for pre-written applications, such as TFTP and BOOTP.
- UDP is a packet protocol, and TCP is a byte-stream protocol. With TCP, you can't predict with certainty how many bytes a read will return, or how many reads you'll need for a given amount of data.

Port numbers are used to match the two ends of the connection. If your local port number is my remote port and vice versa, then we have a connection.

Normally one end performs an active open and the other a passive open. The system performing a passive open is typically running a server application. This system will wait until it receives an indication from a client application performing an active open.

Open, Close, Read, and Write

These four routines (plus the startup and shutdown) are the only user-level network functions required to write an application using USNet. This might surprise you, especially if you have seen network packages that go something like:

```
call TCPwrite
call Ipwrite
call DRIVERwrite
...
```

USNet uses a table-driven protocol stack structure. Each protocol level has only one public symbol: The name of the protocol table. USNet performs all necessary calls through these protocol tables. The user only has to call a general high-level function that is the same for all protocol configurations.

The open function specifies which protocols, and in which order, are to be used. There are no restrictions on the protocol stack as such, but of course not all combinations make sense.

Nopen

Opens a connection.

```
int Nopen(char *to, char *protoc, int lp, int rp, int flags);
```

to — String specifying the name of the remote system. This can take one of the following forms:

"host"	Remote host, shortest route.
"host/network"	Remote host, using named network.
"*"	Any host, used for passive open or broadcast.
"*/network"	Any host, using named network.
"n1.n2.n3.n4"	IP address of remote system.

protoc — String specifying the transport and network layer protocols, separated by a slash. Typical values would be "TCP/IP", "UDP/IP" or "ICMP/IP".

lp — Local port number. For an active open, this is often an ephemeral port, and a suitable random value can be obtained using the utility function ***Nportno()***. For a passive open, the well-known port number should be used.

rp — Remote port number. For an active open, this should be the well-known port for the service used in the connection. For a passive open, this value should be specified as 0, and any remote port will be accepted for the connection.

flags — Normally 0, but for a non-blocking open, you can specify the flag S_NOWA, and the call will return without blocking. In order to determine if the connection is established, use the macro ***SOCKET_ISOPEN()***. Also, for UDP connections, you can use the value S_NOCON to cause the connection to behave in a connectionless manner. When you specify S_NOCON, the connection will accept all UDP messages directed to the local port, regardless of the originating IP address or UDP port. This information is stored so that a call to ***Nread()*** followed by a call to ***Nwrite()*** will respond to the source of the message that was just read.

Nopen() is used for both active and passive opens. The behavior is determined by the parameters supplied to the function. Several examples follow to further illustrate the use of the function.

A passive open will wait indefinitely. An active open for TCP will return when the connection has been made, but it times out in a couple of minutes if there is no answer.

See also: ***Nclose(), Nread(), Nwrite()***

Return Value

conno	A return value >= 0 is a connection number. This is the handle for further communication on the connection.
EHOSTUNREACH	Could not access the remote system.

ENOBUFS	NCONNS in **local.h** is not large enough.
ETIMEDOUT	Timeout.
ECONNABORTED	Remote host refused the connection.

Examples

```
/* An active open from host1 that causes TCP to send out open requests
to port 1000.  The local port number is dynamically and randomly
assigned with the function Nportno().  */

    /* host1 */
    int conno, myport;  /* connection and port number */
    myport = Nportno();
    conno = Nopen("host2", "TCP/IP", myport, 1000, 0);
    if (conno < 0)
        /* process error */

/* A passive open at host2 that waits for and accepts calls from anyone
who asks for port number 1000.  This type of open would be done by a
server */

    /* host2 */
    int conno;       /* connection number */
    conno = Nopen("*", "TCP/IP", 1000, 0, 0);
    if (conno < 0)
        /* process error */

/* A UDP open at host1 for hostA through port serial1 would look like
this: */

    /* host1 */
    conno = Nopen("hostA/serial1", "UDP/IP", 1000, 1010, 0);

/* The specification of "serial1" indicates a specific network interface
on host1, and is not referring to hostA's network interfaces.  This form
of open may be needed if there are two connections between host1 and
hostA.  In this manner, "serial1" serves to identify which local network
interface is being used.  Note "serial1" references field 2 in the
network configuration table in netconf.c.  */

/* To send and receive ICMP messages, you can use the form: */

    /* host1 */
    conno = Nopen("host2", "ICMP/IP", 1000, 1010, 0);

/* This is a special situation; see, for instance, PING.C for the use of
ICMP.  */

/* Perform a non-blocking OPEN and do some processing while polling for
the OPEN connection.  */

    conno = Nopen("*", "TCP/IP", 1000, 0, S_NOWA);
    if (conno < 0 )
        /* handle error condition */
    while ( !SOCKET_ISOPEN(conno))
        /* perform other processing */
```

Nclose

Closes a connection.

```
int Nclose(int conno);
```

conno The connection number previously returned from a call to ***Nopen()***.

Nclose closes a connection, possibly waiting for a complete close handshake. In no case should the application retry the close. In some cases (as with TCP), the connection block will actually be freed after a minute or so, but this is automatic, and the application should not touch the connection after the close.

See also: ***Nopen(), Nread(), Nwrite()***

Return Value

`0`	Normal close.
`EBADF`	The connection number is invalid. No closing was performed.
`ECONNABORTED`	Protocol problem. If you have been writing data to the other system, consider the data unsafe. Connection is closed.

Example

```
int error;       /* error code        */
int conno;       /* connection number */
error = Nclose(conno);   /* close the connection */
if (error < 0)  /* process error */
```

Nread

Reads a message from a connection.

```
int Nread(int conno, char *buff, int len);
```

conno Connection number.

buff Buffer to store message.

len Size of the buffer.

Reads a message from a connection into the specified buffer. For a blocking socket, the call will block until information is available to be read, or until a timeout occurs. The timeout can be adjusted using the ***SOCKET_RXTOUT()*** macro.

For TCP connections, ***Nread()*** may return up to the maximum amount of information that will fit in one internal message buffer. This will be less than MAXBUF bytes. For UDP connections, the data from the next UDP message will be returned.

See also: ***Nclose(), Nopen(), Nwrite()***

Return Value

`0`	The remote system has closed the connection.
`count`	Values > 0 indicate the number of bytes read.
`EBADF`	The connection number is not valid.
`EWOULDBLOCK`	Non-blocking connection can't proceed. Read would be retried.
`ETIMEDOUT`	Timeout. Read can be retried.
`ECONNABORTED`	Protocol problem. Normally the application should close the connection.
`EMSGSIZE`	The message is too long for the supplied buffer.

Example

```
/* user defined input buffer size */

#define MAX_BUFFER_SIZE 80
int error;                     /* error code */
int conno;                     /* connection Number */
char buff[MAX_BUFFER_SIZE];    /* data input buffer */
/* read data into "buff" from connection number "conno" */
error = Nread(conno, buff, sizeof(buff));
if (error < 0)
    /* process error */
```

The constant `MAX_BUFFER_SIZE` could be replaced with the USNet constant `MAXBUF` defined in file **local.h**. A call to ***Nread()*** cannot return more than `MAXBUF` bytes.

Nwrite

Writes a message to a connection.

```
int Nwrite(int conno, char *buff, int len);
```

conno	Connection number.
buff	Buffer containing message.
len	Number of bytes to write.

Nwrite() writes a message to a connection from the specified buffer. The largest buffer passed to ***Nwrite()*** should not exceed the value given by the ***SOCKET_MAXDAT()*** macro. For TCP connections, this will reflect the maximum segment size that is indicated by the remote TCP when the connection is established. For UDP connections, this value will reflect the MTU imposed by the link layer. These values will generally be at least 256 bytes, so it is reasonable to write out small buffers directly.

See also: ***Nclose(), Nopen(), Nread()***

Return Value

count	Values >= 0 indicate the number of bytes written.
EBADF	The connection number is not valid.
ETIMEDOUT	Timeout. With TCP in blocking mode, this probably means the other end did not send acknowledgments as expected. It could also mean an extremely heavy system load and that a timeout occurred before the acknowledgment could be received. The connection should be closed. In non-blocking mode, the write should be retried.
ECONNABORTED	Protocol problem. Normally the application should close the connection.
EMSGSIZE	The message is too large for the internal buffer.

Example

```
/* user defined output buffer size */

#define MAX_BUFFER_SIZE 80
int error;                    /* error code */
int conno;                    /* connection Number */
char buff[MAX_BUFFER_SIZE]; /* data output buffer */
/* write data stored in "buff" to connection number "conno" */
error = Nwrite(conno, buff, sizeof(buff));
if (error < 0)
    /* process error */

/* dynamically sized write buffer */

int error;                    /* error code */
int conno;                    /* connection Number */
int maxwrite;                 /* maximum write size */
char buff[MAXBUF];            /* data buffer */
/* write data stored in "buff" to connection number "conno" */
conno = Nopen("host", "TCP/IP", Nportno(), 1050, 0);
if (conno < 0)
```

```
    /* process error */
maxwrite = SOCKET_MAXDAT(conno);
error = Nwrite(conno, buff, maxwrite);
if (error < 0)
    /* process error */
```

Dynamic Protocol Interface Macros

The following macros are useful for obtaining additional information or setting control information for a connection, and are described in this section:

SOCKET_NOBLOCK	sets the connection for non-blocking operation.
SOCKET_BLOCK	sets the connection for blocking operation.
SOCKET_ISOPEN	checks to see if a connection has entered the ESTABLISHED state.
SOCKET_HASDATA	checks to see if a message is available on a connection.
SOCKET_CANSEND	checks to see if a connection can accept data to be written.
SOCKET_TESTFIN	checks to see if the remote end of the connection has closed.
SOCKET_MAXDAT	provides the maximum size of a buffer than can be written to a connection.
SOCKET_RXTOUT	sets the receive timeout for a connection.
SOCKET_IPADDR	provides the IP address of the remote end of a connection.
SOCKET_OWNIPADDR	provides the IP address of the local end of a connection.
SOCKET_PUSH	sets the `PSH` flag on the next outgoing TCP segment.
SOCKET_FIN	sets the `FIN` flag on the next outgoing TCP segment.

SOCKET_NOBLOCK

Sets the connection for non-blocking operation.

```
SOCKET_NOBLOCK(conno)
```

conno The connection for which non-blocking operation should be set.

When non-blocking operation is set, calls to network functions that normally would need to wait for network activity in order to be completed will return the negative value `EWOULDBLOCK` when such a condition is encountered.

SOCKET_BLOCK

Sets the connection for blocking operation.

```
SOCKET_BLOCK(conno)
```

conno The connection for which blocking operation should be set.

When blocking operation is set, calls to network functions run to completion, or return a timeout error if an associated time limit is exceeded. Blocking operation is the default behavior for network functions, and this call will only be needed to return a non-blocking connection to blocking operation.

SOCKET_ISOPEN

Checks to see if a connection has entered the ESTABLISHED state.

```
SOCKET_ISOPEN(conno)
```

conno The connection that should be checked for the ESTABLISHED state.

This macro will evaluate as `0` if the connection is not in the ESTABLISHED state, and `1` if the connection is in the ESTABLISHED state. This macro is useful for connections that call ***Nopen()*** with the `S_NOWA` flag, so that after requesting a connection, the connection can be checked to see if it has been established.

SOCKET_HASDATA

Checks to see if a message is available on a connection.

```
SOCKET_HASDATA(conno)
```

conno The connection that should be checked for an available message.

This macro will evaluate as `0` if no information is available, or non-zero if data is available.

SOCKET_CANSEND

Checks to see if a connection can accept data to be written.

`SOCKET_CANSEND(conno, len)`

conno The connection that should be checked for room for writing.

len The amount of data to be written.

This macro will evaluate as `0` if the amount of data is more than can be written out immediately, or non-zero if the data length specified can be written.

SOCKET_TESTFIN

Checks to see if the remote end of the connection has closed.

`SOCKET_TESTFIN(conno)`

conno The connection that should be checked for a close from the remote end.

This macro will evaluate as `0` if the remote end of the connection has not yet closed, or non-zero if the remote system has closed.

SOCKET_MAXDAT

Provides the maximum size of a buffer than can be written to a connection.

`SOCKET_MAXDAT(conno)`

conno The connection for which the maximum buffer size should be determined

This macro will evaluate to the maximum number of bytes that can be accepted by the connection in a call to ***Nwrite()***.

SOCKET_RXTOUT

Sets the receive timeout for a connection.

`SOCKET_RXTOUT(conno, tout)`

conno The connection for which the timeout is to be adjusted.

tout The new timeout, in milliseconds.

SOCKET_IPADDR

Provides the IP address of the remote end of a connection.

`SOCKET_IPADDR(conno)`

conno The connection for which the remote IP address is to be returned.

The data type of the result is `Iid`.

SOCKET_OWNIPADDR

Provides the IP address of the local end of a connection.

`SOCKET_OWNIPADDR(`*conno*`)`

conno The connection for which the local IP address is to be returned.

The data type of the result is `Iid`. This macro is useful for systems that have more than one network interface. The IP address returned will be that of the interface that is used for the connection.

SOCKET_PUSH

Sets the `PSH` flag on the next outgoing TCP segment.

`SOCKET_PUSH(`*conno*`)`

conno The connection for which the next outgoing segment should include the `PSH` flag.

The next TCP segment to be written following a call to this macro will have the `PSH` flag set in the TCP header. This is useful for indicating to the TCP on the remote system that all internally buffered segments up through this segment should be delivered to the application as soon as possible.

SOCKET_FIN

Sets the `FIN` flag on the next outgoing TCP segment.

`SOCKET_FIN(`*conno*`)`

conno The connection for which the next outgoing segment should include the `FIN` flag.

The next TCP segment to be written following a call to this macro will have the `FIN` flag set in the TCP header. This is useful for shutting down a connection at the same time that the last segment is sent. Following the write, call ***Nclose()*** to finish closing the connection. ***Nclose()*** will not send a `FIN` segment in this case.

Multicast API (DPI)

In order to receive information associated with a multicast host group, join the multicast group using the ussHostGroupJoin() function described here, specifying the IP address for the group, and the interface that will be used. Once the group has been joined, datagrams on the local network directed to the group will be accepted by the system.

If there is no longer a need to continue receiving datagrams directed to a certain group, the system can stop accepting datagrams directed to the group by using the ussHostGroupLeave() function.

ussHostGroupJoin

Joins a multicast host group.

```
int ussHostGroupJoin(Iid iid, int netno);
```

iid	IP address for multicast host group.
Netno	Index for network interface.

The ussHostGroupJoin() function allows a system to receive multicast messages as part of a multicast host group. The group is identified by the multicast IP address that is passed to the function.

The network interface is identified by an index. The first network interface for a system that occurs in the netdata[] table is identified as 0, the next is 1, and so on. For systems with just one network interface, this value should be 0.

See also: ussHostGroupLeave

Return Value

0	Success.
NE_PARAM	Invalid group address or interface identifier.
ENOBUFS	Insufficient resources to join another group.

Example

```
#define MCTESTIP "224.1.2.3"
rc = ussHostGroupJoin(inet_addr(MCTESTIP), 0);
```

ussHostGroupLeave

Leaves a multicast host group.

```
int ussHostGroupLeave(Iid iid, int netno);
```

iid	IP address for multicast host group.
Netno	Index for network interface.

The ussHostGroupLeave() function removes the system from a multicast host group that has previously been joined.

The network interface is identified by an index. The first network interface for a system that occurs in the netdata[] table is identified as 0, the next is 1, and so on. For systems with just one network interface, this value should be 0.

See also: ussHostGroupJoin

Return Value

`0`	Success.
`NE_PARAM`	Invalid group address or interface identifier.
`EBADF`	Multicast group not found.

Example

```
#define MCTESTIP "224.1.2.3"
rc = ussHostGroupLeave(inet_addr(MCTESTIP), 0);
```

Examples

The following text provides examples of:

- Broadcasting
- TCP File Transfer
- Non-Blocking Operations

Broadcasting Examples

For broadcasting messages to all hosts on the network, use host name "`*`" in the active open, and then, do an ***Nwrite()***. For instance:

```
host1:
conno = Nopen("*/enet", "UDP/IP", 1010, 1000, 0);
.....
stat = Nwrite(conno, buf, len);
```

In this case, "`enet`" is the network name, and "`*`" represents all hosts on that network. The receiving hosts' ***open()*** would generally be a passive open.

```
host2:
conno = Nopen("*", "UDP/IP", 1000, 0, 0);
....
stat = Nread(conno, buf, len);
```

The receiving hosts must be listening on the same port number that the broadcasting host is sending to (e.g., 1000 in this case).

Broadcasting should only be used for data links that support it in hardware, such as Ethernet. It should not be done at the TCP level.

If the broadcasting host connects to several networks, the open call must specify the network name. Broadcasting is done to one network only.

TCP File Transfer Example

This example might be used to write a file to a remote host. Flow control and error checking are handled by TCP.

```
/* Client */

int maxwrite;          /* maximum write size */
char buf[MAXDAT];      /* data buffer */
conno = Nopen("host1", "TCP/IP", Nportno(), 1000, 0);
if (conno < 0)
    /* process error */
maxwrite = SOCKET_MAXDAT(conno);
for (;;)
{
     len = fread(ifile, buf, maxwrite);
     if (len <= 0)
        break;
     stat = Nwrite(conno, buf, maxwrite);
     if (stat < 0)
        /* process error */
}
stat = Nclose(conno);
if (stat < 0)
   /* process error */

/* Server */

char buf[MAXDAT];
conno = Nopen("*", "TCP/IP", 1000, 0, 0);
if (conno < 0)  /* process error */
for (;;)
{
     len = Nread(conno, buf, sizeof(buf));
     if (len < 0)  /* process error */
     if (len == 0) break;
     stat = fwrite(ofile, buf, len);
     if (stat < 0)  /* process error */
}
stat = Nclose(conno);
if (stat < 0)  /* process error */
```

Non-Blocking Operations Example

The following example shows how to read using non-blocking operations. Non-blocking writes will complicate an application quite a bit. If no multitasker is used, there is really no alternative to non-blocking operations. With multitasking, a heavy use (perhaps even any use) of non-blocking mode is not recommended.

```
conno = Nopen("*", "TCP/IP", 1001, 0, S_NOWA);
if (conno < 0)  /* ERROR */
while (!SOCKET_ISOPEN(conno))
    /* perform other work */

SOCKET_NOBLOCK(conno);
for (;;)
{
   YIELD();
   len = Nread(conno, buf, sizeof(buf));
   if (len < 0)
      if (len != EWOULDBLOCK)
         break;  /* error */
      else
         /* perform other work */
   else if (len == 0)
      break;  /* other end closed */
   else
   {
      /* process message */
   }
}
stat = Nclose(conno);
if (stat < 0)  /* ERROR */
```

6. BSD Socket Interface

About BSD Sockets

The BSD 4.3 sockets are the closest thing there is to a standard user interface to TCP/IP. However, they can only be approximated on a non-UNIX system, because many UNIX functions interact with sockets. The UNIX dependencies come in these forms:

- The UNIX sockets are really an intertask communication system, not a networking interface. They can be used to map to the various UNIX file systems, and they can mix files and sockets and even other things in one operation.
- The use of functions ***fcntl()***, ***select()***, ***read()***, ***write()***, and ***close()*** for networking purposes will easily cause conflicts. USNet changes these names by appending "***socket***" to them.
- The UNIX sockets have an interface to the UNIX signals, which again have an interface to just about any UNIX function.
- Some BSD socket features are implicitly not reentrant. These include function ***gethostbyname()*** and all use of `errno`. This is of course more a multitasking question than a networking question.
- The BSD use of TCP urgent data is in conflict with the TCP standard. The USNet module **tcp.c** contains a source-level variable to select either the standard or the BSD method. Best policy in all cases is not to use the BSD out-of-bound data, or the TCP urgent data.

The USNet socket interface is intended to help users in these ways:

- Porting from UNIX
- Porting to UNIX
- Writing new code

Porting from UNIX

In some cases, it may be practical to port an existing application from UNIX to an embedded multitasker. To get started, replace the include statements with:

```
#include "socket.h"
```

Then compile. The rest may not be quite as easy:

- UNIX applications are often non-ANSI C.
- The application may use non-ANSI functions heavily. Some of these may be low-level hardware-specific functions that take a lot of work to replace.
- Any real-time application for UNIX is likely to run in polling mode. Polling under multitasking is a very bad idea.

The last of these is the most serious. Porting an application can sometimes save time, but it does not always produce a good design.

Porting to UNIX

It might be very useful to run an embedded program also in a UNIX workstation. This is typically true for test programs and utilities, but not so much for the entire application. Our test program **sotest.c** is written this way; compiling it with the command-line option `DUNIX` produces a UNIX version. Use the code at the start of **sotest.c** as a model.

The include statement is replaced by the appropriate UNIX include statements. Unfortunately there is some variation in the UNIX practice; the example in **sotest.c** runs in both AIX and Solaris.

1. Define the USNet initialization and termination calls as nulls.
2. Define ***Nprintf()*** as ***printf()***, if used.
3. Rename ***fcntlsocket()***, ***closesocket()***, ***readsocket()***, ***writesocket()***, and ***selectsocket()***.
4. Define `errno` as `extern int`.

Writing New Code

For somebody who already knows the BSD sockets, writing any new code using them makes sense. (The Dynamic Protocol Interface needs quite a bit less space, but the difference in speed is not significant.) To support these users, we have made the USNet sockets as similar to 4.3 BSD sockets as reasonably possible. These points may require special attention:

- Symbolic error codes are not perfectly standardized across different UNIX systems. USNet uses the Solaris names.
- The typical UNIX use of `errno` is not reentrant. If this becomes critical, use ***getsockopt()*** to get the last error code.
- The function ***gethostbyname()*** is not reentrant. Use ***gethostbyname_r()*** instead if this is critical.
- You can't mix files and sockets. For instance, you can't use a ***selectsocket()*** to wait for either a keyboard character or a network packet.
- Avoid non-blocking mode if multitasking is used.

Structures and Definitions

To get in the needed definitions, use:

```
#include "socket.h"
```

Many of the BSD socket routines use a pointer to structure `sockaddr`, which specifies network address information. The `sockaddr` structure is a generic structure that can be used with a number of different communications protocols. USNet only uses the Internet Protocol (IP), and therefore only requires the use of the Internet structure `sockaddr_in`. Values are assigned to `sockaddr_in` and passed into the socket routine via the `sockaddr` parameter. This requires a typecast to `sockaddr *`. The discussion of the ***connect()*** function provides an example. Here are the structure definitions:

```
struct sockaddr {           /* generic socket address */
   unsigned short sa_family;       /* address family */
   char sa_data[14];    /* up to 14 bytes of address */
};
```

In practice, this is used almost as a void pointer. The true Internet address structure is:

```
struct in_addr {             /* Internet address */
    unsigned long S_addr;
};
struct sockaddr_in {  /* Internet socket address */
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

BSD Socket Interface Functions

The USNet BSD Socket Interface provides these function calls:

accept()	accepts a connection on a socket.
bind()	binds a name to a socket.
closesocket()	closes a socket.
connect()	initiates a connection on a socket.
fcntlsocket()	controls socket flags.
gethostbyname()	returns the IP address that corresponds to a host name.
getpeername()	extracts the remote address information for a socket.
getsockname()	extracts the local address information for a socket.
getsockopt()	gets options on sockets.
ioctlsocket()	sets control parameters for a socket.
listen()	listens for connections.
readsocket()	receives a message from a socket ID.
recv()	receives a message.
recvfrom()	receives a message from a connection.
recvmsg()	establishes a connection and receives a message.

selectsocket()	waits for activity on a set of sockets.
send()	sends a message on an established connection.
sendmsg()	sends a message that can be split between buffers.
sendto()	establishes a connection and sends a message.
setsockopt()	sets options on sockets (described with ***getsockopt***).
shutdown()	shuts down part of a connection.
socket()	creates a socket.
writesocket()	sends a message to a socket.

The typical calling sequences for a connection-oriented client and server are shown below.

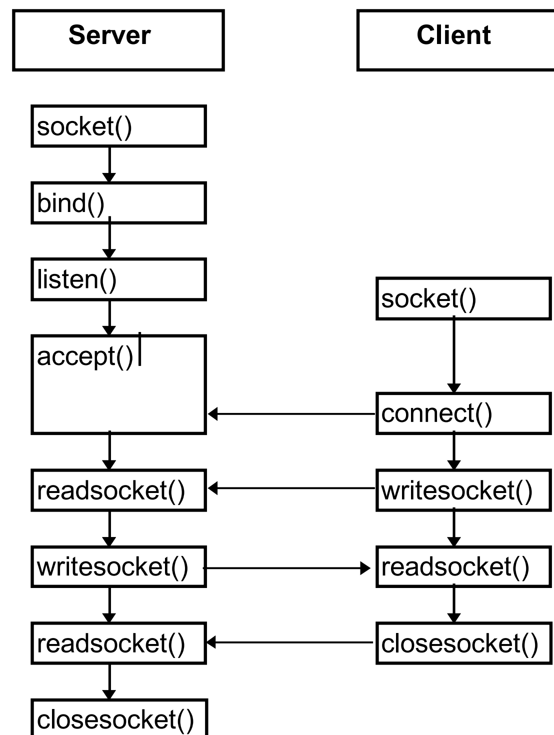


Figure 6-1: Functions Used in a Connection-Oriented System

For a connectionless protocol, the typical functions used by the server and client are shown in the next figure.

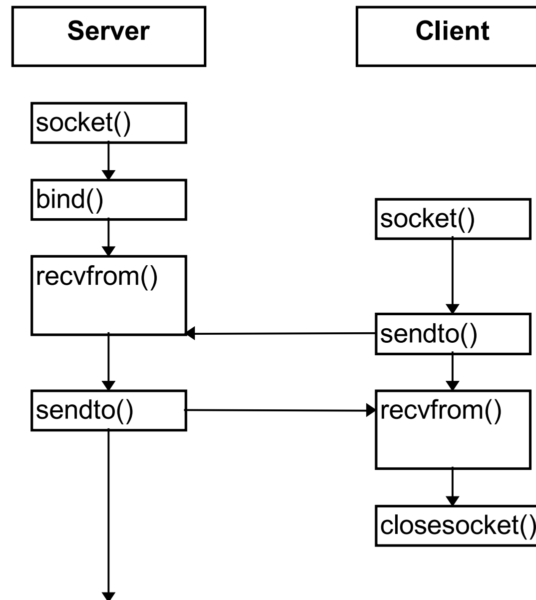


Figure 6-2: Functions Used in a Connectionless System

Most functions return a value of -1 in case of an error. The error code is stored in `errno`, and can also be retrieved using the ***getsockopt()*** function, as in the following example:

```
int errcode, errlen;
.
.
i1 = connect(s, (struct sockaddr *)&socka, sizeof(socka));
if (i1 < 0)
{
    i1 = errno;
    if (getsockopt(s, SOL_SOCKET, SO_ERROR,
                   &errcode, &errlen) >= 0)
        i1 = errcode;
    Nprintf("connect: error %d\n", i1);
    /* additional error handling */
}
```

Here the value of `errno` is saved before calling ***getsockopt(),*** in case this call fails and causes `errno` to be overwritten. The ***getsockopt()*** function should be used when possible in multitasking systems because `errno` is not reentrant.

If a call to ***socket()*** returns -1, there is no socket number to refer to when trying to retrieve the error code. In this case, the error code must be retrieved from `errno`.

The ***gethostbyname()*** functions return a pointer to a host data structure. If these functions fail, then a null pointer is returned.

accept

Accepts a connection on a socket.

```
int accept(int s, struct sockaddr *name, int *namelen);
```

s	Socket identifier.
name	On return, this provides information about the remote end of the connection.
namelen	On entry, this is a pointer to an integer containing the size of the name structure, and on return this pointer points to the size of the returned structure. This size will not change under USNet.

The ***accept()*** call is used by a server application to perform a passive open for a socket. The socket will remain in the LISTEN state until a client establishes a connection with the port offered by the server. The return value from this function is an identifier for a newly created socket over which communication with the remote client can occur. The original socket remains in the LISTEN state, and can be used in a subsequent call to ***accept()*** to provide additional connections.

See also: ***socket, bind, listen***

Return Value

-1	Error.
>= 0	Socket identifier for the established connection.

Example

```
int s1, s2;
int socksz;
struct sockaddr_in socka;
…

socksz = sizeof(socka);
memset(&socka, 0, sizeof(socka));
socka.sin_family = AF_INET;
s2 = accept(s1, (struct sockaddr *)&socka,
            &socksz);
if (s2 < 0)
    Nprintf("Error in accept\n");
```

bind

Binds a name to a socket.

```
int bind(int s, struct sockaddr *name, int namelen);
```

s — Socket identifier.

name — Structure that identifies the remote end of the connection. The `sin_family` member of the structure can be left as 0 to accept connections on any attached network interface.

namelen — Size of *name*.

A server application uses the ***bind()*** function to specify the local Internet address and port number for a connection. The port number is the port that the server will be listening on. A call to ***bind()*** can also optionally be called by a client application before calling ***connect()***.

See also: ***socket, listen, accept, closesocket***

Return Value

-1 — Error.

0 — Success. The Internet address and port number have been associated with the local end of the socket.

Example

```
int rc;     /* return code */
int s;      /* socket identifier */
struct sockaddr_in socka;  /* local port, etc */
…

memset(&socka, 0, sizeof(socka));
socka.sin_family = AF_INET;
socka.sin_port = htons(1100);
rc = bind(s, (struct sockaddr *)&socka,
          sizeof(socka));

if (rc < 0)
    Nprintf("Error in bind\n");
```

In this example, 1100 is the local port number to be used. A client performing a ***connect()*** to this server would also use port number 1100.

closesocket

Closes a socket.

```
int closesocket(int s);
```

s Socket identifier.

The ***closesocket()*** function is used to close a socket. This function is the same as the regular BSD Sockets ***close()*** function, but it has been renamed to avoid conflicts with the ***close()*** function that operates on file descriptors.

See also: ***socket***

Return Value

-1	Error.
0	Close was successful.

connect

Initiates a connection on a socket.

```
int connect(int s, struct sockaddr *name, int namelen);
```

s	Socket identifier.
name	Structure that identifies the remote end of the connection.
namelen	Size of name.

The ***connect()*** function performs an active open, allowing a client application to establish a connection with a remote server. The `name` structure is used to specify the Internet address and port number for the remote end of the connection. The Internet address is usually retrieved using the ***gethostbyname_r()*** function.

See also: ***closesocket***

Return Value

-1	Error.
0	Success. A connection has been established with the remote server.

Example

```
int rc;                        /* return code */
struct sockaddr_in socka;      /* internet address */
                               /* and port number */
struct hostent hostent;        /* for retrieving IP */
                               /* address from host */
unsigned char buff[BUFFLEN + 1];
…
memset(&socka, 0, sizeof(socka));
socka.sin_family = AF_INET;
gethostbyname_r("host1", &hostent, buff,
                sizeof(buff), &rc);

if (rc < 0)
    Nprintf("Error: gethostbyname_r\n");
memcpy((char *)&socka.sin_addr,
       (char *)hostent.h_addr_list[0], Iid_SZ);
socka.sin_port = htons(1100);
rc = connect(s, (struct sockaddr *)&socka,
             sizeof(socka));

if (rc < 0)
    Nprintf("Error connecting to remote server\n");
```

Here you can see that `&socka` which is of type `sockaddr_in *` must be cast to a `sockaddr *` since this is what is expected by ***connect()***. This refers back to the previous discussion on structures and definitions.

fcntlsocket

Controls socket flags.

```
int fcntlsocket(int s, int cmd, int arg);
```

The networking commands are:

F_GETFL get flags

F_SETFL set flags

This should of course be ***fcntl***, but we append "***socket***" to this to avoid naming conflicts.

The ***fcntlsocket()*** function allows a socket to be set to use non-blocking semantics, and also allows the current setting to be retrieved.

Networking uses only one flag: `FNDELAY` (or `O_NDELAY`; both names seem to be in use) for non-blocking I/O.

See also:Non-blocking sockets in Chapter 5, *Dynamic Protocol Interface.*

Return Value

The return value is -1 for error, 0 for successful `SETFL`, the current value of the flags for successful `GETFL`.

gethostbyname

Returns the IP address that corresponds to a host name.

```
struct hostent *gethostbyname(char *name);
```

`name` The name of the host for which the IP address should be obtained.

The ***gethostbyname()*** function is not reentrant. The ***gethostbyname_r()*** function should be used in situations where reentrancy is a requirement. The name is normally of the form "`hostname`", but "`host/network`" can be used when you want to talk using a specific network interface.

See also: ***gethostbyname_r***

Return Value

0 IP address could not be obtained.

!= 0 IP address is in the returned structure.

Example

```
hostentp = gethostbyname("testserver");

if (hostentp != 0)
     memcpy((char *)&socksav.sin_addr,
              (char *)hostentp->h_addr_list[0], 4);
```

gethostbyname_r

Returns the IP address that corresponds to a host name.

```
struct hostent *gethostbyname_r(char *name,
                                  struct hostent*result,
                                  char *buff, int buflen,
                                  int *errcod);
```

name The name of the host for which the IP address should be obtained.

result Structure in which the IP address should be stored.

buff Scratch buffer, which should provide at least 32 bytes.

buflen Size of `buff`.

errcod Return code from function.

The name is normally of the form "`hostname`", but "`host/network`" can be used when you want to talk using a specific network interface. The IP address of the host is placed into the structure `hostent`. This function is reentrant and is available in many but not all 4.3 BSD implementations.

See also: ***gethostbyname***

Return Value

0 IP address could not be obtained.

!= 0 IP address is in the returned structure.

The `hostent` structure is defined as follows:

```
struct hostent {
  char *h_name;       /* name for host */
  char **h_aliases;   /* alias list */
  int h_addrtype;     /* host address type */
  int h_length;       /* length of address */
  char **h_addr_list;  /* list of addresses */
};
```

Example

```
if (gethostbyname_r("testserver", &hostentp,
                    buff, sizeof(buff),&errval))
    memcpy((char *)&socksav.sin_addr,
           (char *)hostentp->h_addr_list[0], 4);
```

getpeername

Extracts the remote address information for a socket.

```
int getpeername(int s, struct sockaddr *name,
                int *namelen);
```

s Socket identifier.

name Structure into which the remote address information should be stored.

namelen A pointer to the length of the *name* structure.

The ***getpeername()*** function retrieves the remote address information and stores it in the supplied structure.

Return Value

-1 Error.

0 Remote address was retrieved.

Example

```
struct sockaddr_in socka;
int rc;    /* return value */
int s;     /* socket identifier */
…

s = socket(PF_INET, SOCK_DGRAM, 0);
…

rc = getpeername(s, (struct sockaddr *)&socka,
                  &socksize);

if (rc < 0)
    Nprintf("Error in getpeername\n");
```

getsockname

Extracts the local address information for a socket.

```
int getsockname(int s, struct sockaddr *name,
                int *namelen);
```

s Socket identifier.

name Structure into which the local address information should be stored.

namelen A pointer to the length of the *name* structure.

The ***getsockname()*** function retrieves the local address information and stores it in the supplied structure.

Return Value

-1 Error.

0 Local address was retrieved.

Example

```
struct sockaddr_in socka;
int rc;     /* return value */
int s;      /* socket identifier */
…

s = socket(PF_INET, SOCK_DGRAM, 0);
…

rc = getsockname(s, (struct sockaddr *)&socka,
                 &socksize);

if (rc < 0)
    Nprintf("Error in getsockname\n");
```

getsockopt, setsockopt

Gets and sets options on sockets.

```
int getsockopt(int s, int level, int optname,
               char *optval, int *optlen);
int setsockopt(int s, int level, int optname,
               char *optval, int *optlen);
```

s Socket handle.

level See Table 6-1 below.

optname See Table 6-1 below.

optval Pointer to option value.

optlen Pointer to the size of the data stored in *optval*.

The functions in the following table manipulate socket options.

Table 6-1: Routines that Manipulate Socket Options

level	optname	Description
IPPROTO_IP	IP_OPTIONS	Options in IP Header
IPPROTO_TCP	TCP_MAXSEG	get TCP maximum segment
	TCP_NODELAY	don't delay send
SOL_SOCKET	SO_BROADCAST	permit broadcast
	SO_DEBUG	debug flag
	SO_DONTROUTE	no routing
	SO_ERROR	get and clear error code
	SO_KEEPALIVE	keepalive probing
	SO_LINGER	linger on close
	SO_OOBINLINE	leave URG data inline
	SO_RCVBUF	receive buffer size
	SO_SNDBUF	send buffer size
	SO_REUSEADDR	local address reuse
	SO_TYPE	get socket type

See also: ***fctlsocket, ioctlsocket***

Return Value

-1 Error.

0 Success. The `optval` pointer points to the option value for ***getsockopt()***; the option was set for ***setsockopt()***.

Example

```
rc = setsockopt(s, SOL_SOCKET, SO_KEEPALIVE, 0, 0);

if (rc < 0)
    Nprintf("Error in setsockopt\n");
```

ioctlsocket

Sets control parameters for a socket.

```
int ioctlsocket(int s, int request, char *arg);
```

`s` Socket identifier.

`request` Request type

`SIOCATMARK` checks out-of-bound mark.

`arg` Optional argument. `arg` is assigned 1 if the socket read is at the out-of-bound mark, 0 otherwise. `arg` is of type "`int *`".

The ***ioctlsocket()*** function behaves the same as the regular BSD Sockets ***ioctl()*** function, except that it only accepts socket identifiers. The optional third argument is used as a pointer for the result. There is some variation in how this function is defined in BSD sockets: The second argument may be "`unsigned long`", and of course the variable arguments are treated differently in non-ANSI C.

See also:***getsockopt, setsockopt***

Return Value

-1 Error.

0 Operation successful.

listen

Listens for connections.

```
int listen(int s, int backlog);
```

s Socket identifier.

backlog Specifies the number of connections that will be held in a queue waiting to be accepted. This value includes connections that are in the SYN_RCVD state and connections that are in the ESTABLISHED state that have not yet been accepted by the application.

The ***listen()*** function is part of the sequence of functions that are called to perform a passive open. This call puts the socket into the LISTEN state.

See also: ***socket, bind, accept***

Return Value

-1 Error.

0 Success.

Example

```
int rc;     /* return code */
int s;      /* socket identifier */
…

rc = listen(s, 5);
if (rc < 0)
    Nprintf("Error calling listen\n");
```

readsocket

Receives a message from a socket ID.

```
int readsocket(int s, char *buf, int len);
```

s Socket identifier.

buf Buffer into which received data will be stored.

len Maximum number of bytes to be received.

The ***readsocket()*** function behaves the same as the regular BSD Sockets ***read()*** function, except that it only accepts socket identifiers.

See also: ***recv, recvfrom, recvmsg***

Return Value

-1 Error.

>= 0 Number of bytes received.

recv

Receives a message.

```
int recv(int s, char *buf, int len, int flags);
```

s — Socket identifier.

buf — Buffer into which received data will be stored.

len — Maximum number of bytes to be received.

flags — Allows for these options:
- `MSG_OOB` returns urgent data.
- `MSG_PEEK` returns information, allowing it to be read again on a subsequent call.

The flag `MSG_WAITALL` is not supported.

See also: ***recvfrom, recvmsg***

Return Value

-1 — Error.

>= 0 — Number of bytes received.

The following error codes could be returned in `errno` or through ***getsockopt()*** if ***recv()*** returns indicating an error:

`EWOULDBLOCK`
Only returns if the socket is set up as non-blocking. If this is the case, then a call to ***recv()*** can check for `EWOULDBLOCK` and try again later, effectively polling.

`EWTIMEDOUT` Would only be returned if previously the macro ***SOCKET_RXTOUT*** was used to adjust the receive timeout of the socket. The application could call ***recv()*** again later.

`EOPNOTSUPP`
1. The call to ***recv()*** asked for out-of-band data (the flags parameter had `MSG_OOB` set), and none was available.

2. The call to ***recv()*** didn't ask for out-of-band data, and there is some that needs to be received.

`EBADF` Invalid socket handle. No need to close, since that call would return an error as well.

`ECONNABORTED`
A definite fatal error. Usually results from a retransmission timeout or reception of a `RST` segment. Time to close the socket.

Example

```
int rc;    /* return code */
int s1, s2;  /* socket identifiers */
unsigned char buff[BUFFLEN]; /* read buffer */
…

s2 = accept(s1, (struct sockaddr *)&socka,
            &socksize);
…

rc = recv(s2, buff, 2, 0);
if (rc < 0)
    Nprintf("Error receiving data.\n");
else if (rc == 2)
    Nprintf("Success: read 2 bytes\n");
else
    Nprintf(Error: did not retrieve 2 bytes\n");
```

Notice in this example that ***recv()*** uses the second socket identifier, the one returned from the ***accept()***, not the original socket which is used as an argument to ***accept()***.

recvfrom

Receives a message from a connection.

```
int recvfrom(int s, char *buf, int len, int flags,
             struct sockaddr *from, int *fromlen);
```

s	Socket identifier.
buf	Buffer in which information will be stored.
len	Number of bytes to receive.
flags	Specifies optional behavior: MSG_OOB returns urgent data. MSG_PEEK returns information, allowing it to be read again on a subsequent call.
from	Specifies the remote host to which the connection should be made.
fromlen	Size of the `from` data structure.

The ***recvfrom()*** function allows a connection to be made and a message to be read from the connection. The flag `MSG_WAITALL` is not supported.

See also: ***recv, recvmsg***

Return Value

-1	Error.
>= 0	Number of bytes received.

Example

The ***accept()*** or ***connect()*** call is not needed here since ***recvfrom()*** establishes the connection before reading.

```
int s1, s2;                /* socket identifiers */
int rc;                           /* return code */
unsigned char buff[BUFFLEN];      /* read buffer */
struct sockaddr_in socka; /* remote host address */
…

memset(&socka, 0, sizeof(socka));
socka.sin_family = AF_INET;
gethostbyname_r(hnp, &hostent, buff, sizeof(buff),
                &i1);

if (i1 < 0)
{
    Nprintf("%s not known\n", hnp);
    closesocket(s2);
    return -1;
}

memcpy((char *)&socka.sin_addr,
       (char *)hostent.h_addr_list[0], Iid_SZ);
socka.sin_port = htons(1100);
rc = recvfrom(s2, buff, 8, 0, (struct sockaddr *)&socka, &socksize);

if (rc != 8)
    Nprintf("Error in recvfrom\n");
```

recvmsg

Receives a message.

```
int recvmsg(int s, msghdr *msg, int flags);
```

s Socket identifier.

msg Pointer to structure that describes how received data should be stored. This structure is shown below.

flags Specifies optional behavior:
- MSG_OOB returns urgent data.
- MSG_PEEK returns information, allowing it to be read again on a subsequent call.

The ***recvmsg()*** function is the most general of the ***recv*** functions. This function allows a connection to be established and read with one call. The flag MSG_WAITALL is not supported.

Here is the definition of the msghdr structure:

```
struct msghdr {              /* Message header for recvmsg */
  char *msg_name;            /* optional address */
  int msg_namelen;           /* size of address */
  struct iovec *msg_iov;     /* scatter/gather arra */
  int msg_iovlen;            /* num of elems in msg_iov */
  char *msg_accrights;       /* access rights */
  int msg_accrightslen;
};

struct iovec {               /* address and length */
  char *iov_base;            /* base */
  int iov_len;               /* size */
};
```

USNet ignores the access rights field in the msghdr structure.

See also: ***recv, recvfrom***

Return Value

-1 Error.

>= 0 Number of bytes received.

selectsocket

Waits for activity on a set of sockets.

```
int selectsocket(int nfds, fd_set *readfds, fd_set
                 *writefds, fd_set *exceptfds,
                 struct timeval *timeout);
```

nfds — Number of sockets. Watch out for "off by one" errors. For example, if the highest value of the descriptors that should be evaluated is n, nfds should be set to n+1.

readfds — Socket identifiers for which ***selectsocket()*** should return if data becomes available or the state of the socket changes.

writefds — Socket identifiers for which ***selectsocket()*** should return if the socket can accept more data or if there is an error.

exceptfds — Socket identifiers for which ***selectsocket()*** should return if out-of-band data is available.

timeout — Specifies time after which ***selectsocket()*** will return if none of the specified conditions occurs.

This is a general UNIX routine, but handles sockets as well as files. The `fd_set` structures specify which sockets (range 0 to `nfds`-1) are considered.

These macros can be used to manipulate `fd_set`:

`FD_ZERO(&fd_set)`	clears the socket list
`FD_SET(s, &fd_set)`	adds socket *s*
`FD_CLR(s, &fd_set)`	removes socket *s*
`FD_ISSET(s, &fd_set)`	non-zero if *s* included

When ***selectsocket()*** returns, there are bits in the `fd-set` structures only for those sockets that satisfied the condition.

Structure `timeval` gives the timeout value:

```
struct timeval {        /* Time-out format for select() */
    long tv_sec;        /* seconds */
    long tv_usec;       /* microseconds */
};
```

A `NULL` pointer means an infinite timeout. If the structure contains the value 0, then the descriptors will be checked once and the call to ***selectsocket()*** will return without delay. This is useful for application-level polling.

USNet does not assume that the operating system supports a select type operation, and performs this call using polling and short sleeps.

Return Value

-1	Error. Note that this should not occur in the current implementation.
0	Timeout occurred.
>0	This number of sockets are ready for the requested operations.

Example

```
int s1, s2, s3;     /* sockets */
int rc;             /* return code */
fd_set socket_set1, socket_set2;
…

FD_ZERO(&socket_set1);
FD_ZERO(&socket_set2);
FD_SET(s1, &socket_set1);
FD_SET(s3, &socket_set1);
FD_SET(s2, &socket_set2);
rc = selectsocket(3, socket_set1, socket_set2, 0, NULL);

if (rc < 0)
    Nprintf("Error, no sockets ready.\n");
else
    Nprintf("%d sockets ready.\n", rc);

if (FD_ISSET(s1, &socket_set1))
    Nprintf("Socket 1 is ready to be read.\n");
else if (FD_ISSET(s2, &socket_set2))
    Nprintf("Socket 2 is ready to be written\n");
else if (FD_ISSET(s3, &socket_set3))
    Nprintf("Socket 3 is ready to be read.\n");
else
    Nprintf("Error.\n");
```

send

Sends a message on an established connection.

```
int send(int s, char *buf, int len, int flags);
```

s Socket identifier.

buf Pointer to data to be sent.

len Number of bytes to send.

flags Allows for these options:
MSG_OOB sends the data as urgent data
MSG_DONTROUTE ensures that the message is not sent through a default router.

The ***send()*** function can be used with sockets for which the connection has previously been established.

See also: ***sendto, sendmsg***

Return Value

-1 Error.

>= 0 Number of bytes sent.

If ***send()*** returns indicating an error, the following error codes could be returned in errno or through ***getsockopt()***:

EBADF The socket descriptor is invalid, or another process is using the socket at the moment.

ESHUTDOWN The application has already requested that the sending side of the socket be shut down. No further data can be sent through this socket.

ECONNABORTED An error has occured on this socket. The socket should be closed.

EMSGSIZE A non-stream socket has been asked to send more information than can be written at once through the socket.

ENOBUFS The system is out of buffers for sending data. The call to ***send()*** can be retried later.

Example

```
int s2;     /* socket identifier */
int rc;     /* return code */
unsigned char buff[BUFFLEN];
…

rc = send(s2, buff, sizeof(buff), 0);
if (rc < 0)
    Nprintf("Error sending data\n");
```

sendmsg

Sends a message that can be split between buffers.

```
int sendmsg(int s, msghdr *msg, int flags);
```

s Socket identifier.

msg Pointer to structure that describes the data to be sent. This structure is shown below.

flags Specifies optional behavior:
MSG_OOB sends the data as urgent data
MSG_DONTROUTE ensures that the message is not sent through a default router.

The ***sendmsg()*** function is a send function that allows the data to be sent from an array of buffers.

Here is the definition of the msghdr structure:

```
struct msghdr {             /* Message header for recvmsg */
  char *msg_name;           /* optional address */
  int msg_namelen;          /* size of address */
  struct iovec *msg_iov;    /* scatter/gather arra */
  int msg_iovlen;           /* num of elems in msg_iov */
  char *msg_accrights;      /* access rights */
  int msg_accrightslen;
};

struct iovec {              /* address and length */
  char *iov_base;           /* base */
  int iov_len;              /* size */
};
```

USNet ignores the access rights field in the msghdr structure.

See also: ***send, sendto***

Return Value

-1 Error.

>= 0 Number of bytes sent

sendto

Send a message.

```
int sendto(int s, char *buf, int len, int flags,
           struct sockaddr *to, int tolen);
```

s — Socket identifier.

buf — Buffer from which information will be sent.

len — Number of bytes to send.

flags — Specifies optional behavior:
MSG_OOB sends the data as urgent data.
MSG_DONTROUTE ensures that the message is not sent through a default router.

to — Specifies the remote host to which the connection should be made.

tolen — Size of the `to data` structure.

The ***sendto()*** function allows a connection to be made and a message to be written to the connection.

See also: ***send, sendmsg***

Return Value

-1	Error.
>= 0	Number of bytes sent.

Example

```
rc = sendto(s, "HIJKLMNO", 8, 0,
     (struct sockaddr *)&socka, sizeof(socka));

if (rc < 0)
    Nprintf("Error sending\n");
```

shutdown

Shuts down part of a connection.

```
int shutdown(int s, int how);
```

s — Socket identifier.

how — Describes type of shutdown:
- 0 shuts down receive data path
- 1 shuts down send data path, TCP sends FIN
- 2 shuts down send and receive path

The ***shutdown()*** function is useful for fully specifying the limited closure of a connection. Normally the ***closesocket()*** function is used to fully close a connection.

See also: ***closesocket***

Return Value

-1	Error.
0	Shutdown successful.

socket

Creates a socket.

```
int socket(int domain, int type, int protocol);
```

domain For USNet, this should always be PF_INET.

type USNet expects one of three constants for this parameter:

SOCK_STREAM	stream socket (TCP/IP)
SOCK_DGRAM	datagram socket (UDP/IP)
SOCK_RAW	raw-protocol interface

protocol This can be specified as 0.

A call to ***socket()*** will create a socket of the specified type. A socket must be created before any other socket calls are used.

See also:***closesocket***

Return Value

-1 Error.

>= 0 The newly created socket can be accessed through this handle.

If ***socket()*** returns with an error indication, the value in errno or obtained through ***getsockopt()*** can be interpreted as follows:

EPROTONOSUPPORT
: The requested protocol is not available. Perhaps SOCK_STREAM was specified, but TCP support is not configured for the underlying stack.

Example

```
int s;    /* a socket */
…

s = socket(PF_INET, SOCK_DGRAM, 0);
if (s < 0)
    Nprintf("Error opening socket\n");
```

writesocket

Sends a message to a socket.

```
int writesocket(int s, char *buf, int len);
```

s	Socket identifier.
buf	Pointer to data to be sent.
len	Number of bytes to send.

The ***writesocket()*** function behaves the same as the regular BSD Sockets ***write()*** function, except that it only accepts socket identifiers.

See also: ***send, sendto, sendmsg***

Return Value

-1	Error.
>= 0	Number of bytes sent.

Multicast API (BSD)

In order to receive information associated with a multicast host group, join the multicast group by performing the following steps:

socket()	Use INET protocol family with SOCK_DGRAM.
setsockopt()	Use SO_REUSEADDR with a value of 1.
bind()	Use a well known port (to receive multicasts on).
setsockopt()	Fill out the mreq structure with an appropriate Multicast address and host interface. If no host interface is given, the default will be used instead. This is defined by the macro, ussDefaultMcNetno, and is declared in net.h.
recvfrom()	Receive Multicasts as they come in on the port that was bound.

7. Network Applications and Protocols

Overview

USNet comes with these networking application routines:

RARP	maps a hardware address to an IP address.
BOOTP	uses the UDP protocol to load a program over the network.
DHCP	delivers host configuration parameters to a client host.
TFTP	is a file transfer program implemented with UDP.
FTP	is a file transfer programs implemented with TCP.
TELNET	is the usual TCP/IP method of remote terminal access.
IGMP	is the multicast protocol.

The following are available as options, at extra cost:

NAT	is the network address translation.

RARP

RARP (Reverse Address Resolution Protocol) is used to map a hardware address to an IP address. The RARP client sends its own hardware address to the RARP server (using a broadcast address). The server will scan its tables for a match, and return the IP address.

To run USNet as a RARP server, just configure the required hardware (Ethernet) addresses in the host you want to act as a server. You can do this in the static table (**netconf.c**), or using the dynamic configuring (see DHCP below).

Get IP Address

The routine will attempt to get an answer from a RARP server on the specified network. If it succeeds, it stores the received IP address into the configuration table.

`int RARPget(int netno)`

The code performs exponential backoff in retries.

If the local host already has an IP address, the call will return immediately with a good status.

The ***RARPget()*** return codes are:

>0	Success
ETIMEDOUT	Timeout

BOOTP

BOOTP uses the UDP protocol to load a program over the network. There are two parts to it: The boot name server (which is really not needed in a small network), and the bootload code itself, which loads a file using UDP. USNet contains the following BOOTP support routines:

- ***BOOTPget()***
- ***BOOTPopen()***
- ***BOOTPread()***

Get Boot Record

This routine will attempt to get an answer from a boot server on the specified network.

`int BOOTPget(int netno)`

The response is stored in static memory, so the function is not reentrant. (This should not be a problem in bootloading.) The code performs exponential backoff in retries.

If the local host is configured with a zero IP address, this function will fill in any IP address it receives from the boot server.

The ***BOOTPget()*** return codes are:

0	Success
ETIMEDOUT	Not successful

Open Connection for Booting

This routine will try to open a connection for bootloading.

`int BOOTPopen(int netno)`

Function ***BOOTPget()*** must have been successfully called for the same network. The code performs exponential backoff in retries.

The function returns the connection number if it's successful, the negative error code `ETIMEDOUT` otherwise.

Read Bootload Data

This function is used to read the bootload file. Function ***BOOTPopen()*** must have been called successfully.

```
int BOOTPread(int conno, char *buff, int len)
```

How a binary load file is interpreted is highly system-dependent, so this function can't do any actual loading, it will just return the data. The transfer uses TFTP, and the records, except for the last one, are always 512 bytes long.

The ***BOOTPread()*** return codes are:

`>0`	Number of bytes
`0`	End of file
`ETIMEDOUT`	Timeout
`EMSGSIZE`	Buffer too small, less that 512 bytes

DHCP

DHCP (Dynamic Host Control Protocol) is a method by which a DHCP server can deliver host configuration parameters to a client host, typically when the client host boots. DHCP can be used within a subnet, and also across subnets, provided that a DHCP server is available, and the appropriate hosts have been set up to forward DHCP messages. DHCP is based on the BOOTP protocol, and provides extensions such as the ability for a server to dynamically assign reusable network addresses.

In USNet, DHCP is used to obtain an IP address for the host. The protocol will be used automatically as part of ***Portinit()*** and ***Portterm()*** if the system is configured by adding a `#define DHCP` line to **local.h**.

The call to obtain an IP address through DHCP is:

```
int DHCPget(int netno, unsigned long lease);
```

The parameter *`lease`* specifies the requested lease time in seconds.

DHCPget() return codes are:

`>0`	IP address allocated or renewed
`0`	No action needed
`ETIMEDOUT`	Timeout

The call to release an assigned IP address is:

```
int DHCPrelease(int netno);
```

The ***DHCPrelease()*** return codes are:

`0`	Success
`ETIMEDOUT`	Timeout

TFTP and FTP

TFTP and FTP are file transfer programs. TFTP is implemented with UDP, and FTP with TCP. The two ends of a file transfer are called a client and a server. The server is the passive component, which sits and waits for requests. To view the source code, refer to files **fttest.c** (see Chapter 8 *Test Programs*) and **ftp.c**.

The FTP server as shipped is configured for ANSI C support. In this mode, only the basic file transfer functions are available. You can configure it for the DOS file system by setting the variable *`EXTENDED_C`* to 1.

Start Server

These calls will start the servers. If you are using a multitasker, you will want to start these as tasks.

```
int TFTPserv()
```

```
int FTPserv()
```

The server never returns. In other words, it sits in an infinite loop.

Send File

This call sends a file.

```
int TFTPput (char *host, char *file, int mode)
```

```
int FTPput (char *host, char *file, int mode)
```

The send file arguments are:

`host`	Name of the server host. The form can be *`host`* or *`host/network`*.
`file`	Name of the local file to be sent. You can also specify *`inputfile outputfile`* in cases where the file should be created under a different name.
`mode`	`ASCII` for a text file, `IMAGE` for a binary file.

The call returns 0 for success, -1 for failure.

TFTP & FTP Examples

```
TFTPput("XX", "test1", ASCII);
            /* test1 => host XX */

FTPput("XX", "t1 /usr/aa/t1", IMAGE);
            /* t1 => host XX target file /usr/aa/t1 */
```

Receive File

This call receives a file.

`int TFTPget (char *host, char *file, int mode)`

`int FTPgetchar (*host, char *file, int mode)`

The receive file arguments are:

host	Name of the server host. The form can be *host* or *host/network*.
file	Name of the file to be received. You can also specify *inputfile outputfile* in cases where the file should be created under a different name.
mode	`ASCII` for a text file, `IMAGE` for a binary file.

The call returns 0 for success, -1 for failure.

FTPget Examples

```
FTPget("XX", "test1", ASCII);
                /* test1 <= host XX */

FTPget("XX", "t1 \tmp\t1", IMAGE);
                /* \tmp\t1 <= host XX t1 */
```

Telnet

Telnet is the usual TCP/IP method of remote terminal access. The client part of Telnet acts as a terminal emulator. The server part depends quite a bit on the circumstances, but is usually a command processor with a remote login. The figure below shows this relationship.

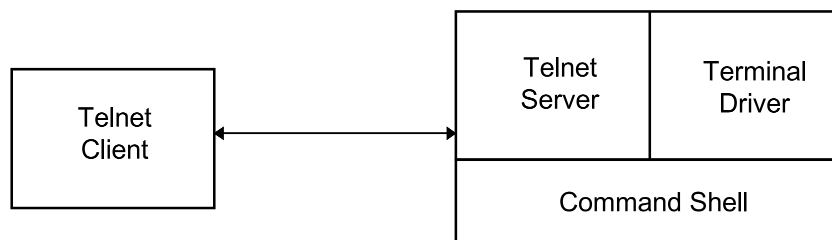


Figure 7-1: TCP Remote Terminal Access

USNet Telnet is packaged as two main programs: Client **telnet.c** and server **tnserv.c**. You may want to make these into tasks (see FTP or TFTP). To run the server under UNIX or DOS, use the command:

```
tnserv
```

To run the client under UNIX or DOS, type:

```
telnet <target name>
```

<Alt-X> terminates the program, as it is currently supplied. Change the variables *ESCNAME* and *ESCTYPE* if you require a different termination character.

USNet is not tied to any specific operating system, so there is no defined command shell. The supplied Telnet server performs only the following functions:

- It exchanges some basic control information with the client.
- It reads command lines, and calls routine ***command()*** for each. The routine as supplied just echoes the command.

The Telnet client (terminal emulator), as supplied, is able to log on to a UNIX computer and to perform other similar tasks.

IGMP / Multicast

IGMP (Internet Group Management Protocol) allows sending messages to multiple hosts in a group.

USNet must be configured to include multicast support code if the application needs to send or receive multicast messages. This setting is made with the USS_IP_MC_LEVEL macro in local.h, and is described in Chapter 4, *Configuration*.

No special application level operations need to be performed when sending information to a multicast group. When the IP address of the destination is a multicast host group, then the physical layer frame will be built appropriately for delivery to the multicast group, and sent on the default multicast interface. The index of the default multicast interface is specified via the constant ussDfltMcNetno which is defined in net.h.

The host group addresses range from 224.0.0.0 to 239.255.255.255.

The USNet multicast application program interface is based on the recommended interface described in RFC 1112.

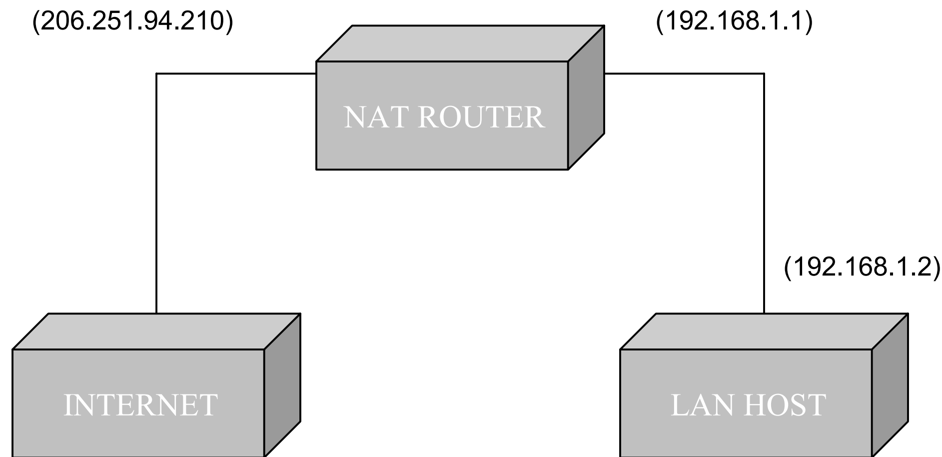
See the DPI or BSD chapter for documentation of the multicast API functions.

NAT

Note: NAT is available as an extra-cost option for USNet.

USNet currently has support for NAPT (Network Address Port Translation). This form of NAT assumes that hosts on the internal LAN will initiate communications with hosts on the external WAN through the USNet NAT router. ICMP, UDP, TCP and other protocols may be used through a USNet NAT router. Support for the FTP protocol ALG (Application Layer Gateway) is also included.

The following diagram represents an example NAT router's network:



This section describes how to build USNet as a NAT router.

Configuration

In file drvsrc\$(CPU)**local.h** set RELAYING to 1 to enable USNet to relay between interfaces:

Change:

```
#define RELAYING 2
```

To:

```
#define RELAYING 1
```

In file NETSRC**netconf.c**, add the NATLOCAL flag to each interface that should behave as the router for a private network. The following netdata example shows one private (internal or LAN) interface and one public (external or WAN) interface.

```
"nat", "int", C, {192,168,1,1}, EA0, NATLOCAL, Ethernet, NE2000, 0,
        "IRNO=10 PORT=0x300",
"nat", "ext", C, {206,251,94,210}, EA0, 0, Ethernet, NE2000, 0,
        "IRNO=11 PORT=0x400",
```

In file NETSRC**nat.c**, several table size definitions exist.

TUTABLESZ — TCP/UDP table size

This value represents the number of entries that may concurrently exist within the NAT TCP/UDP table. All TCP and UDP communications routed through the NAT router must be entered in the TU Table.

ICMPTABLESZ — ICMP table size

This value represents the number of entries that may concurrently exist within the NAT ICMP table. Every ICMP message must have a corresponding entry in the ICMP Table.

UNTABLESZ — Unknown protocol table size

This value represents the number of entries that may concurrently exist within the NAT Unknown protocol table. Entries in this table include all IP protocols other than TCP, UDP, and ICMP. Every transaction taking place via the NAT router must have the protocol registered in the Unknown Protocol Table.

These should be defined to appropriate values for the target networking environment. This is determined by examining the requirements of the LAN hosts. For example, if there are 2 LAN hosts and each host will open no more than 5 concurrent UDP/TCP communication channels with hosts on the Internet, then a maximum of 10 (2x5) entries may need to be maintained. Therefore, TUTABLESZ must be defined to 10 to avoid lost information. The default value in NETSRC\nat.c is 10.

ICMP messages often do not expect replies. This means that only the maximum number of simultaneously routed ICMP messages must be accounted for. As a rule of thumb, this value can be set to the number of hosts on the local network.

The unknown protocol table should include all other Internet communications not using TCP, UDP, or ICMP.

Explanation of table entry replacement:

A modified LRU algorithm is used when the NAT table is full and a new entry is added. Entries that are least used and have the least precedence are replaced first. The precedence is primarily determined by the transport protocol in use. The precedence is ICMP, UDP, Unknown, TCP, and TCP-FTP-control, in order of least to greatest precedence.

If a TCP or UDP channel is replaced in the NAT table, a new local port number will be generated and will disrupt communications using an existing connection.

The cost of adding new entries is linear on a per-datagram basis. In other words, each datagram passed through the NAT router is searched for linearly in the NAT table. As the number of NAT entries increases, the amount of CPU time spent searching for those entries also increases.

As with USNet in general, the debugging trace level may be used to enable printf() debugging from the NAT module. By default, if NTRACE (or TRACE_DEBUG from config.mak) is 5 or greater, the following NAT debugging information will be generated:

Inbound/Outbound IP address mappings (IP.port => IP.port)

TCP/UDP port adjustments (TCP/UDP.port => TCP/UDP.port)

FTP translations (Sequence number, PORT command)

If NTRACE is 7 or greater, NAT will print out:

Table additions/removals

If NAT debugging is to be isolated from the rest of USNet debugging, set NTRACE or TRACE_DEBUG to 0 (or the appropriate value) and modify netsrc\nat.c as follows:

```
#include "net.h"

#include "local.h"

#include "support.h"


#undef NTRACE           /* Undefine USNet NTRACE for this module */

#define NTRACE 7        /* Redefine NTRACE for nat.c only */
```

8. Test Programs

Overview

USNet comes with many test programs, each designed to test a specific aspect of USNet functionality. Test programs and sample applications reside in the application source directory **appsrc**. The list below summarizes these programs. The file name of each demo is the demo name with .c extension (e.g. ping.c). Following this summary, more details are given for some of these tests.

BENCH

Benchmarking test for USNet. Provides performance values for throughput and latency. In order to get a benchmark, the application is run on two machines with one acting as a server, and one (the unit under test) acting as a client. The server should be significantly faster than the client in order to get good results. This test will sometimes fail to run to completion, especially when testing serial links.

This test is important because the error recovery mechanisms in TCP can mask problems in lower layers in the stack. If the benchmark shows performance that is in line with similar implementations, you can be more confident of proper operation.

This test also shows off USNet's efficiency. USNet usually has no problem coming close to the theoretical maximum throughput of a 10 Megabit Ethernet network. Embedded processors typically don't have the horsepower to max out a 100 Megabit network, but that may be changing.

DHCPTEST

Minimal application to start the DHCP server. There is no formal testing framework built into this application.

EMTEST

Embedded FTP client. This is recommended as the second test to run when verifying USNet on a new system, after LTEST. It is documented in Chapter 2, *Quick Start*.

FTTEST

FTP client and server. Has more capabilities than EMTEST but requires a file system.

- Can act as an FTP server or FTP client
- Can act as a TFTP server or TFTP client
- Demonstrates use of built-in file transfer functions (i.e. FTPget(), FTPput())

This test can be used against EMTEST instead of a UNIX or PC FTP server.

FTTEST uses either the FTP or the TFTP protocol to check the reliability of the connection. FTTEST uses the FTP functions presented earlier. It runs in one host as a server, and in another as a client. To start the server, type:

```
FTTEST
FTTEST -UDP
```

To get the test running, type at another host:

```
FTTEST <host>
FTTEST -UDP <host>
```

The `-UDP` flag specifies the TFTP (and therefore, UDP) protocol. The test will run until stopped with the <Escape> key, or until there is an error. The FTTEST client display, using TCP and `TRACE_DEBUG`=1, will be something like this:

```
220 FTP server ready.
331 Password required.
231 User name accepted.
200 OK.
200 OK.
150 Ready to take file.
TX 102400 bytes in 2530 ms = 40474 bytes/sec
226 closing.
221 Goodbye.
220 FTP server ready.
and so on
```

Most of the trace consists of echoing the server replies, so the exact look depends on the server. The above example uses USNet as server.

Most UNIX systems are set up to run the FTP server without any special preparations. You need to check that the **local.h** variables `USERID` and `PASSWD` will log into the server. The client must be configured, usually in **/etc/hosts**. TFTP is typically not active in UNIX systems, and you may not be able to activate it.

FTTEST against a UNIX system may end after a while (at least 10 minutes) by the UNIX system not responding to an open (a SYN message). This only means that UNIX has run out of buffer space. Wait a few minutes and restart the test.

Because FTTEST uses ANSI C stream I/O, it may not run in your embedded system. For this situation, use EMTEST, as described above. However, just sending messages between two tasks is not a particularly good reliability test. In these matters, the dirtier the better. An excellent test is to configure FTTEST to use a 38,400 bps serial line between two PCs. The delay associated with DOS disk I/O will make sure that plenty of data is lost, and there are lots of retries. If the test runs overnight under these conditions, then you can trust that your application will not wither away at the first sign of network trouble.

The FTTEST server does not have a key sequence for quitting, and when run on a PC will require you to reboot your system to exit. The code was written this way with embedded boards in mind. You may get around this if you run the server process in an interactive debugger.

HTTEST

Minimal application to start the http server. Note that the default set up for the http server includes content that demonstrates many of its features.

LTEST

Loopback test. This is highly recommended as the first test to run when verifying USNet on a new system. It sets up a TCP connection through a loopback device driver, so that all communication takes place within the unit under test. This test is documented in Chapter 2, *Quick Start*.

MCRXTEST and MCTXTEST

Multicast test. Fairly simple test that can be used to demonstrate proper reception of IP multicast datagrams. Mostly of interest to those working with multicasting. Run these against each other. TX is the sending side; RX is the receiving side.

To run mcrxtest:

1. Set USS_IP_MC_LEVEL to 2 in include\local.h.
2. Configure the host in netsrc\netconf.c with a valid Ethernet interface entry.

   NOTE: Ensure that the driver being used supports the ussMcastGroupJoinE ioctl() option. The following drivers already have support:

AMD961	drvsrc\amd961.c
NE2000	drvsrc\ne2000.c
EN360	drvsrc\m68k\en360.c

   If the Ethernet controller being used supports a 'promiscuous mode' of operations but does not have the Multicast ioctl() option, then the pattern established in the supported drivers listed above can be used to add the Multicasting functionality to the non-multicasting driver.
3. Build the USNet libraries by typing “omake” in the root install directory.
4. Build mctxtest by typing “omake mcrxtest” on the command line.
5. Load the resulting mcrxtest application file onto the target and execute.

To run mctxtest:

1. Set USS_IP_MC_LEVEL to 1 or 2 in include\local.h.
2. Configure the host in netsrc\netconf.c with a valid Ethernet interface entry. Note that it does not matter in this case what Ethernet driver is being used. The desitnation hardware address is specified at the link layer in this case.
3. Build the USNet libraries by typing “omake” in the root install directory.
4. Build mctxtest by typing “omake mctxtest” on the command line.

5. Execute the mcrxtest application on another machine (see "To run mcrxtest" above).
6. Load the resulting mctxtest application file onto the target and execute.

MTTEST

Multitasking test. It is especially important to run this test to verify USNet after porting to a new multitasking operating system. This test is documented in Chapter 2, *Quick Start*.

PING

Simple ping client utility. Designed as a command line utility, so mostly useful when running USNet from DOS. Runs until interrupted.

PING uses the ICMP protocol to report if a host is responding. USNet contains PING as an ANSI C application. To run the application, type one of these forms:

```
PING -s <size> <host>
PING <host>/<network>
PING *
PING n1.n2.n3.n4
```

The `-s` option allows you to specify the size of the optional data field to be sent with the ICMP echo request message that is generated. If you do not specify this value, the default value of 64 bytes will be sent.

The program will report at one-second intervals, until you press the <Escape> key. Some TCP/IP implementations will not respond to a wild-card ping; USNet does. Note that there must be a server process running at the remote host. Test program FTTEST will serve in this manner. You may also PING a UNIX machine with the standard TCP/IP daemons running. Here is an example of the trace output for PING while it is running:

```
ping pig

NE2000 004005120d6b IR10 P300
mail     nctn      c0.09.c8.03
dog      nctn      c0.09.c8.02
pig      nctn      c0.09.c8.0a

ARP 08005acd6a9d -> 192.9.200.10
1 reply from 192.9.200.10
2 reply from 192.9.200.10
3 reply from 192.9.200.10
4 reply from 192.9.200.10
5 reply from 192.9.200.10
6 reply from 192.9.200.10
7 reply from 192.9.200.10
```

The first line denotes the driver type, its hardware address, IRQ and I/O address. The next three lines show the network connections found in **netconf.c** The ARP is issued from the host “dog” and the PING replies come from host “pig”.

PITEST

An augmented version of PING that allows the interval between ICMP Echo Requests to be adjusted. You can use this test in conjunction with other tests to check performance under heavy network traffic conditions.

PITEST sends ICMP Echo Request packets to the host specified in the command line at a rate that can be adjusted while the test is running. The command line syntax is the same as for PING. To adjust the rate at which packets are sent, enter '**F**' (faster) to reduce the delay between sending packets, and '**S**'(slower) to increase the delay between sending packets. The timing loop used in the program is software-based, so the delay value will need to be adjusted to suit both the machine that is running PITEST, and the host that is being pinged. A good setting is achieved when the status reports show that the host under test is able to reply to almost but not quite all of the packets that it receives.

Here is a sample session:

```
C>pitest pig

NE2000 004005120d6b IR10 P300

ARP 08005acd6a9d -> 192.9.200.10
64 bytes  8442 ==>   <== 8442  10 sec DLY=40
64 bytes  8413 ==>   <== 8413  10 sec DLY=40
user terminated
```

Here is the meaning of the information on the status line:

`64 bytes`	The data field size is 64 bytes
`8442 >`	8442 packets were sent during this period
`< 8442`	8442 packets were received during this period
`10 sec`	The sampling period was 10 seconds
`DLY=40`	The software timing loop is set to 40 iterations

The status report will be updated every 10 seconds until you press <Escape>.

RYTEST

Relay test. A benchmarking program designed specifically to measure USNet performance when acting as a relay between 2 networks.

SOTEST

Socket test. This test exercises the Berkeley Sockets API of USNet. This is probably more commonly used for internal testing by USNet developers, but it may be of interest to curious users.

SOTEST uses both the UDP and the TCP protocol to check various BSD socket functions. It is quite simple, and you may want to modify it to test your particular application. SOTEST is used in the same way as FTTEST: It runs in one host as a server, and in another as a client. To start the server, type:

```
SOTEST
```

To get the test running, type at another host:

```
SOTEST <host>
```

The test will run until stopped with the <Escape> key, or until there is an error. The server process will not exit without rebooting the system. See FTTEST above for more information.

TELNET

Simple telnet client. Most useful when run from a DOS system. Accepts command line arguments and uses character I/O.

TNSERV

Simple telnet server that can be run on any system. Echoes any input that is received. Could be used as a starting point for a command line interface in an embedded system.

UXSERV

This program can be run on a UNIX system as the server counterpart to MTTEST. See the MTTEST section for more information.

9. Porting

Overview

USNet supports many processor, compiler, and RTOS combinations. However, if you need to port to a new one, see the relevant sections in this chapter for guidance.

Compiler and Processor Support

Processor Supported But Not Compiler

If there is support for your processor but not for your compiler, proceed as follows:

1. Create a compiler directory in **<*root*>\config\<*cpu*>** and in **<*root*>\drvsrc\<*cpu*>**. The second one may not be necessary. Copy all the files from an existing compiler directory in **config** and **drvsrc** to the newly created directories.
2. Edit the tool names and command line options in the **compiler.mak** file.
3. Check that any included assembly modules are suitable for the new assembler. For most embedded systems, **LTEST** needs ***start.asm*** (startup code), ***putchr.c*** (character display), and possibly ***suppa.asm*** (miscellaneous low-level support).

Neither Processor Nor Compiler Is Supported

If there is no direct support for your compiler and processor, proceed as follows:

1. Create directories for the new processor and compiler under the **config** and **drvsrc** directories. Use the files from an existing processor/compiler directory pair as a guide to create new files for your processor and compiler.
2. Edit the makefile parameter *`ENDIAN`* to `LITTLE` (little-endian or Intel-type addressing) or `BIG` (big-endian or Motorola-type addressing).
3. For segmented architectures such as the 8086 you also need to add the parameter *`FARDEF`* to the compilation flags. See the 8086 support on how this is done.

Hardware Configuration

There are five areas that need to be considered when interfacing USNet to a new hardware platform. See Table 9-1.

Table 9-1: Configuration Areas

Configuration Area	Description
Timer support	A hardware or software timer that provides USNET with a tick count.
Trace display output support	Character output routines.
Keyboard input support	Character input routines.
Interrupts support	Network driver and timer.
Low-level I/O support	Low level byte, word, or block I/O. ***chksum*** and ***memcpy*** routines.

Timer Support

USNet is shipped with ready-to-go timer routines which operate on a given hardware platform. In most cases this should be sufficient. However, if your embedded system requires the use of a new timer chip, you may support this by replacing the timer routines with your own. The USNet timer routines are described here to assist with that process.

USNet uses time values expressed in milliseconds. It does not need timer interrupts, just a tick count and a clock frequency. In DOS and UNIX environments, USNet takes these values from the ANSI C services. In embedded environments, USNet sets up a hardware clock.

Low-level support for each processor contains the following clock routines:

- ***Nclkinit()*** to set up the clock
- ***Nclkterm()*** to turn off the clock
- ***Nclock()*** to return number of clock ticks as a 32-bit integer

These are in **clock.c** or in **suppa.asm** depending on which hardware platform you installed. Each works for some particular test board; in some cases, several versions are included. You will quite possibly need to modify the code to fit your hardware. The clock frequency is stored into the variable `clocks_per_sec`.

The low-level clock routines are called from the macros ***LOCALSETUP()*** and ***LOCALSHUTOFF()***. The clock frequency is also stored in ***LOCALSETUP()***. (It would be more natural to store it in ***Nclkinit()***, but this is often in assembly code, and in some cases not used.) These macros are defined in **local.h**:

```
extern unsigned int clocks_per_sec;
#define LOCALSETUP 0, clocks_per_sec = 100, Nclkinit()
#define LOCALSHUTOFF() Nclkterm()
```

The number 0 that starts ***LOCALSETUP()*** is simply the return code in this case.

If you can use the ANSI C clock support, you do not need the low-level clock routines ***Nclkinit()***, ***Nclkterm()***, or ***Nclock()*** at all. Just set the clock frequency in the ***LOCALSETUP()*** macro, for instance:

```
#include <time.h>
extern unsigned int clocks_pwer_sec;
#define LOCALSETUP 0, clocks_per_sec =
               (unsigned int)CLOCKS_PER_SEC
#define LOCALSHUTOFF()
```

Most multitasking operating systems handle time-keeping functions. For these, define ***Nclock()*** to return the number of ticks as a long integer, and ***LOCALSETUP()*** to store the clock frequency into `clocks_per_sec`.

Display and Keyboard Support

USNet provides a function, ***Nputchr()***, for displaying characters to a display device. You may replace this function with your own if required by your application or hardware platform.

The application level and the trace feature of USNet uses the function ***Nputchr()*** when displaying characters and ***Ngetchr()*** when reading characters from a keyboard. ***Nputchr()*** is written either in the C file **putchr.c**, or in the assembler file **suppa.asm** (depending on how USNet was installed). In DOS and UNIX environments, the C code uses the ANSI C function ***putchar()***. The following is an example of ***Ngetchr()*** found in **suppa.asm** for the i8086:

```
_Ngetchr:
        mov     ah,00H
        int       16H
        or        al,al
        jz        get3
        mov    ah,0
get3:   ret
```

Interrupts

If USNet already provides support for your processor you may ignore this section. Otherwise, you will need to make the following alterations for the new processor.

1. The **driver.h** macro ***DISABLE()*** disables interrupts, ***ENABLE()*** enables them. Alter these macros so they operate for your processor.

2. For writing the interrupt support, if your compiler supports C-level interrupt functions, you may use the M68k **driver.c** code as a base to start with. For assembly stubs, borrow the HC16 code, and the interrupt stub ***irstub()*** in **suppa.asm** *or* **suppa.s**.

3. If you need to clear the interrupt, do this at the start of each interrupt stub. If these are in C, you may do this by defining macro ***CLEARIR(`irno`)*** in ***driver.h***. See ***driver.h*** for the I8086 for an example.

4. Complete ***driver.c*** routines ***IRinstall()*** and ***IRrestore()***. These may need code to unmask and mask interrupts, as in the x86 versions. You need to install and to retrieve interrupt handlers.

This may use ready-made routines (as for the x86), or assembly code (HC16), or mostly C (M68k).

5. Routine ***mapioadd()*** returns a memory-mapped I/O address as a far pointer. If there are no far addresses, it should just return the argument typecast as a pointer.

Low-Level I/O

Low-level input and output routines are defined in **suppa.asm**, **suppa.s,** and **driver.h**. Due to their nature, these routines are usually written in assembler. Depending on your target requirements, these routines support byte, word, and block I/O for the network driver. To get a feel for how this is done, review the I/O routines for a supported hardware platform other than i8086 (i8086 uses library I/O routines).

This is an example of an input byte routine for a SPARC processor:

```
.global __inb
__inb:
        lduba   [%o0+%g0] 4,%o0          ! load byte
        nop
        nop
        nop
        jmpl    %o7+8,%g0
        nop
```

Porting to a New Multitasking RTOS

USNet works with or without a multitasking RTOS. In a non-multitasking environment, it performs internal tasking to handle timeouts and incoming messages, and can use several connections in parallel (by using multiple ***Nopens*** within an application). However, to run several servers or a server and a client in the same host requires the use of a multitasking environment.

Several macros are provided with USNet to support multitasking. The multitasking macros are contained in file **mtmacro.h**. If you don't specify any multitasker when you install USNet, you should be using a copy of **mtmacro.h** that does not actually use multitasking. If you specify a multitasker, you should be using the **mtmacro.h** specific to that multitasker.

This section covers these topics:

- Multitasking Configuration
- Signaling
- Task Creation
- Yielding Control
- Preemption
- Signaling

Multitasking Configuration

Parameter `MT` in **mtmacro.h** specifies the type of multitasking:

0 = no multitasking

1 = cooperative multitasking

2 = preemptive multitasking

If you are using your own multitasking system, you will also need to rewrite the macros defined in this chapter to call the multitasking services of your environment.

The functions needed are few and simple, and just about any multitasker should do. USNet does not require preemptive task switching or task priorities. The macros are explained in detail below. Differences between single and multitasking environments are also described.

Creating Tasks

The **RUNTASK()** macro is used to create and start a task using entry address *`func`* and task priority *`prior`*.

`RUNTASK(`*`func`*`, `*`prior`*`)`

If task priorities are not supported, just ignore this field. If they are, you need to check that the three parameters shown in Table 9-1 have reasonable values defined in **mtmacro.h**.

Table 9-1: Priority Parameters

Parameter	Priority
SERV_PRIOR	Priority for servers
CLIENT_PRIOR	Priority for client tasks
NET_PRIOR	Priority for the network task (handling of arrived messages)

`NET_PRIOR` must map into a higher priority than the other two. (The network task must never be preempted by a task that uses network services.) The system should be able to process incoming and outgoing network messages before application processes are ready to send or receive them.

The pointer argument is essential to handle the FTP and TFTP servers properly. If your multitasker does not support it, you can pass it to the task through a table indexed by task number.

The definition `TASKFUNCTION` is the function type for your tasks. If the operating system has no special requirements for this, make it `void`. SuperTask! for instance expects all tasks to be `void FAR`.

USNet assumes that a task main-level return terminates a task. If this is not true in your multitasking system, you need to add a task termination call to the end of the FTP and TFTP tasks.

Yielding Control

The ***YIELD()*** macro may be used to allow other tasks an opportunity to execute. ***YIELD()*** is called internally from the "write message" function to ensure the network task (the task that processes arrived messages) gets a chance to execute. If you use preemptive scheduling, you can leave ***YIELD()*** empty since task switching should do this for you.

In some multitaskers, there is no simple way to give up control from a high-priority task. As you should never give the network task a low priority (see above), this should not be a problem.

Preemption

USNet calls the ***BLOCKPREE()*** macro to block preemption, and ***RESUMEPREE()*** to restore it. If there is no preemption, these macros are empty.

In some cases, the best way to implement these functions is by disabling and enabling interrupts. The blocking is never done for long periods of time, and never over an operating system function.

Signaling

USNet relies on the following macros for interprocess communication.

`WAITFOR(`*`condition`*`, `*`signo`*`, `*`msecs`*`, `*`flag`*`)`

This waits for *condition* to become true, with the timeout *msecs*. The *condition* is any standard C language condition statement; *msecs* is an integer value representing the number of milliseconds until the wait times out. *Flag* is a return value. *Signo* assigns an identifying integer to the ***WAITFOR()***.

`WAITNOMORE(`*`signo`*`)`

`WAITNOMORE_IR(`*`signo`*`) (See note.)`

NOTE: Refer to section on *Interrupt Handlers* in Chapter 10, *Device Drivers*.

WAITFOR() is called from a task and causes that task to wait until a condition is met. ***WAITNOMORE()*** is called from a separate task to cause the condition to be retested. When ***WAITFOR()*** is called, execution halts temporarily at that point. It returns a value where success indicates the signal event condition occurred. If the wait times out, it returns a value indicating failed. Both of these conditions should be tested since generally execution will be different depending on the outcome.

In a non-multitasking system these may still be used. ***WAITFOR()*** may be used to wait until an incoming message is received, and ***WAITNOMORE()*** is essentially a null macro (which could be placed within the application for portability issues between single and multitasking systems, for example).

The argument *signo* (signal number) ties these two functions together. Below are the rules for the behavior of ***WAITFOR()*** and ***WAITNOMORE()***:

- If *condition* is true, ***WAITFOR()*** clears the flag and returns.
- If *condition* is not true, ***WAITFOR()*** will delay up to *msec* milliseconds. During this time, each ***WAITNOMORE()*** for *signo* causes ***WAITFOR()*** to retest the condition.

- If the ***WAITFOR()*** implementation has a *blind spot* (see examples below), the macro must accommodate lost signals by retesting the condition after time is up.
- If the condition never became true, ***WAITFOR()*** returns with `flag` set to any non-zero value. In other words, after processing resumes, `flag` is used for determining whether ***WAITFOR()*** timed out or exited due to the condition being met.
- Several tasks can be waiting for the same signal (same `signo`) but only in the ARP and RARP protocols. Preferably a ***WAITNOMORE()*** should wake all of them up, but a failure to do this is not fatal.
- The ***WAITNOMORE()*** is issued by one task only.
- The signal numbers go from `EVBASE` to `EVBASE + 2 * (NCONNS+NNETS) + 1` being computed from the maximum number of connections and the maximum number of networks. The numbers are expressed as macros, so you can change them fairly easily if necessary. These are found in **support.h**.

Please note that USNet does not expect your operating system to provide anything like the above ***WAITFOR()*** capability. You need to create the ***WAITFOR()*** yourself with the multitasking functions available to you. In fact, ***WAITFOR()*** is not necessarily tied to multitasking at all.

The single-tasking version of ***WAITFOR()*** is simply:

```
for (flag=0,endtime=TimeMS() + msec; !(condition); )
{
   if (TimeMS() >= endtime)
   {
      flag = 1;
      break;
    }
    YIELD();
}
```

In a multitasking environment, the condition would be retested each time a task called ***WAITNOMORE()***. However, there is no other task in a system without multitasking, and therefore the condition is tested periodically within this loop. The call to ***YIELD()*** can cause the condition to change, causing the loop to break. ***TimeMS()*** is a USNet support function which returns a network-synchronized base time. The call within the "`for`" statement is simply used to get the starting time of the ***WAITFOR()***, which, when added to `msec`, gives the time when the wait will timeout.

As mentioned already, the signaling macros require that signaling support be provided by the multitasking system. You may then rewrite these macros to include the system calls provided by your system. You can use *semaphores* to implement the needed signaling. These are present in almost all operating systems, defined in different ways, and often called *events*, *counters*, or *resources*. In all cases, you can wait for the semaphore (or event, etc.) to be set, and you can set and clear the semaphore.

Setting the semaphore will in some cases wake up all the waiting tasks, in some cases only one. Some systems use the word *event* for the former and semaphore for the latter feature, but this is not universal by any means. USNet assumes that signaling wakes up all tasks, but will manage even if this does not happen.

A semaphore has to be reset (cleared) at some point, otherwise no actual waiting will be done. In some cases the semaphore (or event, etc.) is cleared automatically by the multitasking system within either the waiting operation or the sending operation. In many cases it is cleared by a separate operating system call, either in the signaling task or in the waiting task. USNet does not really care which method is used.

Here is a pseudocode example using generic system calls ***wait()*** and ***set_semaphore()***, which would be replaced by the multitasking system's actual system calls. In the following discussion, execution will wait until the semaphore is set and will continue when it is cleared. We will also assume that in this multitasking system all tasks are awakened when the semaphore is set. The semaphore will be cleared automatically by the multitasking system.

```
#define WAITFOR(condition, signo, msecs, result_flag)
{
   for ((result_flag = SUCCESS);;)
   {
       if (condition is true)
       {
           result_flag = SUCCESS;
           break;
       }
       if (result_flag == TIMED_OUT
          break;
       result_flag = wait(signo, timeout value);
   }
}

#define WAITNOMORE(signo) set_semaphore(signo)
```

The first time through the loop, we set the `result_flag` to `SUCCESS` to bypass the timeout test below. Also, we must first test the condition so we do not cause an unnecessary wait if the condition is already true. If that occurs, we just drop out of the loop. Assuming the condition is not true, the timed-out test fails, and the wait halts execution. If another task sets the signal by calling ***WAITNOMORE()***, execution proceeds, `result_flag` will be set to `SUCCESS`, we loop again, the condition is retested, and if true, the loop breaks. The other case which can cause the loop to break is if the wait times out. In this case, we begin a new loop. If the condition is now true after the timeout, we set the `result_flag` to `SUCCESS` anyway and break. (It is arguable whether an application would need to do it this way. That is application-specific depending on what information is more important, the timeout, or the condition.) If the condition is still not true, we exit, with `result_flag` equaling `TIMED_OUT`.

Here is a specific example using the SuperTask! pulse event where wait is performed via ***wteset()***, and ***set_semaphore()*** is performed by ***pulsevt()***.

```
#define WAITFOR(condition, signo, msecs, flag)
   for (flag=SUCCESS; ; )
   {
      if (condition)
      {
         flag = 0;
         break;
      }
      if (flag != SUCCESS)
         break;
      flag = wteset(signo, ((long)msecs*CLOCKHZ)/1000);
   }
```

And in ***WAITNOMORE()***:

```
#define WAITNOMORE(signo) pulsevt(signo)
```

The above examples illustrate a problem we will call a blind spot. One should be aware of this when coding a ***WAITFOR()*** macro. The semaphore may be set and cleared (cleared either by a receiver or the sender) after the test for `condition` and before the ***wteset()*** function. In other words, the signal is set, but this task did not catch it in time before it was cleared. In this case the task will be kept waiting until either there is a new signal, or the time limit expires. This is not fatal but may need to be handled differently depending on the needs of the application. How this functions will depend on the capabilities of your multitasking system.

An ideal ***WAITFOR()*** mechanism would be what we might call *true events*. Basically, if a count of the number of signals or events is available, there is no need to clear the signal. These require the following three operations:

- Signal an event
- Get an event count
- Wait until the event count has changed

With these, the ***WAITFOR()*** macro might look like:

```
evcnt = getevent(signo);
While (!condition)
{
   flag = waitevent(signo, evcnt, msecs);
   if (flag)
      break;
}
```

There would be no blind spot since the signal never needs to be cleared. If you are writing your own multitasker, consider including a capability for returning the number of signals which have occurred.

The ***WAITFOR()*** and ***WAITNOMORE()*** macros may also be implemented using messages received across the network. A message acts in this way very much like a semaphore that wakes up only one task, and is automatically cleared by the waiting task. The blind spot here is the case where several tasks are waiting for the same signal. Messages usually require more overhead than semaphores.

10. Device Drivers

Overview

A number of device drivers are already provided with USNet for commonly used network controllers. These include Ethernet, Arcnet, and serial connections. A list of supported controllers is provided in the file **readme.txt**. If your application requires a new controller, you will need to write your own device driver. This chapter describes the steps needed for writing device drivers. If you are using one of the supplied drivers, you may still find this chapter useful, in that it describes the internals of USNet device drivers.

USNet includes two features to assist you with your device driver implementation. Support for interrupt handling is provided. All you need to do is write the interrupt handler as a C function and give USNet the name of this function as the interrupt handler for the device. Secondly, USNet provides several support functions which will assist with the interface between USNet and your driver. These functions and the use of them are presented in this chapter, along with a description of the interrupt handling mechanism. Finally, the functions required for a device driver are described and examples are presented.

Topics discussed are:

- Data Structures
- Support Functions
- Interrupt Handling
- Configuring Interrupt Table Size
- Configuring a New Processor
- Error Codes
- Writing A Device Driver
- Character Drivers
- Block Drivers
- Adapters

Data Structures

The `NET` and `MESSH` data structures may be used within a device driver for storing certain information. You may see how they are used in the function examples given later in this chapter, and also in the source code for the supplied device drivers. Only some of the fields relevant to device driver implementations are discussed here; however, their full definitions may be viewed in **net.h**. This section will describe `NET` and `MESSH` and point out several fields which you may find useful when writing your own driver.

Messh (MESSH) Structure

Message buffers are required by block drivers for storing incoming and outgoing messages. A message buffer consists of a header and the contents of the message itself. The message header is defined by structure MESSH in **net.h** as follows:

```
struct MESSH {          /* internal message header */
unsigned short mlen;    /* message length */
unsigned char netno;    /* network number */
char offset;            /* offset to data */
....
};
typedef struct MESSH MESS;
#define MESSH_SZ ((sizeof(MESS)+3)&~3)
```

A few useful fields of the MESSH Structure are shown in Table 10-1.

Table 10-1: Some Useful Fields of the MESSH Structure

Field	Description
mlen	Length of the message buffer. This includes the message data and the message header. The message length must be less than or equal to the maximum size of a message buffer (MAXBUF in local.h). The size of the message data would be: `mlen - MESSH_SZ;`
netno	Network number. This is an index into the network table (global variable nets) and indicates the network structure defining the network to be used for this message.
Offset	Generally, this is the message header size (MESSH_SZ). Adding this value to the address of the message header (or buffer) itself gives the address of the message data. For instance: `unsigned char *byteptr; /* ptr to message data */` `MESS *messptr; /* ptr to message buffer */` `.....` `byteptr = messptr + messptr->offset;`

Net (NET) Structure

The structure NET defines network connections to USNet. These fields may be useful within a device driver for storing device-specific information. Since device drivers are highly dependent on the architecture of the device, some of the fields of NET may be used in a number of different ways, depending on the requirements for the device.

Explaining all the ways a device driver could be written is certainly beyond the scope of this document. However, the source code for the device drivers may be examined to see some ways NET has been used previously.

```
struct NET {   /* structure defining a network */
    PTABLE *protoc[3];
               /* link, driver, adapter protocol */
    int irno[4];           /* interrupt numbers */
    int port;              /* I/O port */
    char FAR *base[2];     /* for memory-mapped I/O */
    char hwflags;          /* hardware level flags */
    MESS *bufbas;          /* input buffer base */
    MESS *bufbaso;   /* output buffer base */
    long bps;        /* bits per second */
    ....
    /* all hardware net structures must fit in SERIAL,
       use filler if necessary */
    struct SERIAL {
               /* hardware net data for serial lines */
    void (*comec)(int, struct NET *);
                            /* character from driver */
    int (*goingc)(struct NET *);
                            /* character to driver */
    ....
    } hw;
};
```

There are many fields within the NET structure, a few of which are described in Table 10-2.

Table 10-2: Some Useful Fields of the NET Structure

Field	Description
protoc	Protocol path. Each entry of this array stores a structure of pointers to functions. See the section on PTABLES later in this chapter. The functions are used for implementing a protocol level in the protocol stack. Specifically, ***protoc[0]*** stores the functions that implement the link layer, ***protoc[1]*** stores the device driver functions, and ***protoc[2]*** stores the adapter functions.
irno	Array of interrupt numbers for this network controller. This is used to store the interrupt numbers configured in the driver parameter field of the network configuration table, such as irno=5 (see Chapter 4). This value is later used when the device driver initializes (or installs) the interrupt handler function (see ***IRinstall()***).
port	Network controller's I/O port address.
base	May be used to store FAR pointer base addresses.
hwflags	Network controller hardware flags. This may be used if you need to set some flags for your device driver which determine, for instance, different modes of operation, or some other flag driver feature.
bufbas	Input buffer base. May be used for storing the address of an input message buffer in a block driver.
bufbaso	Output buffer base. May be used for storing the address of an output message buffer in a block driver.
bps	Bits per second, useful for storing the baud rate.
hw.comec	Pointer to the function which transfers a byte from the device driver to the link layer. This is used only with character drivers.
hw.goingc	Pointer to the function which transfers a byte from the link layer to the device driver. This is used only with character drivers.

Support Functions

The following macros and functions are provided to assist you with writing your device driver. They are intended to be used as an interface between USNet and a driver. Use them within your driver code to separate device-dependent code from USNet-dependent code. You will find these macros in file **driver.h**. In most cases you should not need to rewrite these.

Clear Interrupt

*CLEARIR(*irno*)*

The macro ***CLEARIR(irno***) is defined in **driver.h** and clears interrupt number `irno` in the interrupt controller. It does not clear the network controller. Note that this is an architecture-dependent function; different architectures will vary greatly on how this is handled. For the processors and controllers supported by USNet, code is supplied to handle clearing interrupts for the given architecture. When writing your own driver, you may need to clear the controller's interrupt within the device driver code. This is shown in the interrupt handler examples later in this chapter.

Clear Interrupt Example

```
void (*irnew[MAXIRNO])(int arg);

char irargs[MAXIRNO];

static void INTERRUPT irhan4() { CLEARIR(4); irnew[4](irargs[4]); }
```

Taken out of context, this example from the file **driver.c** may appear confusing at first. This simply shows that interrupt number 4 is cleared before calling its interrupt handler. `Irnew[]` is an array of pointers to functions (interrupt handlers actually). This code is already within USNet file **driver.c** (and **suppa.asm**) and does not need to be written by you (unless you are using a new processor). It is simply shown here to illustrate the use of this macro.

Disable and Enable Interrupts

Two macros are defined in **driver.h** for disabling and enabling interrupts. This is done as follows:

```
DISABLE();
<< code that cannot be interrupted >>
ENABLE();
```

Most of the supplied drivers do not need to use this. However, if in the course of writing your own driver you need to ensure that a section of code will not be interrupted, you may use these macros to guarantee that. There is an example of their use in **NE2000.C.**

Install Interrupt Vector

This routine installs a new interrupt handler into the interrupt table for interrupt `irno.`

```
void IRinstall(int irno, int netno, void (*irhan)(int netno))
```

`irno` is the interrupt number

`netno` is the network number

`irhan` is the pointer to the interrupt handler function, which takes the network number as a parameter

The interrupt handler is a function you write within the device driver code.

When an interrupt occurs, the handler will be called with the network number as an argument. The network number is an index to the USNet data structure which defines the network connection to be used by the interrupt. This is described further in the *Data Structures* section in this chapter.

IRinstall() automatically saves the old interrupt vector. The intended use of ***IRinstall()*** is to call it from the initialization routine within your device driver code. See the section on function ***init()*** later in this chapter for a specific example of its use.

Restore Interrupt Vector

This routine restores the original interrupt vector which was removed by a previous call to ***IRinstall().***

`void IRrestore(int irno)`

It is intended to be called from the shutdown function within your device driver. See the section on function ***shut()*** later in this chapter for a specific example.

Map I/O Address

This routine converts a flat 32-bit address into a far pointer.

`char FAR *mapioadd(unsigned long flat)`

It is only needed in segmented architectures that expect far pointers. See file **WD8003.C** for an example of its use.

Adding Messages to a Queue

The queue macros are used with block drivers to manipulate arriving and departing messages and the USNet queues which control them. Note that these macros are only relevant to block drivers.

QUEUE_IN Macro

When an interrupt occurs, indicating the network controller has received a new message, your interrupt handler will need to add this new message to the appropriate USNet queue. To queue an arrived message, use the macro:

`QUEUE_IN(ptr, qname, mess)`

ptr — is a pointer to the network structure `NETS[netno]`. The network structure contains fields which are pointers to message queues (see struct `NET` in **net.h**).

qname — allows you to specify which queue to add the message to. It takes a value of either `arrive` or `depart`**.** These are keywords which you do not need to predefine. The macro uses these in its string replacement as names of fields within the `struct NET`.

mess — is a pointer to the message.

QUEUE_IN Examples

This example shows how ***QUEUE_IN()*** would be used in the interrupt handler (discussed later in this chapter) to add an arrived message to the arrived message queue. The four periods and the << >> symbols, in this and subsequent examples, represent additional code which does not directly relate to this example and has been removed.

```
static void irhan(int netno)
{
MESS *mess;
```

```
struct NET *netp;
....
netp = &nets[netno];      /* nets is a global USNet table */
mess = << get message from controller's memory >>
....
QUEUE_IN(netp, arrive, mess);
....
}  /* end irhan */
```

To queue a message for transmission use the departure queue. One place where this may be used is within the ***writE()*** function (also discussed later), which may be used to send a message to the network.

```
static int writE(int conno, MESS *mess)
{
struct NET *netp;
        /* ptr to network structure for this device.*/
....
netp = &nets[mess->netno];
    /* get ptr to network required for this message */
....
QUEUE_IN(netp, depart, mess)
                          /* add to departure queue */
....
} /* end writE */
```

QUEUE_FULL Macro

The ***QUEUE_FULL()*** macro may be used to test for a full queue before attempting a ***QUEUE_IN().*** The syntax is:

`QUEUE_FULL(`*`ptr`*`, `*`qname`*`)`

ptr is a pointer to the network structure `nets[`*`netno`*`]`.

qname is the queue to be tested.

QUEUE_FULL Example

```
static int writE(int conno, MESS *mess)
{
struct NET *netp;
.....
netp = &nets[mess->netno];
      /* get ptr to network required for this message */
if (QUEUE_FULL(netp, depart))
   Nprintf("Error: departure queue full.\n");
else
   QUEUE_IN(netp, depart, mess);
```

Removing Messages from a Queue

You may use the macro ***QUEUE_OUT*()** to remove a message from a given queue and place it in a message structure. Its syntax is similar to ***QUEUE_IN().***

QUEUE_OUT Macro

The ***QUEUE_OUT()*** parameters are identical to those for ***QUEUE_IN().***

QUEUE_OUT(ptr, qname, mess)

QUEUE_OUT Example

This removes a message from the departure queue before writing it to the controller (the device). Refer to **NE2000.C** for a specific example.

```
irhan(int netno)
{
MESS *mess;
struct NET *netp;
....
netp = &nets[netno];
....
QUEUE_OUT(netp, depart, mess);
<<write message to network controller >>
....
}
```

QUEUE_EMPTY Macro

The ***QUEUE_EMPTY()*** macro may be used to test for an empty queue before attempting a ***QUEUE_OUT*().** The syntax for ***QUEUE_EMPTY()*** is:

QUEUE_EMPTY(*ptr*, *qname*)

ptr is a pointer to the network structure nets[netno].

qname is the queue to be tested.

QUEUE_EMPTY Example

Refer to **NE2000.C** for a specific example.

```
irhan(int netno)
{
MESS *mess;
struct NET *netp;
.....
netp = &nets[netno];
....
if (QUEUE_EMPTY(netp, depart))
  << process error >>
else
{
QUEUE_OUT(netp, depart, mess);
<<write message to network controller >>
}
.....
}
```

Writing/Reading to/from the Controller

The PC version of USNet provides macros ***_inb(), _outb(), _inw(),*** and ***_outw()*** for reading and writing bytes and words with a given address. These are used to send and receive data directly to and from the network controller device via a hardware address. These macros are processor and compiler dependent and might not be provided for your architecture. Therefore, this will be an operation you will need to do within your driver. You can see examples of their use in the device driver source code for the PC, for instance, files **WD8003.C, NE2000.C**, and **I8250.C** with USNet installed for the 8086.

Interrupt Handling

If you are using a USNet-supported processor, interrupt support has been placed in module **driver.c**, which uses the header file **driver.h**. The interrupt handlers in the drivers are subroutines called from stub routines in C (**driver.c**) or assembler (**suppa.asm**) depending on the compiler support. The purpose for this design is to separate the processor-dependent interrupt handling from the device driver code. In other words, the interrupt handler within the device driver will be called from one of these interrupt stubs.

The stub performs these tasks:

1. It saves all registers.
2. It clears the interrupt for the processor, if needed, by using macro ***CLEARIR().*** (Clearing it for the device is up to the driver.)
3. It initializes what might be needed to call C code. As an example, in 8086 it must clear the direction flag.
4. It calls the driver handler with one argument: The network number. For the I8086 and I386 versions, this is picked up from a table indexed by the interrupt number. For the other processors, each network has its own stub handler, so the index is simply picked up as a constant.

Now, the network controller's interrupt handler code can be written as an ordinary processor-independent C function.

This simple scheme might leave the interrupts disabled for the entire driver call. In most cases this is just fine in networking. Nested interrupts are properly supported by hardware. Any attempts to augment them with clever schemes are, in the high-pressure world of networking, unwise and unsafe. USNet will not be hurt by a few lost interrupts.

Clearing the interrupt controller (which is not needed in many processors) should normally be done first, not last. As was already mentioned, this is done in the interrupt stub. If you do it after you have cleared the device interrupt, you may clear the next interrupt also. There are cases, though, where interrupt clearing gets so complicated that it has to be written into the driver.

Configuring Interrupt Table Size

For the I8086 and I386 only you may specify the size of the **driver.c** interrupt tables with the parameter `MAXIRNO` in **driver.c**.

Configuring a New Processor

If USNet already provides support for your processor, you may ignore this section. Otherwise, you will need to make these alterations for the new processor:

1. The **driver.h** macro ***DISABLE()*** disables interrupts; ***ENABLE()*** enables them. Alter these macros so they operate for your processor.

2. For writing the interrupt support, if your compiler supports C-level interrupt functions, you may use the m68k **driver.c** code as a base to start with. For assembly stubs, borrow the hc16 code, and the interrupt stub ***irstub()*** in **suppa.asm**.

3. If you need to clear the interrupt, do this at the start of each interrupt stub. If these are in C, you may do this by defining macro ***CLEARIR(irno)*** in **driver.h**. See **driver.h** for the I8086 for an example.

4. Complete **driver.c** routines ***IRinstall()*** and ***IRrestore().*** These may need code to unmask and mask interrupts, as in the x86 versions. You need to install and to retrieve interrupt handlers. This may use ready-made routines (as for the x86), or assembly code (hc16), or mostly C (m68k).

5. Routine ***mapioadd()*** returns a memory-mapped I/O address as a far pointer. If there are no far addresses, it should just return the argument typecast as a pointer.

Error Codes

Two error codes you might want to use as return codes from your driver functions are `NE_HWERR`, and `NE_PARAM`. These are defined in **net.h** (you will need to `#include` **net.h** to use them). `NE_HWERR` is used to return hardware errors occurring in the device driver. `NE_PARAM` is used to indicate that bad parameters were passed to the device in the initialization routine. Some of the example driver routines later in this chapter use these return codes. The driver will send the return codes to USNet, which will in turn pass the error to your application via the user interface functions.

Writing a Device Driver

When you write a device driver, you need to include these functions: ***irhan(), init(), shut(), opeN(), closE(),*** and ***writE().*** Depending on your implementation, some of these may be not be needed. Also, you need to assign these functions to a `PTABLE`. This is a table of pointers to your driver functions and is the mechanism USNet uses to call them.

For example, the format for the driver is:

```
irhan(int netno)
{
   ....
}
init()
{
   ....
}
shut()
{
   ...
}
etc.
PTABLE ptable("driver name", init, shut, etc., );
```

Which function gets called at what time depends on the field it is assigned to within the table, i.e., when USNet expects the ***shut()*** function it will call the third function. Therefore, position within the table is crucial. Each function and the `PTABLE` is described in more detail in this chapter's sections on character and block drivers.

Character Drivers

The function of a *character driver* is to get and to send characters between a network controller and the link layer. It does not know what the characters mean, where they go, or where they come from. The driver does not assemble characters into messages, because this is a protocol-dependent job. Likewise, it does not disassemble a message into characters. A character driver would be typical of a serial driver.

Figure 10-1 shows how incoming data is handled within USNet as the data is transferred between the network controller and the application. The logic flows from top to bottom. The part above the wider line is performed on one character at a time.

device	data available interrupt
driver	gets character calls ***comec()***
link layer	add to message queue when done
TCP/IP	***screen()*** checks, processes
user application	***Nread()*** gets message

Figure 10-1: Incoming Data

Outgoing data shown in Figure 10-2 is handled according to the following diagram, again from top to bottom. The boxes below the wider line are done for each character.

user application	***Nwrite()*** message
TCP/IP	***writeE()*** adds headers
link layer	add to depart queue
driver	calls ***goingc()***, sends character to device
device	transmit interrupt sends character to network

Figure 10-2: Outgoing Data

The code you must write for your own device driver is represented at the driver level in the diagrams above. Therefore, for reading data, you must retrieve the character from the controller (device) and pass it to the link layer via the routine ***comec().*** Similarly, for sending, ***goingc()*** is used. Both ***comec()*** and ***goingc()*** are within the link layer source code. Their purpose is to act as an interface between the driver and link layer, which enables the device driver to be written as a separate module from the link layer. This greatly simplifies the writing of device drivers. Refer to module **slip.c** for an example of how ***comec()*** and ***goingc()*** are implemented.

Interrupting on each character is time-consuming. As a rule of thumb, an Intel 386 can handle at most 38,400 bits per second this way – less if a DOS extender is used. Higher rates than this require either a FIFO buffer in the serial port, or the use of DMA.

USNet character drivers are short and simple, and will work in any protocol stack. A typical size would be 200 lines of code. The easiest way to produce a new driver is probably by editing the I8250 code.

The following text explains the routines and data structures of a character driver. The examples provided are based on the code for the **I8250.C** driver. In some places, some of the original code not relevant to the discussion has been removed and replaced with four period symbols, or with angle brackets and some pseudocode (such as `<< write message to buffer >>`). Refer to the source code in **I8250.C** to see a specific implementation of these routines. Comments which are not part of the original code have been added to the examples for explanation. Recognize that some of the code in these examples is device dependent and will be different for your device, particularly the ***_outb()*** and ***_inb()*** calls. Study the examples for understanding the process, but don't get bogged down in the device-dependent details.

Interrupt Handler

This is a regular C function, called from the interrupt stub which is triggered when a network I/O interrupt occurs. The argument is used to index to the network tables.

`static void irhan(int` *`netno`*`)`

The code for ***irhan()*** should determine the status of the interrupt. If this is *`transmitter empty`*, the handler needs to send a character to the device and must call the routine ***goingc()*** (via the network structure) to get a character to transmit. If ***goingc()*** returns the value -1, there is no data ready for transmission.

If the interrupt is *`data available`*, then the device has data to be received. The driver should take a character from the device and give it to the routine ***comec()*** (again through the network structure).

The handler must make sure that the interrupt is cleared before it returns. In some cases (the I8250 among them), the handler must check for further interrupts before returning.

Interrupt Handler Example

```
/*==============================
   C level interrupt handler.  Called from a stub.
   Returns to the interrupt stub.
*/
.
static void irhan(int netno)
{
int char;                /* character to be sent or received. */
unsigned int tport;      /* device I/O port */
unsigned char status;    /* interrupt status */
struct NET *netp;        /* pointer to network structure */
netp = &nets[netno];     /* assign a ptr to current  */
                         /* network struct via index netno */
tport = netp->port;      /* get address of the device I/O port */
while ((status = _inb(tport+IIR) & 7) != 1)
                         /* which interrupt occurred? */
{
....
switch (status)   /* switch on which interrupt occurred */
{
....
case 2:              /* Transmitter empty interrupt */
   char = netp->hw.goingc (netp);
                     /* get the char to be transmitted */
   if (char != -1)
      _outb(tport+THR, char); /* write char to device */
   else              /* no char available at present */
```

```
        _outb(tport+IER, _inb(tport+IER) & 0xd);
    break;

case 4:              /* data available from device. */
    netp->hw.comec (_inb(tport+RDR), netp);
                     /* inb reads from device */
    break;   /* comec sends to link layer */
     ....
}   /* end case */
}   /* end while */
....
} /* end irhan */
```

All references to device refer to the network controller. ***Comec()*** is accessed via a pointer to the function stored in a field of the network structure. This is pre-assigned by USNet during initialization; all you need to do is call it.

Transmit Routine

Use ***writE()*** to make a character available to the interrupt handler.

`static int writE(int conno, MESS *mess)`

conno is a connection number

mess is a message pointer

The ***writE()*** routine is called whenever your application calls the ***Nwrite()*** function, as explained in the chapter on the USNet user interface. This routine enters the message in the departure queue. This makes the message available to the interrupt handler, which sends it when a transmitter-empty interrupt occurs. If the device is not busy, it generates the transmitter-empty interrupt. It then returns and allows the interrupts to take care of the rest.

Transmit Routine Example

```
/*==============================
   Transmit routine.  Enters the message in the
   departure queue.  If link is busy just returns.
   Otherwise generates the interrupt and returns.
   Returns:
        error:  -1
        queued or started:  0
*/
static int writE(int conno,  MESS *mess)
{
int tport;        /* device I/O port */
struct NET *netp;
      /* ptr to network structure for this message */
(void)conno, (void)protoc;
       /* first two parameters are not needed here */
netp = &nets[mess->netno];
      /* get ptr to network required for this message */
tport = netp->port;                 /* assign I/O port */
mess->offset = mess->netno;
BLOCKPREE;              /* block task switching */
if (QUEUE_FULL(netp, depart))
                     /* if queue is full, process err */
     << process queue full error >>
QUEUE_IN(netp, depart, mess);
```

```
                    /* add message to departure queue */
_outb(tport+IER, _inb(tport+IER) | 2);
                   /* generate the transmit interrupt */
RESUMEPREE;             /* resume task switching */
return 0;
....;
}
```

You can see here that ***writE()*** uses the macros ***QUEUE_FULL()*** and ***QUEUE_IN()*** (discussed earlier in this chapter) to perform the queue operations. In this case, the parameters for connection number and protocol path are not used. Nevertheless, they are required for compatibility with the USNet protocol path data structures which store and call this routine.

Open Connection

Normally no action is needed. If, however, your network controller has some special needs when opening a connection, you may use ***opeN()*** to run it.

`static int opeN(int conno, int flag)`

conno is the connection number for the open connection

flag is a flag that may be used for opening connections

This routine would be run when your application makes a call to ***Nopen().*** The flag may be used for opening connections with different options relevant to some devices. For example, the WD8003 (not a character driver) allows a monitoring option for receiving or rejecting different types of network messages.

Close Connection

Like Open Connection, normally no action is needed. If, however, your network controller has some special needs when closing a connection, you may use ***closE()*** to run it.

`static void closE(int conno)`

This routine would be called when your application makes a call to ***Nclose().*** The parameters are similar to those for ***opeN().***

Configure and Start Up

This routine processes the hardware parameters, calls the optional adapter initialization, sets up the controller, and stores data into the network table.

`static int init(int netno, char *params)`

netno is the network number

params are the device-initialization parameters

The initialization parameters are the same string you configured in the network configuration table (see Chapter 4, *Configuration*). Then it calls routine ***IRinstall()*** to store the interrupt address and enable the interrupt. This routine is called from USNet when your application uses function ***Portinit().***

The initialization parameters for the I8250 are the baud rate, the I/O port address and the interrupt number. Another device might need different parameters; for instance, two separate interrupt numbers.

Configuration Start Up Example

```
/* ========================================================
Configure and start up the 8250 interface.  We process the user-level
text parameters and store the values into the net table.  We initialize
the controller.  Then we store the interrupt address and enable the
interrupt.
*/
static int init(int netno, char *params)
{
   int i1, tport;
   long l1;
   char *cp1, par[16], val[16];
   struct NET *netp;
   netp = &nets[netno];
   for (cp1=params; *cp1; )
   {
      Nsscanf(cp1, "%[^=]=%s %n", par, val, &i1);
      cp1 += i1;
      if (strcmp(par, "IRNO") == 0)
         Nsscanf(val, "%d", &netp->irno[0]);
      else if (strcmp(par, "PORT") == 0)
         Nsscanf(val, "%i", &netp->port);
      else if (strcmp(par, "CLOCK") == 0)
         Nsscanf(val, "%ld", &l1);
      else if (strcmp(par, "BAUD") == 0)
         Nsscanf(val, "%ld", &netp->bps);
   }

   i1 = netp->protoc[2]->init(netno, params);
   if (i1 < 0)
      return i1;
   tport = netp->port;
   _outb(tport+LCR, 0x80);    /* set baud reg access */
   i1 = l1 / netp->bps; /* set baud rate */
   _outb(tport+BRDH, i1>>8);
   _outb(tport+BRDL, i1);
   _outb(tport+LCR, 0x03);          /* set LCR value */
   _outb(tport+IER, 0x03);          /* set IER value */
   _outb(tport+MCR, 0x0b);          /* set MCR value */
   i1 = (int)(char)_inb(tport+LSR);
                      /* clear any line status int */

   if (i1 == -1)
     goto err2;
   (void)_inb(tport+RDR);
                     /* clear any receive interrupt */
   (void)_inb(tport+IIR);
                 /* clear any transmitter interrupt */
   (void)_inb(tport+MSR);
                /* clear any modem status interrupt */
   IRinstall(netp->irno[0], netno, irhan);
#if NTRACE >= 1
   Nprintf("I8250 IR%d P%x BPS%ld\n", netp->irno[0],tport,
                netp->bps);
#endif

   return 0;
err2:
```

```
    return NE_HWERR;
}
```

The device initialization section uses a number of ***_outb()*** and ***_inb()*** calls along with the device I/O port address *tport*. See **I8250.C** for specifics. This type of code is what will be different for your device's architecture.

Shut Down

Shut() turns off the controller and calls the optional adapter shutdown. It also calls routine ***IRrestore()*** to restore original interrupt status that existed before USNet was initialized.

```
static void shut(int netno)
```

netno is the network number

Shut() is called by USNet whenever ***Portterm()*** is called from the application.

Shut Down Example

```
/* =====================================================
Shut down the 8250 interface.  Turns off the controller.  Restores
original IRQ, mask and vector.
*/
static void shut(int netno)
{
    int tport;
    struct NET *netp;
    netp = &nets[netno];
    tport = netp->port;
    while (!(_inb(tport+LSR) & 0x40));
    _outb(tport+IER, 0);
    _outb(tport+MCR, 0);
    IRrestore(netp->irno[0]);
}
```

The ***_outb()*** and ***_inb()*** calls are device-specific commands. If you are writing your own device driver, these calls would be specific to the architecture of your network controller.

Protocol Table

USNet uses a *protocol table* to call functions specific to a given protocol or device. A `PTABLE` is defined as follows:

```
#define PTABLE const struct Ptable
                  /* typedef caused trouble */
struct Ptable {   /* protocol table, end of each module */
    char name[10];            /* name of protocol */
    int (*init)(int, char *); /* initialize */
    void (*shut)(int);  /* shut */
    int (*screen)(MESS *);    /* screen */
    int (*opeN)(int, int);    /* open */
    int (*closE)(int);        /* close */
    MESS *(*reaD)(int); /* receive */
    int (*writE)(int, MESS *); /*send */
    int Eprotoc;        /* external protocol number */
    unsigned char hdrsiz;     /* header size */
};
```

Here, you can see a PTABLE is basically a structure of pointers to functions. USNet uses this structure to call the protocol, link layer and device-specific functions when they are needed. In other words, USNet will call your device driver functions by using pointers to them stored within a PTABLE entry. Be sure you add your function names to the proper fields (see the example below). When USNet expects to call the device driver ***init()*** function, for instance, it should be the ***init()*** function which is assigned to the init field within the PTABLE, otherwise your driver will not operate properly.

PTABLE Example

```
PTABLE I8250_T = {"I8250", init, shut, 0, opeN, closE, 0, writE, 0,
MESSH_SZ};
```

The value "I8250" is the name of the driver; all others are normally fixed as you see here. The zeros are used for functions which are not needed.

In this case, ***reaD(),*** and ***screen()*** are not needed (or implemented for that matter) by the device driver. Protocol layers higher than the device driver level generally use ***screen()***, and ***reaD()*** is generally not needed since the interrupt handles reading data from the device and sending it to the link layer.

Block Drivers

A *block driver (WD8003)* receives and sends whole messages, rather than characters, between the network controller and the link layer. It neither examines nor supplies any message contents. An example of a block driver would be one which communicates with an Ethernet controller. Because whole messages are handled at a time, block drivers are implemented differently from character drivers.

Figure 10-3 shows how incoming data is handled within USNet as the data is transferred between the network controller and the application. The logic flows from top to bottom.

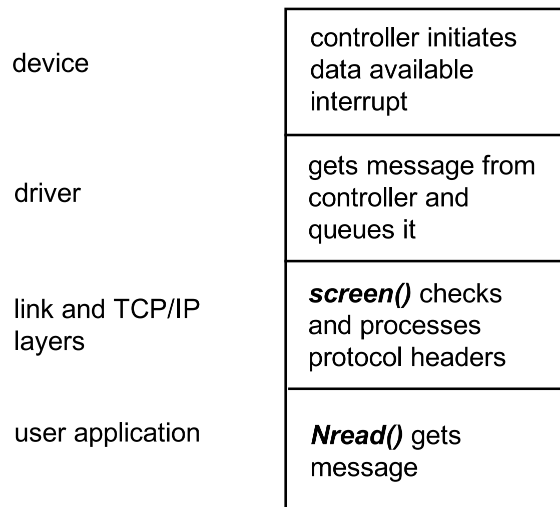


Figure 10-3: Block Driver Incoming Data

Outgoing data, as shown in Figure 10-4, is handled similarly; again, the sequence is from top to bottom.

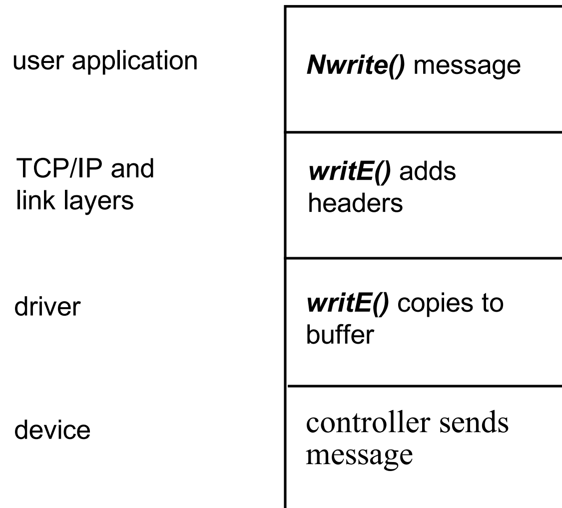


Figure 10-4: Block Driver Outgoing Data

USNet block drivers are short and simple, and will operate in any protocol stack. A typical size would be 350 lines of code. The easiest way to produce a new driver is probably by editing the existing WD8003 driver.

The following text explains the routines and data structures of a block driver. There are two ways of writing this, depending on whether the network controller provides a *transmit interrupt* or not. The transmit interrupt simply allows the sending of a message by the controller to be interrupt controlled. If one is not available, then the ***writE()*** routine must write the message directly to the controller. If it is available, the controller can generate a transmit ready interrupt and the message is then written to the device via the interrupt handler.

The following sections present each block driver function and show examples of their implementation. Use this as a guide for building your own block device driver. The code would be written differently if a transmit interrupt was present than if it was not. Examples of both are given. The transmit interrupt examples are based on the **NE2000.C** code and the non-transmit interrupt examples are based on the **WD8003.C** code. Refer to these files for specific implementations. In some cases, code from the original driver has been replaced by either four periods or `<<pseudocode>>` symbols where it was not relevant for understanding the example. As for character drivers, comments which are not part of the original code have been added to the examples for explanation. Recognize that some of the code in these examples is device dependent and will be different for your device, particularly the ***_outb()*** and ***_inb()*** calls. Study the examples for understanding the process, but don't get bogged down in the device-dependent details.

Interrupt Handler

This is a regular C function, called from the interrupt stub which is triggered when a network controller interrupt occurs.

```
static void irhan(int netno)
```

netno is used to index to the network tables and locate the network structure for this device

This first example shows an interrupt handler that does not use the transmit interrupt.

Interrupt Handler Example

```
/*=======================================================
C level interrupt handler for Ethernet.  Called from a stub, registers
are saved.  Queues the arrived message into the arrive queue of the
network block.  We pad the Ethernet header with an extra 2 bytes for 32-
bit machines.  Returns to the interrupt stub.
*/
static void irhan(int netno)
{
   Int i2;
   unsigned int len, tport;          /* device I/O port */
   unsigned char rcv,               /* interrupt status */
                 start,
                 next;
   char FAR *shp,              /* shared memory pointer */
           *bp;                     /*  buffer pointer */
   MESS *mess;          /* incoming or outgoing message */

   struct NET *netp;
          /* the network definition for this controller */
   netp = &nets[netno];           /* assign network ptr */
   tport = netp->port;          /* get port I/O address */
/* find out which interrupt occurred.  If no message available, record
the error and return.  Record errors in the network structure.*/
if ((rcv = _inb(tport+ISR)) & (MSK_PRX|MSK_RXE))
{
   if ((rcv & SMK_PRX) == 0)
   {
      if (rcv & SMK_CRC) netp->err[3]++;
                                 /* record err and ret */
      if (rcv & SMK_FAE) netp->err[4]++;
      if (rcv & SMK_FO) netp->err[5]++;
      if (rcv & SMK_MPA) netp->err[6]++;
         goto ret;
   }


/* Now, there is a message, so check that it's reasonable and assign
variables and pointers to portions of the message so it may be copied to
USNet's MESSH structure later.  THIS IS VERY DEVICE DEPENDENT.
*/
amess:
start = _inb(tport+BNRY) + 1;
if (start >= OUTPAGE)
{
    netp->err[0]++;
    goto ret;
}
shp = netp->base[0] + (SHAPAGE * start);
shp++;
next = *shp++;
if (next >= OUTPAGE)
{
    netp->err[0]++;
    goto ret;
}
len = *((short FAR *)shp);
shp += 2;
len -= 4;
if (len > MAXBUF - MESSH_SZ)
{
    netp->err[1]++;
```

```
    goto ret3;
}

/* message now looks good, be sure the arrive queue is not full */
if (QUEUE_FULL(netp, arrive))
    goto ret3;
/* Copy the message into a buffer, queue for dispatching. */
if ((mess = NgetbufIR()) == 0)  /* get new message buff */
    goto ret3;
mess->netno = netno;
                  /* assign values to fields of message */
mess->mlen = len + MESSH_SZ;
                           /*struct, see MESSH in NET.H */
mess->offset = MESSH_SZ;
bp = (char *)mess + mess->offset;
if (next && next < start)
{               /* now, copy the message into the buffer */
    i2 = (OUTPAGE - start) * SHAPAGE - 4;
    Nfarcpy(bp, shp, I2);
                   /* copy from shared memory to buffer */
    len -= i2;
    if ((int)len <= 0) goto ret3;
    bp += i2;
    shp = netp->base[0];
}
Nfarcpy(bp, shp, len);
                   /* copy from shared memory to buffer */
QUEUE_IN(netp, arrive, mess);
                          /* add message to arrive queue */
WAITNOMORE_IR(SIG_RN(netno));
                             /* trigger network task if */
                            /* multitasking is enabled. */
ret3:        ....
goto amess;
}
/* At end clear the interrupt, go back to stub. */
ret:
_outb(tport+ISR, -1);
}  /* end irhan */
```

The code for ***irhan()*** first checks to see which interrupt took place. If this is anything but "data in" it does nothing. Otherwise, and if there is a message, the handler takes it, and checks it for a reasonable format (but not for protocol). The handler allocates a buffer with the function ***getbufIR(),*** queues the message with the ***QUEUE_IN()*** macro, and kicks off the network task with the operation ***WAITNOMORE_IR().*** A number of values must be assigned to fields in the USNet `MESSH` structure which defines a message to USNet. The interrupt handler must make sure that the interrupt is cleared before it returns. In some cases (the WD8003 among them), the handler must check for further interrupts before returning. ***Nfarcpy()*** is used to copy between memory locations. It provides an architecture-independent routine which can be used to hide the details of copying, for example, in segmented architectures.

If a transmit interrupt is available, the interrupt handler may also contain code for sending the message to the device. A transmit interrupt is handled in this way:

- If the departure queue is empty, mark the transmitter free.
- If the queue is not empty, take the first element using the macro ***QUEUE_OUT***, and initiate a new transmission.

Irhan Example

```
static void irhan(int netno)
{
    int status;                     /* interrupt status */
    unsigned int tport, len;
                     /* device I/O port, message length */
    MESS *mess;                       /* message buffer */
    struct NET *netp;              /* network structure */

    /* get and clear interrupt status */
    netp = &nets[netno];    /* assign network structure */
    tport = netp->port;               /* assign I/O port */
    _outb(tport+CMDR, MSK_PG0);
    status = _inb(tport+ISR);   /* get interrupt status */
    _outb(tport+ISR, status);

    /* receive interrupt */
    if (status & (MSK_PRX+MSK_RXE))
    {
     << process receive interrupt—removed from example to
        save space >>
    }

    /* transmit interrupt, send out the next message from
       the departure queue */
    if (status & (MSK_PTX+MSK_TXE))
    {                       /* if a transmit int occurs */
lab6:
        if (QUEUE_EMPTY(netp, depart))
                                 /* is the queue empty? */
            netp->hwflags = 0;
                          /* yes, mark transmitter free */
        else                        /* if not, continue */
        {
            QUEUE_OUT(netp, depart, mess);
                    /* get message from departure queue */
            if (mess->offset != netno)
                goto lab6;
            netp->bufbas = mess;
            len = mess->mlen - MESSH_SZ;
            spt = (short *)((char *)mess + MESSH_SZ);
            if (len < ET_MINLEN) len = ET_MINLEN;
            _outb(tport+TBCR0, (len&0xff));
                              /* write message to device */
            _outb(tport+TBCR1, (len>>8));
            << much device dependent code here>>
            mess->offset = MESSH_SZ + LHDRSZ;
            if (mess->id <= bWACK)
            /* test to see if we can release the buffer */

            {              /* this may depend on an ACK */
                if (mess->id == bRELEASE)
                {
                    mess->id = bALLOC;
                    NrelbufIR(mess);
                }
            }
            else
                WAITNOMORE_IR(SIG_WN(netno));
                          /* wake up waiting tasks */
        }
    }
```

Transmit Routine

This is called from USNet when your application performs an ***Nwrite().***

`static int writeE(int conno, MESS *mess)`

The arguments are: Connection number, message address. (But if the protocol argument is zero, the first argument is the network number.)

There are two basic sending strategies: With or without a transmit interrupt. Not using one involves:

1. Copy the data to the controller.
2. Initialize and start the transmission.
3. Wait until it is done.
4. Check status: Return a negative value for error, positive for success.

Transmit Routine Example 1

```
/*=========================================================
Transmit routine.  Copies message to the output area of the shared memory, tells
the controller to send it, waits until all is clear.
Returns:
    error:    -1
    success:  length of message
*/
static int writeE(int conno, MESS *mess)

{
int tport, i1, stat, len;
         /* device I/O port, return status, message length */
char FAR *shp;                    /* shared memory address */
struct NET *netp;                     /* network structure */
i1 = protoc ? connblo[conno].netno : conno;
                             /* get current connection num */
netp = &nets[i1];            /* assign ptr to network def'n */
tport = netp->port;                 /* get port I/O address */
len = mess->mlen - MESSH_SZ;
                                /* mess len minus mess hdr */
shp = netp->base[0] + (SHAPAGE * OUTPAGE);
                /* set shared mem ptr to correct address. */
BLOCKPREE();                       /* block task switching */
Nfarcpy(shp, (char *)mess+MESSH_SZ, len);
               /* copy mess to from buff to shared memory */
             /* device dependent code to transmit message */
if(len < ET_MINLEN) len = ET_MINLEN;
_outb(tport+TBCR0, len);
_outb(tport+TBCR1, (len >> 8) );
_outb(tport+TPSR, OUTPAGE);
_outb(tport+CMDR, MSK_TXP + MSK_RD2);
while (_inb(tport+CMDR) & MSK_TXP);
stat = _inb(tport+TSR);
RESUMEPREE();               /* resume task switching */

if (stat & SMK_FU)
               /* if transmit errors exist, set errors in */
    netp->err[8]++;                  /* network structure. */
if (stat & SMK_COL)
    netp->err[9]++;
if (stat & SMK_ABT)
    netp->err[10]++;
return NE_HWERR;         /* return hardware error to USNet.*/
}
```

Not using a transmit interrupt is simpler (saves some code in the interrupt handler), but may be less efficient. Sending with the transmit interrupt means the interrupt handler does the actual transmit, allowing the ***writE()*** to return without waiting for actual transmission to occur. In this case the ***writE()*** is performed as follows:

1. Check flag for transmitter idle.
2. If yes, copy the data, and start the transmission
   If no, queue the message with the macro ***QUEUE_IN().***
3. Return zero.
4. The interrupt handler will mark the transmission complete, and start the next message from the queue.

Transmit Routine Example 2

```
/*=========================================================
Transmit routine.  If the transmitter is idle, starts the transmission and
returns.  Otherwise adds message to the departure queue; the interrupt handler
will transmit it.  The interrupt is generated by the controller.  Returns:
        error:    -1
        success:  0
*/
static int writE(int conno, MESS *mess)

{
    int tport, i1, len, netno, stat;
    short *spt;
    struct NET *netp;
    netno = protoc ? connblo[conno].netno : conno;
    netp = &nets[netno];         /* get network structure */
    tport = netp->port;    /* get device I/O port address */
    mess->offset = mess->netno;
    BLOCKPREE();                   /* block task switching */
    _outb(tport+IMR, 0x00);          /* disable interrupts */
    if (netp->hwflags)                   /* is device idle? */
    {
        QUEUE_IN(netp, depart, mess);
                 /* no, so add message to departure queue */
        RESUMEPREE();             /* resume task switching */
        stat = 0;     /* return and let interrupt send it */
    }
    else
    {
        netp->hwflags = 1;               /* set device busy */
        RESUMEPREE();             /* resume task switching */
        << write message to device and transmit >>
        stat = 1;
    }
    _outb(tport+IMR, 0x1f);           /* enable interrupts */
    return stat;
}
```

Open Connection

Normally no action is needed unless you have special actions to take when a connection is opened.

`static int opeN(int` *`conno`*`, int` *`flag`*`)`

`conno` is an index to the open network connection

`flag` is a monitoring flag

In the WD8003 driver a small amount of code was written which used the `flag` parameter to process a monitoring flag. The monitoring flag is the device understood for specifying different kinds of messages which could be accepted or rejected. Refer to **WD8003.C** if you are interested.

Close Connection

Like Open Connection, normally no action is needed unless you have special actions to take when a connection is closed.

`static void closE(int conno)`

As in the ***opeN(),*** the WD8003 driver used a small amount of code to process the monitoring flag.

Configure and Start Up

The ***init()*** routine processes the hardware parameters from the network configuration table, calls the optional adapter initialization, sets up the hardware, and stores data into the network table.

`static int init(int netno, char *params)`

netno is the network number

params are the initialization parameters

Then it calls routine ***IRinstall()*** to store the interrupt address and enable the interrupt. The initialization parameters are the same device parameters configured in the network configuration table.

The initialization parameters for the WD8003 are the I/O port address, the interrupt number and the shared buffer address. Another device might need different parameters. The example below is based on the WD8003 driver.

Configure and Start Up Example

```
/*==========================================================
Configure and start up the Ethernet interface.  We process the user-
level text parameters and store the values into the net table.  We take
the address from the Ethernet board, and set up the board.  Then we
store the interrupt address and enable the interrupt.
Returns:
   error:    -1
   success:   0
*/
static int init(int netno, char *params)
{
    int i1, ultra, tport;    /* device I/O port address */
    unsigned long bpar;
    char cc1, cc2, *cp1, par[16], val[16];
                 /* for parsing and storing parameters */
    struct NET *netp;             /* network structure */
    ....
    netp = &nets[netno];    /* assign network structure */
    for (cp1=params; *cp1; )  /* for each parameter.... */
    {
        Nsscanf(cp1, "%[^=]=%s %n", par, val, &I1);
                                /* parse the parameter */
        cp1 += i1;
```

```
        if (strcmp(par, "IRNO") == 0)
                                    /* which param is it? */
            Nsscanf(val, "%d", &netp->irno[0]);
                               /* assign params to correct*/
        else if (strcmp(par, "PORT") == 0)
                           /* fields in network structure */
            Nsscanf(val, "%i", &netp->port);
        else if (strcmp(par, "BUFFER") == 0)
        {
            Nsscanf(val, "%li", &bpar);
            netp->base[0] = mapioadd(bpar);
                    /* convert to a far ptr from 32 bit */
        }
        else return NE_PARAM; /* return parameter error */
    }
    tport = netp->port;
                  /* assign device I/O port address */
    netp->bps = 10000000;
         /* assign bits per second, this is an Ethernet */
    << setup hardware, perform device specific
       Initializations >>
    IRinstall(netp->irno[0], netno, irhan);
                        /* install interrupt handler */
    ....
    return 0;
}
```

This example does not call the adapter initialization. To see how this is done refer to the ***init()*** routine in the *Character Driver* section of this chapter.

Shut Down

This turns off the controller and calls the optional adapter shutdown.

`static void shut(int netno)`

netno is the network number and protocol path

It calls routine ***IRrestore()*** to restore the original interrupt.

Shutdown Example

```
/*=========================================================
Shut down the Ethernet interface.  Restores original IRQ, mask and
vector. Turns off the controller.
*/
static void shut(int netno)
{
    int tport;               /* device I/O port address */
    struct NET *netp;          /* ptr to network struct */
    ....
    netp = &nets[netno];/* assign ptr to network struct */
    tport = netp->port;      /* get address of I/O port */
    << write shutdown info to device using port I/O
       address >>
    IRrestore(netp->irno[0]);
                      /* restore the original interrupt */
}
```

This example does not call the adapter shutdown. To see how this is done refer to the ***shut()*** routine in the *Character Driver* section of this chapter.

Protocol Table

See the *Character Driver* section of this chapter for an explanation of the protocol table. It is identical for block drivers. The example here shows an entry for the WD8003 driver.

PTABLE Example

```
const struct Ptable WD8003_T = {"WD8003", init, shut, 0, open,
close, 0, write, 0, MESSH_SZ};
```

The field "`WD8003`" is the name of the driver; all others are normally fixed. Zeros may be used for functions which are not used.

Adapters

A PCMCIA card connects through a PCMCIA adapter as shown in the figure below. The PCMCIA standard defines a set of service calls (card services and socket services) that are used to program the adapter. As long as the software uses these services, and does not talk directly to the adapter, any PCMCIA card should work with any PCMCIA adapter.

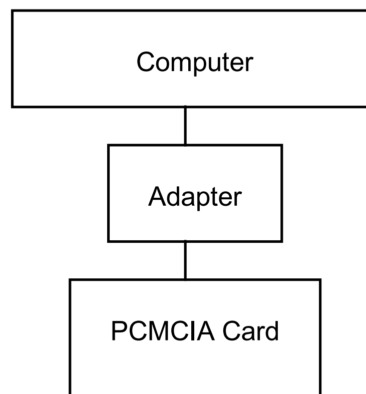


Figure 10-5: PCMCIA Adapter

USNet comes with two adapter modules: PCMCIA1 for card services, and PCMCIA2 for socket services. Use the socket services if you have a choice. However, socket services can only be used if the card services are not loaded, so, if some other part of the system needs the card services, you will have to use them with USNet also.

The PCMCIA2 module will try to map the resources required by the network interface into socket 0 of the adapter. If the network interface resides in a different socket, this can be specified by including the string "SOCKET=n" in the initialization string of the netdata entry for this interface. Here, n is the socket number.

The adapter code is formatted as a driver, but has only the initialization and the shutdown functions. The next figure below shows the relationship.

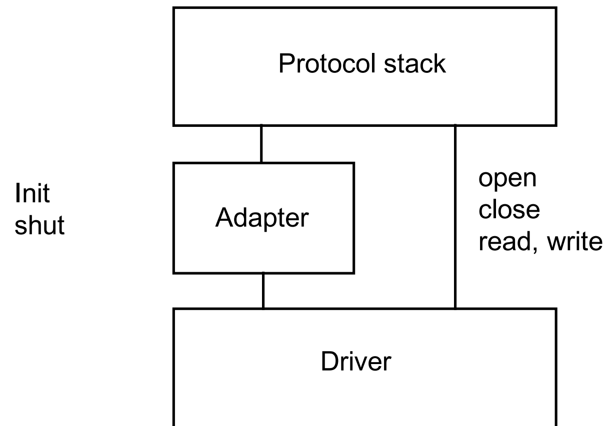


Figure 10-6: PCMCIA Adapter/Driver Relationship

In some cases, of course, the presence of the PCMCIA adapter might also affect the read and write functions. For efficiency reasons, this might require the creation of a separate PCMCIA device driver.

The PCMCIA services may not be available in all cases; for instance, if you build your own PCMCIA interface into an embedded controller. In this case the options are:

- You can create your own socket services, and use them through PCMCIA2.
- You can create a new adapter module, for instance PCMCIA3, that talks directly to your adapter.
- You can create a device driver that handles the PCMCIA interface directly.

The choice mostly depends on how general the solution needs to be.

The adapter concept is of course in no way limited to the PCMCIA adapters.

11. Performance

"There are lies, damned lies, and benchmarks."
Anonymous

Benchmarks

Running benchmarks for TCP/IP software has two important functions:

- This is needed to make sure that the protocols work exactly as they should. A test may succeed even when every packet is sent twice and every ACK times out, but it will not achieve reasonable speeds.
- Ethernet speed using 486-class processors and maximum packet size should be a million bytes per second. If it is, the product works well. If it isn't, something is wrong. (But this could be something as simple as a slow Ethernet controller.)

Published speed and timing information can help users choose the right hardware. This of course requires data about hardware of different types and speeds. If every number is over a million bytes per second, the table will look impressive (not to say a little suspicious), and be of no practical use.

Benchmarks are also used in marketing, and this is where the trouble comes in. The work that goes into improving benchmark numbers does not necessarily improve the product, or help the customer. There have been well-publicized cases of benchmark abuse, and some magazines even refuse to publish any benchmark numbers.

Network benchmarks have one big advantage over most other kinds. There can be little disagreement about what should be measured: Characters per second, and round-trip time. But even here, there's reason to beware.

Elaborate Compiler Options

You don't want benchmark results for some complicated option combinations that nobody can possibly use in an application. Ideally the benchmark should use default options, but some simple optimization is also fine.

The USNet benchmark options are given in the discussion after the data tables.

Special Benchmark Configurations

Beware of benchmarks that use only a part of TCP/IP, or tinker with options and internals. Perhaps something like turning off the SNMP instrumentation is acceptable, but any whiff of "benchmark options" is fatal.

The USNet benchmarks are run with everything: Checksums, fragmentation and reassembly, slow start, congestion control, delayed ACKs, urgent data, silly window avoidance, routing, Karn-Jacobson, and MIB II instrumentation. The serial tests for USNet actually use fragmentation, and even retransmission of fragments.

All USNet benchmarks use the product exactly as it is shipped. There are no benchmark options or benchmark versions.

Lavish Resources

Running everything in a 486 will give high numbers but little information. One number over a million is the same as another number over a million.

A large buffer pool may help the benchmark, and may not be available in an embedded system. All USNet benchmarks were run with a 22.5-kilobyte buffer pool, which probably was never more than half used.

Unusual Test Procedures

The test programs should use ordinary application functions only. The data must be actually sent out, and arrival must be verified. This is especially true for UDP: In some designs the sender may start discarding packets at high speeds.

The USNet benchmark program BENCH is very simple, and the relevant parts are reproduced at the end of this chapter. BENCH is also shipped with USNet.

Design Questions

Sending TCP/IP requires the same well-defined tasks no matter what software is running. The data must be moved and checksummed, headers must be created, interrupts handled. There are no shortcuts in any of this. If test A gives better numbers than test B, there must be clear and simple reasons for this. Until you know the reasons, comparing numbers is of little use.

The biggest, and most obvious, factor in any differences is usually the environment: Speed of hardware and so on. Normally you want to eliminate this factor, and concentrate on the rest. This of course is never easy. Seemingly little details like memory speed or amount of cache can be decisive. There can be hidden and unexpected differences in the environment. Consider as an example benchmarks A and B for Motorola 68000, A using compiler A and B compiler B.

Assume that compiler A includes the following ***memcpy()*** routine:

```
loop: mov.b     (a0),d0
      add.l     #1,a0
      mov.b     d0,(a1)
      add.l     #1,a1
      sub.l     #1,d1
      bne.b     loop
```

And compiler B does the same thing with:

```
loop: mov.b      (a0)+,(a1)+
      dbra       d1,loop
```

Benchmark B will look much better, for reasons that are not relevant in any way. (The example is not invented, although any vendor would likely avoid compiler A in benchmarks, or perhaps use a tailor-made ***memcpy()***.)

What mostly affects the efficiency of a TCP/IP product is its design. Quality of coding comes as a distant second; good code can't undo bad design, but bad code is often improved by a good compiler. The following sections give some design considerations, in rough order of importance.

Copying of Data

USNet copies the user data from the application buffer to a system buffer, and then from the system buffer to the network controller. (This is the direction for sending.) The second of these can be done by DMA, but not typically in an embedded system. The first can't be optimized out by any acceptable means. (Some USNet utilities call the protocol stack directly for speed, but we would never run any benchmarks this way.)

Any data copy besides the above two is unnecessary, and will affect performance noticeably in anything below a RISC or a 486. This might sound obvious and unnecessary to say, but actually all UNIX-based TCP/IP systems do quite a bit of internal copying.

Drivers

The single most important bit for TCP/IP speed is the "enable transmit interrupts" bit in the network controller. The effect varies and can disappear under other factors, but generally there is no way to get high speeds out of Ethernet without using transmit interrupts.

All USNet drivers use transmit interrupts. Not long ago, drivers were often written in assembly language. We don't do this, because it would be a maintenance nightmare, and because you wouldn't notice any difference in speed. In some systems the drivers are written in assembler, for speed, and the transmit interrupts are left out, for simplicity. This tradeoff is like paying a dollar for ten cents.

In some cases, when the processor is slow and the compiler bad, you might consider assembly language for SLIP character interrupts. The easiest way is to compile the driver and **slip.c** with assembly output, and then optimize the assembly code by hand.

Protocol Interfaces

The USNet protocol stack is traversed with indirect function calls, with one or two arguments. There is no queuing or task switching between the protocols.

Function Structure

USNet is in "flat C," not in the more common "nested C." A code section in "nested C" might look like:

```
extract_UDP_data(args);
check_UDP_DATA(args);
```

Whereas “flat C” looks like:

```
<the code to extract UDP data>
<the code to check UDP data>
```

In other words, unique (not general) tasks are not packaged into separate functions. They are performed, one after the other, as part of the protocol stack function, such as ***screen()*** in **udp.c** above. This is partly to eliminate the call-return overhead, but mostly to optimize all duplication and unnecessary tasks out of the code. In the above example, the “flat C” code will quickly suggest that some UDP fields can be checked right out of the header, and there is no need to move them into a temporary location.

You can see an example of this process in the MD5 digestion code in **snmpag.c**. This code was translated from the “nested C” RFC 1321 into “flat C,” and became half as small and quite a bit faster in the process.

Some recent C compilers have a global optimization option that “collapses” small functions into in-line code, and then eliminates any duplication. This is exactly the “flat C” strategy, done by hand in USNet.

There is of course a price to pay for the flat structure. The code does not necessarily look elegant, and understanding it may be hard in places. (But you don’t have to jump in and out of subroutines for details.) Some `goto` instructions are pretty much necessary. We use these sparingly, mostly when the alternative is code duplication, and we never jump back and forth in the old spaghetti style.

Benchmark Results

The speed measurements that follow have been run with the program BENCH. This is included in the USNet release, so that you can run your own speed tests. You need two hosts to run the test. Define the faster of these as `SERVER` in **BENCH.C**, then issue a “`make bench`” (after any needed configuring), then start BENCH first in the server and then in the client.

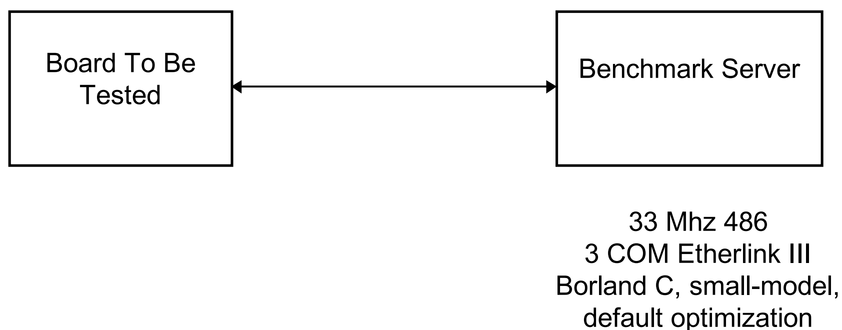


Figure 11-1: Benchmark Configuration

The speed tests presented here use the following configuration. Each test is explained in more detail after the tables. Table 11-1 presents TCP Data Rate Measurements; Table 11-2 presents UDP Data Rate Measurements; and Table 11-3 presents BSD TCP Data Rate Measurements.

In data rate measurements, the client sends data to the server at full speed. UDP has no flow control, so this will work only if the server is faster than the client. The value "`bytes`" is the amount of data in one packet.

Travel time is measured as follows: The client and the server alternate sending and receiving parcels of data. One parcel is one or more 1460-byte packets, as shown in the table. The result is calculated by dividing the total time with the number of shipped parcels. Travel time is one-half of round-trip time.

Table 11-1: TCP Data Rate Measurements

TCP		**Kbytes Per Second**			**Travel Time Ms**		
Hardware	**Mhz**	**Packet Size**			**Data Bytes**		
		512	**1024**	**1460**	**1460**	**5840**	**23360**
386SX	25	216	326	384	5	15	62
386	33	483	591	651	4	10	39
386	40	572	897	1011	3	7	26
68360	25	295	438	508	4	12	49
SPARC	40	610	908	1030	3	7	26
2 hops		219	319	359	8	18	73
ARCNET	2.5	117	135	149	11	40	152
8250	115 kb	10	10	10	190	540	2186

Table 11-2: UDP Data Rate Measurements

UDP		Kbytes Per Second			Travel Time Ms		
Hardware	Mhz	Packet Size			Data Bytes		
		512	1024	1460	1460	5840	
386SX	25	277	396	449	5	14	
386	33	581	642	663	3	9	
386	40	744	1095	1154	2	7	
68360	25	413	555	624	4	11	
SPARC	40	979	1034	1040	2	7	
2 hops		277	391	446	8	17	
ARCNET	2.5	119	136	150	11	39	

Table 11-3: BSD TCP Data Rate Measurements

BSD TCP		Kbytes Per Second			Travel Time Ms		
Hardware	Mhz	Packet Size			Data Bytes		
				1460	1460	5840	23360
386SX	25			380	5	15	63
386	33			651	4	10	39
386	40			1020	3	7	27
68360	25			505	4	12	50
SPARC	40			1030	3	7	27
2 hops				367	8	18	74
ARCNET	2.5			149	11	41	152
8250	115 kb			10	207	559	2200

Benchmark Details

The following sections provide additional detail regarding the individual benchmarks.

AMD 386, ARCNET, 40 Mhz

Compiler: Borland 16-bit version 3.1, default options

OS: DOS

Network: 2.5 Mbps ARCNET, SMC165 in client, SMC20020 in server

ARCNET is a token-passing network. This generally penalizes simple benchmarks, but helps under heavy loads.

USNet ARCNET drivers handle the ARCNET fragmentation as follows:

- Received fragments are assembled in the interrupt handler.
- The first fragment is sent by the driver write function, and later fragments are sent by the transmit interrupt handler.
- There are no intermediate data moves. The driver does not move any filler bytes.

AMD 386, 40 Mhz

Compiler: Borland 16-bit version 3.1, default options

OS: DOS

Network: NE2100 (AMD 7990)

The NE2100 uses DMA, which allows for these high speeds.

AMD 386: 115,200 bps 8250, 40 Mhz

Compiler: Borland 16-bit version 3.1, default options

OS: DOS

Network: serial port, SLIP

The real test here is of course not so much what speed we achieve, but can we keep up with well over 10,000 interrupts per second. We did, but close to the limit. (The same test will not work with PPP.) Character overruns (mostly during DOS timer interrupts) required some retransmissions. Packet sizes other than 512 bytes required fragmentation and reassembly.

UDP in this configuration would require application-level error handling.

Fujitsu SPARClite, 40 Mhz

Compiler: Microtek Research version 1.2, default options

OS: None

Network: Fujitsu MB86960

This is a very fast processor, certainly high-end for typical embedded applications. This particular board uses a 16-bit interface to the Ethernet controller, and even requires `nops` in the transfer loop. As the numbers show, not even this hurt very much.

Intel 386, 33 Mhz

Compiler: Metaware 32-bit version 3.21, default options

OS: Pharlap DOS extender

Network: NE2000 (National Semiconductor 8390)

The NE2000 uses 16-bit input/output instructions to move data, which is clearly a limiting factor. Pharlap interrupt handling takes up about 5% of the time, but of course most Pharlap users want it included.

Sixteen-bit transfer is certainly not state-of-the-art in PCs, but it is very common in embedded systems.

Intel 386SX, 25 Mhz

Compiler: Borland 16-bit version 3.1, default options

OS: DOS

Network: WD8003 (National Semiconductor 8390)

The WD8003 (Western Digital, lately SMC) uses on-board shared memory for data buffers. This method is not very common in embedded NS8390 use, but the WD8003 and the later SMC Ultra are certainly widely used controllers.

The main bottleneck here is processor speed.

Motorola 68360, 25 Mhz

Compiler: Microtek Research version 4.3K, option Ot

OS: None

Network: 68360

The 68360 uses DMA, which would suggest higher rates than what we actually achieved. The limiting factor here is clearly the one remaining data move: To or from the user buffer. (You can't eliminate this without cheating.)

Two-Hop Routing

Compiler: Borland 16-bit version 3.1, default options

OS: DOS

Network: 2 separate Ethernet networks; see Figure 11-2 below

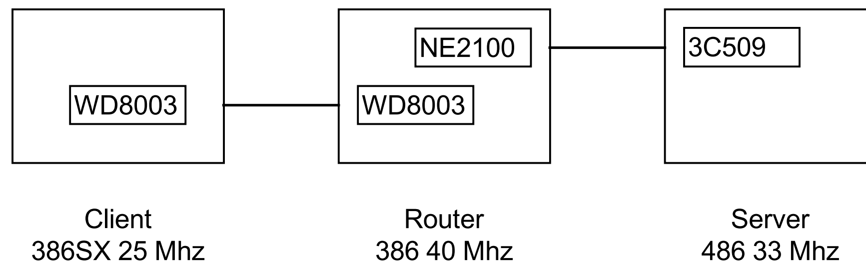


Figure 11-2: Two-Hop Routing

Benchmark Listings

The TCP and the UDP data rate tests used the following write/read loops:

```
client:  for (lc1=0; lc1<NLOOPS; lc1++)
         {
               buff2[0] = (char)lc1;
               status = Nwrite(conno, buff2, bsize);
               if (status < 0)
                     goto err2;
         }

server:  for (lc1=0; lc1<NLOOPS; lc1++)
         {
               status = Nread(conno, buff1, bsize);
               if (status != bsize)
                     goto err3;
               if (buff1[0] != (char)lc1)
                     goto err5;
         }
```

The BSD stream socket data rate test used the following read/write loops:

```
client:  for (lc1=0; lc1<NLOOPS; lc1++)
         {
               buff2[0] = (char)lc1;
               status = send(s, buff2, bsize, 0);
               if (status < 0)
                     goto err2;
         }
```

```
server:  for (lc1=0; lc1<NLOOPS; lc1++)
         {
               status = recv(s2, buff1, bsize, 0);
               if (status != bsize)
               goto err3;
               if (buff1[0] != (char)lc1)
                     goto err5;
         }
```

The TCP and UDP travel time was measured using the following pair of loops:

```
client:  for (lc1=0; lc1<lcnt; lc1++)
         {
               for (lc2=0; lc2<lentab[lc3]; lc2++)
               {
               status = Nread(conno, buff1, bsize);
               if (status != bsize)
                     goto err3;
               if (buff1[0] != (char)lc1)
                     goto err5;
               }
               for (lc2=0; lc2<lentab[lc3]; lc2++)
         {
               buff1[0] = (char)lc1;
               status = Nwrite(conno, buff1, bsize);
               if (status < 0)
                     goto err2;
               }
         }

server:  for (lc1=0; lc1<lcnt; lc1++)
         {
               for (lc2=0; lc2<lentab[lc3]; lc2++)
               {
                     buff2[0] = (char)lc1;
                     status = Nwrite(conno, buff2, bsize);
                     if (status < 0)
                           goto err2;
               }
               for (lc2=0; lc2<lentab[lc3]; lc2++)
               {
                     status = Nread(conno, buff2, bsize);
                     if (status != bsize)
                           goto err3;
                     if (buff2[0] != (char)lc1)
                           goto err5;
               }
         }
```

The BSD socket stream travel time was measured using the following pair of loops:

```
client:  for (lc1=0; lc1<lcnt; lc1++)
         {
               for (lc2=0; lc2<lentab[lc3]; lc2++)
               {
                     status = recv(s, buff1, bsize, 0);
                     if (status != bsize)
                           goto err3;
                     if (buff1[0] != (char)lc1)
                           goto err5;
               }
               for (lc2=0; lc2<lentab[lc3]; lc2++)
               {
                     buff1[0] = (char)lc1;
                     status = send(s, buff1, bsize, 0);
                     if (status < 0)
                           goto err2;
               }
         }

server:  for (lc1=0; lc1<lcnt; lc1++)
         {
               for (lc2=0; lc2<lentab[lc3]; lc2++)
               {
                     buff2[0] = (char)lc1;
                     status = send(s, buff2, bsize, 0);
                     if (status < 0)
                           goto err2;
               }
               for (lc2=0; lc2<lentab[lc3]; lc2++)
               {
                     status = recv(s, buff2, bsize, 0);
                     if (status != bsize)
                           goto err3;
                     if (buff2[0] != (char)lc1)
                           goto err5;
               }
         }
```

12. Technical Background

Overview

USNet was designed and written according to the TCP/IP protocol definitions. The *Recommended Reading* section of Chapter 1 lists books and Internet RFCs that provide more information on protocols and technical background.

USNet was designed especially for embedded environments. Of course there really is no such thing as "embedded TCP/IP;" all TCP/IP implementations must be able to talk to each other in the same way. But the environment affects design and implementation in many ways:

- USNet may have to run using very slow hardware, or very little memory.
- Connections may have high error rates.
- Hosts can be badly congested due to real-time work.
- There are often strict response-time requirements.
- USNet has to run without any operating system at all, and must easily adapt to any real-time multitasker.
- USNet must run in 8-, 16- and 32-bit architectures, either big-endian or little-endian.
- There are no "typical" traffic patterns, no "normal" applications such as the Internet FTP and TELNET.

The following text discusses some related technical subjects, especially from the embedded viewpoint.

TCP Retransmission

When the acknowledgment doesn't arrive, TCP must resend the data. The procedure is as follows:

1. When the timeout `txtout` expires, resend.
2. If no ACK in `txtout`, resend a second time.
3. If no ACK in 2*`txtout`, resend a third time.
4. Keep trying, doubling the timeout until it exceeds a preset value, 30 seconds in USNet.

USNet uses the Jacobson-Karn method to calculate the timeout value as an adjusted average of measured round-trip times. No measurement is done for retransmitted messages. When more than one retry is needed, the timeout is doubled. (This is a slightly simplified explanation.)

The Jacobson-Karn method works well; it has little trouble with variable and completely unknown round-trip times, or modest error rates. However, there are a couple of pitfalls in the implementation:

- Unless the timing granularity is much smaller than the round-trip times, it must be considered in the calculations. The result must be rounded up to at least one clock tick.
- No measurement should be done for any messages that may end up in the receiver's "future message" queue. These messages are normally not resent, but they must be treated as if they were. Ignoring this rule will make Jacobson-Karn unable to handle connections with even modest error rates.

Continuous sharp variation in round-trip time (unfortunately not rare in embedded systems) can cause trouble for Jacobson-Karn. In local networks a solution might be to use a constant timeout value, but this really isn't TCP any more. Some implementations use a fairly large minimum timeout value, to avoid unnecessary retransmission. This is not suitable as a general solution.

Jacobson-Karn will not work well for a very bad connection, where a packet often has to be retransmitted twice. Neither would anything else. The only good way to handle these connections is with error detection and correction at the link level. Someone unnamed has said that TCP/IP will work with two paper cups and a string. This claim may seem a bit misleading to people who have actually worked with marginal connections. TCP/IP will work "over a string," but only using some link-level protocol (such as HDLC) that is not part of the TCP/IP protocol family. SLIP and PPP do not contain any error handling.

Why does the method ignore packets that were resent once? You would quickly see why by commenting out this check and running TCP/IP over a fast serial line. Every now and then a packet would arrive bad (receiver overrun typically) and be resent. Jacobson-Karn would keep doubling the timeout value, but this would have no effect on the error rate, so the timeout would end up at some maximum value, and the throughput would be horrible.

The one retry rule is of course completely artificial. There is no particular reason to believe that one retry means line error, two means timeout value was too short. It might even seem that the rule has its dangers. What about this situation:

- Client has a 50 millisecond timeout.
- Server needs 75 milliseconds to ACK.
- All ACKs arrive a little late, so all transmissions need one retry.
- Timeout value is never updated, so nothing changes, and everything is sent twice.

Fortunately it turns out that this situation is not stable unless the TCP window only allows for one packet. Never configure TCP/IP so that the TCP window is shorter than twice the maximum packet.

It might seem that TCP could try to differentiate between lost packets and late ACKs by keeping track of duplicate ACKs. This has been tried, and the results were not encouraging. In any case the TCP standard does not contain anything like this.

Sliding Window

TCP flow control uses a sliding window. Each ACK can be interpreted as "send window bytes more data." This does not mean "more than you already sent," it means "after the data hereby acknowledged." As data is received and consumed, the host keeps extending the window at its own pace.

This simple window concept can be used in different ways Sometimes the packet exchange will look like Figure 12-1.

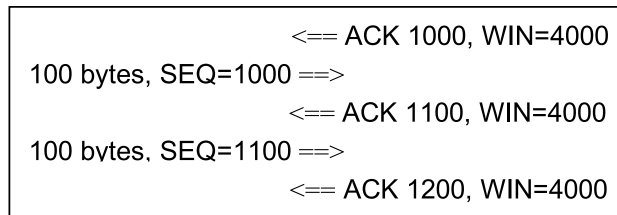


Figure 12-1: Packet Exchange

People sometimes think that there is something wrong here: The client keeps sending data, so how can the window stay the same? But this is what happens if the application in the server has received the data by the time the ACK is sent. Using delayed ACK (see Figure 12-2) can cause this pattern.

Here is another common pattern:

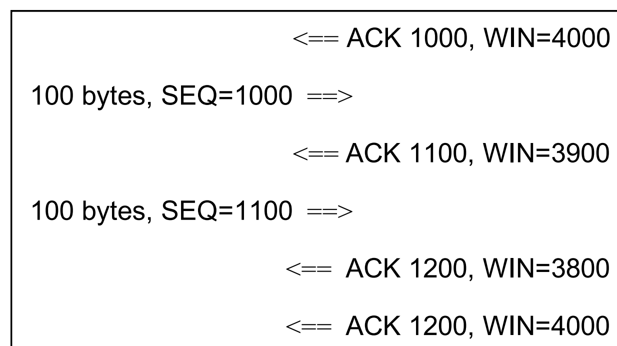


Figure 12-2: Delayed ACK

This suggests that the server ACK'd the two packets immediately, before the application had a chance to read. (Doing this systematically is not acceptable in TCP.) When the application takes the data, TCP sends a window update.

Figure 12-3 shows a situation where the window is exhausted:

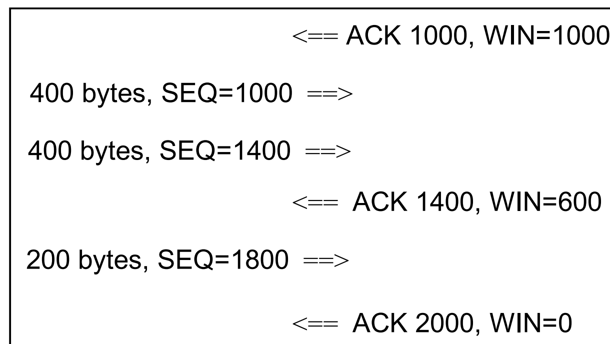


Figure 12-3: Exhausted Window

This sequence might seem strange at first. We have sent 800 bytes into a 1000-byte window, so how come the window is 600 bytes and not 200 bytes? Of course there is no real mystery here. The window is fixed by the ACK number, not by what we have sent, or rather what we think we have sent.

If the application in this last example will not read any data, the window will stay zero, and the sender will have to stop sending. This is not an error situation in any way (though people sometimes think so), and when the reading resumes, the data will flow again. There is no time limit on the pause.

The sender is required to keep probing for window size in some way, because an ACK that carries a window update might get lost, and also to find out if the receiver is still there. Traditionally this has been done with a packet that contains one data byte. It would be simpler, cleaner and more efficient to use an ACK without any data at all, but for some reason this is not done.

USNet uses a 1-byte data probe as described in RFC 1122, and as used by the UNIX-based implementations.

TCP Delayed ACK

USNet follows all the rules for delayed ACKs, as described in RFC 1122. An ACK is delayed until one of the following occurs:

- Data is going out.
- More than one full-size segment of data can be ACK'd. (This is normally 1460 bytes in Ethernet.)
- At least 3 packets can be ACK'd.
- The window grows by at least one full-size segment (by 1460 bytes in normal Ethernet).
- A time limit (defined in **tcp.c** source as 200 ms) expires.

Delaying ACKs is very important on busy serial connections that use short packets.

Congestion Control

The sliding-window flow-control method can run into difficulties when the data is routed. The remote host does not, and cannot, consider the abilities of the routers when it assigns the window size. No doubt the original design assumed that whoever takes on the job of routing should have the resources for that.

RFC 1122 describes a two-part procedure for considering the routing bandwidth. Slow start requires the sender to start up gradually, and keep speeding up as the ACKs arrive. Congestion avoidance defines a way to limit the actually-used window to a value that does not cause difficulties. Originally all this was meant for routed connections only.

Still later another problem started cropping up. What if the 4096 bytes of data is sent as short packets at full speed? If the remote host receives 100 40-byte packets at full speed, nobody should be surprised if it throws away half of them. Of course packets should not be sent like this in TCP, but it is perfectly possible to do so, and with faster and faster hardware these packet bursts can become deadly weapons.

In response to this second problem, slow start and congestion avoidance are now generally used for all connections, though RFC 1122 only requires them for non-local connections.

Slow start and congestion avoidance are absolutely necessary in embedded environments, much more so than in workstation networks. USNet implements these according to RFC 1122, for all connections. It uses actual packet counts in this, not estimates based on sequence numbers.

There is a pitfall in the implementation of a slow-start algorithm. Assume the following situation:

- Client is in slow start, sends one packet and waits for the ACK.
- Client retransmission timeout is 150 ms.
- Server uses delayed ACK, with a 200 ms delay.

The client sends a message, times out, and resends. The ACK arrives. If the client now re-enters slow start (because a retransmission was needed), the circle will never be broken. The client will be in permanent slow start, and the server in permanent ACK delay. The solution to this problem is fortunately simple. A host should enter slow start only if it also recalculates the timeout value.

Silly Window Syndrome

The silly window syndrome consists of the receiver offering very small windows, and the sender sending very short packets. Nobody would have heard of the silly window syndrome if an early TELNET had not managed to combine several unlikely design choices to produce it, and if it didn't have such a catchy name.

USNet does not suffer from the silly window syndrome because of the following design features:

- ACKs are delayed to avoid small windows.
- Stream sockets use Nagle's algorithm to combine short send requests.
- TCP send will wait for a larger window if the whole packet does not fit.

ARP Caching

The ARP table (usually called the ARP cache) gives the Ethernet address (or more generally the media address) for the known local hosts. The cache is built by the ARP protocol.

A local IP-to-Ethernet table saves time, but like all duplicate information, it presents a maintenance problem. Assume that the table now says:

```
192.9.200.3  ==  002324252627
```

Also assume that the computer referred to here suffers an accident. The Ethernet card is quickly replaced. The new Ethernet address is 002324252628. Until the ARP table is updated, the other host can't send anything to 192.9.200.3.

USNet handles ARP updating in the following way:

- An entry (except for a statically configured entry) times out in 60 seconds, counted from the time a packet was last received from this host.
- Any received ARP request is used to update the entry, even if it is still live.

In an embedded environment, even 60 seconds can be an eternity. (It is generally short enough to allow for a TCP retry to succeed, though.) We can't very well make this constant much smaller, because the ARP load might become disturbing. But the host that changed its address can help itself, by sending an ARP request to the network (perhaps to itself) when it goes on-line.

Many TCP/IP systems even a few years ago would not accept an ARP update for a valid ARP entry. The purpose of this was no doubt to keep away hosts that used somebody else's IP address. On embedded networks, this concern should not be overly important.

A. Terminology

CHAP	Challenge Handshake Authentication Protocol. A user and password authentication method used by a PPP connection. Both the user name and password are encrypted.
compiler.mak	Compiler definition makefile located in **config\<*cpu*>\<*compiler*>** subdirectory, where <***cpu***> is the name of the target processor, and <***compiler***> is the name of the compiler.
config.mak	Configuration makefile located in the USNet install directory. This file defines the target processor, toolchain, RTOS, and other relevant application parameters for use when the USNet library is built.
DHCP	Dynamic Host Configuration Protocol. The protocol used by a host to request an IP address from a DHCP server based on the host's name.
DNS	Domain Name Server. This is a machine which tells remote hosts what IP address corresponds to a host name and vice versa.
DPI	Dynamic Protocol Interface. This is USNet's primary interface using stream I/O-like function calls.
FTP	File Transport Protocol. FTP is used to transfer files using TCP connections through port 21 on an FTP server.
Host	**A computer on the network.**
Link Layer	The protocol used over the physical connection between two hosts. USNet supports ARCNET, Ethernet, PPP, and SLIP. The link layer is defined in **netconf.c**.
Opus Make	Opus Make is a powerful make utility distributed with USNet. USNet libraries and test programs are built with makefiles written for use by Opus Make. See the Opus Make manual for further details.
Passive Open	A passive open means a host attempts to open a connection to any remote host wishing to establish a connection. The host will remain in the ***Nopen()*** function indefinitely until a connection is established.
RTOS	Real Time Operating System. Examples: SuperTask!® and SMX®.
TCP	Transmission Control Protocol. TCP is a reliable protocol that insures data is actually received at the remote site.

TFTP	Trivial File Transport Protocol. TFTP is used to transfer files via a UDP connection through port 69 on a TFTP server.
UDP	User Datagram Protocol. UDP is a protocol designed to send data packets to the remote site without guaranteeing reception.

B. Trace Output

Overview

The trace output is used as a diagnostic aid during USNet integration testing and application development. Most hardware platforms support tracing through an RS232 serial port. Trace output is supported with the ***Nputchr()*** function. This function is platform-dependent and may have to be modified to support your platform. See the *Display and Keyboard Support* section of Chapter 9, *Porting*.

Since the USNet trace output covers details of the TCP/IP stack protocols, it is beyond the scope of this manual to provide detailed information here. We can, however, give you a brief definition of the trace codes. You can do a ***grep*** or ***search*** on ***NTRACE*** in the protocol modules such as **tcp.c** or **udp.c** for more information.

Displaying Trace Data

Within the USNet-supplied makefile, a macro is defined which allows you to control the level of trace data output. To change this, modify the ***TRACE_DEBUG*** macro. You may use any value from 0 through 9, with 0 representing no trace and increasing numbers representing an increasing amount of trace output. At installation ***TRACE_DEBUG*** is set to 3 and should not have to be changed. The NTRACE macro is set to the value of TRACE_DEBUG automatically by config.h, which is generated by the make.

NTRACE also controls the level of error reporting. Doing a ***grep*** or ***search*** on ***NTRACE*** will show what levels of error reporting are used.

Below is a TCP trace fragment captured from a target while executing EMTEST. The fields and their contents have meanings defined, and can be used to characterize networking anomalies. The Trace Fields are defined on the next page. Looking at the first highlighted row, the fields are defined as:

```
SC 59869865 C1/1b9c ST2 DL0 W5840/15340 SQ2f4d1202 AK39310a6 10
SC 59869865 C1/1b9c ST3 DL0 W5840/16060 SQ2f4d1202 AK39310a6 11
TX 59869865 C1/1b9c ST5 DL0 W5840/16060 SQ39310a6 AK2f4d1203 10
TX 59869865 C0/1a9c ST1 DL6 W5840/16060 SQ39189a2 AK2f4c18de 18
SC 59869865 C0/1a9c ST1 DL24 W5840/16054 SQ2f4c18de AK39189a8 18
RX 59869865 C0/1a9c ST1 DL24 SQ2f4c18de AK39189a8 18
```

Appendix B

Field	Definition
Field 1	Identifies the type of protocol operation. TCP and UDP have their own unique operation codes. The example above is for TCP, and its codes are defined as follows: TCP CODES: FQ - future queue OP - open connection CL - close connection SC - screen TX - transmit RETX - retransmit RX - receive UDP CODES: UO - open UC - close US - screen UR - read UW - write
Field 2	The timestamp for each transaction. This time snapshot is taken from the clock state. This is the clock defined in the module **clock.c**.
Field 3	The connection number/port number.
Field 4	The TCP state.
Field 5	The net data length. This does not include any headers.
Field 6	The local (self)/remote window sizes.
Field 7	The sequence number. This number is randomly generated to comply with RFC recommendations.
Field 8	The acknowledge number.
Field 9	The TCP flag.

C. RTOS-Specific Information

MTOS

About the Multitasker:

Vendor Industrial Programming Inc.

Targets m68k, 80x86 real mode, x86 protected mode, R3000, Intel 80860

Compilers Several

Installation

The following instructions assume DOS. The same general rules apply even if you work under UNIX. How the source is actually brought into a UNIX system will vary from case to case, but should normally be fairly simple.

1. Create a test directory, called here **\test**, and under it directories **mtos**, **usnet** and **usnetmt**.
2. Mount the MTOS disk and install MTOS into directory **mtos**:

```
copy a:*.* \test\mtos
```

3. Install USNet without multitasking.
4. Install USNet with multitasking.

These MTOS makefiles are included in USNet:

i386
metaware

Configuring

Configure MTOS and USNet for your actual environment and target. This will generally require editing of the makefile, and any hardware support files as needed. See the MTOS and USNet documentation for details.

The USNet makefile for MTOS, as distributed, is compatible with MTOS. If you need to create a new makefile, check the MTOS documentation for rules on the compilation options. USNet will in general accept any reasonable options.

Testing

The procedure given below includes a considerable amount of testing. If you feel sure of your ground, you can skip some tests. But don't skip them to save time: It is truly amazing how many hours a saved minute can cost.

USNet without RTOS

Compile, load and run the following test programs, in this order:

LTEST This is a loopback test that runs in the actual target but performs no actual network I/O.

EMTEST This runs against any standard FTP server, and tests the actual network.

MTTEST This is the non-multitasking version of the multitasking test. It needs a server, either another MTTEST running in a PC, or UXSERV running in a UNIX system.

USNet with RTOS

Compile, load and run MTTEST. Use the same server as in the previous step. As a prelude to the actual network test, MTTEST checks that multitasking operates properly. This tests the following:

- Checks that all needed event flags work properly
- Checks that macro ***RUNTASK()*** works
- Checks that macro ***WAITFOR()*** waits when the condition is false, exits when the condition is true, and wakes up for ***WAITNOMORE()***
- Checks that ***WAITFOR()*** times out properly
- Checks that ***WAITFOR()*** sets the flag correctly according to the condition
- Checks that the macros ***BLOCKPREE()*** and ***RESUMEPREE()*** work properly

Any errors here cause MTTEST to show a message and quit. If this happens, contact us for support.

If MTTEST appears reliable, change the *`TRACE_DEBUG`* argument in config.mak to 1 (both in the client and in the server), and run an overnight test.

For an extremely tough test, use PITEST to generate additional load in the target running the MTTEST client. PITEST program will send ping requests at an adjustable rate.

Creating Applications

The user interface for USNet is exactly the same with and without multitasking. There are of course some functional differences:

- The makefile is a little different. The multitasker files are given in the link commands. The C flags include the multitasker header files.
- MTOS is a pre-emptive multitasker. The ***YIELD()*** macro is normally not needed in user code. Server tasks can execute concurrently, not just one at a time. MTOS is initialized with various system-level calls; no special initialization is needed at the start of a user program.

Features Used

WAITFOR() and ***WAITNOMORE()*** use event flags. These are created in routine ***NMTinit()*** in **multi.c.** The keys used are "NET0", "NET1" and so on, 16 flags for each key. The total number of events needed is `2*(NCONNS+NNETS+1)`.

Default priorities are defined as follows (high number = high priority, up to 255):

```
SERV_PRIOR    100
CLIENT_PRIOR  100
NET_PRIOR     110
```

The macro ***YIELD()*** is defined as `pause(NXTICK)`, which will delay to the next clock tick. USNet does not call ***YIELD()*** automatically in pre-emptive multitasking.

Task type is defined as:

```
#define TASKFUNCTION void
```

RUNTASK() is a subroutine in **multi.c**. It builds the `TCD` structure and calls ***crtsk()***. Task keys have values "`NET1`", "`NET2`" and so on. Task "`NET0`" is ***relaytask()***, created in ***NMTinit()***.

Macro ***WAITNOMORE()*** uses function ***srsefg()***. ***WAITNOMORE_IR()*** is empty; see below about signaling from interrupts.

Macro ***WAITFOR()*** uses function ***waiefg()***, called from inside subroutine ***Nwtems()***. This function checks separately for the delay values 0 and 0xffffffff.

BLOCKPREE() and ***RESUMEPREE()*** are subroutines that disable and enable interrupts.

Function ***Nclock()*** gets the time in milliseconds from function ***getime()***. Function ***Nclkinit()*** sets *`clocks_per_sec`* to 1000.

Signaling from Interrupts

Signaling from interrupts is done indirectly, from task ***relaytask()***, which is run immediately after the interrupt.

Task ***relaytask()*** is created in the initialization function ***NMTinit()***. Function ***IRinstall()*** uses a call to ***contsk()*** to tell MTOS that ***relaytask()*** is to be run after a network interrupt.

Relaytask() calls ***srsefg()*** to set the appropriate event flag.

MultiTask!

About the Multitasker:

Vendor	United States Software
Targets	8051, 80x96, 80x86 real mode, x86 protected mode, 80960, Z180, 68k, 68HC11, 68HC16, MIPS, SPARC
Compilers	Many

Installation

The following instructions assume DOS. The same general rules apply even if you work under UNIX. How the source is actually brought into a UNIX system will vary from case to case, but should normally be fairly simple.

1. Create test directory, called here **test**, and under it directories **mt**, **usnet** and **usnetmt**.
2. Install MT into directory **mt**.
3. Install USNet without multitasking.

4. Install USNet with multitasking.

Configuring

Configure MT and USNet for your actual environment and target. This will generally require editing of the makefile, and any hardware support files as needed. See the MT and USNet documentation for details.

The USNet makefile for MT, as distributed, uses the same compilation options as MT. If you need to make changes to these, check that the two makefiles remain compatible.

The user-level configuration file in MT is called **depends.h**. All USNet test programs will run with the default MT configuration.

Testing

The procedure given below includes a considerable amount of testing. If you feel sure of your ground, you can skip some. But don't skip them to save time: It is truly amazing how many hours a saved minute can cost.

Compiling and Testing the RTOS

1. Go to directory **test/mt**. Compile and link **coretest**:

```
omake coretest
```

2. Then load the test into the target board and run it.

Testing USNet without RTOS

Compile, load, and run the following test programs, in this order:

LTEST This is a loopback test that runs in the actual target but performs no actual network I/O.

EMTEST This runs against any standard FTP server, and tests the actual network.

MTTEST This is the non-multitasking version of the multitasking test. It needs a server, either another MTTEST running in a PC, or UXSERV running in a UNIX system.

Testing USNet with RTOS

Compile, load and run MTTEST. Use the same server as in the previous step. As a prelude to the actual network test, MTTEST checks that multitasking operates properly. This tests the following:

- Checks that the timer ticks convert into milliseconds correctly.
- Checks that macro ***RUNTASK()*** works.
- Checks that macro ***WAITFOR()*** waits when the condition is false, exits when the condition is true, and wakes up for ***WAITNOMORE()***.
- Checks that ***WAITFOR()*** times out properly.
- Checks that ***WAITFOR()*** sets the flag correctly according to the condition.
- Checks that the macros ***BLOCKPREE()*** and ***RESUMEPREE()*** work properly.

Any errors here cause MTTEST to show a message and quit. If this happens, contact us for support.

If MTTEST appears reliable, change the *TRACE_DEBUG* argument in config.mak to 1 (both in the client and in the server), and run an overnight test.

For an extremely tough test, use PITEST to generate additional load in the target running the MTTEST client. PITEST program will send ping requests at an adjustable rate.

Creating Applications

The user interface for USNet is exactly the same, whether multitasking is used or not. There are of course some functional differences:

- The makefile is a little different. The multitasker library is given in the link commands. The C flags include the multitasker header files.
- MT is a preemptive multitasker. The ***YIELD()*** macro is normally not needed in user code. Server tasks can execute concurrently, not just one at a time.
- The main entry must contain code to initialize the multitasking system. Below is how this is done in MTTEST. For further details see the multitasker manual.

```
uint32 free_memory[(MEM_ALLOCATION+7)/4];

Mtinitialize();  /* initialize MT! */
usrclk_init();    /* Initializeuser clock */
MTmeminit(&free_memory[1], MEM_ALLOCATION);
status1 = RUNTASK(task1, CLIENT_PRIOR);
status2 = RUNTASK(task2, CLIENT_PRIOR);
MTstart();       /* begin multitasking */
```

The user tasks are now running, a call to ***MTterminate()*** returns here.

```
usrclk_term();   /* stop user clock */
return 0;
```

Features Used

WAITFOR() and ***WAITNOMORE()*** use events. The first event is 2 (numbers 0 and 1 are reserved by the multitasker itself). Total number of events needed is 2*(NCONNS+NNETS+1).

Default priorities are defined as:

```
SERV_PRIOR    100
CLIENT_PRIOR  100
NET_PRIOR     110
```

(For MT, high number = high priority, up to 255.)

The macro ***YIELD()*** is defined as ***scdtsk()***. This will give control to the next task with the same priority.

Task type is defined as:

```
#define TASKFUNCTION void       x86 protected mode
#define TASKFUNCTION void FAR   others
```

Macro ***RUNTASK()*** uses function ***runtsk()***.

Macro ***WAITNOMORE()*** uses function ***setevt()***. ***WAITNOMORE_IR()*** uses ***MTqcmd_c()***, that is the command-queuing function.

Macro ***WAITFOR()*** uses function ***wteset()*** if the timeout value is not zero, **chkevt**() if it is zero. (In MT, timeout 0 means wait forever.)

Macro ***BLOCKPREE()*** is translated as ***mask_ints()***, ***RESUMEPREE()*** as ***unmask_ints()***.

Function ***Nclock()*** is mapped into function ***get_sys_time()***. ***Nclkinit()*** stores the multitasker value `CLOCKHZ` as the clock frequency.

Hitachi HI-SH7

About the Multitasker:

Vendor	Hitachi
Targets	Hitachi SH-7000*
Compilers	Hitachi SH Series C

Installation

When running with the HI-SH7* operating system, USNet needs to include a number of source code files that are distributed with the operating system. Some of these files are not modified from the standard HI-SH7 distribution, and others have been adapted for use with USNet.

1. First install the Hitachi HI-SH7 operating system and compiler. Typically this will create a file structure as follows:

   ASM
   HI-SH7
   HI-SH7/sh7604
   SHC

2. Create a test directory, called here **\test**, and under it directories **usnet** and **usnetmt**.

3. Install USNet without multitasking.

4. Install USNet with multitasking.

Configuring

Once you have installed the necessary files on your host system, you will need to specify configuration information. Configure USNet for your actual environment and target. This will generally require editing of the makefile, and any hardware support files as needed. See Chapter 4, *Configuration* for details.

Testing

The procedure given below includes a considerable amount of testing. If you feel sure of your ground, you can skip some. But don't skip them to save time: It is truly amazing how many hours a saved minute can cost.

Compiling and Testing the RTOS

Follow the procedure supplied with HI-SH7 to ensure that the multitasker will run correctly in your development environment.

USNet without RTOS

Compile, load and run the following test programs, in this order:

LTEST This is a loopback test that runs in the actual target but performs no actual network I/O.

EMTEST This runs against any standard FTP server, and tests the actual network.

MTTEST This is the non-multitasking version of the multitasking test. It needs a server, either another MTTEST running in a PC, or UXSERV running in a UNIX system.

USNet with RTOS

Compile, load and run MTTEST. Use the same server as in the previous step. As a prelude to the actual network test, MTTEST checks that multitasking operates properly. This tests the following:

- Checks that the timer ticks convert into milliseconds correctly.
- Checks that macro ***RUNTASK()*** works.
- Checks that macro ***WAITFOR()*** waits when the condition is false, exits when the condition is true, and wakes up for ***WAITNOMORE()***.
- Checks that ***WAITFOR()*** times out properly.
- Checks that ***WAITFOR()*** sets the flag correctly according to the condition.
- Checks that the macros ***BLOCKPREE()*** and ***RESUMEPREE()*** work properly.

Any errors here cause MTTEST to show a message and quit. If this happens, contact us for support.

If MTTEST appears reliable, change the *TRACE_DEBUG* argument in config.mak to 1 (both in the client and in the server), and run an overnight test.

For an extremely tough test, use PITEST to generate additional load in the target running the MTTEST client. PITEST program will send ping requests at an adjustable rate.

Creating Applications

The user interface for USNet is exactly the same, whether multitasking is used or not. There are of course some functional differences:

- The makefile includes the multitasker support files and library in the default rules.
- HI-SH7 is a preemptive multitasker. The ***YIELD()*** macro is normally not needed in user code. Server tasks can execute concurrently, not just one at a time.
- The system file **hisuptbl.c** is set up so that ***main()*** is the initial task scheduled for execution, with a priority of 1. Although a priority of 1 works for the sample MTTEST program which launches additional tasks and exits, you may want to adjust this to a lower priority if the ***main()*** function continues to execute. Since the Network Task is set to execute at priority 2, any task using network functions should have a lower priority.

At the start of ***main()***, all of the multitasker initialization has been performed. Here is an example:

```
TASKFUNCTION main(void)
{
    exd_tsk();
}
```

Features Used

The event functions provided under uITRON are not used directly. However, the ***wai_tsk()*** and ***rel_wai()*** functions serve in a similar manner to efficiently wake tasks that are waiting for certain conditions.

Default priorities are defined as follows (for uITRON, high number = low priority):

```
SERV_PRIOR    4
CLIENT_PRIOR  4
NET_PRIOR     2
```

The macro ***YIELD()*** is defined as ***wai_tsk(1)***. This will put the task into a wait state for one unit of time, allowing other tasks to execute.

Task type is defined as:

```
#define TASKFUNCTION TASK
```

Macro ***RUNTASK()*** is implemented as the ***RunTask()*** function defined in **multi.c**. This function creates and starts a task using the ***cre_tsk()*** and ***sta_tsk()*** functions. The ID of the new task is returned. If no more task IDs are available, then an error value `E_NOEXS` is returned. Newly created task IDs range from 1 to `MAX_TSK_ID` which is defined in **mtmacro.h**.

The ***KILLTASK()*** macro is implemented with the ***exd_tsk()*** function.

The ***WAITNOMORE()*** and ***WAITNOMORE_IR()*** macros are used to wake up a task that has entered the wait state while waiting for a condition. The `TaskId[]` array is used to correlate each USNet signal with a task that may be waiting for the signal. When a ***WAITNOMORE*** macro is called, the corresponding task is woken using the ***rel_wai()*** or ***irel_wai()*** function.

The ***WAITFOR()*** macro is implemented with code that checks for the specified condition and timeout, stores the task's ID in the `TaskId[]` element for the appropriate signal, and calls ***wai_tsk()***. If the desired event occurs and the task is in the wait state, then it will be woken, and the macro will complete showing that the condition was satisfied.

If the condition does not occur, the check for a timeout eventually evaluates as true, and the macro returns with the flag showing that a timeout occurred.

Macro ***BLOCKPREE()*** is translated as ***Ndisable()***, ***RESUMEPREE()*** as ***Nenable()***. These are assembly language functions that disable and enable interrupts.

Function ***Nclock()*** is mapped into function ***get_time()***. ***Nclkinit()*** stores the multitasker value 100 as the clock frequency.

VRTX

About the Multitasker:

Vendor	Microtec Research
Targets	80x86 real mode, x86 protected mode, 68k SPARC, PowerPC
Compilers	Several

Installation

The following instructions assume DOS. The same general rules apply even if you work under UNIX. How the source is actually brought into a UNIX system will vary from case to case, but should normally be fairly simple.

1. Create a test directory, called here **\test**, and under it directories **usnet** and **usnetmt**.
2. Install USNet without multitasking.
3. Install USNet with multitasking.

These VRTX makefiles are included in USNet:

i386
metaware

Configuring

Configure VRTX and USNet for your actual environment and target. This will generally require editing of the makefile, and any hardware support files as needed. See the VRTX and USNet documentation for details.

The USNet makefile for MT, as distributed, is compatible with VRTX. If you create a new makefile, check the VRTX documentation for rules on compilation options. USNet will in general accept any reasonable options.

Testing

The procedure given below includes a considerable amount of testing. If you feel sure of your ground, you can skip some tests. But don't skip them to save time: It is truly amazing how many hours a saved minute can cost.

Compile and Test RTOS

Follow the VRTX procedures to ensure that the multitasker will run correctly in your environment.

USNet without RTOS

Compile, load, and run the following test programs, in this order:

LTEST This is a loopback test that runs in the actual target but performs no actual network I/O.

EMTEST This runs against any standard FTP server, and tests the actual network.

MTTEST This is the non-multitasking version of the multitasking test. It needs a server, either another MTTEST running in a PC, or UXSERV running in a UNIX system.

USNet with RTOS

Compile, load and run MTTEST. Use the same server as in the previous step. As a prelude to the actual network test, MTTEST checks that multitasking operates properly. This tests the following:

- Checks that macro ***RUNTASK()*** works.
- Checks that macro ***WAITFOR()*** waits when the condition is false, exits when the condition is true, and wakes up for ***WAITNOMORE()***.
- Checks that ***WAITFOR()*** times out properly.
- Checks that ***WAITFOR()*** sets the flag correctly according to the condition.
- Checks that the macros ***BLOCKPREE()*** and ***RESUMEPREE()*** work properly.

Any errors here cause MTTEST to show a message and quit. If this happens, contact us for support.

If MTTEST appears reliable, change the *TRACE_DEBUG* argument in config.mak to 1 (both in the client and in the server), and run an overnight test.

For an extremely tough test, use PITEST to generate additional load in the target running the MTTEST client. PITEST program will send ping requests at an adjustable rate.

Creating Applications

The user interface for USNet is exactly the same, whether multitasking is used or not. There are of course some functional differences:

- The makefile is a little different. The multitasker libraries are given in the link commands. The C flags include the multitasker header files.
- VRTX is a pre-emptive multitasker. The ***YIELD()*** macro is not needed in user code. Server tasks can execute concurrently, not just one at a time.

VRTX is initialized with various system-level calls; no special inititalization is needed at the start of the user program.

Features Used

WAITFOR() and ***WAITNOMORE()*** use events. These come in groups of 32 events each. The needed events are created in the initialization routine ***NMTinit()***. Total number of events needed is `2*(NCONNS+NNETS+1)`.

Default priorities are defined as follows (low number = high priority):

```
SERV_PRIOR    5
CLIENT_PRIOR  5
NET_PRIOR     3
```

The macro ***YIELD()*** is defined as empty.

Task type is defined as:

```
#define TASKFUNCTION void
```

RUNTASK() is defined as a function. It calls ***sc_tcreate()*** to create and start a task.

WAITNOMORE() uses function ***sc_fpost()***. ***WAITNOMORE_IR()*** is exactly the same as ***WAITNOMORE()***.

Macro ***WAITFOR()*** calls function ***Nwtems()*** (defined in **multi.c**) to wait for an event, and system function ***sc_flear()*** to clear the event. ***Nwtems()*** waits with the help of system function ***sc_fpend()***.

Macro ***BLOCKPREE()*** disables interrupts, ***RESUMEPREE()*** enables them.

Function ***Nclock()*** is mapped into function ***sc_gtime()***. ***Nclkinit()*** stores the multitasker value *SYSTEM_TICKPSEC* as the clock frequency.

D. Driver-Specific Information

3C509

About the device:

type	Ethernet
chip	3C509*, 3C595TX*
card	EtherLink III*
buffer memory	on-chip or on-board, varies
data transfer	16 or 32-bit input/output
interrupts	RX, space available

This is a drop-in driver for the 3Com Etherlink III adapter, both for the ISA and the PCI version. (We have not tested EISA or PCMCIA, but the required changes, if any, should be small.) The driver works with regular Ethernet and fast (100 megabits per second) Ethernet.

The basic 3C509 has 4K of on-chip memory. Other models have up to 128K of external (on-board) memory. EtherLink III handles its own buffers, so the driver is not affected by the memory size.

The 3C509 driver does not use the PCI bus master transfer (equivalent to DMA) available in the 3C595TX. This DMA is, in practice, limited to memory-to-memory moves, because the EtherLink III FIFO is half-duplex. You could tell the 3C595 to place a packet into memory and interrupt, but then you couldn't send anything. In a 90 Mhz Pentium, the 32-bit PCI input/output transfer seems about as fast as the DMA. Of course DMA would free up the processor for other work, but only at the expense of an extra interrupt.

Configuring

Interrupt number and port address are needed, for instance:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, _3C509, 0,
"IRNO=10 PORT=0x340",
```

To configure the board, run the 3Com utility **install**. The actual board configuration must match the **netconf.c** parameters. An incorrect configuration is most likely detected in the driver initialization, but don't count on this.

Interrupt Handling

The driver clears the interrupt using the "Interrupt Latch" command.

Sending

All buffer handling is done by the chip. The driver just tells the chip the packet size, and then copies the data. The send logic is:

1. If `hwflags` is 1, no buffer space is available. Queue up the message, and return 0 for "pending".
2. Otherwise, read the available buffer space. If this is big enough for the packet, copy the data, and return 1 for "done".
3. If not big enough, request the chip to interrupt when enough space is available, set `hwflags` to 1, and return 0 for "pending".

The "space available" interrupt will perform these steps:

1. If queue is empty, set `hwflags` to 0.
2. Otherwise check if available buffer space will take the packet. If not, request a "space available" interrupt.
3. If there is enough space, copy the data, and go back to check for more packets.

Receiving

All buffer handling is done by the chip. Whenever there is a receive interrupt, the driver allocates a USNet buffer, copies the message into it, and notifies the network task.

The error counters are updated for these cases:

`IfInErrors`	The status bit "`incomplete`" or "`error`" is set, the message length is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

DC21040

About the device:

type	Ethernet
chip	Digital Equipment 21040*
card	PCI
buffer memory	host memory
data transfer	DMA
interrupts	RX, TX

The driver was tested using the EtherPCI* card by LINKSYS. If there are other cards using this chip, the driver may need modifications for these. These should be very small, because there is little on the card besides the 21040.

Configuring

The hardware parameters are configured automatically by the PCI services. The **netconf.c** entry defines PCI as the driver:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, PCI, 0, 0,
```

PCI interrupts are always level-triggered. Make sure that the interrupt controller is cleared after the driver interrupt handler is called, not before. (This code is in **driver.c** or **suppa.asm**, but may also be part of an operating system.)

Clearing a level-triggered interrupt immediately will cause unwanted interrupts, and can in an extreme case generate a stack overflow.

The driver inititalization has the following error returns:

-1	The device code is not in `pcitab` in **PCI.C**. The table uses code `0x00021011` for DC21040. Change this if your board has a different code. The actual code is shown in an error message.
`NE_PARAM`	A configuration parameter is not recognized by the driver.
`NE_HWERR`	Reading of the hardware address fails. The board is broken, or not properly configured.

The adapter does not go online. The board is broken.

Sending

The send logic is as follows:

1. If *`hwflags`* is 1, the buffer is in use; queue up the message and return 0 for "pending".
2. Otherwise, set *`hwflags`* to 1, request the chip to transmit, and return 0 for "pending".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message and starts the transmission. If the queue is empty, it sets *`hwflags`* to 0.

Receiving

The receive code acquires *`NRECBUFS`* (default 2) receive buffers, and sets up the receiver. The interrupt handler performs the following steps for an arrived message:

1. Allocates a new buffer. If none is available, discards the message, and restarts the receive process.
2. Queues the message, and notifies the network task.
3. Adds the new buffer to the tail of the receive list.

The error counters are updated for the following cases:

`IfInErrors`	The error bit for the packet is set, or the message length is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

DC21140

About the device:

type	Ethernet
chip	Digital Equipment 21140*
card	PCI
buffer memory	host memory
data transfer	DMA
interrupts	RX, TX

This is the driver for the Fast Ethernet controller DC21140. The driver was tested using the EtherPCI card by TRENDNET. The driver may need modifications for other cards, and for embedded use, but these should be small.

Configuring

The hardware parameters are configured automatically by the PCI services. The **netconf.c** entry defines PCI as the driver:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, PCI, 0, 0,
```

PCI interrupts are always level-triggered. Make sure that the interrupt controller is cleared after the driver interrupt handler is called, not before. (This code is in **driver.c** or **suppa.asm**, but may also be part of an operating system.)

Clearing a level-triggered interrupt immediately will cause unwanted interrupts, and can in an extreme case generate a stack overflow.

The source-level variable *MEDIUM* is used to define what kind of network connection is used. The values are:

0 10BASE-T

1 BNC

3 100BASE-TX

4 10BASE-T full-duplex

5 100BASE-TX full-duplex

6 100BASE-T4

7 100BASE-FX

8 100BASE-FX full-duplex

The hardware uses MII (Media-Independent Interface), so these codes should work in any board that has a DEC-compatible serial EEPROM. Of course not all interfaces are available in all cases. The default is

```
#define MEDIUM 3        /* normal fast Ethernet */
```

The driver inititalization has the following error returns:

-1	The device code is not in `pcitab` in **PCI.C.** The table uses code `0x00091011` for DC21140. Change this if your board has a different code. The actual code is shown in an error message.
`NE_PARAM`	A configuration parameter is not recognized by the driver. *`MEDIUM`* is not supported by the hardware.
`NE_HWERR`	Reading of the hardware address fails. The board is broken, or not properly configured.

The adapter does not go online. The board is broken.

Sending

The send logic is as follows:

1. If *`hwflags`* is 1, the buffer is in use; queue up the message and return 0 for "pending".
2. Otherwise, set *`hwflags`* to 1, request the chip to transmit, and return 0 for "pending".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message and starts the transmission. If the queue is empty, it sets *`hwflags`* to 0.

Receiving

The receive code acquires *`NRECBUFS`* (default 2) receive buffers, and sets up the receiver. The interrupt handler performs the following steps for an arrived message:

1. Allocates a new buffer. If none is available, discards the message, and restarts the receive process.
2. Queues the message, and notifies the network task.
3. Adds the new buffer to the tail of the receive list.

The error counters are updated for the following cases:

`IfInErrors`	The error bit for the packet is set, or the message length is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

Special Situations

Some Pentium motherboards have difficulty handling the high-speed DMA load generated by the DC21140. In particular, the 386 instruction lodsd executed in real mode can fetch bad data while the DMA is in progress. This is obviously a very serious problem, but it is caused by a flawed PC motherboard, not by the DC21140.

If you are using DC21140 in real mode, change the parameter *`CPU`* in **suppa.asm** to 3 and run **BENCH**. If **BENCH** finishes without any particular difficulties, you are safe. If not, you can still run USNet by changing *`CPU`* back to 0. However, you might want to look for a better PC.

EN360

type	Ethernet
chip	Motorola 68360*
card	-
buffer memory	host memory
data transfer	DMA
interrupts	RX, TX

This is the driver for embedded Motorola 68360, a CPU that includes multi-purpose communication channels. The driver was tested using the Motorola MC68360 QUADS board.

Configuring

The contains the following source-level parameters:

QUADS Define this for the QUADS board. This is the default:

```
#define QUADS
```

In addition, the driver needs the base address (parameter *BASE*), the interrupt number (parameter *IRNO*) and the Ethernet address. If there is an Ethernet address in a known location (typically EPROM), you can use parameter *ENA* to specify where this is.

The configuration we used for the Motorola QUADS board was:

```
"m68k", "tnet", C, {192,9,200,3}, {0,0,1,2,3,4}, EA0, 0, Ethernet,
EN360, 0, "BASE=0x22000 IRNO=0x60"
```

This kind of hard-coded Ethernet address is of course for testing purposes only.

If the address is in EPROM, use the *ENA* parameter. In some cases getting the Ethernet address requires special processing, for instance reading a serial EPROM. In that case, you can replace the ***memcpy()*** in the initialization code with your own code that places the Ethernet address into `netp->id`.

The actual configuration must match the **netconf.c** parameters. An incorrect configuration will most likely result in a crash or a hang. The driver initialization has the following error returns:

`NE_PARAM` The given interrupt number is not a multiple of 32. An unrecognized parameter is used.

Sending

The initialization code creates 2 transmit buffer descriptors. The send logic is as follows:

1. If *hwflags* is 1, the transmitter is busy; queue up the message and return 0 for "pending".
2. Otherwise, set *hwflags* to 1, request the chip to transmit, and return 0 for "pending".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message and starts the transmission. If the queue is empty, it sets *hwflags* to 0.

Receiving

The initialization code creates one receive buffer descriptor, and sets up the receiver. The interrupt handler performs the following steps for an arrived message:

1. Allocates a new buffer. If none is available, discards the message, and restarts the receive process.

2. Queues the message, and notifies the network task.
3. Starts a new receive with the new buffer.

The error counters are updated for the following cases:

`IfInErrors`	The fatal error bit is set, or the message length is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

I82557

About the device:

type	Ethernet
chip	Intel 82557*
card	EtherExpress PRO/100*
buffer memory	host memory
data transfer	DMA
interrupts	RX, TX

This is a drop-in driver for the Intel EtherExpress PRO/100.

Configuration

The following parameters are available in the driver source:

NORB — Number of receive frame descriptors. Each reserves a packet buffer. Number needed depends on CPU speed and worst-case interrupt latency. Example:

```
#define NORB 4
```

DUPLEX — Normally full or half duplex is automatically negotiated by the physical link, but the outcome can be full duplex that does not work. Values: 0 = half duplex, 1 = full duplex, 2 = automatic. Example:

```
#define DUPLEX 0
```

In addition to these, the driver needs an interrupt number (parameter IRNO) and the port address. These are normally supplied by the PCI BIOS support, the **netconf.c** record is simply:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, PCI, 0, 0
```

PCI interrupts are always level-triggered. Make sure that the interrupt controller is cleared after the driver interrupt handler is called, not before. (This code is in **driver.c** or **suppa.asm**, but may also be part of an operating system.)

Clearing a level-triggered interrupt immediately will cause unwanted interrupts, and can in an extreme case generate a stack overflow.

The driver inititalization has the following error returns:

-1	The device code is not in `pcitab` in **PCI.C**. The table uses code `0x12298086`. Change this if your board has a different code. The actual code is shown in an error message.
`NE_PARAM`	A configuration parameter is not recognized by the driver.

`NE_HWERR`	Self-test failed. The board is broken, or not properly configured.

Sending

The send logic is as follows:

1. If *hwflags* is 1, the transmitter is busy; queue up the message and return 0 for "pending".
2. Otherwise, set *hwflags* to 1, request the chip to transmit, and return 0 for "pending".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message and starts the transmission. If the queue is empty, it sets *hwflags* to 0.

Receiving

The initialization code acquires *NORB* (default 4) receive buffers, and sets up the receiver. The interrupt handler performs the following steps for an arrived message:

1. Allocates a new buffer. If none is available, discards the message, and restarts the receive process.
2. Queues the message, and notifies the network task.
3. Adds the new buffer to the tail of the receive list.

I82595

About the device:

type	Ethernet
chip	Intel 82595TX*
card	EtherExpress PRO/10*
buffer memory	32K on the adapter card
data transfer	16 or 32-bit input/output
interrupts	RX, TX

This is a drop-in driver for the Intel EtherExpress PRO/10 adapter. The driver would work for an embedded 82595 with some small changes, but since the chip is designed for use in a PC, we don't cover this possibility here.

EtherExpress PRO/10 is a replacement for EtherExpress 16 and EtherExpress 32, but it is not software-compatible with either. The 82595 bears no similarities to either the 82586 or the 82596.

Configuring

Interrupt number and port address are needed, for instance:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, I82595, 0,
"IRNO=10 PORT=0x340",
```

To configure the board, or to find out the actual settings, use the Intel program **softset2**. The interrupt number given in **netconf.c** will override the number in the board EPROM. The port number in **netconf.c** must match the port number set with **softset2**, or the USNet initialization will fail.

The USNet driver initialization has these error returns:

`NE_PARAM`	The given interrupt number is not supported by the board. Use **softset2** to find out the allowed values.
`NE_HWERR`	No board ID was seen. The board is broken, or not at this port address.
Board reset fails	The board is broken.

The driver as shipped will do 32-bit transfers. To run it in a sub-386 computer, change the **I82595.C** source line:

```
#define XFER 32
```

to:

```
#define XFER 16
```

Interrupt Handling

Only the `RX` and `TX` interrupts are used. The interrupt handler masks off all 82595 interrupts, to force clearing of interrupts in edge-triggered systems.

Sending

The send routine uses two transmission buffers, at 0x7400 and 0x7a00. The logic is:

1. If *`hwflags`* is non-zero, both buffers are in use; queue up the message and return 0 for "pending".
2. If *`hwflags`* is zero, copy the message into the current buffer. Then check for transmitter idle: If yes, start the transmission; if no, set *`hwflags`* to 1. Return 1 for "done".

Whenever *`hwflags`* was set to 1, the interrupt handler will perform these steps:

1. Start transmission for current `TX` buffer.
2. If queue is empty, set *`hwflags`* to 0. If not, copy message into current `TX` buffer.

Receiving

The receive code uses the buffer pool from 0x0000 to 0x75FF. (It appears that the `RX` buffer pool must start at zero.) Whenever there is a receive interrupt, the driver allocates a buffer, copies the message into it, and notifies the network task.

The error counters are updated for these cases:

`IfInErrors`	The status bit "`RCV OK`" is not set, or the message length is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

I82596

About the device:

type	Ethernet
chip	Intel 82596*
card	-
buffer memory	host memory
data transfer	DMA

interrupts RX, TX

This is the driver for embedded Intel 82596. (The chip was also used in EtherExpress 32, now replaced by EtherExpress PRO/10.) The driver was tested using a Motorola MVME-162LX board.

The driver assumes only one 82596, because quite a bit of static memory is needed for each.

Configuring

The 82596 is not really a memory-mapped device, and the software interface will vary from case to case. Some of the configuring is in **netconf.c**, but parts must be done in driver source.

The driver must be able to send an attention signal to the 82596, and to give a command to it. You need to check your board documentation on how this is done. These two functions are defined as macros at the start of the driver:

```
#define ATTENTION() - - -

#define COMMAND(lo, hi) - - -
```

In addition to these, the driver needs the interrupt number (parameter *IRNO*) and the Ethernet address. If there is an Ethernet address in a known location (typically EPROM), you can use parameter *ENA* to specify where this is.

The configuration we used for the Motorola MVME-162LX board was:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, I82595, 0,
"BASE=0xfff46000 IRC=0xfff4202a ENA=0xfffc1f2c IRNO=0x57"
```

The *BASE* parameter here is used for the ***ATTENTION()*** and the ***COMMAND()*** macros. You can, of course, hard-code the needed addresses directly into the macros, in which case *BASE* is not needed. The *IRC* parameter is used for interrupt control in the Motorola evaluation board. If you don't define *IRC*, this code will not be executed.

In some cases getting the Ethernet address requires special processing, for instance reading a serial EPROM. In that case, you can replace the ***memcpy()*** in the initialization code with your own code that places the Ethernet address into `netp->id`.

The actual configuration must match the **netconf.c** parameters. An incorrect configuration will most likely result in a crash or a hang. I82596 has no identification registers, and the driver initialization has no error returns.

Sending

The send logic is:

1. If `hwflags` is 1, the transmitter is busy; queue up the message, and return 0 for "pending".
2. Otherwise, set `hwflags` to 1, request the chip to transmit, and return 0 for "pending".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message and starts the transmission. If the queue is empty, it sets `hwflags` to 0.

Receiving

The initialization code acquires *NORB* (default 4) receive buffers, and sets up the receiver. The interrupt handler performs these steps for an arrived message:

1. Allocates a new buffer. If none is available, discards the message, and restarts the receive process.
2. Queues the message, and notifies the network task.
3. Adds the new buffer to the tail of the receive list.

MB86960

About the device:

type	Ethernet
chip	Fujitsu MB86960* family
card	-
buffer memory	8-64K on-board
data transfer	16-bit alternate space
interrupts	RX

This is an embedded driver for the Fujitsu MB86960 (NICE*), MB86964 (Etherstar*) and the MB86965 (Ethercoupler*).

Configuring

The chip type is given in driver source:

```
#define TYPE 0              /* 0 = 86960 (NICE) /*
                            /* 1 = 86964 (Etherstar) */
                            /* 2 = 86965 (Ethercoupler) */
```

The initial value for configuration register 0 will vary; as a reminder this is defined in the source:

```
#define DLCR6VAL 0x45
          /* 0x45 for SPARClite evaluation board */
          /* 0x55 for Fujitsu ISA board */
```

Check your hardware documentation for the proper value.

Macro ***PORTADD()*** at the start of the MB86960 source defines how the hardware registers are mapped to port or memory addresses:

```
#define PORTADD(reg) (mapping expression)
```

Typically SPARClite uses only the low 16 bits of each long word, which maps as:

```
(reg+reg+2-(reg&1))
```

An ISA board would typically map one-to-one, so the macro is simply **(reg)**. These two examples by no means exhaust all possibilities, so check your hardware documentation.

Netconf.c needs the interrupt number and the I/O base address, for instance:

```
"sparc", "tnet", C, {192,9,200,19}, {1,2,3,4,5,6},
0, Ethernet, MB86960, 0, "BASE=0x20000000 IRNO=14",
```

The above example hard-codes the Ethernet address. If there is an Ethernet address in a known location (typically EPROM), you can use parameter *ENA* to specify where this is, for instance:

```
ENA=0xffff0000
```

In some cases getting the Ethernet address requires special processing, for instance reading a serial EPROM. In that case, you can replace the ***memcpy()*** in the driver initialization with your own code that places the Ethernet address into `netp->id`.

The USNet driver initialization has these error returns:

`NE_PARAM`	Unknown parameter supplied.

Interrupt Handling

The reason for not using the `TX` interrupt is that in some members of the chip family the transmit operation occasionally freezes, and the chip has to be reset to recover from this. The detection of the freeze depends on a timeout, and would be fairly complicated with `TX` interrupts. The speed penalty for not using the `TX` interrupt is not very large.

To get rid of unwanted packets, the interrupt handler calls the assembly-language routine ***Ninhdsc()***. This routine must be fast, otherwise the chip may stop working under extremely heavy loads.

There is a skip packet command, but this is unreliable under extremely heavy loads.

Sending

The send routine waits for carrier off, then moves the packet length and the packet data. Then it clears the `TX done` bit in the `TX status` register, and starts the operation. Clearing the `done` bit is absolutely necessary, even though undocumented.

Then the routine waits for `TX done`. If this does not come in 500 milliseconds, it clears the controller and tries again.

Receiving

Whenever there is a receive interrupt, the driver allocates a buffer, copies the message into it, and notifies the network task.

If the control information (packet size, status) is invalid, the driver stops and restarts the Ethernet controller.

The error counters are updated for these cases:

`IfInErrors`	The message length is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

SPECIAL NOTE:

The reception of an Ethernet frame over 2047 bytes long will cause the MB86960-family controllers to hang. The driver will recover from this situation by stopping and restarting the chip.

Several Ethernet controllers do support oversize packets, but they should have no business sending these to the MB86960, or even as broadcasts. Therefore, the problem is unlikely to be serious in practice.

NE1000

About the device:

type	Ethernet
chip	National Semiconductor 8390*
card	NE1000*
buffer memory	8K on the adapter card
data transfer	8-bit input/output
interrupts	RX, TX

This is a drop-in driver for the Novell Standard NE1000 adapter for 8-bit PCs. Novell does not build boards any more, but the NE1000 has been adopted by several board manufacturers. It is still commonly used in various single-board computers.

There is a separate driver for embedded NS8390.

Configuring

Interrupt number and port address are needed, for instance:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, NE1000, 0,
"IRNO=5 PORT=0x340",
```

Most, if not all, NE1000 boards configure with jumpers.

Since the boards come from various manufacturers, we can't give exact instructions. (Also note that there are two different Novell NE1000 boards.) The Novell NE1000 allows for these settings:

IRQ number:	2, 3, 4, 5
Port address:	300, 320, 340, 360

The actual board configuration must match the **netconf.c** parameters. An incorrect configuration will most likely result in a crash or a hang. NE1000 has no identification registers, and the driver initialization has no error returns.

Interrupt Handling

The interrupt handler masks off all chip interrupts, to force clearing of interrupts in edge-triggered systems.

One peculiarity of the NS8390 is the receiver overrun error, called Buffer Ring Overflow in the documentation. To continue from this condition, the chip must be stopped, cleared, and restarted. There are differing versions of how exactly this should be done. USNet follows the instructions given in *Local Area Networks Databook*, 1993 second edition, by National Semiconductor. We have tested the error recovery using artificially induced overruns.

The overrun recovery contains a 2-millisecond wait in the interrupt handler. This may not be acceptable in an embedded system. If this becomes a problem, you may want to look into the reasons for the overrun. In a PC, the only way to get overrun errors is to disable interrupts for unreasonable amounts of time. In an embedded system the overrun errors might also mean that the hardware is overloaded in some way.

Sending

The send routine uses a transmission buffer at 0x2000. The logic is:

1. If *`hwflags`* is 1, the buffer is in use; queue up the message and return 0 for "pending".
2. Otherwise, set *`hwflags`* to 1, copy the message into the current buffer, start the transmission, and return 1 for "done".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message, copies it into the `TX` buffer, and starts the transmission. If the queue is empty, it sets *`hwflags`* to 0.

Receiving

The receive code uses the buffer pool from 0x2600 to 0x3FFF. Whenever there is a receive interrupt, the driver allocates a buffer, copies the message into it, and notifies the network task.

The error counters are updated for these cases:

`IfInErrors`	The status bit "no errors" is not set, or the message length is invalid, or the message pointer is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

NE2000

About the device:

type	Ethernet
chip	National Semiconductor 8390
card	NE2000*
buffer memory	16K on the adapter card
data transfer	16-bit input/output
interrupts	RX, TX

This is a drop-in driver for the Novell Standard NE2000 adapter. Novell does not build boards any more, but the NE2000 has been adopted by several manufacturers, and is still extremely popular. Many of the boards do not actually contain an NS8390.

There is a separate driver for embedded NS8390.

Configuring

Interrupt number and port address are needed, for instance:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, NE2000, 0,
"IRNO=10 PORT=0x340",
```

Since these boards come from various manufacturers, we can't give instructions on how to configure them. Typically, the older boards configure with jumpers, the newer boards with a configuration program.

The actual board configuration must match the **netconf.c** parameters. An incorrect configuration will most likely result in a crash or a hang. NE2000 has no identification registers, and the driver initialization has no error returns.

Interrupt Handling

The interrupt handler masks off all chip interrupts, to force clearing of interrupts in edge-triggered systems.

One peculiarity of the NS8390 is the receiver overrun error, called Buffer Ring Overflow in the documentation. To continue from this condition, the chip must be stopped, cleared, and restarted. There are differing versions of how exactly this should be done. USNet follows the instructions given in *Local Area Networks Databook*, 1993 second edition, by National Semiconductor. We have tested the error recovery using artificially induced overruns.

The overrun recovery contains a 2-millisecond wait in the interrupt handler. This may not be acceptable in an embedded system. If this becomes a problem, you may want to look into the reasons for the overrun. In a PC, the only way to get overrun errors is to disable interrupts for unreasonable amounts of time. In an embedded system the situation may not be that simple; the overrun errors might also mean that the hardware is overloaded in some way.

Sending

The send routine uses a transmission buffer at 0x0000. The logic is:

1. If *hwflags* is 1, the buffer is in use; queue up the message and return 0 for "pending".
2. Otherwise, set *hwflags* to 1, copy the message into the current buffer, start the transmission, and return 1 for "done".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message, copies it into the TX buffer, and starts the transmission. If the queue is empty, it sets *hwflags* to 0.

Double buffering would speed up the transmission a little, but we couldn't get it to work reliably in some NE2000 boards, so we are not using it.

Receiving

The receive code uses the buffer pool from 0x0600 to 7FFF. Whenever there is a receive interrupt, the driver allocates a buffer, copies the message into it, and notifies the network task.

The error counters are updated for these cases:

`IfInErrors`	The status bit "no errors" is not set, or the message length is invalid, or the message pointer is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

NE2100

type	Ethernet
chip	Advanced Micro Devices 7990*
card	NE2100*
buffer memory	host memory
data transfer	DMA
interrupts	RX, TX

This is a drop-in driver for the Novell Standard NE2100 adapter. Novell does not build boards any more, but the NE2100 has been adopted by several manufacturers.

There is a separate driver for embedded AMD 7990.

Configuring

Interrupt number, port address and DMA channel are needed, for instance:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, NE2100, 0,
"IRNO=4 PORT=0x340 DMA=5",
```

As these boards come from various manufacturers, we can't give instructions on how to configure them. Typically, the older boards configure with jumpers, the newer boards with a configuration program.

The actual board configuration must match the **netconf.c** parameters. NE2100 has an EPROM checksum, so a bad board or a bad configuration may be detected in the initialization, but there is no guarantee of this.

The driver inititalization has the following error returns:

`NE_PARAM`	The configuration parameter is not recognized by the driver.
`NE_HWERR`	Checksum was bad. The board is broken, or not at this port address.

The stop or the start command did not work. The board is broken.

Sending

The send logic is as follows:

1. If *hwflags* is 1, the buffer is in use; queue up the message and return 0 for "pending".
2. Otherwise, set *hwflags* to 1, request the chip to transmit, and return 0 for "pending".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message and starts the transmission. If the queue is empty, it sets *hwflags* to 0.

Receiving

The recive code acquires *NRECBUFS* (default 2) receive buffers, and sets up the receiver. The interrupt handler performs the following steps for an arrived message:

1. Allocates a new buffer. If none is available, discards the message, and restarts the receive process.
2. Queues the message, and notifies the network task.
3. Adds the new buffer to the tail of the receive list

The error counters are updated for the following cases:

`IfInErrors`	The error bit for the packet is set, or the message length is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

NS8390

About the device:

type	Ethernet
chip	National Semiconductor 8390*
card	-
buffer memory	variable *POOLSIZE*
data transfer	8 or 16-bit input/output
interrupts	RX, TX

This is the driver for embedded National Semiconductor 8390. It will handle these members of the family:

- DP8390D, also called NS32490D
- DP83901A
- DP83902A, also called ST-NIC
- DP83905, also called AT/LANTIC

Configuring

The driver needs the interrupt number, the port address (a memory address in memory-mapped systems), and the Ethernet address. If there is an Ethernet address in a known location (typically EPROM), you can use parameter *ENA* to specify where this is:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, NS8390, 0,
"IRNO=5 PORT=0xffffe000 ENA=0xfffff000",
```

In some cases getting the Ethernet address requires special processing, for instance reading a serial EPROM. In that case, you can replace the ***memcpy()*** in the initialization code with your own code that places the Ethernet address into `netp->id`.

The NS8390 source contains two configuration variables:

POOLSIZE	is the size of the buffer pool (default 8k)
XFER	is 8 or 16 for data transfer width (default 8)

The actual configuration must match the **netconf.c** parameters. An incorrect configuration will most likely result in a crash or a hang. NS8390 has no identification registers, and the driver initialization has no error returns.

Interrupt Handling

The interrupt handler masks off all chip interrupts, to force clearing of interrupts in edge-triggered systems.

One peculiarity of the NS8390 is the receiver overrun error, called Buffer Ring Overflow in the documentation. To continue from this condition, the chip must be stopped, cleared, and restarted. There are differing versions of how exactly this should be done. USNet follows the instructions given in *Local Area Networks Databook*, 1993 second edition, by National Semiconductor. We have tested the error recovery using artificially induced overruns.

The overrun recovery contains a 2-millisecond wait in the interrupt handler. This may not be acceptable in an embedded system. If this becomes a problem, you may want to look into the reasons for the overrun. The overrun could mean that interrupts are disabled for too long, or that the hardware is overloaded in some way.

Sending

The send routine uses a transmission buffer at 0x0000. The logic is:

1. If `hwflags` is 1, the buffer is in use; queue up the message and return 0 for "pending".
2. Otherwise, set `hwflags` to 1, copy the message into the current buffer, start the transmission, and return 1 for "done".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message, copies it into the TX buffer, and starts the transmission. If the queue is empty, it sets *hwflags* to 0.

Receiving

The receive code uses the buffer pool from 0x0600 to *POOLSIZE*-1. Whenever there is a receive interrupt, the driver allocates a buffer, copies the message into it, and notifies the network task.

The error counters are updated for these cases:

`IfInErrors`	The status bit "no errors" is not set, or the message length is invalid, or the message pointer is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

SMC91C92

About the device:

type	Ethernet
chip	SMC 91C90*, 91C92*, 91C94*
card	SMC91C92*
buffer memory	on-chip, amount varies
data transfer	8 or 16-bit, input/output
interrupts	RX, TX, allocation

This is a drop-in driver for the Standard Microsystems Corporation 91C92 Ethernet adapter. This driver should also work, possibly with small changes, for an embedded 91C9x.

Configuring

Interrupt number and port address are needed, for instance:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, SMC91C92, 0,
"IRNO=5 PORT=0x340",
```

See the SMC booklet on how to configure the board. The actual board configuration must match the **netconf.c** parameters. An incorrect configuration will most likely cause a crash or a hang.

Interrupt Handling

The driver clears the interrupt by masking off all 91C92 interrupts at the start of the interrupt handler. This is to guarantee that an edge-triggered system (such as the PC) will see the next interrupt.

Sending

The driver appends 0x20 after an odd-sized packet, 2 zeroes after an even-sized packet.

The buffer handling is done by the chip, but the driver must explicitly allocate and free the space. The send logic is as follows:

1. If `hwflags` is 1, no buffer space is available; queue up the message, and return 0 for "pending".
2. Otherwise, ask for buffer space. If this is available, copy the data, start the transmission, return 1 for "done".
3. If space is not available, set `hwflags` to 1, queue the packet into the departure queue, enable the allocation interrupt.

The allocation interrupt will perform the following steps:

1. If queue is empty, set `hwflags` to 0.
2. Otherwise, request buffer space. If available, copy the data, start the transmission, go back to check for more packets.
3. If space is not available, make a note to exit the interrupt handler with allocation interrupts enabled.

The transmit interrupt releases the buffer space, leaving all other transmit work to the allocation interrupt.

Receiving

All buffer handling is done by the chip. Whenever there is a receive interrupt, the driver allocates a USNet buffer, copies the message into it, and notifies the network task.

In case of a receive overrun error, the driver clears the overrun and restarts the receiver.

The error counters are updated for the following cases:

`IfInErrors`	Any of the fatal error bits is set, or message length is invalid.
`IfInDiscards`	The input queue is full, or no USNet buffers are available.

WD8003

About the device:

type	Ethernet
chip	National Semiconductor 8390*
card	WD8003*, SMC Ultra*
buffer memory	8K on the adapter card
data transfer	shared memory, 16-bit
interrupts	RX, TX

The original WD8003 is not made any more. Western Digital sold its network operations to Standard Microsystems Corporation, or SMC. SMC makes a successor to the WD8003, called the SMC Ultra. The WD8003 driver handles both cards.

There is a separate driver for embedded NS8390.

Appendix D

Configuring

Interrupt number, port address and buffer address are needed, for instance:

```
"test", "tnet", C, {192,9,200,3}, EA0, 0, Ethernet, WD8003, 0,
"IRNO=5 PORT=0x340 BUFFER=0xd0000",
```

The WD8003 jumpers are used as follows:

```
  IRQ number W2:  IRQ      11 9 7 5 3 1
                  2        I - - - - -
                  3        - I - - - -
                  4        - - I - - -
                  5        - - - I - -
                  6        - - - - I -
                  7        - - - - - I


  port address W1:         2 4 6 8 10
                  0x200    - I I I I
                  0x220    - - I I I
                  0x240    - I - I I
                  0x260    - - - I I
                  0x280    - I I - I
                  0x2A0    - - I - I
                  0x2C0    - I - - I
                  0x2E0    - - - - I
                  0x300    - I I I -
                  0x320    - - I I -
                  0x340    - I - I -
                  0x360    - - - I -
                  0x380    - I I - -
                  0x3A0    - - I - -
                  0x3C0    - I - - -
                  0x3E0    - - - - -
```

To configure the SMC Ultra, use the SMC program **ezstart**.

The actual board configuration must match the **netconf.c** parameters. The board has a checksum, so the driver initialization will normally detect an incorrect port address. An incorrect interrupt number will most likely result in a crash or a hang.

Interrupt Handling

The interrupt handler masks off all chip interrupts, to force clearing of interrupts in edge-triggered systems.

WD8003 does not contain handling for the NS8390 Buffer Ring Overflow. The WD8003 is very unlikely to be used in an embedded system, and the overflow error should never happen in a PC. If you really need the overrun handling, you can lift it from the NE2000 driver.

Sending

The send routine uses a transmission buffer at 0x1A00. The logic is:

1. If *hwflags* is 1, the buffer is in use; queue up the message and return 0 for "pending".
2. Otherwise, set *hwflags* to 1, copy the message into the current buffer, start the transmission, and return 1 for "done".

The transmit interrupt handler will check the transmit queue. If this is not empty, it takes the top message, copies it into the TX buffer, and starts the transmission. If the queue is empty, it sets *hwflags* to 0.

Receiving

The receive code uses the buffer pool from 0x0000 to 0x19FF. Whenever there is a receive interrupt, the driver allocates a buffer, copies the message into it, and notifies the network task.

The error counters are updated for these cases:

`IfInErrors`	The status bit "no errors" is not set, or the message length is invalid, or the message pointer is invalid.
`FInDiscards`	The input queue is full, or no USNet buffers are available.

E. Dynamic Configuration of the Routing Table

Overview

The USNet utility directory **unsupp** contains the dynamic routing table utility in file **confupd.c**. To use this utility, place this file in the **netsrc** directory and add the file to the `FILES` list in the makefile. The use of this utility is not covered by our technical support.

Dynamic configuration supports "on-the-fly" modification of the entries in the structure `netconf`. This is the same table you statically configured before building your network application.

The configuration updating must be bracketed by calls to ***ConfLock()*** and ***ConfFree()***:

```
ConfLock();
- - -
<<perform the updating>>
BuildRoutes();      /* Rebuild routing table */
- - -
ConfFree();
```

The routing table is not rebuilt automatically. If you want it rebuilt, call the routine ***BuildRoutes()*** before the call to ***ConfFree()***. This call may take a while, depending on the size of the configuration table.

Configuration entries use structure `netconf`, defined in **support.h**. The following fields in `netconf` are significant in dynamic configuration:

```
char name[9];        /* host name */
char pname[9];       /* network name */
Iid Imask;           /* address mask, 0 = host part */
Iid Iaddr;           /* internal (Internet) address */
struct Eid Eaddr;    /* external (Ethernet) address */
```

Routing Table Configuration Functions

The following functions are discussed in this section:

ConfLock() marks the start of updating.

ConfFree() terminates updating.

ConfFind() finds a configuration entry.

ConfDel() deletes a configuration entry.

ConfAdd()	adds a configuration entry.
ConfRename()	changes an IP address.
ConfDisplay()	displays the configuration table.

ConfLock

Marks the start of updating.

```
void ConfLock(void)
```

Make this call before updating configuration information.

ConfFree

Terminates the updating.

```
void ConfFree(void)
```

Make this call after completing configuration update tasks.

ConfFind

Finds a configuration entry.

```
int ConfFind(int startix, struct NETCONF *argp)
```

`startix` specifies where in the configuration table the search begins

`argp` is a pointer to a configuration structure

ConfFind returns the configuration index for a given configuration record. The fields in this structure act as "keys" in the search; you can use any combination of host name, network name and Internet address. Set to zero any value that should not be checked.

Return Value

The routine returns the index of the first match. No match is returned as -1.

Example

The first argument would typically be 0, but you can use it to scan for duplicates, for instance:

```
memset(&confrec, 0, sizeof(confrec));
strcpy(confrec.name, "server2");
for (index=0; index>=0; index++)
{
    index = ConfFind(index, &confrec);
    /* process the record */
}
```

ConfDel

Deletes a configuration entry.

```
int ConfDel(int argix)
```

ConfDel deletes the specified configuration record. The argument gives the index in the table. If necessary, use ***ConfFind*** to get the index first.

Return Value

The return value is the given index, or -1 for an error. The error cases are:

- Index was too large
- The connection is local or already deleted or static

ConfAdd

Adds a configuration entry.

```
int ConfAdd(struct NETCONF *argp)
```

This call adds a configuration record to the configuration table. If there's an old one for the same IP address, it will be replaced. The argument is a pointer to a properly filled configuration structure. Host name and IP address are mandatory.

Return Value

The return value is the new configuration index, or -1 if the attempt failed. The error cases are:

- There was no host name
- Attempt to add a local connection
- The IP address was zero
- There was no room to add an entry

ConfRename

Changes an IP address.

```
int ConfRename(int argix, Iid iaddr)
```

ConfRename changes an IP address in the configuration table. This function is intended as a logical rename, without any physical configuration changes. Both local and foreign addresses can be changed.

The old address should not be in use at the time of the change.

Return Value

The return value is the configuration index, or -1 if the index was too large.

ConfDisplay

Displays the configuration table.

```
void ConfDisplay(void)
```

This call displays the configuration table for debugging purposes. The format is:

```
host1   port        n1.n2.n3.n4
host2   port        n1.n2.n3.n4
- - -
```

Index

_

_inb() macro, 135
  and character drivers, 139
  and init(), 143
  and shut(), 143
  and transmit interrupts, 145
_inw() macro, 135
_outb() macro, 135
  and character drivers, 139
  and init(), 143
  and shut(), 143
  and transmit interrupts, 145
_outw() macro, 135

3

3C509 processor, driver info, 187

A

accept() BSD function, 76
  example, 76
adapters, 43, 153
  configuring, 42
    example, 43
  initialization, 141, 151
  shutdown, 143, 152
address mask, example, 43
addressing
  Motorola-type, 117
AFLAGS macro, 40
AMD 386 benchmarks, 161
application
  beginning, 23
  developing, 23
  macros to define, 40
application development, 35
applications
  supplied with USNet, 2
architecture, segmented, 132
Arcnet drivers provided, 127
ARCNET, 40 Mhz AMD 386
  benchmarks, 161
ARP, 42
ARP cache, 172
ARP table, 172
arrive queue, 132
AS macro, 40
assembler, 40
  command-line options, 40
assembly stubs, 136
  customizing, 119
authenticating user, 51
auto-generated configuration files, 38

B

baud rate, 43
  for I8250, 142
BENCH, 111
benchmark test, 111
benchmarks
  for AMD 386, 161
  for AMD 386 ARCNET, 161
  for Fujitsu SPARClite, 162
  for Intel 386SX, 162
  for Motorola 68360, 162
  for Two-Hop Routing, 163
  overview, 155
  results, 158
bind() BSD function, 77
  example, 77
blind spot, 125
  and WAITFOR(), 123
  example, 124
block drivers
  data structures, 145
  description, 144
BLOCKPREE() multitasking macro, 122
boot name server, 104
booting
  getting boot record, 104
  opening connection for, 104
  reading bootload data, 105
BOOTP program, 41
  description, 104
BOOTPget() routine, 104

BOOTPread() routine, 105
broadcasting
  example, 68
BSD, 23
BSD functions
  accept(), 76
  bind(), 77
  closesocket(), 78
  connect(), 79
  fcntlsocket(), 80
  for connectionless protocol, 75
  gethostbyname(), 81
  gethostbyname_r(), 82
  getpeername(), 83
  getsockname(), 84
  getsockopt(), 85
  ioctlsocket(), 87
  listen(), 88
  readsocket(), 89
  recv(), 90
  recvfrom(), 92
  recvmsg(), 94
  return values, 75
  selectsocket(), 95
  send(), 97
  sendmsg(), 98
  sendto(), 99
  shutdown(), 100
  socket(), 101
  typical calling sequences, 74
  writesocket(), 102
BSD socket interface, 71
BSD sockets
  writing new code, 72
buffer space, 4
buffers
  code for checking, 24
  code for constructing, 23
  code for server.c, 27
  Ethernet requirements, 48
  setting number available, 48
  setting size, 48
BuildRoutes() function, 209

C

CC macro, 40
CFLAGS macro, 40
CHAP, definition, 173
character drivers
  description, 137, 139
chksum_INASM Macro, 50
CLEARIR() macro, 119
  and unsupported processors, 136
  description, 131
  example, 131
client
  data collection loop, 28
  defining, 24
  FTP, 106
  required features, 25
  role of, 24
  slow start, 171
  Telnet, 107, 108
  terminating USNet, 28
client.c file
  compiling, 29
  structure, 28
clock routines, summary list, 118
clocks_per_sec
  and LOCALSETUP(), 119
clocks_per_sec variable, 118
closE() routine
  description, 141, 151
closesocket() BSD function, 78
code
  reentrant, 4
  ROMmable, 5
  source, 4
comec() routine
  accessing, 140
  and irhan() function, 139
  for reading data, 138
compiler, 40
  command-line options, 40
  defining, 38
COMPILER macro, 38
compiler.mak, 39
compiler.mak file
  contents, 38
  definition, 173
  editing, 39
compilers

unsupported, 117
compiling
application, 29
USNet, 11
ConfAdd() routing table function
description, 211
ConfDel() routing table function
description, 211
ConfDisplay() routing table function
description, 212
ConfFind() routing table function
description, 210
ConfFree() routing table function
description, 210
config.mak, 38
config.mak file
contents, 38
definition, 173
config.sys file
and NDIS drivers, 46
configuration, 37
compiler.mak, 39
config.mak, 38
makefiles, 38
start up example, 142
configuration files
auto-generated, 38
configuration parameters, 37
configuration table
displaying, 212
opening a connection, 59
ConfLock() routing table function
description, 210
ConfRename() routing table function
description, 211
connect() BSD function, 79
example, 79
connection
establishing, 26
connections
accepting on sockets, 76
active open, 57, 58
example, 59
closing, 60
general description, 57
in parallel, 120
initiating on a socket, 79
listening for, 88
opening, 58
passive open, 57, 58, 68
example, 59
receiving messages from, 61, 92
shutting down, 100
writing messages to, 62
control parameters
setting for socket, 87
control, yielding for multitasking, 122
counters, and signaling, 123
customization
overview, 117

D

data
incoming, and block driver, 144
outgoing, and block driver, 145
reading, and character drivers, 138
sending, and character drivers, 138
transfer between controller and application, 137, 144
data collection loop, 28
data structures, 127
fd_set, 95
hostent, 82
include files needed for, 73
MESSH, 127, 128
msghdr, 94, 98
NET, 127, 129
sockaddr, 73
sockaddr_in, 73
timeval, 95
DBG_ID macro, 40
DC21040 processor, driver info, 188
DC21140 processor, driver info, 190
departure queue, 132, 140
design considerations, 167
developing first application, 23
development
application, 35
development system specifications, 13
device driver macros
DISABLE(), 131
ENABLE(), 131
QUEUE_FULL(), 133

QUEUE_IN(), 132
QUEUE_OUT(), 134
device drivers, 5, 127, 187
bad parameters, 136
called from PTABLE, 144
clearing controller interrupt, 131
code you write, 138
format, 137
interface, 127
interrupt handler, 131
NDIS, 46
PCMCIA, 154
restoring interrupt, 132
support functions, 130
transmit interrupts, 157
types allowed, 45
using struct NET, 129
writing your own, 137
DHCP, 41
definition, 173
description, 105
DHCP macro, 50
DHCPget() routine, 105
DHCPrelease() routine, 106
DHCPTEST, 111
diagnostics, 175
DISABLE() macro, 119
altering for unsupported processors, 136
description, 131
DMA, 49, 138
DNS
definition, 173
DNS macro, 50
documentation, 10
DOS
display and keyboard support, 119
installing USNet on, 9
running Telnet, 107
timers, 118
with NDIS, 46
DOS extender
interrupt handling capacity, 138
DPI, 53
definition, 173
DPI (Dynamic Protocol Interface), 23
drivers
configuring, 45
ODI, 46
dynamic protocol functions
Nclose(), 60
Ninit(), 54
Nopen(), 58
Nread(), 61
Nterm(), 55
Nwrite(), 62
Portinit(), 55
Portterm(), 56
Dynamic Protocol Interface, 53
blocking mode, 53
non-blocking mode, 53
overview, 53
Dynamic Protocol Interface macros
SOCKET_BLOCK(), 64
SOCKET_CANSEND(), 65
SOCKET_FIN(), 66
SOCKET_HASDATA(), 64
SOCKET_IPADDR(), 65
SOCKET_ISOPEN(), 64
SOCKET_MAXDAT(), 65
SOCKET_NOBLOCK(), 64
SOCKET_OWNIPADDR(), 66
SOCKET_PUSH(), 66
SOCKET_RXTOUT(), 65
SOCKET_TESTFIN(), 65
summary list, 63

E

EMTEST, 17, 111
and trace output, 175
configuration requirements, 18
development requirements, 18
goals, 17
pass indicators, 19
EN360 processor, driver info, 192
ENABLE() macro, 119
altering for unsupported processors, 136
description, 131
ENDIAN parameter, 117
errno
and BSD functions, 75

error codes, 136
ESCNAME variable, 108
ESCTYPE variable, 108
Ethernet
configuring, 42
example, 43, 44, 45
multiple connections, 40
drivers provided, 127
example of interrupt handler, 146
example of starting and configuring, 151
shutting down, 152
using block driver, 144
events
and signaling, 123
experience, 6

F

FARDEF parameter
for segmented architectures, 117
fcntlsocket() BSD function, 80
fd_set structure, 95
file transfer
example, 69
files
receiving, 107
sending, 106
firstapp.h file, 24
flags, 117
configuring, 42
example, 43
flow control, 57, 169
for congestion, 171
testing with MTTEST, 20
fragmentation, 49
FRAGMENTATION macro, 49
FTP
and FTTEST, 112
definition, 173
description, 106
multitasking, 121
FTP server
PC, 17
UNIX, 18
FTP test, 17, 111, 112
FTPget() routine, 107
examples, 107
FTPput() routine, 106
FTPserv() routine, 106
FTTEST, 18, 19, 112
and UNIX, 112
quitting server, 113
server for PING, 114
Fujitsu SPARClite benchmarks, 162

G

getbufIR() function
and irhan() function, 147
getenv() ANSI C function, 51
gethostbyname() BSD function, 81
example, 81
gethostbyname_r() BSD function, 82
example, 82
getpeername() BSD function, 83
example, 83
getsockname() BSD function, 84
example, 84
getsockopt() BSD function, 85
example of retrieving errno, 75
goingc() routine
and irhan() function, 139
for sending data, 138

H

hardware
configuring, 43, 45, 51, 118
parameters, 141
configuring, 43
hardware address, 42
configuration example, 43
writing to controller, 135
hc16, 136
and assembly stubs, 119
header files, including, 25
Hitachi HI-SH7
configuring, 182
installation, 182
operating system notes, 182
testing, 182
Hitachi HI-SH-7
creating applications, 183
features used, 184

host name, 41, 51
getting IP address for, 81, 82
hostent structure, 82
HTTEST, 113

I

I/O
mapping addresses, 132
I386 processor
configuring interrupt table size, 136
interrupt handling capacity, 138
I386SX processor
benchmarks, 162
I8086 processor
configuring interrupt table size, 136
I8250 processor
configuring, 43
initialization parameters, 142
PTABLE example, 144
I82557 processor, driver info, 193
I82595 processor, driver info, 194
I82596 processor, driver info, 195
ICMP protocol, 59, 114
include files needed, 54
identifying user, 51
IGMP, 108
BSD API, 102
DPI API, 66
implementation considerations, 167
include files, 25, 54
init() routine
description, 141
init_char_driver, 151
initialization
and Ninit(), 54
initializing USNet, 25
functions required, 25
installation
directory tree, 10
for UNIX, 9
for Windows or DOS, 9
Internet address, 41
configuration example, 43
interprocess communication, 122
interrupt addresses
block drivers, 151
character drivers, 141
interrupt handler, 127, 135
block device, 145
example for character drivers, 139
example with block driver, 146
example without transmit interrupt, 146
installing, 131
irhan() description, 139
with transmit interrupt, 145
with writE(), 140
interrupt number
clearing, 131
configuring
example, 44
for a device, 43
for I8250, 142
for WD8003, 151
installing, 131
interrupt stubs
for block drivers, 145
for character drivers, 139
tasks performed, 135
interrupt tables
configuring size, 136
interrupt vectors
installing, 131
restoring, 132
interrupts
and unsupported processors, 119
configuring table size, 136
disabling, 136
disabling and enabling, 119, 122
support, 127
ioctlsocket() BSD function, 87
IP addresses
changing, 211
getting, 104
getting for host name, 81, 82
IPOPTIONS macro, 49
irhan() function
and block drivers, 145
and character drivers, 139
example, 148
example for block drivers, 147
example for character drivers, 139
IRinstall() function, 119, 131, 151
and init(), 141

and unsupported processors, 136
description, 132
Irnew() function
and CLEARIR(), 131
IRNO parameter, 43
IRrestore() function, 119
and shut(), 143, 152
and unsupported processors, 136
description, 132
irstub(), 119, 136

K

KEEPALIVETIME macro, 50

L

LFLAGS macro, 40
LIBR macro, 40
librarian, 40
libraries
building, 11
libraries, defining, 39
link layer, 52
called from PTABLE, 144
configuration example, 43
network configuration, 42
linker, 40
command-line options, 40
listen() BSD function, 88
example, 88
LNK macro, 40
local address
getting for socket, 84
local parameters, 47
local.h
configuration options, 47
local.h file, 12
and protocol selection, 38, 52
contents, 25
contents and location, 38
macros defined in, 118
LOCALHOSTNAME macro, 14, 51
LOCALSETUP macro, 51
LOCALSETUP() macro
and clock routines, 118, 119
LOCALSHUTOFF macro, 51
LOCALSHUTOFF() macro
and clock routines, 118
loopback test, 15, 113
low-level I/O, 120
LTEST, 15, 45
configuration requirements, 16
development requirements, 16
goals, 16
pass indicators, 16
LTTEST, 113

M

m68k, 136
and interrupts, 120
macros
AFLAGS, 40
AS, 40
CC, 40
CFLAGS, 40
COMPILER, 38
Dynamic Protocol Interface, 63
for device drivers, 130
for multitasking, 120
LFLAGS, 40
LIBR, 40
LNK, 40
LOCALHOSTNAME, 14
RTOS, 39
TRACE_DEBUG, 39
USROOTDIR, 38
makefiles
editing, 38
manuals, 10
mapioadd() routine, 120, 132
and unsupported processors, 136
MAXBUF macro, 48
Maxbuf parameter, 61
MAXIRNO parameter
configuring interrupt table size, 136
MB86960 processor, driver info, 197
MCRXTEST, 113
MCTXTEST, 113
message buffers, 128
messages
adding to a queue, 132
broadcasting, 68
reading from a connection, 61

receiving, 90, 94
receiving from connection, 92
receiving from socket, 89
removing from a queue, 134
sending, 97, 98, 99
sending to socket, 102
writing to a connection, 62
MESSH structure, 147
uses, 127
MIB2 macro, 50
MMODL macro, 40
modularity, 5
Motorola 68360
benchmarks, 162
Motorola-type addressing, 117
msghdr structure, 98
definition, 94
mtmacro.h file, for multitasking macros, 120
MTOS
configuring, 177
creating applications, 178
features used, 178
installation, 177
operating system notes, 177
testing, 178
MTTEST, 20, 114
configuration requirements, 21
development requirements, 22
goals, 20
pass indicators, 22
multicast, 108
BSD API, 102
DPI API, 66
multicast test, 113
MultiTask!, 121, 124
configuring, 180
creating applications, 181
features used, 181
installation, 179
operating system notes, 179
testing, 180
multitasker
porting, 120
multitasking
configuration, 121
macros for, 120
preemption, 122
signaling, 122
specifying type of, 121
yielding control, 122
multitasking macros
BLOCKPREE(), 122
RESUMEPREE(), 122
RUNTASK(), 121
WAITFOR(), 122
WAITNOMORE(), 122
YIELD(), 122
multitasking test, 20, 114

N

names
binding to sockets, 77
NAPT, 108
NAT, 108
NBUFFS macro, 48
Nclkinit() timer routine, 118
Nclkterm() timer routine, 118
Nclock() timer routine, 118, 119
Nclose() function, 28
and closE(), 141
description, 60
example, 60
NCONFIGS macro, 48
NCONNS macro, 48
used for signaling, 123
NDIS, 46
drivers, 46
NE_HWERR error code, 136
NE_PARAM error code, 136, 152
NE1000 processor, driver info, 198
NE2000 processor, driver info, 200
NE2000.C file
transmit interrupt examples, 145
NE2100 processor, driver info, 201
NET structure
code example, 129
description, 129
uses, 127
net.h file
contents, 25
defining adapters in, 43

- defining drivers in, 45
- defining link layer, 42
- defining protocols in, 52

netconf structure
- and dynamic configuration, 209

netconf.c, 13
- and netdata table, 40

netconf.c file, 11
- fields in each entry, 14
- selecting NDIS driver, 46

netconf[], 48

netdata, 40
- example, 44

netdata[], 48

netdata[] table
- and initialization, 25
- configuring, 13

network
- configuring, 151
- initialization, 54
- initializing interfaces, 55
- shutting down, 55, 152
- shutting down interfaces, 56
- starting, 151
- turning off, 152

network address, 41

Network Address Translation, 108

network applications, 103

network configuration table, 45, 51
- device parameters, 141, 151

network controller, 42
- clearing interrupt, 131
- configuring, 43
- drivers provided, 127
- identified to USNet via network name, 41
- interrupt, 145
- receiving data from, 135
- turning off, 143
- using for hostname, 51
- writing to, 135

network interfaces
- initializing, 55
- shutting down, 56

network name, 41

network structure, 41

network task
- and irhan() function, 147
- priority of, 122
- yeilding control, 122

networking application routines
- BOOTP, 104
- DHCP, 105
- RARP, 103
- summary list, 103
- Telnet, 107
- TFTP, 106

Nfarcpy()
- and irhan(), 147

Ngetchr() function, 119
- example, 119

Ninit(), 51

Ninit() function
- and initialization, 25
- description, 54
- example, 54

NNETS macro, 48
- used for signaling, 123

non-blocking operations
- example, 70

Nopen() function, 26
- and opeN(), 141
- description, 58
- examples, 59
- parameters, 26

Novell Ethernet, 46

Nportno() function, 28

Nputchr() function, 119
- and trace output, 175

Nread() function, 26, 27, 61, 62
- description, 61
- example, 61
- parameters, 27

NS8390 processor, driver info, 202

Nterm(), 51

Nterm() function, 28
- description, 55
- example, 55

NTRACE macro, 39, 175

Nwrite() function, 26, 27, 149
- and writE(), 140, 149
- description, 62

example, 62
parameters, 27

O

ODI drivers, 46
open
active, 57
passive, 57, 76, 88
opeN() routine, 141
description, 150
Opus Make, 9, 11
definition, 173

P

packets
exchanging, 169
short, 171
passive open, 26, 76, 88
definition, 173
PASSWD macro, 51
password parameter
EMTEST, 18
PCMCIA, 42
adapter, 153
device driver, 154
performance, 155
PING, 59, 114
PITEST, 115
point to point, 41
port address, 43
configuring
example, 44
device, 143
for I8250, 142
for WD8003, 151
port numbers, 57
example, 59
porting, 117
compiler, 117
processor, 117
Portinit() function
and init(), 141
and initialization, 25
description, 55
examples, 55
Portterm() function, 28
and shut(), 143
description, 56
examples, 56
PPP, 41
preemption
and multitasking, 122
processor, defining, 38
processors
benchmarks for, 161
configuring unsupported, 136
requirements, 4
target, 38
unsupported, 117, 119, 131
PRODLIST macro, 38
protocol stack, 57
and opening connections, 57
with block drivers, 145
with character drivers, 138
protocol table, 153
structure definition, 143
protocol.ini file
and NDIS drivers, 46
protocols, 103
link-level, 5
selecting, 52
PTABLE, 137
description, 144
example, 144
structure definition, 143
PTH symbol, 40
pulsevt()
multitasking example, 124

Q

QUEUE_EMPTY() macro
description, 134
example, 135
QUEUE_FULL() macro, 141
description, 133
example, 133
QUEUE_IN() macro, 141
and irhan() function, 147
description, 132
examples, 132
QUEUE_OUT() macro, 134
description, 134

example, 134
queues
  adding messages to, 132
  removing messages from, 134
  testing if empty, 134
  testing if full, 133

R

RAM
  fixed, 4
RARP, 41
  description, 103
RARPget() routine, 104
reaD(), 144
readme.txt file, 127
readsocket() BSD function, 89
recv() BSD function, 90
  example, 91
recvfrom() BSD function, 92
  example, 93
recvmsg() BSD function, 94
relay test, 115
RELAYING macro, 50
remote address
  getting for a socket, 83
resources, and signaling, 123
RESUMEPREE() multitasking macro, 122
ROM, 5
routing table configuration entries
  adding, 211
  deleting, 211
  finding, 210
routing table configuration functions
  ConfAdd(), 211
  ConfDel(), 211
  ConfDisplay(), 212
  ConfFind(), 210
  ConfFree(), 210
  ConfLock(), 210
  ConfRename(), 211
RTOS, 2, 177
  and MTTEST, 20
  definition, 173
  porting, 120
  supported, 39
RTOS macro, 39
RUNTASK() multitasking macro, 121
RYTEST, 115

S

screen(), 144
segmented architectures, 117
selectsocket() BSD function, 95
  example, 96
semaphores, and signaling, 123
send() BSD function, 97
  example, 97
sendmsg() BSD function, 98
sendto() BSD function, 99
  example, 99
serial drivers
  and character drivers, 137
  provided, 127
serial FIFO buffer, 138
serial network, 43
  configuring
    example, 43, 44
    multiple connections, 40
server
  defining, 24
  required features, 25
  role of, 24
  terminating USNet, 28
server.c file
  and include files, 25
  code to add, 27
  compiling, 29
servers
  client in same host, 120
  FTP, 106
  running multiple, 120
  starting, 106
  Telnet, 107, 108
  with FTTEST, 112
  with PING test, 114
set_semaphore()
  multitasking example, 124
setsockopt() BSD function
  example, 86
shared buffer address, 43
  WD8003, 151

shut() routine, 152
description, 143, 152
example, 143, 152
shutdown() BSD function, 100
signaling
for multitasking, 122
WAITFOR(), 122
silly window syndrome, 171
sliding window
for flow control, 169
SLIP
configuring, 42
example, 43
SMC91C92 processor, driver info, 204
SNMP agent, 50
sockaddr structure, 73
sockaddr_in structure, 73
socket interface, 71
socket test, 115
socket() BSD function, 101
example, 101
socket.h file, 25
SOCKET_BLOCK() macro
description, 64
SOCKET_CANSEND() macro
description, 65
SOCKET_FIN() macro
description, 66
SOCKET_HASDATA() macro
description, 64
SOCKET_IPADDR() macro
description, 65
SOCKET_ISOPEN() macro
description, 64
SOCKET_MAXDAT() macro
description, 65
SOCKET_NOBLOCK() macro
description, 64
SOCKET_OWNIPADDR() macro
description, 66
SOCKET_PUSH() macro
description, 66
SOCKET_RXTOUT() macro
description, 65
SOCKET_TESTFIN() macro
description, 65
sockets
accepting connections on, 76
binding names to, 77
blocking, 64
closing, 78
controlling flags, 80
creating, 101
getting local address for, 84
getting options, 85
getting remote address for, 83
initiating a connection on, 79
non-blocking, 64, 80
receiving messages, 89
sending messages to, 102
setting control parameters for, 87
setting options, 85
testing, 115
waiting for activity on, 95
SOTEST, 115
subnet mask, 41
support.h file, contents of, 25

T

target system, 2
design, 7
target system specifications, 13
tasks
defining function type, 121
TCP
and flow control, 57
and FTTEST, 112
and MTTEST, 20
and SOTEST, 115
compared to UDP, 23
compared with UDP, 57
data rate tests, 163
definition, 173
delayed ACKs, 170
file transfer example, 69
flow control, 169
retransmission, 168
timeout, 168
TCP/IP, 167
and PING, 114
and relaying, 50
benchmarks, 155

embedded, 167
protocol relationships, 2
protocols supported, 1
size, 4
user interface, 71
TCP_SACK macro, 51
Telnet
client, 108
description, 107
programs, 107
server, 108
TELNET, 116
terminating USNet, 28
terminology, 173
test programs, 12, 111
test programs, summary list, 111
testing, 175
integration, 12
overview, 111
TFTP
and FTTEST, 112
definition, 174
description, 106
example, 106
multitasking, 121
TFTPget() routine, 107
TFTPput() routine, 106
TFTPserv() routine, 106
TimeMS() function
and WAITFOR(), 123
timeout
non-multitasking system, 120
WAITFOR(), 122, 124
timer interrupts, 118
timer routines
summary list, 118
timeval structure, 95
TNSERV, 116
toolchain
specifying path to, 39
trace
and EMTEST results, 19
and FTTEST, 112
and LTEST, 15
and LTEST results, 16
displaying output, 175
field definitions, 175
Nputchr(), 119
output from PING, 114
overview, 175
trace output, 175
TRACE_DEBUG macro, 7, 39
and application development, 35
and EMTEST configuration, 18
and FTTEST, 112
and MTOS, 178
and MTTEST, 181, 183
and testing, 13
and VRTX, 186
settings, 175
transmission
timeout, 168
transmit interrupt, 145, 150, 157
and irhan() function, 147
transmit routine
description, 149
examples, 140, 149
writeE(), 140
transmitter empty, 139, 140
transmitting
routines for, 149
TRG_ID macro, 40
true events
and WAITFOR(), 125
Two-Hop Routing
benchmarks, 163

U

UDP
and FTTEST, 112
and SOTEST, 115
compared to TCP, 23
compared with TCP, 57
data rate tests, 163
definition, 174
UNIX
and FTTEST, 112
and PING, 114
display and keyboard support, 119
installing USNet on, 9
porting from, 72
porting to, 72

running Telnet, 107
sockets, 71
timers, 118
unsupported processor, 131
interrupt handling, 119, 136
updating
marking start of, 210
terminating, 210
user
authenticating, 51
identifying, 51
USER_INCS macro, 39
USER_LIBS macro, 39
USERID macro, 51
userid parameter
EMTEST, 18
USNet
design, 3
overview, 1
USROOTDIR, defining, 38
USS_IP_MC_LEVEL macro, 49, 113
USS_PROXYARP macro, 51
USSBUFALIGN macro, 49
ussHostGroupJoin, 67
ussHostGroupLeave, 67
UXSERV, 116

V

version, 10
VRTX
configuring, 185
creating applications, 186
features used, 186
installation, 185
operating system notes, 184
testing, 185
vsnlog.txt, 10

W

WAITFOR() multitasking macro
description, 122
example, 124
WAITNOMORE() multitasking macro
description, 122
WAITNOMORE_IR() multitasking macro
and irhan() function, 147
description, 122
WD8003 processor, 43, 44
and close connection, 151
and open connection, 151
block driver, 144, 147
network configuration, 43
PTABLE example, 153
receiving messages, 141
transmit interrupt examples, 145
WD8003 processor, driver info, 205
window
exhausted, 170
for flow control, 169
silly window syndrome, 171
Windows
installing USNet on, 9
WRAP driver, 45
writE() routine
and block drivers, 149
description, 140
example, 140
without transmit interrupt, 145
writesocket() BSD function, 102
wteset()
multitasking example, 124

Y

YIELD() multitasking macro, 122, 123