smxNS™
User's Manual

Version 3.10

February 2024

U S SOFTWARE
EMBEDDED EXCELLENCE

Micro Digital

Copyright and Trademark Information

For support contact Micro Digital.

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.

Revision History

Revision	Date	Notes
2.58	April 2006	Based on USNet 2.52.1 June 2005.
2.58	January 2007	Updated for new features.
2.59	July 2007	Added PPP, SNMP, and Web Server sections.
2.59	September 2007	Updated information about add-ons and demos.
2.59	February 2008	Added RTOS porting information.
2.60	July 2008	Updated naming, described Telnet debug.
2.63	October 2009	Updated naming, APIs, debug information.
2.63	July 2010	Updated naming, APIs.
2.70	October 2010	Changed porting layer to use smxBase.
2.70	August 2011	Updates, corrections, and new sections.
2.80	December 2013	Updates and addition of IPv6, mDNS Responder, and others.
2.81	July 2014	Added notes to DHCP server, DNS client, MTU.
2.90	July 2015	Updates for DPI error codes, CGI, SNMP and misc clarifications.
2.90	October 2015	Corrections and updates.
2.91	February 2017	Corrections and updates.
2.92	February 2018	Adds NC-SI, updates SNMP, misc updates.
2.92	September 2018	Portconfig() options, web server functions, AJAX/jQuery, misc
3.10	February 2024	Porting information removed.

Contents

1. Introduction 1

Overview 1

What is Supplied 2

smxNS Design Considerations 3

Size 3
Clarity 3
External Support 4
Packaging 4
Reentrancy 4
ROM Residence 4
Device Drivers 4
Modularity 4

Recommended Reading 5

Books 5
RFCs Supported 6

Your Experience 8

Overview of the Development Process 8

Analyzing the Design Problem 9
Obtaining Design Tools and Verifying Your System 9

2. Quick Start 10

Directory Structure 10
Version 10
Documentation 10

Configuration 10

Building the smxNS Code 11

Running the Main Test Programs 11

Guidelines for Testing 11
nsdemo 11

3. Beginning Your Application 17

Developing a Simple Application 17

Include Files 19
Initializing smxNS 19
Establishing a Connection 20
Terminating smxNS 22
Compiling Your Application 24
Code Listings 24

Developing Your Application **31**

4. Configuration 33

Overview **33**

Configuring the Build Settings (nscfg.h) **34**

Configuring Local Parameters (nscfg.h) **34**

SNS_MIN_RAM Macro 35
SNS_HW_RX_CHECKSUM Macro 35
SNS_HW_TX_CHECKSUM Macro 35
SNS_CPU_CACHE_DATA Macro 35
SNS_BUFFS_IN_SRAM Macro 35
NCONNS Macro 36
NBUFFS Macro 36
MTU Macro 36
MAX_REASSEM Macro 36
USSBUFALIGN Macro 36
FRAGMENTATION Macro 37
IPOPTIONS Macro 37
USS_IP_MC_LEVEL Macro 37
IP_MC_DFLT_NETNO Macro 37
KEEPALIVETIME Macro 38
RELAYING Macro 38
chksum_INASM Macro 38
DNS Macro 38
NDNSS Macro 38
TCP_SACK Macro 39
LOCALHOSTNAME Macro 39
USERID Macro & PASSWD Macro 39
USS_PROXYARP Macro 39
FILE_SUPPORT Macro 39
SNS_DEBUG_LEVEL Macro 40
NNETS Macro 40
NNETISRS Macro 40

Selecting Protocols **40**

Selecting Drivers **41**

5. Dynamic Protocol Interface 43

Overview **43**

Blocking Versus Non-Blocking Operation **44**

Include Files **44**

Initialization and Termination **44**

Ninit 44
Nterm 45
Portcreate 45
Portconfig 46

Portinit....49
Portstate....50
Portterm....51

Connections....51
Open, Close, Read, and Write....52
Nopen....54
Nclose....56
Nread....57
Nwrite....58
Dynamic Protocol Interface Macros....59
SOCKET_NOBLOCK....60
SOCKET_BLOCK....60
SOCKET_ISOPEN....60
SOCKET_HASDATA....60
SOCKET_CANSEND....61
SOCKET_ISSENDING....61
SOCKET_TESTFIN....61
SOCKET_ISFATAL....61
SOCKET_MAXDAT....62
SOCKET_RXTOUT....62
SOCKET_REMADDR....62
SOCKET_LOCADDR....62
SOCKET_REMPORT....63
SOCKET_LOCPORT....63
SOCKET_PUSH....63
SOCKET_FIN....63
SOCKET_FAMILY....63
SOCKET_HASMYADDR6....64
SOCKET_LOCSITEADDR6....64
SOCKET_REMADDR6....64
SOCKET_LOCLINKADDR6....64

Multicast API (DPI)....65
ussHostGroupJoin....65
ussHostGroupLeave....65

Error Handling....66

Examples....66
Broadcasting Examples....67
TCP File Transfer Example....67
Non-Blocking Operations Examples....68

6. BSD Socket Interface....71

About BSD Sockets....71

Structures and Definitions....72

BSD Socket Interface Functions....72
accept....75
bind....76
closesocket....77

connect 78
fcntlsocket 79
freeaddrinfo 79
gai_strerror 80
getaddrinfo 81
getpeername 83
getsockname 84
getsockopt, setsockopt 85
inet_ntop 87
inet_pton 88
ioctlsocket 89
listen 90
readsocket 91
recv 92
recvfrom 94
recvmsg 95
selectsocket 96
send 98
sendmsg 100
sendto 101
shutdown 102
socket 103
writesocket 104

Multicast API (BSD) 105

7. Network Applications and Protocols 107

Overview 107

ARP 108
Proxy ARP 108

DHCP 109
DHCP Client Configuration 109
DHCP Server Configuration 111
DHCP Server Operation Restrictions 112
DHCP Testing 113

DNS 116
SetDNS() 116
DNSresolve() 116

FTP and TFTP 117
Start Server 117
Send File 118
Receive File 118

HTTP Client 119
Retrieve a Web Page 120
Web Page Callback Function 120

IGMP / Multicast 120

iperf 121

IPv6 ..... **121**

mDNS Responder **122**

NAT **126**
NAT Configuration 126

NC-SI **128**

PPPoE **129**
PPPoE Configuration 129

SLIP **131**
Using SLIP with Windows Computers 131

SMTP **132**

SNTP **134**
Get Time using SNTP 134

Telnet **135**

8. Point To Point Protocol (PPP) 137

Overview **137**

PPP in Theory **137**
LCP Phase 138
Authentication Phase (PAP/CHAP) 138
NCP Phase 138

PPP in Practice **139**
Usage 139
Configuration 140
Scripting 143
Notes on Special Cases 146

PPP ioctl Routines **154**
Description 154
Option Listing 154
Using PPP ioctl() routines 156

PPP dialapi Routines **160**
Description 160
Definitions of API 160
Dynamically Configuring smxNS PPP Dial Scripts 161

PPP pppsig Routines **162**
Description 162
Definition of Signals Available 162
Using PPP Signaling Routines 163

9. Simple Network Management Protocol (SNMP) 165

Introduction **165**
SNMP Overview 165
Design of smxNS SNMP 166

Building an Application **167**
Build-time Configuration 167
Agent Use of Build-time Constants 171
Application Interface 172

Customizing the Agent **180**
Configuring the Agent MIB 180
Adding New MIBs 185
Configuring the Transport Mapping 193
Exercising the Agent 195

10. Web Server 197

Web Server Overview **197**
Web Server Requirements 198

Example Web Server **198**
Building the Example Web Server for Your Target 198
Connecting to the Example Web Server 199
Adding Web Pages Using a File System 199

Using the Web Server **200**
User Server Functions 200

HTTP Server Request Structure **204**
Modules and Handlers 206

Module Function Descriptions **207**
MODchkaccess() 207
MODchkauth() 208
MODchkloc() 209
MODchktype() 209
MODgetuser() 210
MODlog() 211
MODtranslate() 211

Request Structure **212**
Using nsbldpg 213
Server Configuration File 213
MIME Information 219
AddType Command 220
Page Configuration File 221
Variable Configuration File 223
Access Configuration File 224

CGI Function Programming Interface **225**
System Support Routines 226
CGI Routines 231
CGI Environment Variables 237

USMETA Programming Interface **240**
#echo 241
#exec 242
#include 243
#memory 244

#system 244
AJAX and jQuery 245

11. Device Drivers 247

Overview 247

Data Structures....247
Messh (MESSH) Structure....248
Net (NET) Structure....249

Support Functions....250
Disable and Enable Interrupts....250
Install Interrupt Vector....251
Restore Interrupt Vector....251
Map I/O Address....251
Adding Messages to a Queue....251
Removing Messages from a Queue 253
Interrupt Handling....254
Interacting with an Ethernet PHY 254

Configuring a New Processor....255

Error Codes 255

Writing a Device Driver....255

Character Drivers 255
Interrupt Handler....257
Transmit Routine 258
Open Connection 259
Close Connection 260
Configure and Start Up 260
Shut Down 261
Network Protocol Table....262

Block Drivers 263
Interrupt Handler....264
Transmit Routine 267
Configure and Start Up 270
PHY Support Functions 271
Polling....273
Shut Down 274
Protocol Table....275

12. Technical Background....277

Overview 277

TCP Retransmission 277

Sliding Window 278

TCP Delayed ACK....280

Congestion Control 280

Silly Window Syndrome **281**
TCP Window Probe **281**
Address Conflict Detection **281**
ARP Caching **282**

A. Terminology **283**

B. Debugging Techniques **285**

Overview **285**
Displaying Trace Data **285**
Debug over Telnet **287**
arpstat: Dump the ARP Table 287
bufstat: Display Details for Frame Buffers 288
ifstat: Display Network Interface State 289
logdump: Display smxNS Log 290
memdump: Display Memory 290
netstat: Display Connection Status 290
nqstat: Show the State of Connections 291
routestat: Display Routing Information 292
Other Commands 292
Network Analyzers **292**
Windows Utilities **293**
Web Servers **293**
Verification Testing **293**

C. Dynamic Configuration **295**

Overview **295**
Configuration Functions **295**
SetDefaultRouter 295

D. Driver-Specific Information **297**

ACT10100 ..... **297**
AT91 ..... **297**
CFFEC **298**
DC21140 ..... **298**
EP93XX **300**
I8255X **301**
LAN91CXXX **302**

LM3S **303**
LPC2XXX **304**
NE2000 ..... **304**
RTL8139 ..... **306**
STRXXX **306**
USBD **307**
USBH **307**
WiFi **308**

E. Serialized MAC Addresses **310**

F. Memory Usage and Performance **311**
Memory Usage (KB) **311**
Performance **312**

Index **313**

1. Introduction

Overview

smxNS™ began from USNet® v2.58. Much has been changed, and improvements continue to be made.

smxNS is a set of software routines that support TCP/IP protocols and runs on SMX RTOS. It supports the TCP/IP protocols shown in Table 1-1.

Table 1-1: smxNS 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. Both IPv4 and IPv6 are supported.
ICMP	Internet Control Message Protocol: Part of IP for practical purposes
ARP	Address Resolution Protocol: Retrieves a host's netwo controller's hardware address, given the host's Interne address

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

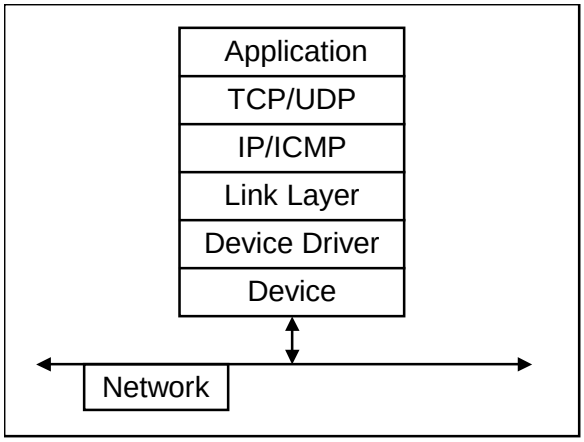


Figure 1-1: Protocol Stack

smxNS's TCP/IP protocol suite allows diverse systems to communicate with each other. It implements a dual IPv4/IPv6 stack. IPv4 support and IPv6 support can be enabled individually or together. More information about IPv6 is presented in the IPv6 section of Chapter 7, Network Applications and Protocols.

Typically, smxNS 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. smxNS takes on the responsibility of providing a reliable connection and reliable data transport when using TCP/IP.

smxNS 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

smxNS includes full source code and sample application protocols and test programs that are useful when building networking into your application.

nsdemo.c includes support for the following protocols:

- FTP client
- FTP server
- Loopback test (exercises core stack protocols)
- DHCP server
- mDNS Responder
- Ping client

- POP email retrieval
- SMTP email sending
- SMTP email server
- SNMP agent
- Telnet server
- TFTP client (like FTP file transfer, but using UDP)
- TFTP server
- Web server

Support for these protocols can be turned on and off using switches at the top of the file.

nstels.c is a simpler application that includes support for a Telnet server.

smxNS Design Considerations

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

- Size
- 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. 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. Stack usage is kept to a minimum by avoiding deep function nesting and excessive autovariables.

Clarity

The main code does not contain any conditional controls for different compilers or processors. Only some of the porting files have code of this 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.

Packaging

smxNS 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.

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

smxNS 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, or PPP.

Modularity

In addition to the main stack, smxNS offers various add-on modules, such as a web server, NAT support, mDNS responder, and SNMP. By separating these from the main stack, you are saved cost and memory space by omitting them if they are not needed.

Recommended Reading

This manual documents smxNS 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

SNMP, SNMPv2, SNMPv3, and RMON 1 and 2
Practical Network Management
William Stallings
ISBN 0-201-48534-6

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

Foundations of WWW Programming with HTML & CGI
IDG Books
ISBN 1-56884-703-3

CGI Programming in C and Perl
Thomas Boutell
Addison Wesly
ISBN 0-201-42219-0

CGI Developers Guide
Eugene Eric Kim
Sams Net
ISBN 1-57521-087-8

Zero Configuration Networking
The Defnitive Guide
Stuart Cheshire & Daniel H. Steinberg
O'Reilly
ISBN 0-596-10100-7

There are many books on web page design. This one is very good for low-level protocols, and has cross-references to RFCs:

Internet Protocols Handbook
Dave Roberts
Coriolis Group Books
ISBN 1-883577-88-8

RFCs Supported

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 by searching with a web browser. The most important ones for smxNS are:

RFC 768	UDP
RFC 791	IP
RFC 792	ICMP
RFC 793	TCP
RFC 821	SMTP
RFC 822	SMTP
RFC 959	File Transfer Protocol
RFC 1034	DNS
RFC 1035	Domain Names - Implementation and Specification
RFC 1101	DNS
RFC 1112	Host Extensions for IP Multicasting
RFC 1122	Explanations and clarifications of all the above, plus additions and corrections
RFC 1144	Compressing TCP/IP Headers for Low-Speed Serial Links
RFC 1157	Simple Network Management Protocol (SNMP)
RFC 1213	SNMP MIB-II
RFC 1320	The MD4 Message-Digest Algorithm

RFC 1321 The MD5 Message-Digest Algorithm

RFC 1332 The PPP Internet Protocol Control Protocol (IPCP)

RFC 1334 PPP Authentication Protocols

RFC 1661 The Point-to-Point Protocol (PPP)

RFC 1662 PPP in HDLC-like Framing

RFC 1725 POP

RFC 1867 Form-based File Upload in HTML

RFC 1869 SMTP

RFC 1876 DNS

RFC 1982 DNS

RFC 1989 PPP Link Quality Monitoring

RFC 1990 The PPP Multilink Protocol (MP)

RFC 1994 PPP Challenge Handshake Authentication Protocol (CHAP)

RFC 2018 TCP Selective Acknowledgment Options

RFC 2045 MIME: Format of Internet Message Bodies

RFC 2046 MIME

RFC 2047 MIME

RFC 2048 MIME

RFC 2049 MIME

RFC 2065 DNS

RFC 2068 HTTP

RFC 2131 Dynamic Host Configuration Protocol

RFC 2132 DHCP Options and BOOTP Vendor Extensions

RFC 2236 Internet Group Management Protocol, Version 2

RFC 2433 Microsoft PPP CHAP Extensions

RFC 2461 Neighbor Discovery for IPv6

RFC 2462 IPv6 Stateless Address Autoconfiguration

RFC 2463 ICMPv6

RFC 2516 A Method for Transmitting PPP Over Ethernet (PPPoE)

RFC 2663 IP Network Address Tranlator (NAT) Terminology and Considerations

RFC 2863 The Interfaces Group MIB

RFC 3411 An Architecture for Describing SNMP Management Frameworks

RFC 3414 User-based Security Model for SNMPv3

RFC 3174 Secure Hash Algorithm 1 (SHA1)

RFC 3826 The AES Cipher Algorithm in the SNMP User-based Security Model

RFC 3927 Dynamic Configuration of IPv4 Link-Local Addresses

RFC 4022 Management Information Base for TCP

RFC 4292 IP Forwarding Table MIB

RFC 4293 Management Information Base for IP

RFC 5227 IPv4 Address Conflict Detection

RFC 5322 Internet Message Format

RFC 5681 TCP Congestion Control

RFC 6056 Recommendations for Transport-Protocol Port Randomization

RFC 6234 US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)

RFC 6762 Multicast DNS

RFC 6763 DNS-Based Service Discovery

Your Experience

This manual assumes you are familiar with TCP/IP and related protocols, C programming, your compiler toolsuite, 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.

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 smxNS.

These are the main 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 your SMX release, which includes smxNS.
4. Verify that the Network Controller hardware, network servers, and network cables are functional.
5. Add XNS files and paths to the application project if not already there.
6. Build and run the SMX Protosystem with the smxNS demo enabled (\SMX\APP\DEMO\nsdemo.c).
7. Develop and debug your application.

8. Generate your production code. Set the macro **SNS_DEBUG_LEVEL** in nscfg.h to 0 (to optimize code space). Configure Ethernet interfaces with the ENA option so that each device uses a unique MAC address.

Steps 1 and 2 are covered in the remainder of this chapter. The remaining steps are discussed in the following chapters.

Analyzing the Design Problem

Proper configuration of smxNS 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. 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 smxNS and SMX 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

Directory Structure

smxNS is organized in a hierarchical directory structure under \SMX\XNS, as shown:

doc	Additional documentation
drvsrc	Drivers and CPU support
<*cpu*>	CPU-specific files
include	smxNS header files
netsrc	Core smxNS source code
supsrc	Low-level code common across other products in this family

Other directories may be present if you have purchased smxNS add-on packages.

ipv6src	smxNS Internet Protocol version 6
pppsrc	smxNS PPP support package
snmpsrc	smxNS SNMP package
websrc	smxNS Web Server package

Version

The smxNS version number is indicated by SNS_VERSION in \SMX\XNS\include\smxns.h.

Documentation

Manuals are supplied in PDF format at www.smxrtos.com/doc. Also see the text files in the SMX\XNS\doc directory for important additional information.

Release notes are supplied in the \SMX\DOC and \SMX\XNS\doc directories. Please take time to review these files.

Configuration

It should be possible to run the packaged smxNS demo program with few or no changes. The IP address of the system running smxNS is set with the LOCALIP macro at the top of the **nsdemo.c** file in the \SMX\APP\DEMO directory.

If LOCALIP is set to 0.0.0.0, then smxNS will retrieve an IP address from a DHCP server, or you could set this to an appropriate fixed address for your network. Other commonly adjusted settings are collected at the top of nsdemo.c. There is a series of macros that specify which clients and servers the test program will launch. There are also test specific settings such as the IP address of a test FTP server, and account information for logging in to the test FTP server.

Other smxNS configuration options are documented in **Chapter 4, *Configuration***.

Building the smxNS Code

Add the XNS source files and paths to the application project, if not already in it.

One source code file might require modification in order to run smxNS's test programs. **nscfg.h**, resides in the **include** directory and is used to define how smxNS 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, neither file should need to be modified. File **nscfg.h** and its parameters are described in section *Configuring Local Parameters* of Chapter 4, *Configuration*.

Running the Main Test Programs

Test programs which were separate in USNet have been combined into a single demo file **nsdemo.c** in the \SMX\APP\DEMO directory.

Guidelines for Testing

- Test using the smxNS trace output. (See Appendix B, *Displaying Trace Data*.)
- 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 smxNS 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.
- If you make experimental changes to the test program, 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 SNS_DEBUG_LEVEL = 3 in nscfg.h to help report error conditions in the stack. Do a **grep** or **search** on "DEBUG_MSG" in the stack modules to locate error traps.
- The header file **net.h** contains error return number translations and meanings.
- Use the function ***Nprintf()*** *or* ***Nputstr()*** 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.

nsdemo

nsdemo.c combines several test programs and example servers into one demo. The specific tests are controlled by a series of #define switches at the top of the file, and these are summarized below.

TEST_CRYPTO	Run a test to confirm cryptographic functions.
TEST_DHCP_SERVER	Start a DHCP server.
TEST_FTP_CLIENT	Run an FTP client that continuously uploads and downloads a test file.
TEST_FTP_SERVER	Start an FTP server.

TEST_LOOPBACK	Test the core of the TCP/IP stack.
TEST_MDNS_RESP	Start an mDNS Responder.
TEST_PING_CLIENT	Run a Ping client.
TEST_POP_RECEIVE	Retrieve an email message from a POP server.
TEST_SMTP_SEND	Send an email message using an SMTP server.
TEST_SMTP_SERVER	Start an SMTP email server.
TEST_SNMP_AGENT	Start an SNMP agent so that the smxNS system will respond to queries from an SNMP manager.
TEST_SNTP_CLIENT	Run the SNTP client that will retrieve the current time from a time server.
TEST_SSL_SERVER	Start a version of the Web Server that uses the Secure Sockets Layer.
TEST_TELNET_SERVER	Start a Telnet server. The Telnet server provides a simple command line that allows the state of the network stack to be displayed.
TEST_TTCP_SERVER	Run the TTCP server so that network performance can be measured.
TEST_WEB_SERVER	Start a Web server that will respond with canned web pages.

See Chapter 7 Network Applications and Protocols for information about these. The sections below give details about these tests.

To run nsdemo, add it to the SMX Protosystem project. Also add compiler command line defines for SMXNS and SMXNS_DEMO.

The nsdemo application and the smxNS stack provide feedback by logging messages with the DEBUG_MSG() macro. For example, this line appears in nsdemo.c:

```
DEBUG_MSG2_PAR0("smxNS Portinit for enet Failed\n");
```

These log messages are sent to both the debug terminal and the smxAware print buffer. The debug terminal output is usually sent to an RS232 port on your target.

The debug macro is of the form DEBUG_MSGd_PARp, where d is the debug level from 1 to 6, and p is the number of parameters in the format string from 0 to 10.

When running under an IDE, the strings directed to the smxAware print buffer can be reviewed by opening the smxAware text display window and expanding the Print node in the object list. Viewing the log messages this way allows you to see all of the most recent trace messages, and is more useful for debugging.

FTP Client Test Overview

The FTP client test sets up the system under test to act as an FTP client. The system writes a file to an FTP server, and then reads it back and verifies that the data has been transferred correctly. This test will run in a continuous loop until the Escape key on the keyboard is pressed.

If you don't already have an FTP server in the local network, the following are freely available and relatively easy to set up:

- **FileZilla Server**, available at http://sourceforge.net/projects/filezilla/

- **War FTP Daemon**, available at http://www.warftp.org/

The following definitions in nsdemo.c should be reviewed before running the FTP client. You will likely need to adjust these definitions and perhaps set up a user account under your FTP server in order for the FTP client test to run successfully.

FTPSERVER	IP address or DNS name for the FTP server.
FTPUSERID	User name for the FTP account.
FTPPASSWD	Password for the FTP account.

FTP Client Pass Indicator

If all is going well, you should see status report messages similar to the following.

```
9 FTtest OK
```

This indicates that the test program has completed 9 passes in which the test file has been uploaded to the FTP server and then downloaded and compared. Additional information may be available from the log messages or from the FTP server's user interface.

If the test is not successful, you could verify that the FTP user account settings are working by playing the role of the test program and logging into the server from a command line. For more detailed debugging, you could increase the setting of SNS_DEBUG_LEVEL in XNS\include\nscfg.h for more verbose logging, and you could review network activity using a network sniffer.

FTP Server Test Overview

The FTP server demonstrates the use of ftp_session_main() function to implement an FTP server. The ftp_session_main() function handles all aspects of an FTP session with a client once the control connection has been established. File system support may be provided either through the minimal RAM based file system, or through traditional file system support such as smxFS.

Loopback Test Overview

The loopback test uses a wrap driver while executing read/write tests on your target. 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.

The loopback test sends trace update messages during execution, and if the test is successful it will display about 30 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().

In order to set up the loopback test, follow these steps.

1. Edit APP\DEMO\nsdemo.c so that just TEST_BASE_NETWORK and TEST_LOOPBACK are enabled
2. Compile and download the top level project
3. Start execution and allow it to run for about 20 seconds

Loopback Test Pass Indicators

The Loopback test will display the following trace output if the test passed.

This concise listing was created with the SNS_DEBUG_LEVEL constant set to 3 in nscfg.h.

```
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 the Loopback Test

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.

POP Email Retrieval Test Overview

The POP email retrieval test will call the POPreceive() function to retrieve an email message from an email server. The following items should be configured for this test.

TEST_POP_RECEIVE	Set to 1 to launch the test.
TEST_POP_SERVER	The IP address or URL of the POP server.
TEST_POP_USER	The user name for the account on the POP server.
TEST_POP_PASSWORD	The password for the account on the POP server.

SMTP Email Send Test Overview

The SMTP send test will call the SMTPsend() function to send a canned email message to an SMTP server. The following items should be configured for this test.

TEST_SMTP_SEND	Set to 1 to perform test.
TEST_EMAIL_ADDRESS	The email address to which the test message will be sent. This address is parsed in order to determine the SMTP server that is used when sending the message.
TEST_SMTP_FLAGS	Normally set to 0. Set to SMTP_USE_SSL to use SMTP over SSL. This requires an SSL library.
MULTIPART	Set to 1 to send a multipart message.

SMTP Server Test Overview

An SMTP server may be launched as part of nsdemo. Log messages will be written as email messages are received.

SNMP Agent Test Overview

The SNMP agent may be started as one of the tasks run in nsdemo. The SNMP agent acts as a server and provides network status information in response to requests sent by a Network Manager application. The SNMP agent is discussed further in Chapter 9.

Telnet Server Test Overview

A Telnet server may be launched as part of nsdemo. telnetd_task_main() sets up connections, and sns_TelnetCli() interprets command lines and provides a response.

The Telnet session function sns_TelnetSessionMain() provides the following functions.

- It exchanges some basic control information with the client.
- It provides simple editing by allowing the backspace key to remove the last character typed from the command line, and then redisplays the line.
- It reads command lines, and calls the routine sns_TelnetCli() for each. The example in nsdemo.c passes the command to sns_DebugCli() so that the command can be processed by the debug interpreter. If the optional HTTP client module is configured (SNS_PROTO_HTTPC = 1), then instead of running the command through the debug interpreter, the command string is interpreted as a URL, and the HTTP client function attempts to retrieve the web page at that location. nstels.c also implements a Telnet client, and if this version is built rather than nsdemo.c, then the Telnet command processor simply echoes back the command that was received.
- If sns_TelnetCli() returns 0, then the telnet session will continue supplying command lines. If sns_TelnetCli() returns –1, the session will be closed. The example in nsdemo.c returns –1 when it is called with the string “quit”.

The command interpreter in nsdemo.c or nstels.c could serve as a starting place for a full featured command line interface for the system running smxNS.

Web Server Test Overview

The Web server may be launched as part of nsdemo. The following configuration items may be useful.

TEST_WEB_SERVER	Set to 1 to enable the Web server.
LOCALIP	The Web server will be accessible at this address.
NUM_WEBS_TASKS	This specifies how many tasks will be launched to fulfill individual requests to retrieve a resource. Setting this to 0 will fulfill a request in the context of the main Web server task.

When the Web server is running correctly, you should be able to enter the IP address of the system running smxNS as the URL in a web browser, and a default web page should be displayed. More information on the Web server is presented in Chapter 10.

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 code presented in this section is intended to illustrate smxNS'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 smxNS'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 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 smxNS'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 */
   .
   .
   .
u16 count;                 /* Index counter */
u8 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 )
   DEBUG_MSG2_PAR0(" 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. Since both the server and client will be running an image built from the smxNS source code, some small differences need to be established so that one system will act as the server and the other as the client.

The server will be running the server application that listens for a network connection and the client will be running the client application that establishes the connection and makes requests of the server. This can be accomplished by using files specific to these network applications when building the top level project. These files will also establish IP addresses and port numbers for the server and client. In this example, one application file is called nsserver.c and the other is nsclient.c.

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 local 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 CLIENT_IP      "10.1.1.2"  /* Client IP address */
#define SERVER_IP      "10.1.1.3"  /* Server IP address */
#define CLIENT_MAC "00:01:02:03:04:05" /* Client hardware addr */
#define SERVER_MAC "00:01:02:03:04:06" /* Server hardware addr */
#define LOCALMASK "255.255.255.0"  /* Set subnet mask here */
#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 **nscs.h**. A listing of **nscs.h** is included at the end of this section. 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 **nsserver.c** and the client, **nsclient.c**. Both these files must reside in the **demo** directory. Any program using smxNS requires four main features:

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

Include Files

One smxNS header file must be included at the top of **nsserver.c**. The file is:

```
   .
   .
   .
#include "smxns.h"              /* Prototypes and definitions */

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

The file **smxns.h** in turn #includes the files **nscfg.h**, **net.h**, **mtmacro.h**, **support.h**, and **socket.h**. The file **nscfg.h** contains smxNS's configuration, such as the number of physical connections, buffers, and options. The file **net.h** contains the function prototype information and type definitions. The file **mtmacro.h** contains definitions associated with the multitasking environment. The file **support.h** contains prototypes of internal support functions. Finally, the file **socket.h** specifies the BSD sockets API. If the application requires any application-specific information stored in a header file, that file should also be included.

Initializing smxNS

Two functions are required to initialize smxNS. The first, ***Ninit()***, will zero all data structures, move the `netdata[]` table from ROM to RAM, and initialize the stack. This is called by SMX module initialization and doesn't need to be included in the network application. The second initialization function, ***Portinit()***, initializes a network interface driver and prepares it for sending and receiving network frames.

In smxNS, the call to Ninit() is integrated with the rest of the system start up. The calling sequence is as follows.

ainit() [implemented in main.c]

Calls smx_modules_init() [implemented in smxmods.c]

Calls smxns_init() [implemented in smxmods.c]

and smxns_init() calls Ninit(). If there is a fatal error in networking initialization, system start up will fail. Otherwise, ainit() continues to launch applications.

ainit() [implemented in main.c]

Calls appl_init() [implemented in app.c]

Calls nsdemo_init() [implemented in nsdemo.c]

and nsdemo_init() launches the test program tasks.

When shutting down, a similar process occurs. Here the chain is exitx_main() calls smx_modules_exit() calls smxns_exit() calls Nterm().

From the perspective of the application developer, the network application code can be considered to start with a function modeled on nsdemo_init(), which typically launches one or more tasks that create and use network sockets.

Let's say that the server task is named server_task_main(). Here is some sample code that shows typical start up of a network application task.

```
void server_task_main(uint_dummy){
   int error_code;
               .
               .
               .
   error_code = Portinit("enet", "");
   if( error_code < 0 ) return;
               .
               .
               .
```

Function ***Ninit()*** does not take any parameters. Function ***Portinit()*** takes two parameters defining the physical connection to be initialized and any special parameters for the initialization.

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;
         .
         .
         .
```

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. To do an active open to a client, the "*" could be replaced with a string containing the IP address of the client.
second	Tells smxNS what protocol will be used in the connection. Other valid options are "UDP/IP" or "ICMP/IP".
third	Tells smxNS 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 smxNS 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 **nsserver.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_size, sizeof(data_size);
   if( error_code < 0 ) return;
         .
         .
         .
   /*
```

```
   ** Convert data_req buffer to integer data_requested here
   */
         .
         .
         .
   /*
   **  Write data
   */
   error_code = Nwrite(conno, junk, data_request);
   if( error_code < 0 ) return;



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

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 smxNS

After the data exchange is complete, both sides of the application are ready to terminate smxNS. 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, smxNS is terminated by calling ***Nterm()***, which actually calls the ***Portterm()*** function to shut down the physical connections. It is common for a system to run its networking functions at all times, so the call to Nterm() may be omitted. Add the following code to **nsserver.c**:

```
         .
         .
         .
   /*
   **  Terminate smxNS
   */
   Nclose(conno);                      /* close the connection */
   return;
}                                      /* 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 "`enet`" 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 **nsserver.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 **nsclient.c**, the overall structure in the program is the same, with two differences between the smxNS calls themselves. The include files, defined constants, and the call to ***Ninit()*** are the same as in **nsserver.c**. The first difference is in the call to ***Nopen()***. Program **nsclient.c** will do an active open to the server and the TCP port on the server. The following code should be in **nsclient.c**:

```
   .
   .
   .
conno=Nopen(SERVER_IP, “TCP/IP”, Nportno(), SERVER_PORT, 0);
   .
   .
   .
```

When the system running nsclient 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 local 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 **nsserver.c**.

A source code listing of **nsclient.c**, with comments, is included following the listing of **nsserver.c**.

Compiling Your Application

The IDE project files delivered with smxNS are designed to handle building an application without major modifications. Make a copy of the smx Protosystem directory to work in and create your application files there. In this example, make two copies of the Protosystem and create **nsclient.c** in one and **nsserver.c** in the other. Add each to the project file in its directory, in place of nsdemo.c.

Run the make and 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 that the server and client are connected via Ethernet.
2. Run the **nsserver** executable on the server.
3. Run the **nsclient** executable on the client.

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 **nscs.h**, **nsserver.c**, and **nsclient.c**.

Listing of nscs.h

```
/*
**  Copyright 1997  U S Software Corp.
**
**  nscs.h – Header file used by nsserver.c and nsclient.c
*/

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


/*
**  Useful constants. These should be included in nsserver.c and nsclient.c.
*/
#define CLIENT_IP      “10.1.1.2”  /* Client IP address */
#define SERVER_IP      “10.1.1.3”  /* Server IP address */
#define CLIENT_MAC “00:01:02:03:04:05” /* Client hardware addr */
#define SERVER_MAC “00:01:02:03:04:06” /* Server hardware addr */
#define LOCALMASK “255.255.255.0” /* Set subnet mask here */
#define SERVER_PORT         1500  /* Server listens at this TCP port */
#define DATA_SIZE            200  /* Size of data buffer in bytes */
#define ITERATIONS            10  /* Number of times to send data buffer*/

#endif                          /* _NSCS_H */
```

Listing of nsserver.c

```
/*
* nsserver.c                                                Version 2.70
*
* smxNS simple server test application.  To be used
* in conjunction with nsclient.c
*
* New code and modifications:
* Copyright (c) 2006-2011 Micro Digital Inc.
* All rights reserved. www.smxrtos.com
*
* USNet sample code:
* Copyright (c) 1997 United States Software Corporation
*
* This software is confidential and proprietary to Micro Digital Inc.
* It has been furnished under a license and may be used, copied, or
* disclosed only in accordance with the terms of that license and with
* the inclusion of this header. No title to nor ownership of this
* software is hereby transferred.
*
* Author: Richard Ames
*
* Portable to any ANSI compliant C compiler.
*
*****************************************************************************/

#ifdef SMXNS_DEMO

/*
**  Include at least the following files for an application
**  using the Dynamic Protocol Interface.
*/
#include “smxns.h”
/*
**  Useful constants.  This is where any application-specific
**  information would be included.
*/
#include “nscs.h”
/*
**  Server starts here.
*/

#define MAIN_STACK_SIZE 1200

#ifdef __cplusplus
extern "C" {
#endif
void nsdemo_init(void);
void nsdemo_exit(void);
#ifdef __cplusplus
}
#endif

static TCB_PTR server_task;

void server_task_main(uint dummy)
{
   int error_code;        /* Error codes returned by interface */
   int conno;             /* Connection to remote client */
   uint count;            /* Count index in junk[] */
   uint pass;             /* Number of times data sent to client */
```

```
uint data_request;     /* Number of bytes client requested */
char junk[DATA_SIZE];  /* Sample junk data */
char data_size[2];     /* Buffer of number of bytes client wants */
char status[10];       /* Client status */
(void)dummy;

/*
** Attempt to initialize the physical connections on this
** host.
*/
DEBUG_MSG3_PAR0("Server attempting a Portinit()\n");
Portcreate(“enet”);
Portconfig(“enet”, “IP”, SERVER_IP);
Portconfig(“enet”, “MASK”, LOCAL_MASK);
Portconfig(“enet”, “MAC”, SERVER_MAC);
Portconfig(“enet”, “LINK”, “Ethernet”);
Portconfig(“enet”, “DRIVER”, “ETHCTRL”);
error_code = Portinit("enet", “”);
if ( error_code < 0 )
{
   DEBUG_MSG1_PAR1(
     "Failed to initialize ports due to code %d\n", error_code);
   Nterm(); /* Terminate smxNS */
   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.
*/
DEBUG_MSG3_PAR1("Server doing an Nopen() on %d\n",SERVER_PORT);
conno = Nopen("*", "TCP/IP", SERVER_PORT, 0, 0);
if ( conno < 0 )
{
   DEBUG_MSG1_PAR1("Failed to open connection due to code %d\n",conno);
   Nterm(); /* Terminate smxNS */
   return;
}
/*
** Connection has been established. Begin writing buffer
** the number of times specified by ITERATIONS.
*/
DEBUG_MSG3_PAR1("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 )
   {
      DEBUG_MSG1_PAR1("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]);
         DEBUG_MSG3_PAR1("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 )
         {
            DEBUG_MSG1_PAR1("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 )
         {
            DEBUG_MSG1_PAR1("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.
         */
         DEBUG_MSG3_PAR1(" Pass %d complete\n", pass+1);
      }
      DEBUG_MSG3_PAR0("Server program completed successfully\n");
      /* Close down the connection */
      Nclose(conno);
      return;
}


/*****************************************************************
* DEMO INITIALIZATION / CLEANUP
*****************************************************************/
void nsdemo_init(void)
{
   DEBUG_MSG3_PAR0("Server Start\n");

   server_task = smx_TaskCreate(server_task_main, PRI_NORM, MAIN_STACK_SIZE, 0,
                                "server_task");
}


void nsdemo_exit(void)
{
}

#endif /* SMXNS_DEMO */
```

Listing of nsclient.c

```
/*
* nsclient.c                                                Version 2.70
*
* smxNS simple client test application.  To be used in conjunction with
* nsserver.c.
*
* New code and modifications:
* Copyright (c) 2006-2011 Micro Digital Inc.
* All rights reserved. www.smxrtos.com
*
* USNet sample code:
* Copyright (c) 1997 United States Software Corporation
*
* This software is confidential and proprietary to Micro Digital Inc.
* It has been furnished under a license and may be used, copied, or
* disclosed only in accordance with the terms of that license and with
* the inclusion of this header. No title to nor ownership of this
* software is hereby transferred.
*
* Author: Richard Ames
*
* Portable to any ANSI compliant C compiler.
*
*****************************************************************************/

#ifdef SMXNS_DEMO

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


/*
**  Useful constants.  This is where the application specific
**  information would be included.
*/
#include "nscs.h"

/*
**  Client starts here.
*/

#define MAIN_STACK_SIZE 1200

#ifdef __cplusplus
extern "C" {
#endif
void nsdemo_init(void);
void nsdemo_exit(void);
#ifdef __cplusplus
}
#endif

static TCB_PTR client_task;
```

```
void client_task_main(uint dummy)
{
   int error_code;        /* Error codes from function calls */
   int conno;             /* Physical connection number */
   uint count;            /* Count index in junk[] buffer */
   uint pass;             /* Number of times server sent data */
   uint client_port;      /* Client-side port number */
   uint data_request;     /* Number of bytes requested by client */
   int data_read;         /* Number of bytes read by client */
   char junk[DATA_SIZE];  /* junk buffer */
   char data_size[2];     /* Request sent to server */
   (void)dummy;

   /*
   ** Attempt to initialize the physical connections on
   ** this host.
   */
   DEBUG_MSG3_PAR0("Client attempting a Portinit()\n");
   Portcreate(“enet”);
   Portconfig(“enet”, “IP”, CLIENT_IP);
   Portconfig(“enet”, “MASK”, LOCAL_MASK);
   Portconfig(“enet”, “MAC”, CLIENT_MAC);
   Portconfig(“enet”, “LINK”, “Ethernet”);
   Portconfig(“enet”, “DRIVER”, “ETHCTRL”);
   error_code = Portinit("enet", “”);

   if ( error_code < 0 )
   {
      DEBUG_MSG1_PAR1(
        "Failed to initialize ports due to code %d\n",error_code);
   }
   /*
   ** 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();
   DEBUG_MSG3_PAR2("Calling Nopen() local port %d remote port %d\n",
            client_port, SERVER_PORT);
   conno = Nopen(SERVER_IP, "TCP/IP", client_port, SERVER_PORT, 0);
   if ( conno < 0 )
   {
      DEBUG_MSG1_PAR1(" Failed to open connection due to code %d\n", conno);
      return;
   }
   /*
   ** Connection has been established. Begin writing buffer
   ** the number of times specified by ITERATIONS.
   */
   DEBUG_MSG3_PAR1("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 */
      DEBUG_MSG3_PAR1("Sending request for %u bytes\n",data_request);
      error_code = Nwrite(conno, data_size, sizeof(data_size));
      if( error_code < 0 )
      {
         DEBUG_MSG1_PAR1(
           "Failed on send data request due to code %d\n",error_code);
         Nclose(conno);
         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 )
      {
         DEBUG_MSG1_PAR1("Failed on reading data due to code %d\n",error_code);
         Nclose(conno);
         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 )
         {
            DEBUG_MSG1_PAR0("Bad Data Received:\n");
            DEBUG_MSG1_PAR1(" Byte number %d ",count);
            DEBUG_MSG1_PAR1("is %d ",junk[count]);
            DEBUG_MSG1_PAR1("but should be %d\n", count%0x256);
            Nclose(conno);
            return;
         }
      }
      /*
      ** Send the status to the server to indicate that the
      ** client successfully read the data.
      */
      DEBUG_MSG3_PAR1(" Data was intact. Read %u bytes\n",data_read);
      error_code = Nwrite(conno, "All Done", 8);
      if( error_code < 0 )
      {
         DEBUG_MSG1_PAR1("Failed on writing data due to code %d\n",error_code);
         Nclose(conno);
         return;
      }
      /*
      ** Got this far? If so, we had a successful pass.
      */
      DEBUG_MSG3_PAR1(" Pass %d complete\n",pass+1);
   }
   DEBUG_MSG3_PAR0("Client program completed successfully\n");
   Nclose(conno);  /* Close the connection */
   return;
}
```

```
/*********************************************************************
* DEMO INITIALIZATION / CLEANUP
*********************************************************************/
void nsdemo_init(void)
{
   DEBUG_MSG3_PAR0("Client Start\n");
   client_task = sb_TaskCreate(client_task_main, PRI_NORM, MAIN_STACK_SIZE, 0
                               "client_task");
}


void nsdemo_exit(void)
{
}

#endif /* SMXNS_DEMO */
```

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 `SNS_DEBUG_LEVEL = 3` in nscfg.h to help report error conditions in the stack. Do a **grep** or **search** on ***DEBUG_MSG*** in the stack modules to locate error traps.
- The header file **net.h** contains error number translation.
- Use ***DEBUG_MSG()*** 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 `SNS_DEBUG_LEVEL = 0` in nscfg.h. 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 smxNS.

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
Build Settings	nscfg.h	<root>\XNS**include**\nscfg.h
Local Parameters	nscfg.h	<root>\XNS**include**\nscfg.h
Protocol Selection	nscfg.h	<root>\XNS**include**\nscfg.h
SSL Support	nscfg.h	<root>\CFG\iararm.h

Notes for Table 4-1:

<*root*> = \SMX

<xxx.yyy> = Build directory, as standard for SMX. xxx is compiler; yyy is CPU, such as IAR.ARM.

Build Setting configuration: **nscfg.h** specifies macros to enable smxNS add-ons. The project file compiles all files.

Local parameter configuration: **nscfg.h** contains site-dependent 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 **nscfg.h**.

SSL Support: CSL_USSL should be defined as 1 to enable HTTPS or SMTP over SSL.

Configuring the Build Settings (nscfg.h)

nscfg.h contains various configuration settings. Add-on selection is done in this file too.

Configuring Local Parameters (nscfg.h)

smxNS is configured mainly by editing file **nscfg.h** in the **include** directory. Other files are also configurable, but do not have the scope of **nscfg.h**. These are the macros in the order they appear in the file. Following this summary is more detailed information for each macro, except the first three which are simple.

*SNS_PROTO_*	selects which application and mid-level protocols to enable. The stack can be configured to use IPv4, IPv6, or both (dual-stack) for the network layer.
*SNS_DRV_*	selects which Ethernet driver to enable.
*SNS_CRYPTO_*	selects which cryptography functions to enable.
SNS_MIN_RAM	selects options to minimize RAM usage.
SNS_HW_RX_CHECKSUM	enables inbound hardware checksum calculation.
SNS_HW_TX_CHECKSUM	enables outbound hardware checksum calculation.
SNS_CPU_CACHE_DATA	locates Ethernet receive buffers in non-cacheable memory.
SNS_BUFFS_IN_SRAM	locates network message buffers in SRAM.
NSDAR_SPACE	amount of memory to reserve for web server.
NCONNS	sets the maximum number of open logical connections in one host.
NBUFFS	sets the number of message buffers.
MTU	sets the Maxiumim Transmission Unit size.
USSBUFALIGN	sets the alignment boundary for the message buffer array.
FRAGMENTATION	sets whether the code to fragment and reassemble IP packets is included.
IPOPTIONS	is the IP option support.
USS_IP_MC_LEVEL	sets the level of support for IP multicast.
IP_MC_DFLT_NETNO	sets the default interface for IP multicast.
KEEPALIVETIME	is the BSD socket keepalive time.
RELAYING	defines whether or not host is to relay.
chksum_INASM	tells smxNS 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 smxNS have the checksum routine ***Nchksum()*** in assembly.
DHCP	configures support for DHCP client functions.
DNS	configures support for DNS client functions.
NDNSS	Number of DNS servers.

TCP_SACK	enables selective ACK for TCP.
LOCALHOSTNAME	obtains smxNS's host name.
USERID	identifies a user for a PPP session.
PASSWD	authenticates a user for a PPP session.
USS_PROXYARP	enables proxy ARP feature.
FILE_SUPPORT	configures file system support.
SNS_DEBUG_LEVEL	sets the amount of debug output.
NNETS	sets the maximum number of network controllers in one host.
NNETISRS	specifies the number of interrupt vectors used by the network interfaces.

SNS_MIN_RAM Macro

This option selects a "minimum RAM" configuration. It influences the default settings of other options and a few sections in the code. A default setting based on the processor type is already set up, but can be changed depending on the system needs.

SNS_HW_RX_CHECKSUM Macro

This option enables hardware checksum calculations for inbound traffic by the Ethernet controller. Checksums in the IP, TCP and UDP headers are calculated. If the checksum is incorrect, the incoming frame is dropped. SNMP statistics are not maintained for frames that are dropped this way. The Ethernet controller and driver must support hardware checksums. Enabling this setting decreases host processing for incoming frames and should increase network thoughput.

SNS_HW_TX_CHECKSUM Macro

This option enables hardware checksum calculations for outbound traffic by the Ethernet controller. Checksums in the IP, TCP and UDP headers are calculated. The Ethernet controller and driver must support hardware checksums. Enabling this setting decreases host processing for building outgoing headers and should increase network throughput.

SNS_CPU_CACHE_DATA Macro

This option is used to locate buffers that store incoming Ethernet frames in non-cached memory. This is intended to avoid inconsistent memory values due to the cache controller not recognizing data written via DMA by the Ethernet controller. In practice, turning on this setting has been useful even in situations where special handling of the Ethernet frame buffers doesn't appear to be necessary.

SNS_BUFFS_IN_SRAM Macro

This setting specifies that Ethernet frame buffers should be located in SRAM. This may desirable for reasons of cache consistency, or for performance reasons. This is typically enabled if SNS_CPU_CACHE_DATA is enabled since internal SRAM is not cached.

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.

When using the Sockets API, the diagnostic counter sns_TcpSynDrops will count the number of times an incoming TCP connection attempt is dropped due to insufficient connections. This count can be displayed using a source level debugger or by using the Telnet debug interface and entering the netstat command. You can use this information to help tune the setting of NCONNS.

NBUFFS Macro

This is the number of working message buffers available to smxNS. When smxNS passes packets up and down the stack, it uses these buffers. These buffers are also used for internal purposes. smxNS 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.

MTU Macro

Maximum Transmission Unit size, in bytes, for the system. This sets the size of the largest unfragmented IP datagram that can be sent or received. The MTU directly affects the size of the frame buffers.

Ethernet supports an MTU of 1500 bytes, but it can be set to 576 bytes to conserve memory, SLIP interfaces are typically set to 576 bytes and PPP interfaces are typically 1500 bytes.

The MTU for the system should be the largest of any of the desired network interface MTUs. When the system is configured to forward between interfaces and at least one interface is Ethernet, the MTU should be set to 1500 bytes, since hosts on the Ethernet network won't be aware that smxNS could be running with a reduced MTU.

MAX_REASSEM Macro

Maximum size IP datagram that can be reassembled. If the system should be able to reassemble datagrams larger than the MTU, change this value to the largest datagram size. All hosts are required to reassemble a datagram of at least 576 bytes in size (per RFC 791).

Note that a typical setting for MAX_REASSEM is simply equal to the MTU, and the system normally doesn’t need to reassemble fragmented datagrams. If the MAX_REASSEM size is adjusted to be larger than the MTU, make sure the MTU is 1500 bytes.

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, so the IP layer will chop up or assemble large packets depending on this switch.

The largest IP datagram that can be reassembled depends on the size of the frame buffers, which is set with the MAXBUF macro. The largest datagram size is MAXBUF – MESSH_SZ – LHDRSZ bytes, which is typically MAXBUF – 46. Fragmented datagrams are not common, and typically are created to accommodate link layers with unusually small frame sizes. Under most conditions, the default setting for MAXBUF will be fine for use with fragmentation support enabled.

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.

The value is taken from RFC 1112, which defines the following IP multicast conformance levels:

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

The default setting is 0, which is fine for any system that makes no use of multicast IP datagrams.

Note that in order to receive multicast datagrams through an Ethernet interface, the device driver for the interface must also include support for receiving multicast frames.

IP_MC_DFLT_NETNO Macro

This specifies the default network interface for IP multicasting. Multicast frames will be sent on this interface unless the application changes the setting.

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.

RELAYING Macro

This specifies whether smxNS 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 smxNS supports do not have an assembly routine, such as PowerPC, so this should be undefined.

DNS Macro

This value specifies if DNS support code should be included, and if it should be automatically called when looking up the remote end of a network connection.

The following settings are defined:

undefined	No DNS support code will be included.
1	DNS support code will be included, but not called automatically. It will be left to the application to make a call to DNSresolve() when a domain name needs to be looked up.
2	DNS support code will be included, and DNSresolve() will be called as part of Nopen() or gethostbyname().

The default setting for DNS is undefined.

Note that a DNS server must be known to the system in order for DNSresolve() to succeed. This information can be directly specified using the SetDNS() function, or it can be retrieved automatically when the DHCPget() function is called.

NDNSS Macro

This is the number of DNS servers available for DNS look ups. The default value is 2. The DNS server IP addresses may be specified by calling SetDNS() or retrieved automatically through mechanisms such as DHCP.

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

smxNS 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 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 smxNS should use when connecting to a remote site, or the name smxNS expects when someone connects to smxNS. These are used in PPP, and Dial-up connections. They are used for establishing a PPP connection using PAP and/or CHAP.

USS_PROXYARP Macro

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

FILE_SUPPORT Macro

This specifies file system support. Since smxNS may be paired with a number of file systems, with differing APIs, this macro is used to specify the particular interface. The following file systems have been defined.

0	Minimal RAMdisk. Supplied by smxNS.
1	smxFS
2	smxFFS
3	POSIX API

SNS_DEBUG_LEVEL Macro

This specifies the amount of information that is generated for use in debugging. The value set between 0 and 6. When set to 0, no information is generated, and when set to 6, all debug messages are written. The meaning of the levels is as follows:

0: Disables all debug output and debug statements are null macros

1: Only output fatal error information

2: Output additional warning information

3: Output additional status information

4: Output additional device change information

5: Output additional data transfer information

6: Output interrupt information

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.

NNETISRS Macro

This is the number of ISRs associated with network interfaces. For processors with built in Ethernet controllers, a value appropriate for the on board controllers is defined. For systems with external network interfaces, the value will depend on the particular drivers and the number of interfaces.

Selecting Protocols

Any network protocols that you do not need can be configured out of the build by defining the protocol as 0 in the local configuration file **nscfg.h**. The following is an example of how this is done:

```
#define SNS_PROTO_UDP 0          /* User Datagram Protocol */
```

Several of smxNS's high level protocols are only supported with TCP and not UDP. Therefore the following smxNS high level protocols will not run under a UDP-only build of smxNS:

ftp*.c	File Transfer Protocol

The following smxNS high level protocols do use UDP only and will therefore run:

dhcp*.c	Dynamic Host Configuration Protocol
dns*.c	Dynamic Name Service
tftp.c	Trivial File Transfer Protocol

The stack can be configured to use IPv4, IPv6, or both (dual-stack) for the network layer. To configure the network layer, set SNS_PROTO_IPV4, SNS_PROTO_IPV6 or both to 1.

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

Selecting Drivers

Drivers for the network interfaces can be configured out of the build similar to the way that this is done for network protocols. Certain drivers will not compile for certain architectures. In order to allow one project file containing a number of possible drivers to be used across a family of processors, a facility is included that allows individual drivers to be turned on or off. If a driver is not selected, a stub file will be generated when compiling that driver.

The significant point here is to make sure that the driver for the network interface used in your system **is** enabled. If it is not, you should receive a link error when you build the final project. Also, if an unneeded driver is causing compiler errors when building the network code, the driver can easily be disabled using this facility.

The list of drivers follows the list of protocols in the file **nscfg.h**. Here is an example showing the selection of the CF5485 Fast Ethernet Controller, and not the CF5282 controller.

```
#define SNS_DRV_CF5282  0  /* ColdFire FEC used on most ColdFires */
#define SNS_DRV_CF5485  1  /* ColdFire FEC used on 5485/75 */
```

5. Dynamic Protocol Interface

Overview

This chapter details the usage of smxNS's Dynamic Protocol Interface. The Dynamic Protocol Interface provides a simple and efficient interface to the smxNS stack. It is an alternative to the BSD Sockets Interface (Chapter 6).

The Dynamic Protocol Interface contains some functions that are used to initialize or shut down the network system. These functions are Ninit(), Portcreate(), Portconfig(), Portinit(), Nterm() and Portterm(). Systems that implement their network applications using the BSD Sockets API will still use these DPI functions for system start up.

The Dynamic Protocol Interface is recommended for

- Applications with individual read and write sizes smaller than the MTU. Note, for example, that an MTU of 1500 bytes typically allows a buffer of 1460 bytes to be written or read at the application level.
- Simple code
- Developers looking to minimize the learning curve

The BSD Sockets API is recommended for

- Developers already familiar with this API
- Ports from existing applications or new development that should share common network code across systems
- Applications where it is desirable to be able to pass an arbitrarily sized buffer in the read and write calls. With the BSD Sockets API, the underlying layers will take care of dividing up the transfers if needed.

The following issues are covered in this chapter:

- 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
- Error Handling
- 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 smxNS routines need to contain the following include statement:

```
#include "smxns.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. Ninit() is called as part of smx_modules_init(), so the network application doesn't call Ninit() directly.

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.

All error conditions are < 0

NE_CFGERR Configuration error. Check log for details.

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();
```

Portcreate

Creates a network interface.

```
int Portcreate(const char *ifname);
```

ifname The name to be associated with the network interface that is created. The maximum size of the interface name is set by the struct NET definition in support.h. The current limit is 11 characters. If a longer string is specified, it will truncated to the maximum length.

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

Return Value

>= 0 Interface created. Value is interface index.

All errors are < 0

NE_CFGERR Configuration error. No room for creating an interface. Room for more interfaces can be made by increasing the value of NNETS.

Example

```
Portcreate("enet");
```

Portconfig

Configures a network interface.

```
int Portconfig(const char *name, const char *key, const char *value);
```

name The name of the interface.

key A string that identifies the parameter to be configured. The string is not case sensitive, and only the first four characters of the string are evaluated.

value A string containing the value to be configured.

Portconfig() configures a network interface. When a network interface is created, its properties are initialized to 0. Portconfig() can be called repeatedly to assign values as needed.

Summary of parameters:

IP	IP address, expressed as a dotted decimal
MASK	Mask for IP address, dotted decimal
IPV6	IPv6 address for static configuration
LINK	Link layer
DRIV	Driver name
MAC	MAC address
FBIP	Fallback IP address, dotted decimal
FBMK	Mask for fallback IP address, dotted decimal
FBCO	Fallback count, switch to fallback IP after FBCO attempts
IP2	Alias IP, dotted decimal
MK2	Mask for alias IP, dotted decimal
NAT	Enable Network Address Translation on interface
DIAL	Enable serial dial out on interface
PEER	IP address of peer in PPP link
PCP	Priority Code Point for VLAN tag
VID	VLAN ID for VLAN tag

Details on parameters

IP: This is the primary IP address associated with the interface. If an address has already been assigned to the interface, calling Portconfig() to set an IP address will also kick off address conflict detection for the new address to qualify it for use.

Special values can be assigned as follows

“0.0.0.0”- Use DHCP to obtain an IP address

“169.254.x.x” - Use a link local IP address. This address range is also known as the Auto-IP address range. The initial setting for this address will be tested for an address collision. If there is no collision, then that address will be adopted. If there is a collision, then another randomly generated address in the link local address range of 169.254.1.0 to 169.254.254.255 will be tried until a free address is found. Note that other address conflicts can lead to the system adopting the fallback IP address, so if you want to just use a local IP address, you should set both IP and FBIP to this range.

Any other address – The address will be used, provided there are no other systems on the local network using this address.

IPV6: This configuration option can be used to assign a static IPv6 address to the system. The string that provides the address should be in hexadecimal with groups of four digits separated by colons. Leading zeros may be omitted, and a double colon can be used to represent one or more groups of zeros, for example “2001:db8:85a3::8a2e:370:7334”.

Link layer: Should be one of "Ethernet", "PPP" or "SLIP".

Driver name: The driver is identified based on a string in the NPTABLE structure for the driver. Most wired Ethernet drivers can use "ETHCTRL".

Fallback IP: Fallback IP address to use if the primary IP address cannot be used. If the primary IP address is set for DHCP, the fallback address is used if the attempt to obtain an address from a DHCP server fails. If the primary IP address is a static address, the fallback address is used if a conflict is detected when probing for a duplicate of the primary address. If the first attempt to establish the fallback address is not successful, it will continue to be retried.

Fallback Count: Number of times to retry establishing the primary IP address before switching to the fallback IP address.

Alias IP: If a non-zero Alias IP is specified, the network interface will accept traffic for this address as well as the primary address.

NAT: Enable Network Address Translation on interface. The string that indicates the state should be either “ENABLE” or “DISABLE”. There are more notes on NAT configuration in Chapter 7.

DIAL: Enable serial dial out on interface. The string that indicates the state should be either “ENABLE” or “DISABLE”. There are notes on using a modem with serial communication in Chapter 8.

PEER: IPv4 address of peer in PPP link. The IP address should be supplied in dotted decimal format. This setting is optional for a PPP link.

PCP: The Priority Code Point specifies the frame priority for a VLAN tagged frame. The priority level range is 0 to 7.

VID: The VLAN ID is a 12-bit value. When a VLAN ID is defined, outgoing frames will include a VLAN tag. The string containing the VID should be in hexadecimal and of the form “0x123”. If a VLAN tag is defined for an interface, the tag will be included in all frames sent on the interface.

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

Return Value

0	Value stored.
All errors are < 0	
NE_PARAM	Run-time parameter error. The named interface was not found, key not found, or invalid value. See log for details.
NE_CFGERR	Configuration error. See log for details.

Examples

The following code is typical for a static IP address. It sets an address of 10.1.1.20.

```
Portcreate("enet");
Portconfig("enet", "IP", "10.1.1.20");
Portconfig("enet", "MASK", "255.255.255.0");
Portconfig("enet", "MAC", "00:01:02:03:04:05");
Portconfig("enet", "LINK", "Ethernet");
Portconfig("enet", "DRIVER", "ETHCTRL");
if (Portinit("enet", "") < 0)
{
   DEBUG_MSG1_PAR0("smxNS Portinit for enet failed\n");
}
```

Here's a more involved example that starts with an address obtained via DHCP and then transitions to a static IP address.

Note that it is possible to leave the interface active while changing the type of IP address that is used, i.e. one doesn't need to go through the Portterm(), Portinit() sequence again in order to change to a new local IP address. All application level connections should be shut down before changing the address though.

```
Portcreate("enet");
Portconfig("enet", "MAC", "00:01:02:03:04:05");
Portconfig("enet", "LINK", "Ethernet");
Portconfig("enet", "DRIVER", "ETHCTRL");
if (Portinit("enet", "") < 0)
{
   DEBUG_MSG1_PAR0("smxNS Portinit for enet failed\n");
}

while (Portstate("enet") != NETIF_READY)
   smx_DelayMsec(500);

/* System is now using address from DHCP server */

Portconfig("enet", "FBIP", "10.1.1.100");
Portconfig("enet", "FBMK", "255.255.255.0");
DHCPrelease(GetPortIndex("enet"));

while (Portstate("enet") != NETIF_READY)
   smx_DelayMsec(500);

/* System is now using address 10.1.1.100 */
```

The first time the interface is set up, no IP address is defined, so the default value 0.0.0.0 will be in place and the DHCP client will be started to obtain an IP address. After the port is initialized, the loop that calls Portstate() will continue looping until an address is established.

In order to transition to a static IP address, the new address and mask should be stored in the fallback IP and fallback mask slots, and the DHCP leased address should be released. This way, the DHCP server is informed that the leased address is no longer in use, and the DHCP client state machine will pick up the fallback address after the leased address is turned in.

If the call to DHCPrelease() were immediately followed by a call to Portinit() to set the IP address directly there is a chance that the DHCP client state machine would restart before the static IP address was in place.

Portinit

Initializes a network interface.

```
int Portinit(const char *ifname, const char *initstring);
```

ifname The name associated with the network interface to be initialized.

initstring A string that can contain additional initialization information. Device drivers may obtain information from this string.

Portinit() initializes a network interface. 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. Note that Ethernet interfaces with 10/100 PHYs may take around 6 seconds to negotiate link parameters.

Although the call to Portinit() may immediately return successfully, there may be a delay before frames can be sent or received. An attempt to establish an active connection will fail if the network interface has not come up yet. The nsdemo.c file contains code that will wait until at least one network interface is up. This code appears in the example below.

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

Return Value

0	Initialization successful.
All errors are < 0	
NE_PARAM	Parameter error. ifname not found, interface already initialized, error in initialization string or hardware error.
NE_CFGERR	Configuration error. Link or driver layer not defined, insufficient resources configured.
NE_HWERR	Hardware error. Hardware behavior was not as expected.
NE_NOBUFS	Not enough memory resources to initialize.

Additional details on error conditions are available in the log.

Example

```
Portcreate("enet");
```

```
Portconfig(“enet”, “IP”, “10.1.1.20”);
Portconfig(“enet”, “MASK”, “255.255.255.0”);
Portconfig(“enet”, “MAC”, "00:01:02:03:04:05");
Portconfig(“enet”, “LINK”, “Ethernet”);
Portconfig(“enet”, “DRIVER”, “ETHCTRL”);
if (Portinit(“enet”, “”) < 0)
{
   DEBUG_MSG1_PAR0(“smxNS Portinit for enet failed\n”);
}
while (Portstate("*") != NETIF_READY)
   smx_DelayMsec(500);
```

Portstate

Checks the state of one or more network interfaces.

```
int Portstate(const char *name);
```

name If “*****”, then all network interfaces are checked; otherwise, this should be a network interface name specified in a call to Portcreate().

Checks the state of a network interface. This is useful for determining when an interface has reached the NETIF_READY state so that one can be sure connections can be actively establilshed and network traffic can be sent.

All interfaces can be checked at once, in which case the state of the network that is closest to or at "NETIF_READY" is reported.

Return Value

NETIF_UNITIALIZED	Network interface not initialized.
NETIF_NOLINK	No link established for interface (often cable disconnected)
NETIF_NEGOTIATING	Interface is linked but IP address not yet established
NETIF_READY	Interface is ready to transmit

All errors are < 0

NE_PARAM	Parameter error. name not found

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

Examples

```
while (Portstate("*") != NETIF_READY)
   smx_DelayMsec(500);
```

Portterm

Shuts down one or more network interfaces.

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

name If "*", then all network interfaces for this host will be shut down; otherwise, this should be a network interface name specified in a call to Portcreate().

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. Any network connections associated with the interface are marked as fatal. 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.

smxNS 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.
- 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 smxNS. This might surprise you, especially if you have seen network packages that go something like:

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

smxNS uses a table-driven protocol stack structure. Each protocol level has only one public symbol: The name of the protocol table. smxNS 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.

Beginning with smxNS v2.90, the error codes returned from Nopen(), Nclose(), Nread() and Nwrite() no longer include overlapping POSIX error codes, i.e. EBADF, ECONNABORTED, etc. Instead, smxNS specific error codes are used as appear in the table below.

smxNS v2.8 and earlier	**smxNS v2.9 and later**
EBADF	NE_BADF
ECONNABORTED	NE_CONNABORTED
EHOSTUNREACH	NE_HOSTUNREACH
ENETUNREACH	NE_NETUNREACH
EMSGSIZE	NE_MSGSIZE
EWOULDBLOCK	NE_WOULDBLOCK
ENOBUFS	NE_NOBUFS
ETIMEDOUT	NE_TIMEDOUT

smxNS creates definitions for the POSIX error codes if they are not present using negative values using code like the following from support..h.

```
#ifndef EHOSTUNREACH
#define EHOSTUNREACH -10
#endif
```

If the error code is defined, then the existing definition is retained. In some build environments, these error codes have positive value, which is not compatible with the convention that DPI functions return a negative value on error. For this reason, the new error code definitions were introduced in smxNS version 2.90.

Network applications that use the DPI functions may need to be adjusted if they include error handling that uses the old error codes. In order to update the code, one should substitute the new error code name. Here is an example:

Change:

```
rc = Nread(s, buf, buflen);
if (rc == ETIMEDOUT)
{
   ...
```

To:

```
rc = Nread(s, buf, buflen);
if (rc == NE_TIMEDOUT)
{
   ...
```

Nopen

Opens a connection.

```
int Nopen(const char *to, const 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%ifname"	Remote host, using named interface.
"*"	Any host, used for passive open or broadcast.
"*%ifname"	Any host, using named interface.
"n1.n2.n3.n4"	IP address of remote system in IPv4 format.
"x:x:x:x:x:x:x:x"	IP address of remote system in IPv6 format, as specified in RFC 4291, section 2.2. "Text Representation of Addresses". It is a series of eight 16-bit address segments separated by colons. Leading zeros may be omitted. Sequences of one or more groups of zeros may be abbreviated as ::, but only once.

protoc String specifying the transport and network layer protocols, separated by a slash. Typical values would be "TCP/IP", "UDP/IP" or "ICMP/IP". If a listening connection specifies IP as the bottom half of the protocol, IPv4 and IPv6 clients are accepted. If IPv6 is specified, only IPv6 clients are accepted.

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.
All errors are < 0	
NE_PARAM	Run-time parameter error. Protocol not recognized.
NE_CFGERR	Out of connection blocks.
NE_HOSTUNREACH	No route to host.
NE_CONNABORTED	Remote host sent RST when opening connection.
NE_NOBUFS	No frame buffers available when opening connection.
NE_TIMEDOUT	Time out trying to create 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. */

/* 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. */
```

```
/* 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.
`-1`	Error occurred in attempting to close the connections. Possible reasons are an invalid connection number or a protocol problem.

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.
>0	Indicates the number of bytes read.
NE_BADF	The connection number is not valid.
NE_WOULDBLOCK	Non-blocking connection can't proceed. Read would be retried.
NE_TIMEDOUT	Timeout. Read can be retried.
NE_CONNABORTED	Protocol problem. For example, the peer TCP sent a RST segment. Normally the application should close the connection.
NE_MSGSIZE	The message is too long for the supplied buffer. The incoming TCP segment or UDP message is dropped and no data is transferred to buff, but the application can continue to use the connection.

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 smxNS constant MTU defined in file **nscfg.h**. A call to ***Nread()*** cannot return more than MTU bytes.

Nwrite

Writes a message to a connection.

```
int Nwrite(int conno, const 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.

By default, when ***Nwrite()*** writes a TCP segment, the PSH flag will not be set. This flag is a hint to the receiving TCP that a usable set of information has been sent and that it should be processed by the receiving network application. The PSH flag can be set by using the ***SOCKET_PUSH()*** macro prior to calling ***Nwrite()***. If the receiving TCP is slow to process incoming information, it may help to set this flag.

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

Return Value

>= 0	Indicates the number of bytes written. For TCP connections, this indicates that the buffer has been written, but not necessarily that the remote end has received the information. Ensuring delivery is handled in the background.
NE_BADF	The connection number is not valid.
NE_TIMEDOUT	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.
NE_CONNABORTED	Protocol problem. Normally the application should close the connection.
NE_MSGSIZE	The message is too large for the internal buffer.
NE_WINZERO	The peer TCP window is not large enough to accept the data. This only occurs in non-blocking mode. See the Non-Blocking Operations Example section for workarounds.

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_ISFATAL	checks for an unrecoverable error on the connection.
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_REMADDR	provides the IP address of the remote end of a connection.
SOCKET_LOCADDR	provides the IP address of the local end of a connection.
SOCKET_REMPORT	returns the remote port number for a connection
SOCKET_LOCPORT	returns the local port number for 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_FAMILY	returns the address family for a given connection.
SOCKET_HASMYADDR6	checks if the IPv6 site local address has been allocated.
SOCKET_LOCSITEADDR6	returns the IPv6 site local address.

SOCKET_REMADDR6 returns the remote host's IPv6 address.

SOCKET_LOCLINKADDR6 returns the IPv6 link local address.

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_ISSENDING

Checks to see if all data that has been written by the application has been acknowledged by the peer TCP.

SOCKET_ISSENDING(*conno*)

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

This macro will evaluate as non-zero if outgoing data has not yet been acknowledged by the peer TCP. The macro will evaluate as 0 if all outgoing data has been acknowledged, or if there has been an unrecoverable error on the connection.

If the application calls SOCKET_ISSENDING() immediately after calling Nwrite(), it will typically return true. Outgoing data is typically acknowledged within a couple hundred milliseconds.

This macro may be useful for tracking status of a transfer or in creating recovery mechanisms for lengthy transfers. Note that even though the peer TCP may have acknowledged receiving a TCP segment, this does not guarantee that the application running on the peer system has successfully read the information. Closing the connection and checking for success is a more reliable mechanism for verifying a complete transfer.

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_ISFATAL

Checks for an unrecoverable error on a connection.

SOCKET_ISFATAL(*conno*)

conno The connection that should be checked for errors.

This macro will evaluate as `0` if there are no unrecoverable errors on the connection, or non-zero if an unrecoverable error has occurred. As an example, an unrecoverable error occurs when a peer TCP sends a RST segment to the local end of the connection. The socket should still be closed when this condition is detected.

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. The default timeout is set by TOUT_READ in net.h.

`SOCKET_RXTOUT(`*`conno, tout`*`)`

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

`tout` The new timeout, in milliseconds. For an infinite timeout, use the value SB_TMO_INF.

SOCKET_REMADDR

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

`SOCKET_REMADDR(`*`conno`*`)`

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

The data type of the result is `Iid`.

SOCKET_LOCADDR

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

`SOCKET_LOCADDR(`*`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_REMPORT

Provides the TCP or UDP port number of the remote end of a connection.

SOCKET_REMPORT(*conno*)

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

The data type of the result is unsigned short.

SOCKET_LOCPORT

Provides the TCP or UDP port number of the local end of a connection.

SOCKET_LOCPORT(*conno*)

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

The data type of the result is unsigned short.

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.

SOCKET_FAMILY

Returns the address family for a given connection.

SOCKET_FAMILY(*conno*)

conno The connection for which to return the address family.

For IPv6 connections, returns AF_INET6. For IPv4 connections, returns AF_INET.

SOCKET_HASMYADDR6

Checks if the IPv6 site local address has been allocated.

SOCKET_HASMYADDR6(*conno*)

conno The connection for which the site local address should be checked.

This macro evaluates as 1 when the IPv6 site local address has been allocated. The macro evaluates as 0 when the address has not be allocated.

SOCKET_LOCSITEADDR6

Returns the IPv6 site local address.

SOCKET_LOCSITEADDR6(*conno*)

conno The connection for which the IPv6 site local address should be returned.

This macro evaluates to data type I6id. The macro SOCKET_HASMYADDR6(conno) can confirm if the IPv6 site local address has been allocated.

SOCKET_REMADDR6

Returns the remote host's IPv6 address.

SOCKET_REMADDR6(*conno*)

conno The connection for which the remote host's IPv6 address should be returned.

This macro evaluates to data type I6id.

SOCKET_LOCLINKADDR6

Returns the IPv6 link local address.

SOCKET_LOCLINKADDR6(*conno*)

conno The connection for which the IPv6 link local address should be returned.

This macro evaluates to data type I6id.

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);
```

Error Handling

When a DPI call returns ECONNABORTED, no further communication over the connection is possible. If the connection was previously opened successfully, then the application must call Nclose() on the connection. Otherwise memory and network data structures might still be assigned to it.

Note that a connection can go from a good state to a failed state at any time. Consider the case where the system at the remote end of a TCP connection unexpectedly goes offline shortly before a client running on an smxNS system sends a query using Nwrite(). The call will likely return a positive value equal to the number of bytes in the buffer being written. This may be confusing, but the meaning of the return value is that smxNS has taken responsibility for delivery this number of bytes to the remote system. It does not necessarily mean that these bytes have been delivered.

The TCP specification describes how a segment will be retransmitted if the remote system does not send a timely acknowledgement. smxNS will perform this retransmission in the background. If these attempts fail, the next time the application calls a function involving the connection, the function will return ECONNABORTED.

The macro SOCKET_ISFATAL() can be used at any time to check for a failed connection.

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 Examples

The following example shows how to read using non-blocking operations. Non-blocking writes will complicate an application quite a bit. 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 (;;)
{
   SNS_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 */
```

The return code from Nwrite() will be EWINZERO if you are in non-blocking mode and the TCP window is not large enough to take your packet.

So, if you are using non-blocking I/O and there is a possibility that the remote host's window may close (this happens when the remote host does not read the received data), then you must use one of the following workarounds:

1) Write less data. The remote window is stored in

```
connblo[conno].window
```

Examine the window and resend using a packet size smaller than the remote window.

2) Enable the TCP window probe. To do this, you must revert to blocking mode and rewrite the data. The write will block while performing the window probe.

```
len = Nwrite(conno, buff, sizeof(buff))
if (len == EWINZERO) {
    SOCKET_BLOCK(conno);
    len = Nwrite(conno, buff, sizeof(buff));
    /* check error, etc */
}
```

3) Close the connection.

4) Use SOCKET_CANSEND() before you write to evaluate whether the connection can send data. This will let you avoid getting into the situation for which you need to test for EWINZERO, but will not solve the problem that there is no probe in non-blocking mode.

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. smxNS 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 smxNS 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.

For somebody who already knows the BSD sockets interface, 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 smxNS 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. smxNS 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 "smxns.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. smxNS 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 smxNS 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.
getaddrinfo()	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.

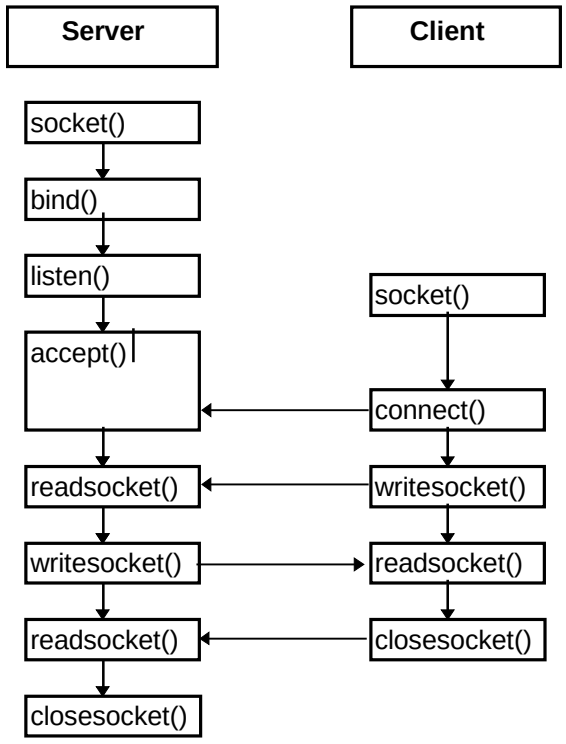


Figure 6-1: Functions Used in a Connection-Oriented System

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

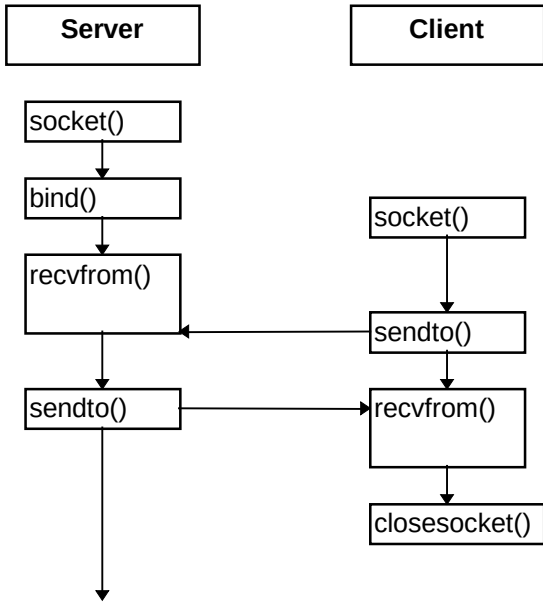


Figure 6-2: Functions Used in a Connectionless System

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;
    DEBUG_MSG2_PAR1("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 smxNS.

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)
    DEBUG_MSG2_PAR0(“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)
    DEBUG_MSG2_PAR0("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.

There is a special situation that may need to be addressed when using non-blocking sockets. Although calling selectsocket() on the write descriptor prior to calling send() will normally take care of most error conditions, there is one case where this may fail. If a lot of data is sent using send() and then closesocket() is called immediately also in non-blocking mode, a portion of data may remain unsent. The easiest solution is to add an additional call to selectsocket() prior to calling closesocket(). See the example section.

See also: ***socket***

Return Value

-1	Error.
0	Close was successful.

Example

```
void wait_for_write(int sockfd)
{
    fd_set wset;
    struct timeval tm;
    do {
        tm.tv_sec = 10;
        tm.tv_usec = 0;
        FD_ZERO(&wset);
        FD_SET(sockfd, &wset);
    } while ( ! selectsocket(sockfd + 1, 0, &wset, 0, &tm) );
}

void write_data(int sockfd, char *buff, int buffsz)
{
    int len, totlen;
    int noblock = 1;
    ioctlsocket(sockfd, FIONBIO, &noblock);
    do {
        wait_for_write(sockfd);
        len = send(sockfd, buff[totlen], buffsz - totlen, 0);
        if (len < 0) {
            /* Handle the error condition */
                ...
        }
        totlen += len;
    } while (totlen < buffsz);
    /* This extra call to select avoids lost data */
    wait_for_write(sockfd);
    closesocket(sockfd);
}
```

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)
    DEBUG_MSG2_PAR0("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)
    DEBUG_MSG2_PAR0("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.

freeaddrinfo

Release the memory allocated for the given addrinfo structure.

```
void *freeaddrinfo(struct addrinfo *res)
```

res (Input) Pointer of the address structure to release

The linked list acquired with getaddrinfo() is released.

See also: ***getaddrinfo***

Return Value

none

Example

```
struct addrinfo *ai;
freeaddrinfo(ai);
```

gai_strerror

Convert an error code from getaddrinfo() into a character string.

```
const char gai_strerror(int errcode);
```

errcode (Input) Error code.

Return Value

Pointer to the corresponding character string.

Example

```
int errcode;
char *errorstr;
errorstr = gai_strerror(errcode);
```

getaddrinfo

Obtain address information based on host and port information.

```
int getaddrinfo(const char *hostname, const char *servname, const
struct addrinfo *hints, struct addrinfo **res);
```

hostname	(Input) Host name or IP address
servname	(Input) Service name or port number string
hints	(Input) Additional optional specifications for the type of address
res	(Output) Address storage area

hostname specifies the acquired host name or IP address.

servname specifies the port number as a character string.

The type and the protocol of the desired socket are specified via the *hints* parameter.

The result of the request is provided in the *res* parameter.

The memory dynamically allocated uses one message buffer (MESS structure).

The following ai_flags options in the hints field are supported.

AI_PASSIVE

AI_NUMERICHOST

AI_ADDRCONFIG

It is necessary to release the allocated memory with freeaddrinfo().

See also: ***freeaddrinfo()***

Return Value

0	Success
!= 0	Check error associated with socket
EAI_ADDRFAMILY	The requested address family for the given hostname is not available
EAI_FAMILY	The requested address family is not available
EAI_SERVICE	The requested service cannot be used by the requested socket type
EAI_NONAME	The requested name is illegal
EAI_MEMORY	Insufficient memory
EAI_FAIL	The name server failed in responding to the request
EAI_SYSTEM	Other system error occurred

Example

```
struct addrinfo hints;
char portstr[10];
int port = 80;
char *hostname = “(Ipv6 address)”;
struct addrinfo *ai;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = 0;
hints.ai_flags |= AI_NUMERICSERV;
sprintf(portstr, “%u”, (int)port);
if (getaddrinfo(hostname, portstr, &hints, &ai)) != 0)
   return –1;
```

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)
    DEBUG_MSG2_PAR0(“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)
    DEBUG_MSG2_PAR0(“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. Refer to the table below for the data type.
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	Type	Description
IPPROTO_IP	IP_ADD_MEMBERSHIP	struct ip_mreq	Join multicast group
	IP_DROP_MEMBERSHIP	struct ip_mreq	Leave multicast group
	IP_MULTICAST_IF	struct in_addr	Set multicast interface
	IP_OPTIONS	char	Options in IP header
	IP_TTL	unsigned int	TTL in IP header
IPPROTO_TCP	TCP_MAXSEG	unsigned int	Get TCP maximum segment
	TCP_NODELAY	unsigned int	Don't delay send
SOL_SOCKET	SO_BROADCAST	unsigned int	Permit broadcast
	SO_DEBUG	unsigned int	Debug flag
	SO_DONTROUTE	unsigned int	No routing
	SO_ERROR	unsigned int	Get and clear error code
	SO_KEEPALIVE	unsigned int	Keepalive probing
	SO_LINGER	struct linger	Linger on close
	SO_OOBINLINE	unsigned int	Leave URG data inline
	SO_RCVBUF	unsigned int	Receive buffer size
	SO_REUSEADDR	unsigned int	Local address reuse
	SO_SNDBUF	unsigned int	Send buffer type
	SO_TYPE	unsigned int	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)
    DEBUG_MSG2_PAR0("Error in setsockopt\n");
```

inet_ntop

Convert an address structure into a string.

```
char *inet_ntop(int af, void *src, void *dst, int cnt);
```

af	(Input) Address family.
src	(Input) Pointer to the network address structure.
dst	(Output) Area where the result is stored.
cnt	(Input) Size of area where the result is stored.

The ***inet_ntop()*** function converts network address structure *src* of address family *af* into a character string. This function copies the string into memory at location *dst* (length *cnt* bytes).

af specifies AF_INET or AF_INET6.

If the value in *af* is not supported, errno is set to EAFNOSUPPORT. If the resulting string would occupy more than *cnt* bytes, errno is set to ENOSPC.

Return Value

NULL	Error.
Pointer to *dst*	Success.

Example

```
example
```

inet_pton

Convert a string into a network address structure.

```
int inet_pton(int af, char *src, char *dst);
```

af (Input) Address family.

src (Input) Pointer to the address of the character string.

dst (Output) Area where the conversion result is stored.

The ***inet_pton()*** function converts the string pointed to by *src* of the *af* address family into a network address structure, and stores it at address *dst* (of length *cnt* bytes).

af specifies AF_INET or AF_INET6.

The function returns a negative value and sets errno to EAFNOSUPPORT if the value for *af* is not supported. When *src* is not a valid address, the function returns 0.

Return Value

> 0	Success
< 0	The address family is not supported.
0	The address of the character string is illegal.

Example

```
example
```

ioctlsocket

Sets control parameters for a socket.

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

s	Socket identifier.
request	Request type. See table below.
arg	Optional argument. See table below

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.

request	argument type	description
FIONBIO	int *	Sets non-blocking I/O if *arg* points to an int of non-zero value. Sets blocking I/O otherwise.
FIONREAD	int *	*arg* is assigned the number of bytes that have not yet been read.
SIOCATMARK	int *	*arg* is assigned 1 if the socket read is at the out-of-bound mark, 0 otherwise.

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 value of *backlog* must be greater than 0 for a subsequent call to accept() to succeed. If there are no connections available at the time a SYN segment is received, the incoming segment will be dropped and the diagnostic counter sns_TcpSynDrops will be incremented. NCONNS can be adjusted up if sns_TcpSynDrops shows dropped SYNs.

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)
    DEBUG_MSG2_PAR0("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.
0	The remote side closed the connection.

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. For non-stream connections, excess bytes will be discarded.
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.
0	The remote side closed the connection.

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.

ETIMEDOUT
: 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)
    DEBUG_MSG2_PAR0("Error receiving data.\n");
else if (rc == 2)
    DEBUG_MSG3_PAR0("Success: read 2 bytes\n");
else
    DEBUG_MSG2_PAR0(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. For non-stream connections, excess bytes will be discarded.
flags	Specifies optional behavior: MSG_OOB returns urgent data. MSG_PEEK returns information, allowing it to be read again on a subsequent call.
from	Indicates the remote host from which the information was received.
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 */
int socksize;
unsigned char buff[BUFFLEN];    /* read buffer */
struct sockaddr_in socka;       /* remote host address */
…

memset(&socka, 0, sizeof(socka));

rc = recvfrom(s2, buff, 8, 0, (struct sockaddr *)&socka, &socksize);
```

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 */
};
```

smxNS ignores the access rights field in the msghdr structure.

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

Return Value

-1	Error.
> 0	Number of bytes received.
0	The remote side closed the connection.

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.

smxNS uses the SIG_SEL signal to support the select operation. SIG_SEL is raised when traffic comes into the stack or maintenance functions run that might change the state of a connection.

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)
    DEBUG_MSG2_PAR0(“Error, no sockets ready.\n”);
else
    DEBUG_MSG3_PAR1(“%d sockets ready.\n”, rc);

if (FD_ISSET(s1, &socket_set1))
    DEBUG_MSG3_PAR0(“Socket 1 is ready to be read.\n”);
else if (FD_ISSET(s2, &socket_set2))
    DEBUG_MSG3_PAR0(“Socket 2 is ready to be written\n”);
else if (FD_ISSET(s3, &socket_set3))
    DEBUG_MSG3_PAR0(“Socket 3 is ready to be read.\n”);
else
    DEBUG_MSG2_PAR0(“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.
EWINZERO	The receiving TCP window is not large enough to take accept the data. See the examples section for a workaround.

Example

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

rc = send(s2, buff, sizeof(buff), 0);
if (rc < 0)
    DEBUG_MSG2_PAR0("Error sending data\n");
```

The errno from send() will be EWINZERO if you are in non-blocking mode and the TCP window is not large enough to take your packet.

If you are using non-blocking I/O and there is a possibility that the remote host's window may close (this happens when the remote host does not read the received data). Then you must use a workaround.

You can do one of several options:

1) Write less data. The remote window is stored in connblo[conno].window

Examine the window and resend using a packet size smaller than the remote window.

2) Enable the TCP window probe. To do this, you must revert to blocking mode and rewrite the data. The write will block while performing the window probe.

```
int noblock = 1;    /* Set to non-blocking mode */
ioctlsocket(sockfd, FIONBIO, &noblock);
...
len = send(sockfd, buff, sizeof(buff), 0);
if (len < 0) {
    int i1, errval, sz;
    sz = sizeof(val);
    i1 = getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &errval, &sz);
    if (errval == EWINZERO) {
        noblock = 0;
        ioctlsocket(sockfd, FIONBIO, &noblock);
        len = send(sockfd, buff, sizeof(buff), 0);
        if (len > 0) {
            noblock = 1;
            ioctlsocket(sockfd, FIONBIO, &noblock);
            /* Continue normal execution */
        }
    }
    else if (errval != EWOULDBLOCK) {
        DEBUG_MSG2_PAR1("Error %d\n", errval);
        closesocket(sockfd);
        return -1;
    }
}
```

3) Close the connection.

4) Call selectsocket() with a write set enabled to check. This will let you avoid getting into the situation for which you need to test for EWINZERO, but will not solve the problem that there is no probe in non-blocking mode.

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 */
};
```

smxNS 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)
    DEBUG_MSG2_PAR0("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 smxNS, this should always be PF_INET.

type smxNS 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)
    DEBUG_MSG2_PAR0("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, IP_MC_DFLT_NETNO, and is declared in nscfg.h.
recvfrom()	Receive Multicasts as they come in on the port that was bound.

7. Network Applications and Protocols

Overview

smxNS offers support for a number of networking applications, and also special features at the stack level. Some are included and some are extra-cost options.

ARP	maps an IP address to a hardware address.
DHCP	delivers host configuration parameters to a client host.
DNS	allows hosts to be specified by name rather than IP address.
FTP	is a file transfer programs implemented with TCP.
HTTP	transfers web pages. A simple client is provided.
IGMP	is the multicast protocol
IPv6	directs datagrams to the destination host using 128-bit IPv6 addresses
mDNS	allows network hosts to discover local services
NAT	is the network address translation.
PPP	connects two hosts over a serial link.
PPPoE	is a protocol typically used with DSL equipment.
SLIP	a minimal protocol for connecting over a serial link.
SNTP	allows hosts to synchronize time information.
TELNET	is the usual TCP/IP method of remote terminal access.
TFTP	is a simple file transfer program implemented with UDP.
Web Server	serves web pages.

The discussions of PPP and the Web Server are lengthy and are presented in separate chapters which follow this one.

ARP

ARP (Address Resolution Protocol) is used to map an IP address to a hardware address. The ARP client checks its ARP cache first for the IP address of the destination host, to get its hardware address. If there is no entry in the cache, it sends a broadcast message to all the hosts on the network segment. The host with the desired IP address responds with its hardware address, and the requestor adds it to its ARP cache.

Proxy ARP

The Proxy ARP feature allows a system running smxNS to answer ARP requests on behalf of another system. This is useful when smxNS acts as a transparent bridge, making it appear that systems that are reachable **through** the system running smxNS are directly connected to an Ethernet network.

There are three steps to enable the Proxy ARP capability in smxNS:

1.) Uncomment the following line in include\nscfg.h:

```
#define USS_PROXYARP
```

this will enable the proxy ARP feature in arp.c and ip.c.

2.) Add the definitions of the Proxy ARP hosts with the PROXYARP flag in the flags field:

```
"net186",  "enet1", C, {206,251,94,253}, EA0, PROXYARP, Ethernet,
AMD961, "IRNO=4 PORT=0x200",
```

3.) You should have at least two interfaces for the local host defined in the netdata[] table. For example, say the local host is named server:

```
"server",  "enet0", C, {206,251,94,224}, EA0, 0, Ethernet, NE2000,
"IRNO=10 PORT=0x0300",

"server",  "enet1", C, {206,251,94,252}, EA0, 0, Ethernet, CS8900,
"IRNO=5 PORT=0x0320 BASE=0xC800",
```

Note that the port name, which is the second field in the definition, is different for the two interfaces defined for the local host and that our proxy ARP host uses the port name of the second interface definition. The order is important. smxNS will take the first subnet address match that it finds when it decides where to send its messages.

We have added logic to ip.c to scan the table for the proxy ARP host and its matching interface definition on the local host. So we need to have the "other" interface specified first in the table so that smxNS will find that when it scans for the subnet address match.

To test this feature, you need two hosts connected to each other on a dedicated network with the host doing the proxy ARP also connected to a second network. Use another host on the second network to send a ping to the host that is on the dedicated network.

The host on the dedicated network should respond to the ping that should be indicated by a ping reply message. After the ping has executed, the ARP cache (use arp -a) on the sending host should have a new entry with the IP address of the host on the dedicated network and the ethernet address of the proxy ARP host.

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 smxNS, DHCP is used to obtain an IP address for the host. The protocol will be used automatically as part of ***NetTask()*** and ***Portterm()*** based on the setting of the `#define SNS_PROTO_DHCPC` line in **nscfg.h**.

The call to obtain an IP address through DHCP is:

```
void DHCPget(int netno);
```

This function is called automatically from the NetTask() network background task.

The current state of the DHCP client is visible via the nets[0].DHCPstate variable. Normally it should have the value DHCP_BOUND, meaning that the system is using the IP address acquired from the DHCP server.

The call to release an assigned IP address is:

```
int DHCPrelease(int netno);
```

The ***DHCPrelease()*** return codes are:

`0`	Success
`ETIMEDOUT`	Timeout

The smxNS DHCP Server follows RFC's 2131 and 2132 with the restrictions noted below.

- DHCPserv() starts the server.
- The server should never return.

DHCP Client Configuration

To use DHCP for address assignment with smxNS:

1) Set the primary IP address to 0.0.0.0 like:

Portconfig(“enet”, “IP”, “0.0.0.0”);

2) #define SNS_PROTO_DHCPC 1 in nscfg.h

3) To adjust the number of times the DHCP client retransmits the DHCPDISCOVER message when trying to locate a DHCP server, adjust DHCPC_DISCOVER_MAX_RETRY in dhcp.h. Setting this macro to 0xFFFFFFFF will allow the call to DHCPget() to retry indefinitely waiting for a DHCP server to become available.

4) To get a router from DHCP:

#define DHCP_CONFIG = 1 or 2

When DHCP_CONFIG is set to 1, the client will request only an IP address from the DHCP server. When DHCP_CONFIG is set to 2, the client will request an IP address, a subnet mask, a router, and a DNS server.

* * * * * IMPORTANT NOTE * * * * *

For network environments where the system running smxNS and the DHCP server may be on different subnets, the DHCP_CONFIG=2 setting should be the most reliable. This setting should ensure that the DHCP server includes the router option in its response.

5) Additional configuration options:

If additional configuration options are desired, then edit dhcp.h and modify the discopts declaration. The options with DHCP_CONFIG = 2 are as follows:

```
static const u8 discopts[] =
   {99, 130, 83, 99, 53, 1, DHCPDISCOVER, 55, 3, 1, 3, 6};
```

Option 55 is a parameter request list, 3 is the number of parameters requested, 1 is the subnet mask option, 3 is the router option, and 6 is the DNS server option in the example above. Valid option codes are given in RFC 2132. To remove options, remove the appropriate one and change the number of parameters accordingly. Do not change any options before or including option 55.

6) DHCP is automatically called from NetTask() if #define SNS_PROTO_DHCPC 1 is set in nscfg.h.

When initializing more than one interface using DHCP, they need to be initialized separately.

Example:

```
Portinit("eth1", “”); /* initializes interface 1 */
Portinit("eth0", “”); /* initialized interface 0 */
```

7) Fallback behavior:

If the initial attempt to obtain an IP address from the DHCP server fails (perhaps because there is no DHCP server on the network), it is possible to have smxNS use an alternate method to obtain an IP address. The alternate method is specified in the FallbackAddr field of the network data structure, which is set with Portconfig() using the “FBIP” key.

If the value here is 0.0.0.0, then smxNS will continue attempting to use DHCP to obtain an IP address. Since the network data structure is cleared to zero as part of initialization, this is the default value, so by default smxNS will stick to DHCP.

If FallbackAddr is set to an address in the range 169.254.x.x, then smxNS will generate a link-local address (also known as an AutoIP address), which will be a random address in this same range.

Any other value for FallbackAddr will be considered a fixed IP address, and that address will be adopted.

In order to set a link-local address or fixed IP address as the fallback address, set the value before calling Portinit(), as in this example.

```
Portconfig(“enet”, “FBIP”, “192.168.1.5”);
Portinit(“enet”, “”);
```

8) Lease renewal:

smxNS will automatically track the time left on a DHCP lease to renew it. The lease time and the renewal time are stored in the smxNS NET structure.

In order to suggest a lease time to the DHCP server, fill in a value for the SuggestedLease field in the nets[] data structure before calling Portinit(). This value is in units of 1 second. For example

```
nets[0].SuggestedLease = 7200;  /* 2 hour lease */
Portinit("enet", "");
```

DHCP Server Configuration

1) Server configuration file:

The name of the server configuration file is defined as CONF_FILE in dhcp.h. The default name is "dhcp.con" with no path. This file contains the configuration parameters that the server will give to clients. Most entries are self-explanatory. The range entry is the range of IP addresses you wish to give your clients. To configure with no router or domain name server, put 0 for the number of entries, with no IP addresses to follow To configure no domain name or bootfile name enter none.

This file must have the following format with no lines omitted:

```
 netname
subnet_mask x.x.x.x
 range x.x.x.x x.x.x.x
 router number_of_routers x.x.x.x [x.x.x.x ...]
 domain_name_server number_of_dns x.x.x.x [x.x.x.x ...]
 domain_name name
 bootfile name or none if not needed
```

Here is a specific example:

```
enet
subnet_mask 255.255.255.0
 range 192.168.1.150 192.168.1.159
 router 2 192.168.1.1 192.168.1.3
 domain_name_server 2 192.168.1.1 192.168.1.3
 domain_name ussw.com
 bootfile none
```

If the server configuration file does not exist when sns_DHCPServerConfig() is called, a new one will be created from the cfgstr string that is passed as a parameter to sns_DHCPServerConfig(). Here is an example cfgstr definition from nsdemo.c,

```
char cfgstr[] = {
  "enet\r\n"
  "subnet_mask 255.255.255.0\r\n"
  "range 192.168.1.150 192.168.1.159\r\n"
  "router 2 192.168.1.1 192.168.1.3\r\n"
  "domain_name_server 2 192.168.1.1 192.168.1.3\r\n"
  "domain_name ussw.com\r\n"
  "bootfile none\r\n"
};
```

The DHCP server can service multiple interfaces. When configuring the DHCP server for multiple interfaces, the configuration information blocks follow one after the other. Here is an example cfgstr definition for two interfaces. (Also, the enet entry has been simplified.)

```
char cfgstr[] = {
  "enet\r\n"
  "subnet_mask 255.255.255.0\r\n"
  "range 192.168.1.150 192.168.1.159\r\n"
  "router 1 192.168.1.1\r\n"
  "domain_name_server 1 192.168.1.1\r\n"
```

```
  "domain_name none\r\n"
  "bootfile none\r\n"
  "wifinet\r\n"
  "subnet_mask 255.255.255.0\r\n"
  "range 192.168.2.10 192.168.2.19\r\n"
  "router 1 192.168.2.1\r\n"
  "domain_name_server 1 192.168.2.1\r\n"
  "domain_name none\r\n"
  "bootfile none\r\n"
};
```

2) Server lease file:

The name of the server lease file is defined as LEASE_FILE in dhcp.h. The default name is "dhcp.lea" with no path. Create this as an empty file when running the server for the first time. Otherwise, do not edit this file.

3) General configuration:

a) dhcp.h contains two configuration switches:

i) DHCP_PROBE : defining DHCP_PROBE enables an ICMP echo request probe of each potential address before the server gives it out. This enables the server to detect addresses in use and mark them as unavailable to give. #undef DHCP_PROBE disables it.

ii) DHCP_BROADCAST: The DHCP server will unicast all replies to the client's hardware address and to yiaddr (the IP address it is trying to give the client). This behaviour corresponds to #undef DHCP_BROADCAST in dhcp.h. If the TCP/IP stack on your client is unable to receive unicast messages before the IP address is configured, then #define DHCP_BROADCAST and all messages will be broadcast to all clients. Note that a smxNS client can receive unicast messages before the client is configured if DHCP is enabled.

b) #define DHCP_SERVER "server_name" in dhcp.h

c) #define DHCP 2 in nscfg.h

d) The DHCP server must be configured with a static IP address. The server IP address must be in the same subnet as the client address range set in the CONF_FILE.

e) The task stack size must be large, possibly as much as 5000 bytes.

Please read the information below under File Access for information on how the smxNS DHCP server access a filesystem.

DHCP Server Operation Restrictions

The smxNS DHCP server is not a complete implementation of RFC 2131. It is subject to the following limitations:

Options allowed for minimal implementation:

Option codes are from RFC 2132

Code	Bytes	Option
1	4	Subnet Mask
3	4n	Router

6	4n	DNS Server
15	n	Domain Name
50	4	Requested IP Address
51	4	IP Address Lease Time
—		
53	1	DHCP Message Type
54	4	Server Identifier

Client requests for options other than the ones above the line will be ignored.

Restrictions and Requirements:

1) smxNS's DHCP server will not interact with relay agents. The client must be on the same subnet as the server.

2) smxNS's DHCP server will assume there are no other DHCP servers on the same subnet.

3) smxNS's DHCP server will not have support for limited lease times. All lease times will be infinite.

4) smxNS's DHCP server will deliver a boot file name, but will not provide a mechanism for delivering the file.

5) The smxNS DHCP server only allows “dynamic allocation”. This means that addresses are always assigned from a pool. The smxNS server does not support the ability to always associate a single address with a particular client.

File Access:

The DHCP server uses persistent storage for:

1) Lease file - record of client bindings and

2) Configuration file - DHCP server configuration.

The lease file is accessed with the functions find_lease(), read_lease(), and write_lease(). The configuration file is accessed with the function read_conf(). File access is done using the C <stdio.h> functions. These file access functions should be changed to the appropriate methods for accessing non-volatile storage on your system. The include file dhcp.h includes <stdio.h> if EOF is not already defined. This include will also need to be changed if a different method of file access is used.

DHCP Testing

The smxNS DHCP client has been tested against the smxNS DHCP server and against the Internet Software Consortium DHCP server (www.isc.org).

The smxNS DHCP server has been tested against the smxNS client, and against Windows 95 and Windows 98 DHCP clients.

The details of the testing procedure are given below.

smxNS DHCP server testing against smxNS clients:

All addresses below are 192.168.1.xxx

The address range in dhcp.con is defined as 192.168.1.150 to 192.168.1.160 for this test (unless specified otherwise).

Acquire is performed by configuring client to use DHCP and starting fttest on the client. Release is performed by stopping the fttest client with the <ESC> key. For example, client B in test 2 performs: fttest 192.168.1.151, starts and acquires an address, then is stopped with <ESC> and address is released. Unless otherwise noted, all tests are performed on 80x86 platform, compiled with Borland C compiler v4.5. (Tests also verified for Microsoft C compiler v8.00). Note: fttest was a standalone USNet application which has been moved into nsdemo.c.

Test	Client	Action(s) (A = Acquire, R = Release address)
1	A	A 151
2	B	A 152 / R 152
3	B	A 152 / R 152
4	C	A 152 (simultaneous)
	D	A 153

** Reboot Clients C and D without releasing address

5	C	A 152
	D	A 153
6	B	A 154

** Reboot server and restart it

7	any	A previous address

** Reboot server, delete lease file, and restart server. Also, reboot client B.

8	B	A 154

** Reboot server, reboot all clients. Delete lease file. Edit DHCP configuration file to have range of one address (151).

9	A	A 151
10	B	Fails in acquiring address

What does each test prove?

1	Basic ability to acquire address.
2	Tests release of address.
3	Re-acquire gives client same address.
4	a) Client can reclaim unused address b) Simultaneous client requests work
5	Clients get same address back even if they didn't release it.
6	Client binding of B has address now in use by client C.

7	Server remembers client bindings through persistent storage
8	DHCP_PROBE defined: Address probe detects addresses in use even though there are not bindings for these clients (since we deleted the lease file). Server gives an address not in use. DHCP_PROBE undefined: Server will attempt to give an address in use with the address probe disabled. Client will send DHCPDECLINE because it also probes the offered address (and we haven't disabled this probe). Server handles DHCPDECLINE, and offers next address until it gets to one which the client accepts.
9	—
10	Server prints warning message, doesn't attempt to give address when there are no more left.

The smxNS DHCP server passed all the above tests for 4 server configurations:

DHCP_BROADCAST	DHCP_PROBE
defined	defined
defined	undefined
undefined	defined
undefined	undefined

BIG endian vs. LITTLE endian test:

The smxNS DHCP client was run on the SH3 platform (which is BIG endian). The smxNS server is run on 80x86 (LITTLE endian). This tests whether there are any byte ordering problems in how the server handles messages.

smxNS DHCP Server testing against Windows 95 and Windows 98 clients

Test	Client	Action(s)
1	A	A 151 / R 151
2	B	A 151
3	A	A 152
4	B	R 151

Windows acquire/release performed with winipcfg, multiple acquire/release, and renew all work.

DNS

DNS, or Domain Name System, is a protocol that allows a system to be located based on its host name. This introduces a useful level of indirection when specifying the end of a connection that can allow a system to continue to function even though changes may occur in the way the endpoints are attached to the Internet.

When the DNS macro is set to 2 in nscfg.h, the DNS look up will be invoked automatically for calls that accept a string to specify a host name. For example, Nopen() could use www.smxrtos.com rather than a dotted decimal IP address as the first parameter in the function call that specifies the host at the remote end of the connection.

In order for the DNS look up to succeed, at least one DNS server must be available to smxNS. If the smxNS system uses DHCP, then this information can be retrieved automatically as part of that process. Otherwise, the SetDNS() function can be used to manually specify DNS hosts. Up to NDNSS (default 2, set in nscfg.h) DNS servers may be specified. The DNS server can be located on another network, so long as a router is available.

The DNS resolver can also map from a local host name to an IP address using a legacy mDNS query. In this case, a DNS server does not need to be defined. Details are in the DNSresolve() section below.

Here is the function that allows a DNS server to be specified.

SetDNS()

```
int SetDNS (char *ip, char *index)
```

The function arguments are:

ip	IP address of the DNS server, as a string in dotted decimal format.
index	The index for the DNS server entry. Any existing entry will be overwritten. Indices 0..NDNSS-1 are valid.

The call returns 0 for success, -1 for failure.

Applications can also call the DNS resolver directly using the ***DNSresolve()*** function (described next).

DNSresolve()

Resolves a domain name to an IP address.

```
int DNSresolve(const char *fullname, IPaddr *iidp);
```

fullname	domain name
iidp	pointer to the address of the returned IP address

DNSresolve() stores the IP address at this location if *fullname* is non-zero.

DNSresolve() can start with either a domain name or IP address. If there's an @ in the name, ***DNSresolve()*** tries to find a mail host (IP address). If the first letter in the name is between 0 and 9, it's a pointer to an IP address, and ***DNSresolve()*** tries to find the domain name.
DNSresolve() can also attempt to obtain an IP address from a local host by sending a legacy mDNS query. In this case, the fullname parameter should end in ".local". For example, calling DNSresolve()

on "myhost.local" will return the IP address of host "myhost" if it is on the local network and running an mDNS Responder.

Return Value

>= 0	Successful lookup
-1	IP address could not be obtained from the DNS server(s)
ENOBUFS	Not enough buffers available for query (defined in **support.h**)

Example

```
IPaddr ipa;
char *hostname;

hostname="localhost";
stat = DNSresolve(hostname,ipa);
if (stat<0)
    ERROR();
```

FTP and TFTP

FTP and TFTP are file transfer protocols. FTP is implemented with TCP. TFTP is implemented with UDP so it is smaller but less reliable. TFTP is less secure and less capable, so it is of limited use.

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 XNS\netsrc**ftpc.c**, XNS\netsrc**ftps.c**, XNS\netsrc\tftp.c, and APP\DEMO**nsdemo.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.

The FTP server supports the internal commands APPE, CDUP, CWD, DELE, EPRT, LIST, MKD, MODE, NLST, PASS, PASV, PORT, PWD, RETR, RMD, RNFR, RNTO, STOR, STRU, TYPE, USER, QUIT, XCUP, XCWD, XMKD, XPWD and XRMD.

See Chapter 2 for more information on the FTP server and client test programs in nsdemo.c.

Start Server

These calls will start the servers. If you are using a multitasker, you will want to start these as tasks.

```
int FTPserv()

int TFTPserv()
```

The server never returns. In other words, it sits in an infinite loop.

Send File

This call sends a file.

```
int FTPput (char *host, char *lfile, char *rfile, char *userid, char
*pw, int mode)

int TFTPput (char *host, char *lfile, char *rfile, int mode)
```

The send file arguments are:

host Name of the server host. The form can be *host* or *host/network*.

lfile Name of the local file to be sent.

rfile Name of the file to be stored on the remote system.

userid Name of user account on the remote system. Not needed for TFTP.

pw Password for the user account on the remote system. Not needed for TFTP.

mode ASCII for a text file, IMAGE for a binary file.

The call returns 0 for success, -1 for failure. Note that the FTP protocol sends the user ID and password information as cleartext.

FTP & TFTP Examples

```
FTPput("XX", "t1", "/usr/aa/t1", "user", "password", IMAGE);
            /* t1 => host XX target file /usr/aa/t1 */

TFTPput("XX", "test1", "test1", ASCII);   /* test1 => host XX */
```

Receive File

This call receives a file.

```
int FTPget(const char *host, const char *lfile, const char *rfile,
           const char *userid, const char *pw, int mode)

int TFTPget (char *host, char *lfile, char *rfile,  int mode)
```

The receive file arguments are:

host Name of the server host. The form can be *host* or *host/network*.

lfile Name of the local file to be saved.

rfile Name of the file to be retrieved from the remote system.

userid Name of user account on the remote system. Not needed for TFTP.

pw Password for the user account on the remote system. Not needed for TFTP.

mode ASCII for a text file, IMAGE for a binary file.

The call returns 0 for success, -1 for failure. Note that the FTP protocol sends the user ID and password as cleartext.

FTPget Examples

```
FTPget("XX", "test1", "test1", "user", "pw", ASCII);
              /* test1 <= host XX */

FTPget("XX", "\tmp\t1", "t1", "user", "pw", IMAGE);
              /* \tmp\t1 <= host XX t1 */
```

HTTP Client

Support for retrieving a web page is available via the HTTPget() function in the http.c . The use of this function is demonstrated in the nstels.c application. This function is not intended for use as a general purpose browser, but rather as a mechanism for automated retrieval of information that is available via a web page.

When running nstels, you can log in to the smxNS Telnet server and then retrieve a web page by typing in the web server's host name followed directly by the path to the page. Here is an example session.

C:>telnet 192.168.2.2
smxNS skeleton Telnet server
smxNS Telnet Server
www.smxrtos.com/
Calling HTTPget() for host www.smxrtos.com with path /
1016 004: HEAD
1028 005: TITLE
…
3000 006: /TABLE
3006 005: /BODY
3018 005: /HTML
0400 000:

In this example, the default page from www.smxrtos.com is retrieved. The page contents are delivered via the HTTPdisplay() callback function, which parses the information into chunks of one HTML tag or word at a time. The callback function also includes flags and a length field, which are the first two values that appear on each line. In the nstels.c demo, the output is directed to the Telnet connection. An application that needed to extract a specific piece of information from a page could simply scan the results for a keyword and throw the rest away.

In addition to web servers on the Internet, a local server on the LAN should be a practical way to develop applications that use this function. The appendix contains a pointer to a simple web server that may be used this way. Also note that many network devices such as consumer routers provide web server based status and configuration, and these may be useful for a quick test of this function.

Retrieve a Web Page

This call starts the process to retrieve the contents of a web page.

```
int HTTPget (char *host, char *rsrc)
```

The arguments are:

host Name of the server host. The form can be *host* or *host/network*.

rsrc The path to the web page to be retrieved.

The call returns 0 for success, < 0 for failure.

Web Page Callback Function

This call returns the parsed web page to the application.

```
int HTTPdisplay (int flags, u8 *chunk, int len)
```

The arguments are:

flags Flags describing the returned HTML tag or parsed word.

```
0x0100  text
0x0200  precede with space
0x0400  follow with new line
0x10xx  html control <something>
0x20xx  html control off </something>
0x40xx  special character &something;
```

chunk The parsed HTML tag or word from the body text.

len The length of chunk. The last element from a page has lenth 0.

The application should return 0 for success.

Each time the function is called, either an HTML tag or a word from the body is delivered.

IGMP / Multicast

IGMP (Internet Group Management Protocol) allows sending messages to multiple hosts in a group.

smxNS 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 nscfg.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 IP_MC_DFLT_NETNO which is defined in nscfg.h.

The host group addresses range from 224.0.0.0 to 239.255.255.255.

The smxNS 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.

iperf

iperf is a program for measuring network performance. The smxNS version of iperf is based on iperf version 3.0.3. A repository of iperf code is at http://downloads.es.net/pub/iperf/. The original iperf files were adapted to work with smxNS and are located in the XNS/iperfsrc directory.

An iperf test is run by running two instances of the iperf program on two hosts that can communicate over a network. To enable the iperf program in the smxNS build, add all .c files in the iperfsrc directory to the project, and add an include path to XNS/iperfsrc

$PROJ_DIR$\..\..\..\XNS\iperfsrc\

and set TEST_IPERF_SERVER to 1 in APP/DEMO/nsdemo.c.

The other host can be a Linux computer. In order to run that instance, you can follow these steps:

Download and extract the source code archive

Move to the top level directory of the archive, build and run iperf using these commands

$ cd iperf-3.0.10
$ ./configure
$ make (many warnings may be reported)
$ cd src
$ ./iperf3 -v (this will show the version and confirm the executable is present)

To invoke iperf with smxNS running at 192.168.1.100, you can use

iperf3 -l 1460 -c 192.168.1.100 -V

iperf3 -l 1460 -R 192.168.1.100 -V

The first command is for smxNS to send bulk information and the second is for smxNS to receive bulk information.

After the test runs for 10 seconds, a summary of the test results is displayed, for example

```
Test Complete. Summary Results:
[ ID] Interval          Transfer     Bandwidth       Retr
[  4]  0.00-10.00  sec  12.5 MBytes  10.5 Mbits/sec    0    sender
[  4]  0.00-10.00  sec  12.5 MBytes  10.5 Mbits/sec         receiver
```

IPv6

IPv6 is a network layer that uses 128-bit addresses. An IPv6 node can assign an IP address automatically and does not depend on a DHCP server.

The smxNS IPv6 implementation does not currently support ICMP redirect, SLIP or PPP.

When the network interface is initialized (by calling the Portinit() function), the network prefix (fe80::/64) for the link local address and the interface ID are combined and the link local address is automatically created. The Interface ID is created based on the MAC address of the Ethernet interface.

RFC 2464 specifies a mechanism for generating a link local address based on the MAC address of an Ethernet interface. For example, given a MAC address of 34:56:78:9A:BC:DE, the Interface ID is as follows.

36:56:78:FF:FE:9A:BC:DE

The byte sequence FF:FE is inserved for the 4th and 5th bytes. The first byte is exclusive ORed with 0x02.

The corresponding link local address becomes

FE80::3656:78FF:FE9A:BCDE

If a Router Advertisement is received from a router, an IPv6 address will be created based on the Router Advertisement and the Interface ID.

The IPv6 address is stored in the fourth member of the struct NETDATA6 structure regardless of whether the address is statically configured or automatically set up.

The IPv6 stack should check for a duplicate IPv6 address. This check should be performed for both manually configured and autoconfigured addresses. The prospective address is a temporary address, and cannot be used until the check is completed.

The Duplicate Address Detection logic is executed once a candidate address has been created. The check typically takes one second.

The Duplicate Address Detection check is also performed on the link local address. Thus there is a delay following the time the network interface is initialized (by calling the Portinit() function) before the link local address can be used.

Configuration of Duplicate Address Detection is performed in the file XNS\include\nd6.h.

#define ND6_DAD_COUNT 1 /* DupAddrDetectTransmits */

Duplicate Address Detection is performed ND6_DAD_COUNT times every second. When ND6_DAD_COUNT is set to 0, Duplicate Address Detection is disabled, and the IPv6 address can be used immediately.

mDNS Responder

The mDNS (Multicast Domain Name Service) Responder allows local hosts on the network to discover services running on the smxNS system. For example, if the smxNS system is running a print server, the mDNS Responder can advertise and answer queries from other systems to help them locate this service on the network.

Example code for setting up and starting the mDNS Responder is provided in nsdemo.c. To enable the mDNS Responder:

- Set TEST_MDNS_RESP to 1 at the top of nsdemo.c.
- Set SNS_PROTO_IGMP to 1 in XNS\include\nscfg.h.
- Set USS_IP_MC_LEVEL to 2 in XNS\include\nscfg.h.
- Add XNS\netsrc\igmp.c to the project.

The service or services that the mDNS Responder maintains are organized as a set of associated records. The record types are defined as part of the DNS protocol, and this framework is extended in the mDNS protocol.

- A PTR (pointer) record associates a service with an instance name.
- A SRV(service) record associates an instance with a listening port and a network host name.
- A TXT (text) record associates an instance with a text string, which may be empty.

Here is an example of a data structure used to initialize the mDNS Responder:

```
GLOBALCONST RR_RECORD dns_rec[] =
{
  {“superprint”, “_printer._tcp”, “paper=A4”, 631, NULL, 0}
};
GLOBALCONST RESPONDER_CONTEXT mdns_rc =
{
   dns_rec,
   1
};
```

This structure provides the information used to construct the PTR, SRV and TXT records that are used in responses to mDNS queries.

The fields are used as follows:

“superprint”: This is the instance name for the service. It is intended to be a user-friendly name, and some implementations may provide a mechanism to allow the end user to configure this name. A service instance name must be unique on the local network. If the provided name is not unique, the mDNS Responder will modify the name by appending an index so that it becomes unique. This is handled as part of the mDNS protocol.

“_printer._tcp”: This is the service name. There are well known service names such as _printer._tcp and _ftp. _tcp, and these are currently registered at the Internet Assigned Numbers Authority.

“paper=A4”: This is the string used for the TXT record. In order to configure multiple key/value pairs in a text string, use the separator 0x01 between pairs.

631: This is the port that the service listens on. This information is used in creating the SRV record.

NULL, 0: These are empty fields that can be used to define subtypes of the service. An example appears below.

The RESPONDER_CONTEXT structure contains a pointer to the record initialization data and a count of the number of services. Multiple services can be advertised by placing additional entries in the RR_RECORD structure.

Service name subtypes are useful in some circumstances to allow mDNS queriers to find a subset of instances that support a service. To define one or more subtypes that are associated with a service, create a list of subtype strings, and include a pointer to this list in the corresponding RR_RECORD definition. For example

```
GLOBALCONST char *subtypes[] = {“_coremote._sub”, “_dbupdate._sub”};
GLOBALCONST RR_RECORD dns_rec[] =
{
  {“superprint”, “_http._tcp”, “”, 80, subtypes, 2}
}
```

Here the subtypes _coremote._sub._http._tcp and _dbupdate._sub._http._tcp are subtypes of the service _http._tcp. The value (2) that follows the pointer to the list of subtypes is the number of strings in the list.

To advertise additional services you can simply add another resource record to the list. For example, you might add an ftp service with the instance name “helper” which listens on port 23 and has no subtypes:

```
GLOBALCONST RR_RECORD dns_rec[] =
{
  {“superprint”, “_http._tcp”, “”, 80, subtypes, 2},
  {“helper”, “_ftp._tcp”, “”, 23, NULL, 0}
};

GLOBALCONST RESPONDER_CONTEXT mdns_rc =
{
   dns_rec,
   2
};
```

In order to run the mDNS Responder, pass the initial configuration using the sns_mDNSResponderInit() function, and then repeatedly call the sns_mDNSResponderCheck() function.

```
void mdns_task_main(uint dummy)
{
  int i1;
  struct RESPONDER_STATE *mdns_state;
  int name_established;
  name_established = 0;
  i1 = sns_mDNSResponderInit(&mdns_rc);
  if (i1 >= 0)
  {
    do
    {
      mdns_state = sns_mDNSResponderCheck();
      if ((mdns_state->state == MDNS_RESPONDING) && (name_established == 0))
      {
        DEBUG_MSG3_PAR1(“Instance name is %s\n”, sns_mDNSGetInstance(0));
        name_established = 1;
      }
    } while (mdns_state->error == 0);
    sns_mDNSResponderShut();
  }
  return;
}
```

The sns_mDNSResponderCheck() function maintains the state machine and receives and responds to mDNS traffic as needed. The function returns a structure that includes an error indication and the state machine state. If an error occurs, the mDNS Responder should be shut down and restarted. The state information is useful to determine when the instance names have been established. Once the state reaches MDNS_RESPONDING, the names have been established. Under normal circumstances the name will not change, but if there is another host on the local network configured to use the same name for a service, then an index will be appended.

The sns_mDNSGetInstance() function can be used to obtain a pointer to an instance name. The value passed to the function is a 0-based index using the same order as the list of services used to initialize the mDNS Responder.

The sns_mDNSSetUniqueCallback() function can be used to specify a user-defined function to make the instance name unique. This must be called following the call to sns_mDNSResponderInit().

The sns_mDNSResponderShut() function sends goodbye messages to time out the advertised services and closes the sockets associated with the mDNS Responder.

The mDNS Responder also maintains an address record (A record) that maps the smxNS host name to the network interface IP address. The smsNS host name may also be modified by the mDNS Responder. The name is established once the state reaches MDNS_RESPONDING.

The host name may be updated at runtime by calling the function SetHostname(char *) and passing a pointer to a host name string. The current value of the host name string can be retrieved by calling GetHostname(), which returns a pointer to the string. The host name string does not include the domain name when used with these functions.

The mDNS Responder implementation is based on the current Internet-Drafts for Multicast DNS and DNS-Based Service Discovery as of April 2013. The mDNS Responder was exercised using the Mac OS X command line dns-sd utility and the Linux command line avahi-browse utility. Sample command lines follow

> dns-sd –B _printer._tcp

```
Browsing for printer.tcp
Timestamp    A/R    Flags    if    Domain    Service Type    Instance Name
11:47:27.564 Add        2     4    local.    _printer._tcp   superprint
```

> avahi-browse –r –t _printer._tcp

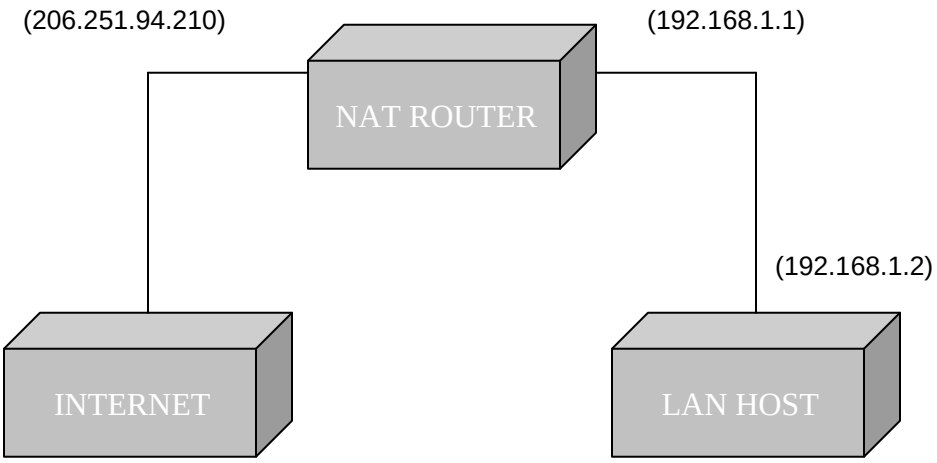
```
+   eth0 IPv4 superprint    printer
=   eth0 IPv4 superprint    printer local
   hostname = [MDI-System.local]
   address = [192.168.1.12]
   port = [1234]
   txt = []
```

NAT

Note: NAT is available as an extra-cost option for smxNS.

smxNS 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 smxNS NAT router. ICMP, UDP, TCP and other protocols may be used through a smxNS 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 smxNS as a NAT router.

NAT Configuration

In file include**nscfg.h** set RELAYING to 1 to enable smxNS to relay between interfaces:

Change:

```
#define RELAYING 2
```

To:

```
#define RELAYING 1
```

Set the NAT flag to each interface that should behave as the router for a private network. The following Portconfig() example shows one private (internal or LAN) interface and one public (external or WAN) interface.

```
Portconfig("eth0", "IP", "192.168.1.1");
Portconfig("eth0", "NAT", "ENABLE");
Portconfig("eth1", "IP", "206.251.94.210");
```

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 smxNS in general, the debugging trace level may be used to enable printf() debugging from the NAT module. By default, if SNS_DEBUG_LEVEL is 3 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 SNS_DEBUG_LEVEL is 5 or greater, NAT will print out:

Table additions/removals

If NAT debugging is to be isolated from the rest of smxNS debugging, set SNS_DEBUG_LEVEL to 1 (or the appropriate value) and modify netsrc\nat.c as follows:

```
#include "smxns.h"

#undef SNS_DEBUG_LEVEL

#define SNS_DEBUG_LEVEL 5
```

NC-SI

Note: NC-SI is available as an extra-cost option for smxNS.

NC-SI (Network Controller Sideband Interface) is a protocol that allows a host processor and Ethernet controller acting as a BMC (Baseboard Management Controller) to control one or more NICs (Network Interface Controllers). This design allows flexible out-of-band management of the NICs and is applied in certain networking equipment.

The NC-SI feature in smxNS is implemented by replacing the usual direct interface between the Ethernet controller and the PHY with one that uses the NC-SI protocol to allow an Ethernet controller to communicate with one or more NICs. With NC-SI, operations such as checking PHY link status are performed by calling a function that creates an Ethernet frame that is directed to a NIC, and then listening for a response frame from the NIC.

The smxNS NC-SI implementation provides most of the commands described in the Network Controller Sideband Interface Specification and allows AENs (Asynchronous Event Notifications) to be received via a callback function.

The NC-SI feature is driven by the NC-SITask() function that is launched as part of smxNS start up. The first steps of the task are to configure the attached NIC including setting the MAC address so that it can send and receive Ethernet frames.

There are approximately 25 NC-SI functions that a network application can call to issue an NC-SI command. All command functions include parameters to specify the NIC package and channel and fill in an ncsi_status structure to provide details on how the command executed. The call to the command function blocks while it is executing, and this includes exchanging messages with the NIC. If the NIC responds to the command, the function will return 0.

Here is an example function prototype

```
int NCSIGetNCSIStatistics(uint pkg, uint ch, struct ncsi_ncsi_stats
*ns, struct ncsi_status *s);
```

This function returns statistics using the ncsi_ncsi_stats structure. Here is an example use of the function.

```
struct ncsi_ncsi_stats ns;
struct ncsi_status resp;
int stat;
uint pkg = 0;
uint ch = 0;
stat = NCSIGetNCSIStatistics(pkg, ch, &ns, &resp);
DEBUG_MSG3_PAR1("Get NC-SI Statistics returns %d\n", stat);
if (stat == 0)
{
  DEBUG_MSG3_PAR2("  Response 0x%04x Reason 0x%04x\n",
                  resp.response_code, resp.reason_code);
  DEBUG_MSG3_PAR1("NC-SI Commands Received %d\n",
                  ns.ncsi_commands_received);
```

```
  DEBUG_MSG3_PAR1("NC-SI Control Packets Dropped %d\n",
                  ns.ncsi_control_packets_dropped);
  DEBUG_MSG3_PAR1("NC-SI Command Type Errors %d\n",
                  ns.ncsi_command_type_errors);
  DEBUG_MSG3_PAR1("NC-SI Command Checksum Errors %d\n",
                  ns.ncsi_command_checksum_errors);
  DEBUG_MSG3_PAR1("NC-SI Receive Packets %d\n",
                  ns.ncsi_receive_packets);
  DEBUG_MSG3_PAR1("NC-SI Transmit Packets %d\n",
                  ns.ncsi_transmit_packets);
  DEBUG_MSG3_PAR1("AENs Sent %d\n", ns.aens_sent);
}
```

Definitions for the NC-SI structures are in XNS/include/ncsi.h. You can review these structures to see which fields can be used to pass inbound parameters or retrieve outbound parameters.

An application can access the information in AEN packets by registering a callback function that is called when an AEN is received. Note that this callback function is executed in the context of smxNS's high priority NetTask(), so it should perform its function promptly. The AEN information is delivered in an ncsi_aen_info structure and includes the channel ID, AEN type and 4 bytes of data specific to the AEN. Here's an example of the use of an AEN callback.

```
/* From XNS/include/ncsi.h */
struct ncsi_aen_info {
  int channel_id;
  int type;
  int data;
};

/* Network application code */
void callback(struct ncsi_aen_info *p)
{
  Nprintf("AEN status type %d\n", p->type);
}

NCSIRegisterAENCallback(callback);
```

PPPoE

Note: PPPoE is available as an extra-cost option for smxNS.

PPPoE (Point-to-Point Protocol Over Ethernet) encapsulates PPP frames in Ethernet frames. This is useful in certain applications, especially in DSL-related equipment that uses PPP features for access control and accounting. smxNS provides support for building both PPPoE Hosts and PPPoE Access Concentrators.

PPPoE Configuration

Here are the necessary steps to configure and build PPPoE with smxNS.

1. Test smxNS on the target without PPPoE integration. Run nsdemo with PPPoE disabled in nscfg.h.

Note that PPP must be configured even though there may not be a serial interface on the target.

2. Build the PPPoE version

If you purchased the PPPoE option, there should be a target defined that will build smxNS with PPPoE support.

3. Define the target interface

In configuring the PPPoE interface, change the link layer setting from Ethernet to PPPOE.

For example:

Previously, the configuration may have been:

```
Portconfig("eth0", "LINK", "Ethernet");
```

It should then be changed to the following:

```
Portconfig("eth0", "LINK", "PPPOE");
```

If the IP address is defined by the Access Concentrator, define the IP address as 0.0.0.0. Additionally, create an entry for the peer host so that PPP can store the remote IP address for later.

```
"ac", "pppoe", C, X, EA0, ROUTER, 0, 0, 0,
"test", "pppoe", C, X, EA0, 0, PPPOE, PCI, 0,
```

If smxNS is being run as an Access Concentrator, additional entries in the netdata[] table can be set up so that they are distributed to PPPoE hosts. Here is an example configuration for this

```
"host", "tnet", CC, W, EA0, PROXYARP, 0, 0, 0,
"test", "tnet", CC, X, EA0, 0, PPPOE, NE2000, "IRNO=10 PORT=0x300",
"test", "enet", CC, Y, EA0, 0, Ethernet, NE2000, "IRNO=5
PORT=0x320",
"gw",   "enet", CC, Z, EA0, ROUTER, Ethernet, 0, 0,
```

In addition, for use as an Access Concentrator, the following settings are suggested for nscfg.h.

```
#define RELAYING 1
#define USS_PROXYARP
```

3. Run nsdemo with PPPoE enabled in nscfg.h.

4. Further configuration items specific to the operations of the PPPoE host are contained within netsrc\pppoe.c. Edit the file configuration options as necessary. The default settings should be a reasonable starting point.

The corresponding file for the Access Concentrator version is netsrc\pppoeac.c. The following notes describe the configurable values at the top of the file.

```
#define PPPOE_TIMER_GRANULE 1000
```

The PPP timeout function for PPPoE sessions will be called using a period defined by this constant. The default value sets a frequency of once per second.

```
#define PPPOE_ACNAME "AC-0000"
```

This string is delivered in the AC-Name tag when the Access Concentrator sends its PPPoE Active Discovery Offer (PADO) packet. This Access Concentrator name may be useful to the PPPoE host in deciding whether or not to set up a PPPoE with this Access Concentrator. In practice, this information is commonly ignored.

```
#define MAX_SERVICE_NAME_LEN 16
```

This defines the size of the buffer that stores the string associated with the Service-Name tag. The Access Concentrator is set up to use a liberal policy on service names, accepting any name that is suggested by the host. This policy is suggested in the Security Considerations section of RFC 2516.

Similar buffer length definitions exist for the Host-Uniq, AC-Cookie and Relay-Session-Id tags.

```
#define PNETS 2
```

This defines the number of physical network interfaces. State information for PPP sessions is stored in the network interface structure nets[]. Typically, each network interface is associated with a physical network interface, which may be a serial interface for PPP, or an Ethernet interface for a PPPoE host. A PPPoE Access Concentrator may support multiple PPPoE sessions over the same Ethernet interface.

In order to support this, some interface structures are used as "virtual interfaces". Interfaces with an index between 0 and PNETS - 1 correspond to physical interfaces. Indices between PNETS and NNETS - 1 correspond to virtual interfaces, which are mainly used to store PPP session state.

Note that NNETS which is defined in nscfg.h needs to be larger than the number of physical interfaces. The default value of 4 happens to provide a little room for this.

The Access Concentrator will provide up to NNETS – PNETS PPPoE sessions. Once this limit is reached, the Access Concentrator will respond to incoming PPPoE Active Discovery Request (PADR) packets with a PPPoE Active Discovery Session-confirmation (PADS) packet that contains an AC-System-Error tag.

SLIP

SLIP is a link layer that connects two hosts over a serial connection. In order to configure an interface to use SLIP, the SLIP protocol table should be specified using the “LINK” keyword when calling Portconfig(). An example is provided in the Configuration chapter.

Using SLIP with Windows Computers

An smxNS system running SLIP may be connected to a larger network by using a Windows XP computer as a gateway. In order to set up a SLIP connection on Windows XP, follow these steps.

1. Select Start, then right click on My Network Places and select Properties.
2. Select "Create a new connection". Select "Next".
3. Select "Set up an advanced connection".
4. Select "Connect directly to another computer".
5. Select "Guest".
6. Enter a name for the connection, for example "SLIP".
7. Select the serial port for the connection.
8. Select "Finish".
9. Windows displays a connection window. Select "Properties".
10. Select "Configure..." from the "General" tab.
11. Deselect all checkboxes.

12. Set speed to 115200 bps (this is the smxNS default, adjust as needed).
13. Select "OK" to close the modem configuration window.
14. Select the "Options" tab in the SLIP properties window.
15. Deselect all except "Display progress".
16. Select the "Networking" tab.
17. Select "SLIP: Unix Connection".
18. Under "This connection uses the following items:", select only "Internet Protocol (TCP/IP)" and "QoS Packet Scheduler".
19. Again under "This connection uses the following items:", highlight "Internet Procotol (TCP/IP)" and select "Properties".
20. Select "Use the following IP address:", and enter "192.168.2.1".
21. Select "Advanced". Deselect all "Advanced TCP/IP Settings".
22. Select the "WINS" tab. Deselect "Enable LMHOSTS lookup" if it is selected. Select "Disable NetBIOS over TCP/IP".

Please refer to the section on Null Modem Links in the PPP chapter for additional details on direct serial links and networking through a Windows system.

SMTP

Note: SMTP is available as an extra-cost option for smxNS.

SMTP is the Simple Mail Transfer Protocol, used for sending email. The optional smxNS email client package allows an application to send an email message from a host connected to the internet.

`int SMTPsend(struct outgoing_email *request)`

The argument is:

`request` Pointer to an outgoing_email structure. This structure is filled in to describe the email that should be sent.

```
struct outgoing_email {
  const char *mailserver;
  uint port;
  const char *to;
  const char *cc;
  const char *from;
  const char *subject;
  const char *username;
  const char *domain;
  const char *password;
  const char *type;
  uint flags;
  time_t localtime;
  uint status;
};
```

mailserver is a pointer to the name of the mail server that should receive the email.

port is the UDP port for the SMTP transfer.

to, cc, from, and subject are the familiar values one sees in email headers.

The to and cc strings can contain commas to specify multiple addresses.

The from field can be left blank if the username and domain fields are filled in.

The username, domain and password fields are also used as part of the authentication exchange if the server requests CRAM-MD5, NTLM, LOGIN or PLAIN authentication methods.

The type string is used to fill in a "Content-type:" email header if it is present.

The flags field can contain the SMTP_USE_SSL bitflag to direct the email client to set up an SSL connection to the email server. It can also contain the SMTP_MUST_AUTHENTICATE flag to force the SMTP transaction to be authenticated. If the server does not attempt to authenticate and this flag is set, the call will fail and no information will be transferred.

The localtime field can be filled in with the local time in time_t format, or it can be left empty and SMTPsend() will call smx_SysStimeGet() to retrieve time information.

The call returns 0 for success, < 0 for failure.

More detailed information on error returns can be obtained by examining the status field in the outgoing email structure after the function call completes. The following values may be returned: SMTPC_SUCCESS, SMTPC_GENERAL_ERROR, SMTPC_CONNECTION_ERROR, SMTPC_AUTHENTICATION_ERROR, SMTPC_PARAMETER_ERROR, SMTPC_RCPT_ERROR.

The body text of the email message comes from a callback function that the email sending application needs to implement. The function prototype looks like this

int SMTPgetdata(char *buff, int buflen, struct part_info *i);

The first time SMTPgetdata() is called, buflen will be set to 0. This is a hint that the application should fill in the the part_info structure as a response to this call. The following call will have a non-zero value to indicate it is time to pass the content data. Here's the part_info structure.

```
struct part_info {
  int passthrough;
  const char *encoding;
  const char *type;
  const char *filename;
};
```

The part_info structure is cleared before SMTPgetdata() is called, so if no options are needed the structure can be left alone. Here are some notes on the options.

passthrough is a flag to indicate if the data returned in buff should be returned directly to the SMTP server. If passthrough is left as 0, the outgoing buffer will be terminated with CRLF before passing the information to the server, otherwise the data is transferred without modification.

encoding, type and filename are used to set up headers if needed. Example values are "base64", "application/octet-stream" and "example.bin" respectively.

For the initial call in a sequence (where buflen = 0), SMTPgetdata() should return 0 to indicate that a text message is about to be returned, and 1 to indicate that a multipart message will be returned.

After the initial call, the SMTP sending function will provide a buffer to hold the outgoing content and will indicate the amount of room in the buffer in buflen. The SMTPgetdata() implementation

should fill in buff and return the number of bytes transferred as the return value. When there is no more data left to transfer, SMTPgetdata() returns 0.

For multipart transfers, SMTPgetdata() will continue to be called after the first part is complete. When all parts have been transferred, SMTPgetdata() should return –1.

SNTP

SNTP stands for Simple Network Time Protocol, and it is a simplified form of NTP, the Network Time Protocol. An NTP server can service both NTP and SNTP clients. smxNS includes an SNTP client function, so that time information can be retrieved from a time server.

NTP time servers are capable of delivering 64 bits of time information, or better, with a resolution on the order of nanoseconds. For the simplified SNTP version, 64 bits of time information are used, and this can be used to correct error in the local time source. The time returned is based on an epoch of January 1, 1900, and this may need to be converted for use with the local time support. The NTP server provides universal time (UTC), and will need to be adjusted for time zone and daylight savings time if desired.

Get Time using SNTP

The routine will attempt to retrieve the time from the specified NTP server. The function accepts a string for the host name, and this can be either be an IP address or a name that can be looked up via DNS.

Three functions are supplied that work together to retrieve time information from an NTP server and adjust the local time.

```
s64 sns_SntpGet(char *timeserver)
```

The ***sns_SntpGet()*** return codes are:

!= 0	Adjustment for NTP time
0	Time request failed

The 1900 epoch for NTP timestamps may be different from the convention supported by the C library or other system software. For example, a system might use an epoch of January 1, 1970. The sample code in nsdemo.c translates to this epoch by adjusting based on a point in time that is common to both systems. In this case, the calculation uses UTC on January 1, 1972, which is established to be 2,272,060,800 as an NTP timestamp.

```
void sns_LocalNtpTimeAdjust(s64 adj)
```

sns_LocalNtpTimeAdjust() adjusts an internally maintained time offset so that a local time reading can be combined with the NTP server information to provide an adjusted time.

```
u64 sns_LocalNtpTimeGet(void)
```

sns_LocalNtpTimeGet() reads the local time using the local time function and combines it with the offset provided by the NTP server. The return value is an unsigned 64-bit value in NTP format. The upper 32 bits represent whole seconds and the lower 32 bits hold fractional seconds. The epoch for NTP is January 1, 1900.

Here is an example that combines these three functions to update and then retrieve the adjusted current time in NTP format, adjusted to include only include the elapsed seconds information.

```
u32 ntptime;
sns_LocalNtpTimeAdjust(sns_SntpGet(SNTPSERVER));
ntptime = sns_LocalNtpTimeGet() >> 32;
```

The combination of sns_LocalNtpTimeAdjust() and sns_SntpGet() should be called periodically to adjust for drift in the local time source. sns_LocalNtpTimeGet() can be called any time to get the current reading of the time adjusted by the offset provided by the NTP server.

A public pool of time servers has been organized, and is available using the name “pool.ntp.org”. More information on this project is available at http://www.pool.ntp.org. This should be a good choice for the timeserver name, so long as DNS support is available.

The NTP messages are sent over UDP, and there is the chance that they will be lost. This function does not contain retry logic, but this could be implemented at the application level. The tests we conducted show the communication with servers to be reliable, despite the transport protocol.

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.

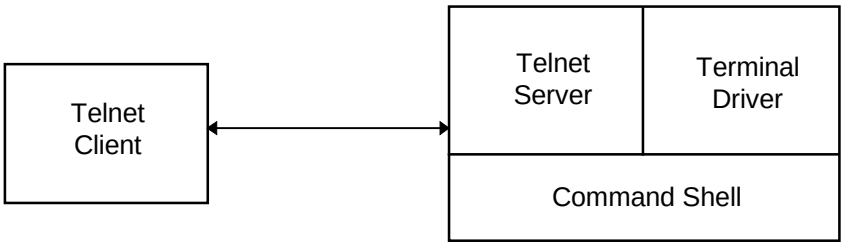


Figure 7-1: TCP Remote Terminal Access

smxNS Telnet support is implemented as a server function that handles Telnet sessions. The function takes a connected Telnet connection as an argument, and uses a callback function to submit the command line supplied by the user and retrieve a response.

8. Point To Point Protocol (PPP)

Overview

The Point to Point Protocol (PPP) is a link layer protocol that connects two hosts over a serial connection. This is commonly used in data acquisition and Internet connectivity. PPP is commonly used to provide TCP/IP networking for end node systems that have at least one serial port, but no Ethernet controller.

For dial-up purposes (that is, using a modem and telephone line), a dialer is included. It is also an option to use a personal or vendor specific dialer in place of our mechanism, though we cannot support this. smxNS dialing does require the use of a precompile-time interpreter, provided in DOS executable format with source code. A compiler/linker for the development OS should not have problems turning this into an executable file. It is written in ANSI C.

smxNS PPP is based on RFC 1661, and this is the most current specification of PPP at the time of this writing. Related RFCs that were used in the PPP implementation include:

1332 IPCP
1334 PAP
1662 HDLC framing
1990 MP
1994 CHAP
2433 MS-CHAP

PPP in Theory

The Point to Point Protocol is not a server/client system. It is commonly used that way, but only because it is convenient to do so. We will use the following conventions: the side who initiates communications is the client and the side who is waiting to be contacted is the server. The host is the side of reference (i.e. “this” side); the peer is the side opposite the reference (i.e. "that" side). So the server can be peer or host and the client can be peer or host (or vice versa for either). The peer may also be called a remote host.

There are two necessary phases within PPP: Link Control Protocol phase and Network Control Protocol phase. These are referred to as LCP and NCP respectively. The LCP used by PPP is most recently specified in RFC1661. The NCP phase is specified by the network layer protocols used. In smxNS, the Internet Protocol (IP) is used in our network layer, so we use the Internet Protocol Control Protocol (RFC1332). A third phase, commonly included at the end of the LCP phase and before the NCP phase, is authentication. Generally, the Password Authentication Protocol (PAP from RFC1334) or Challenge Handshake Authentication Protocol (CHAP from RFC1994 or MS-CHAP from RFC2433) is used.

LCP Phase

The LCP phase determines the requirements and capabilities of both sides of a PPP link before actual communications begin. Let us refer to the client as the host and the server as the peer. Typically, the client (host) sends a configure-request packet ("conf-req") to the server (peer) to initiate communications. This packet contains a list of options that the host would prefer to use in the future. The peer should respond with either a configure-acknowledge ("conf-ack") or a configure-negative-acknowledge ("conf-nak") according to its satisfaction with the options within the conf-req. Also, when a peer receives a conf-req, it will send a conf-req back with the options it would like to use, so the process is mutual. If the host receives a conf-nak, then the peer was dissatisfied with the options enabled and the host must reconfigure itself and send a new set of options corresponding to the wishes/abilities of the peer in a new conf-req. If the options nak'd (negatively-acknowledged) are necessary for correct functioning of the host, the host must terminate the link negotiations.

If the host received a conf-ack, the host must wait to receive the peer's conf-req. If the host gets the conf-req and the options requested are acceptable, the host must send a conf-ack. At this point, the LCP phase is Open and the next phase should be initiated. If a timeout occurs before the conf-req is received, the host must re-send its conf-req and restart its half of the negotiations.

Authentication Phase (PAP/CHAP)

Authentication is used to decide what level of access the authenticatee should have to the authenticator. This is usually a "all- or-nothing" sort of thing. Using the same pair from LCP as an example, we continue on to authentication. Let us assume that the peer (server) requested PAP in its conf-req. This would require the host (client) to now send an authentication-request ("auth-req"). This packet includes a user ID and a password. smxNS does not encrypt these. If the peer finds the user ID and the password acceptable, the host should receive an authentication-acknowledge ("auth-ack") and authentication would be completed. If the peer finds the user ID and password unacceptable, the host should receive an authentication-negative-acknowledge ("auth-nak") and the link should be terminated by the peer (this is not necessarily true, however).

Let us go back to the end of LCP and assume that the peer had requested CHAP in its conf-req instead of PAP. The peer (server) would then send a challenge (some unique value to be hashed). The host (client) would then tag on its password (secret) to the challenge and hash it with MD5. It would place this hashed value in a response and send it back. The peer would hash what should be the same thing on their side and compare it to the original. If they match, the peer would send a success packet and authentication would be concluded; otherwise, it would send a failure packet and the link should terminate (although it may continue on). There are two distinct advantages about CHAP over PAP. Primarily, the raw password is never sent over the network (this does mean that both sides must maintain a copy of the password). Secondarily, the authenticator authenticates the authenticatee (i.e. sends the first packet) rather than forcing the authenticatee to authenticate itself to the authenticator.

MS-CHAP is different than CHAP. It makes use of the MD4 algorithm to hash the password.

Mutual authentication is appropriate, and often suggested as a means of increasing security, though most "servers" will not allow this. smxNS will allow this, though some work may need to be done for its role as an authenticator. smxNS has no pre-configured mechanism for storing a table of User IDs and secrets (passwords) for potential peers, though the structure to access that table is in place.

NCP Phase

Once the LCP is finished (and authentication if necessary), the NCP phase(s) must start. We use IPCP, as mentioned earlier. The behavior is nearly identical to the LCP phase, but its purpose is not

to set up link layer communications but to set up network layer communications for the IP protocol, including the IP address.

Optionally, smxNS allows a host to use Van Jacobson TCP/IP header compression. It is negotiated during IPCP. Throughput should increase slightly when using this.

PPP in Practice

Usage

Set up the network interface to an appropriate state. Here are examples for use with PPP:

```
Portcreate(“ppp0”);
Portconfig(“ppp0”, “IP”, “0.0.0.0”);
Portconfig(“ppp0”, “LINK”, “PPP”);
Portconfig(“ppp0”, “DRIVER”, “NS16550”);
Portinit(“ppp0”, “IRNO=3 PORT=0x2F8 CLOCK=115200 BAUD=9600”);
```

"ppp0" — A smxNS host that connects to other hosts through a null modem. It has no IP address assigned statically so it is assumed that the peer will provide one during IPCP.

```
Portcreate(“pppd0”);
Portconfig(“pppd0”, “IP”, “0.0.0.0”);
Portconfig(“pppd0”, “LINK”, “PPP”);
Portconfig(“pppd0”, “DRIVER”, “NS16550”);
Portconfig(“pppd0”, “DIAL”, “ENABLE”);
Portinit(“pppd0”, “IRNO=4 PORT=0x3F8 CLOCK=115200 BAUD=9600”);
```

"pppd0" — A smxNS host that connects to other hosts through a modem. It has no static IP address so it is assumed that the peer will assign one during IPCP. The only difference between this interface configuration and "ppp0" is the “DIAL” attribute is turned on. The macro, DIALD, needs to be configured to 1 in include\pppconf.h in order to use this entry.

```
Portcreate(“pppd1”);
Portconfig(“pppd1”, “IP”, “206.251.94.242”);
Portconfig(“pppd1”, “LINK”, “PPP”);
Portconfig(“pppd1”, “DRIVER”, “NS16550”);
Portconfig(“pppd1”, “DIAL”, “ENABLE”);
Portconfig(“pppd1”, “PEER”, “206.251.94.243”);
Portinit(“pppd1”, “IRNO=3 PORT=0x2F8 CLOCK=115200 BAUD=9600”);
```

"pppd1" — A smxNS host that connects to other hosts through a modem. This host has an IP address. If a peer dials into it, this host will be able to assign the peer the IP address from "PEER". The macro, DIALD, will need to be configured to 1 in include\pppconf.h in order to use this entry.

Note that for PPP connections, the PPP peer will act as the default router unless another default router is configured. If necessary, the host may have other interfaces to which subnetting still applies. If anything is not in that subnet, the default router, specified by SetDefaultRouter(), will be used.

The BIN directory contains the file prefrmt.exe. This is in DOS executable format. The source code for this file is in the BIN\PREFRMT directory under the name prefrmt.c. If the development machine cannot execute DOS applications, prefrmt.c should be compiled for the appropriate OS. The source code's only dependancy is having script.h and script2.h in the include path. If script2.h does not exist,

make an empty file in the same directory as script.h called "script2.h" (it is normally generated during the standard build process). Make sure the resulting executable file ends up in the BIN directory.

If scripted dialing will be used (DIALD == 1), the script files may require modifications to interact more correctly with the modem being used.

dial-in.scr — This is used to allow a remote host to dial into smxNS. It uses manual answer mode but may be changed to use auto answer.

dial-out.scr — This is used to dial out to a remote host over a line. At least the phone number will have to be changed along with any special considerations for flow control or other modem or line specific properties.

dial-dwn.scr — This is made to de-initialize a modem after a session has ended. This is not absolutely necessary, but it makes it easier to bring the modem up the next time.

See the "Scripting" section later in this manual for assistance with the function of these files.

When first starting or if scripts or pppconf.h options are changed, consider turning PPP_DEBUG to 1. This will make changes and their effects more readily apparent. It will also reveal areas that may need adjustment.

Configuration

All PPP related macro values are defined in include\pppconf.h. They are quite extensive and some of them interact with each other, so it is important to understand what they do when changing them. In the state it is shipped in, PPP should be able to establish a link with most implementations using a null modem.

PPP_DEBUG

smxNS PPP comes with a module called pppdebug.c which can parse and print out, with Nprintf(), the frames that are sent and received by the link. This macro enables/disables this capability. It is useful to set this macro to 1 while configuring the PPP link. Once the link is behaving appropriately, this can be set to 0 and only warnings and errors will be printed out with Nprintf(). SNS_DEBUG_LEVEL takes precedence over this value.

DIALD
This specifies whether PPP will use the dialer automatically. See later sections of this document for further information.

DBUFFER
PPP starts negotiations when the application forces the link up explicitly or when the first datagram is transmitted. This option tells PPP to buffer datagrams while the link comes up. By default this is on.

DBUFFER_SZ
This tells PPP how many buffers to queue up while waiting for the link to become established. The default value is NBUFFS/NNETS so that PPP doesn't starve the rest of smxNS out of buffers but has enough to effectively perform the function of dial-on-demand.

IDLE_TOUT
This value specifies the amount of vacant time in seconds (TimeMS()/1000) in the link before it is closed manually. As delivered, it is disabled with a value of 0.

ECHO_TOUTMS
This value specifies the amount of time (in TimeMS() milliseconds) in an open link between echo-request packets being sent. This can be used to check the link quality or to check if the peer has disappeared (if the peer loses connectivity without warning).

ECHO_RETRIES

This value specifies the number of echo-request packets sent without a reply before the link is deemed bad and is set to close. Setting ECHO_TOUTMS to a positive non-zero value enables this.

PPP_USERID

Because the PPP authentication user ID may differ from the application level user ID, we provide this value. It defaults to the application layer user ID. This value is set in Portinit() and can be changed thereafter through the ioctl routine (see the PPP ioctl Routines section).

PPP_PASSWD

Because the PPP authentication password may differ from the application level password, we provide this value. It defaults to the application layer password. This value is set in Portinit() and can be changed thereafter through the ioctl routine (see the PPP ioctl Routines section).

AUTHENT

We support PAP, CHAP and MS-CHAP authentication. This macro specifies which of those we will allow a peer to use on us. For client-oriented applications, this will usually be set to allow all three. For server-oriented applications, most people turn this off to save code space. All three are enabled by default.

USE_NT

Set this to one to use NT style challenge response. Set to zero for Lan Manager style challenge response. It is best to leave this on unless the remote host is a Lan Manager or an old Windows machine.

REQAUTH

This specifies which authentication will be requested by the smxNS host. For CHAP/MS-CHAP, AUTH_ALG must also be set (see below). For PAP, it is what it is.

AUTH_ALG

For MS-CHAP, this value must be set to CHAPalg_MD4; for normal CHAP, the value must be set to CHAPalg_MD5.

TOUTMS

This is the elapsed time in milliseconds (TimeMS()) before time out. Our default is 2.5 seconds (2500) though RFC 1661 sets the default at three seconds. It has been noted that race conditions occur more frequently with smaller values, though every link is different. Links that come up slowly may need a smaller timeout period. Links that do not come up at all may require a longer timeout period.

TOUT_GROW

This specifies whether or not the restart timer should start small and grow to the maximum timeout value (TOUTMS) as link quality is assessed to be poor. It is off by default. When on, this may cause more retransmissions than necessary at the start of negotiations.

MAXCONF

This is the value in the restart counter for both LCP and IPCP. It should default to ten. The configuration packet will be resent this many times without response before the link is set to close.

MAXTERM

This is the value in the restart counter for LCP when closing. It should default to three. The terminate request packet will be resent this many times without acknowledgement before the link is forced closed.

COMPRESSION

This can be set to request and support protocol field and address/control field compression and/or VJ TCP/IP header compression. It is generally best to leave this at 3 to support both types as this will increase your throughput slightly. If code size is favored, it is best to leave this at either 1 (for

address/control/protocol field compression) or 0 (for no compression). VJ compression requires a great deal of code, but the others do not.

MAXSLOTS
Maximum slots for TCP/IP (VJ) header compression. See RFC 1144 for more information or leave them at their default values. They basically correspond to the number of TCP connections coexisting on the link.

PPP_MRU
Specifies whether or not the host will negotiate the MRU (Maximum Receive Unit) for smxNS. This value is equivalent to (MAXBUF - MESSH_SZ - LHDRSZ) in smxNS. Unless you are planning on reducing buffer size, this is not necessary.

MAGICNUM
Specifies whether or not the host will use Magic number with LCP. Unless you really want to save on the amount of data sent, leave this on. It is standard for almost all PPP links.

ASYNC
RFC 1662 tells of HDLC framing and the character escaping mechanism. This option will request that the peer use the RACCM value (see below) as its character map when sending to us. This option is enabled by default.

RACCM
This is the Remote Asynchronous Control Character Mapping. This option is only negotiated if ASYNC is enabled (see above). It is a 32-bit field where each bit corresponds to a character < 0x20. If the bit is set, PPP HDLC encoding must escape the character. Therefore, a value of 0x00000000 increases throughput the most but decreases reliability. A value of 0xffffffff escapes all characters and decreases throughput. The default value is 0x00000000.

IPCP_DNS
RFC 1877 includes extensions for PPP that allow configuration of DNS addresses during IPCP. This is not recommended except for dedicated devices with minimal application functionality and is disabled by default.

There are two parts to this option. The active configuration and the passive configuration. When the active portion is enabled (by setting bit 0 to 1), the host will send a configure-request with the current DNSiid. Typically, this will not have been configured by the application and will be 0.0.0.0 for both primary and secondary addresses. When the passive portion is enabled, the host will do nothing unless a configure-request is received for either the primary or secondary DNS in which case the host will reply with a Nak of the address if it does not match the host's DNSiid as configured by the application through SetDNS().

MP
The Multilink Protocol (RFC 1990) is enabled by a new Maximum Reconstructed Receive Unit. The endpoint discriminator is negotiated along with the MRRU. This option is not tested and is not considered a supported feature.

MPBUF
The number of buffers MP packets can occupy. Reasonably, no host should fragment any packet into more pieces than the number of physical connections.

AUTH_ACK_REPLY
When the peer passes authentication, this string is sent. It does not matter what it is, though the peer application may see it.

AUTH_NAK_REPLY
When the peer fails authentication, this string is sent. In MS-CHAP, a result code and retry flag is sent instead. If SNS_DEBUG_LEVEL >= 5 and PPP_DEBUG is on, the Message field of this packet will appear as strange characters because of the MS-CHAP result code.

QUALITY
This allows the peer to use link quality report monitoring. Very few implementations support this so you will want to leave this off. If you do wish to use this, contact Micro Digital PPP support.

Scripting

In order for PPP to function over a modem, there are three non-error cases that must be handled:

1. If PPP needs to actively establish a link to a remote host over a modem line, the modem needs to dial out to the remote host prior to the initiation of PPP.

2. If PPP is waiting for a remote host to establish a link over a phone line, it must configure the modem to wait for such an event and perform some actions when the event occurs.

3. If either the host or the peer terminates PPP, the modem should be configured to a default state to wait for further action.

In order to facilitate this operation, there are several scripts used by smxNS dial-on-demand:

1. pppsrc\dial-out.scr — This is written to configure the modem to dial a phone number to a remote host for active links. Once it is completed successfully, PPP is initiated. It checks the condition of the modem, changes it to off-line mode if necessary and dials the phone number. If the attempt to connect fails because of a modem error condition (e.g. no dialtone), the script will try a few more times. If the modem is not responding, the script will attempt to bring the modem to off-line mode (see dial-dwn.scr below). The phone number is defined as a global variable that can be changed in your application. There is more information on variables in scripts in the "Commands" section following number 3 below.

2. pppsrc\dial-dwn.scr — This is written to configure the modem to be in terminal mode after being in on-line mode. It hangs up the line. In order to force a modem to off-line mode, most require that the string "+++" be sent surrounded by a guard time of 1 or 2 seconds of silence. The default script will successively increase the guard time from 1 to 5 seconds if the modem is not responding. At that point the script will fail. If the modem returns to terminal mode, the script succeeds.

3. pppsrc\dial-in.scr — This is written to configure the modem to wait for an incoming call and answer it when it comes. PPP will wait passively once the script finishes with success. With the current setup, this script is executed while PPP is down for all links. If you will only be dialing out, an empty file (null script) can be used in its place in order to save a little CPU time. The current script should never finish unless an incoming call is received.

There are three additional scripts for logging into Windows based machines when using a physical null modem instead of a conventional modem.

1. pppsrc\ms-out.scr — This is written to send the string CLIENT to the passive machine. The string, CLIENTSERVER, is expected in response after which the script is successfully completed and PPP data can flow.

2. pppsrc\ms-dwn.scr — This is written to disconnect from a directly connected Windows machine. It sends the string, None, and terminates.

3. pppsrc\ms-in.scr — This script waits forever for the string, CLIENT. If received, the string CLIENTSERVER is sent followed by some carriage returns and the script completes successfully so that PPP data can flow.

Commands used by scripts:

FILE name
Name the file being used ('name' in this case).

```
GOTO 3
   # Go to tag 3 unconditionally.

:3
   # This marks the position for tag 3

CHECK 1 3
   # If the internal status is good
   #    go to tag 1,
   # Else
   #    go to tag 3.

INIT
   # Make the internal status good.

%retries 5
   # Set the initial value of %retries to 5.  Future
   #   references to %retries will become an integer value
   #   at run time.  The name is arbitrary (like any
   #   variable name).  Any characters can be in a variable
   #   name except for '\r', '\n' or ' '.
   #      %bad\rvariablename
   #      %bad variable name
   #      %good_variable_name
   #      %bad_variable_name_this_is_just_too_long
   #
   #   The default variable name length is 30 characters.

-- %retries
   # Decrement the value of %retries by one.  Include a
   #   space between the "--" and the '%'.

++ %retries
   # Increment the value of %retries by one.  Include a
   #   space between the "--" and the '%'.

 (%retries < 0) 1 2
   # If the value of %retries is less than 0,
   #   goto 1,
   # Else
   #   goto 2.
   #
   # >, <, >=, <=, == are supported operators.
   #
   # Either of the two comparison values can be an integer
   #   variable or integer value.  Any variables must be
   #   locally defined in a script.

$USERID test
   # Declare USERID as a pointer to "test".  Future
   #   references to $USERID will become "test".  Note that
   #   USERID is a global that can be referenced by your C
   #   application.  It is a char *.  Don't be afraid to
   #   point it to a new location.
   #
   #   #include "..\pppsrc\script.h"
   #   char *new_string = "new_string";
   #   void func(void)
```

```
    #   {
    #       USERID = new_string;
    #   }

SEND ATD 1 $PHONENUM \r
    # Send 8-bit data string to modem.  This particular
    #   sequence will dial the phone number prefixed with
    #   a 1.  The "\r" is necessary for most modems to
    #   signal the end of a command sequence.  NOTE: a ' '
    #   (space character) is required between a variable
    #   name and any other form of data.  This makes
    #   variables distinct while parsing the script.

EXPECT 5 OK
    # If "OK" is not received in 5 seconds, make the internal
    #   status bad and go on.
    # If "OK" is received in 5 seconds or less, just go on.

PAUSE 5
    # Yield control for five seconds.

DEBUG 3 Hello, World!
    # If SNS_DEBUG_LEVEL >= 3,
    #   print "Hello, World!" to stdout.
    # Else,
    #   preprocess this command out.

>logArray 20
    # Create an array of 20 bytes to log with

LOG >logArray
    # Start putting incoming data into array

NOLOG
    # Stop saving incoming data
```

These scripts may need modification for any particular environment. The rules for doing so are mentioned below. While going though them, it is recommended that a dial script be at hand to make references tangible. It is important to remember that the script is parsed into a series of arrays at build time. During execution of the application the arrays are stepped through to discover their outcomes.

Read time rules

- Each command must be on its own line.

- Anything following '#', the comment character, on a single line will be ignored.

- A file line will be truncated after 80 characters.

- prefrmt always expects the scripts to be in the order dial in, dial out and dial down.

Run time rules

- The status flag is internal. The status becomes good when INIT is performed. The status becomes bad when an EXPECT or SEND times out (i.e. the expected data is not received or sent data never sent). Until the status flag is reset, SEND and EXPECT operations will be skipped.

- Except for GOTO, CHECK and the if-else operation, all of which move the script index to the appropriate TAG value, the commands are performed in the order of the script.

Variable usage

- Variables must be declared before they are used. String and log variables have global scope, however, so they can be declared in a prior module and used in a later one:

    dial-in declares ussDialVar

    dial-out uses ussDialVar without declaring it

- Do not define a single variable multiple times. Remember, string and log variables have global scope.

- Variables must be surrounded by space characters so that they can be differentiated from other strings. Upon transmission or reception, the spaces will be disregarded.

- For SEND or EXPECT, data is concatenated at run time as if there were no space or tab characters. Carriage returns can be placed in a string by using "\r" or "^M". These can be used without spaces between them and other strings; however, there must be a space between them and variable references.

Suggestions

- We recommend that SEND and EXPECT are always performed in pairs. If a DEBUG statement or other time consuming command is performed in between, chances are that the EXPECT will not be installed early enough to receive the data provided from the modem as a response to the SEND information.

- Loops that may require a significant amount of time should have a SEND, EXPECT, PAUSE or DEBUG statement inside of them. All other commands are concatenated in execution and will block. For example, the following should never be done because it will starve the rest of smxNS.

```
INIT
:1
%temp 30000
(%temp >= 0) 1 2
-- %temp
:2
```

  Adding the following just after tag 1 will remedy the situation.

```
PAUSE 0
```

Notes on Special Cases

Dial On Demand

Although it is good to be able to send/receive data regardless of the state of the link, our present implementation has no direct access to the driver. This means that certain important functions (like checking the wire status) must be performed with timeouts and assumptions.

A second concern with dial-on-demand is that the link often requires a great deal of time to come up (configuring the modem, dialing out, bringing LCP, authentication and NCP up). If the upper layers require timely feed back from the remote host, replies may not arrive quickly enough. For purposes such as forwarding segments periodically over an open connection, it may serve a valuable purpose. There is a method to force the link to be either up or down. See the appendix for details.

One last area of concern is the IP address negotiations used by PPP. If the dial-up server requires that the host change its IP address, the connection may become invalid. Therefore, make sure that the original IP address remains through each demand dial session. One method of gaining this information is through the PPP ioctl() function and the PPP signal functions. Modify the following macro in pppsig.h:

```
extern Iid previous_ip;
#define PPPSIG_IPCP_UP(netno) \
        do { \
           Iid ip; \
           ussPPPTable.ioctl(&nets[netno], \
              ussPPPHostAddressGetE, &id, 4); \
           if (id.l != previous_ip.l) { \
              PPPSIG_PRINT("Warning: IP address changed!\n", netno); \
              abort_application_connections(); \
           } \
        } while (0)
```

The PPPSIG_IPCP_UP() macro is chosen because it always procedes the IPCP layer where IP addresses are negotiated. The do { ... } while(0) phrase is used to encapsulate the command as a single expression.

Null Modem Links to Window Machines

Three special scripts were created to support this operation more efficiently. They are ms-in.scr, ms-out.scr, and ms-down.scr in the pppsrc directory.

Special considerations:

Windows 95 – This operating system would not behave as a dial-up server so smxNS was always the passive host when connecting to Win95 machines.

Windows 98 – We were not able to install a null modem into the operating system and were forced to seek a third party solution. There was no problem using this operating system with conventional telephone modems. Contact Micro Digital PPP support for information about the driver used if the null modem cannot be configured.

Windows NT – A special null modem cable or adapter was required to allow a physical connection between the hosts. The Microsoft web site has ample information about this issue. Otherwise, everything worked as expected.

Windows 2000 – A null modem cable is required. Steps for setting up the connection:

1. Select Start, Settings, Network and Dial-up Connections.
2. Double click on "Make New Connection".
3. Both a "Network Connection Wizard" and a "Location Information" dialog box will appear. In the "Location Information" dialog box, fill in a dummy value in the "What area code (or city code) are you in now?" box, and select "OK".
4. From the "Phone And Modem Options" dialog box, select "OK".
5. Now the "Network Connection Wizard" dialog box remains on the screen. Select "Next>".
6. Select "Connect directly to another computer".
7. Select "Guest".

8. Select the serial port for the connection.
9. Select "For all users".
10. Enter a name for the connection, for example "PPP".
11. Select "Finish".
12. Windows displays a connection window. Select "Properties".
13. Under the "General" tab, make sure the communication port that you selected earlier is still selected.
14. Select "Configure..." from the "General" tab.
15. Deselect all checkboxes.
16. Set speed to 115200 bps (this is the smxNS default, adjust as needed).
17. Select "OK" to close the modem configuration window.
18. Select the "Options" tab in the PPP properties window.
19. Deselect all except "Display progress".
20. Select the "Networking" tab.
21. Select "PPP: Windows 95/98/NT4/2000, Internet".
22. Under "This connection uses the following items:", select only "Internet Protocol (TCP/IP)".
23. Again under "This connection uses the following items:", highlight "Internet Procotol (TCP/IP)" and select "Properties".
24. Select "Use the following IP address:", and enter "192.168.2.1". Any unused IP address should work, but one that is "next" to the address assigned to the attached PPP port is easier to keep track of.
25. Select "Advanced". Deselect all "Advanced TCP/IP Settings".
26. Select the "WINS" tab. Deselect "Enable LMHOSTS lookup" if it is selected.
27. Select OK. You will be asked "This connection has an empty primary WINS address. Do you want to continue?". Select "Yes".
28. Select OK two more times to close the configuration dialog boxes.

Windows XP – A null modem cable is required. Here are the steps for setting up the connection.

1. Select Start, then right click on My Network Places and select Properties.
2. Select "Create a new connection". Select "Next".
3. Select "Set up an advanced connection".
4. Select "Connect directly to another computer".
5. Select "Guest".
6. Enter a name for the connection, for example "PPP".
7. Select the serial port for the connection.
8. Select "Finish".

9. Windows displays a connection window. Select "Properties".
10. Select "Configure..." from the "General" tab.
11. Deselect all checkboxes.
12. Set speed to 115200 bps (this is the smxNS default, adjust as needed).
13. Select "OK" to close the modem configuration window.
14. Select the "Options" tab in the PPP properties window.
15. Deselect all except "Display progress".
16. Select the "Networking" tab.
17. Select "PPP: Windows 95/98/NT4/2000, Internet".
18. Under "This connection uses the following items:", select only "Network Monitor Driver", "Internet Protocol (TCP/IP)" and "QoS Packet Scheduler".
19. Again under "This connection uses the following items:", highlight "Internet Procotol (TCP/IP)" and select "Properties".
20. Select "Use the following IP address:", and enter "192.168.2.1". Any unused IP address should work, but one that is "next" to the address assigned to the attached PPP port is easier to keep track of.
21. Select "Advanced". Deselect all "Advanced TCP/IP Settings".
22. Select the "WINS" tab. Deselect "Enable LMHOSTS lookup" if it is selected. Select "Disable NetBIOS over TCP/IP".
23. Select OK until all the configuration dialog boxes are closed.

To enable traffic to flow over this connection, first start the system running smxNS, then bring up the Network Connections dialog box by selecting Start | My Network Places | View Network Connections. Now double click on the name of connection that you configured, i.e. “PPP”, and you should see the connection status change from “Disconnected” to “Connected”.

Once the connection has been established, the directly linked XP host computer will be able to contact the smxNS system at its IP address. In order for other systems to establish connections with the smxNS system, they must be updated with routing information that indicates that the smxNS system is accessible via the linked XP host. In addition, the XP host must have IP forwarding turned on in order for IP datagrams to be forwarded between its LAN connection and the SLIP connection.

In order to turn on IP forwarding on a Windows XP computer:

1. Start Registry Editor (regedit.exe).
2. Open the following registry key:
3. HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters
4. If the value of IPEnableRouter is not 1, change it to 1.
5. Reboot the computer.

The Windows route command can be used to install a static route that allows other Windows computers on the LAN to reach a system that is connected via SLIP or PPP. This is best illustrated by an example. An smxNS system is configured to use PPP over an interface with an IP address of 192.168.2.2. The computer that is directly connected to the smxNS computer has an Ethernet interface with the address 192.168.1.100, and the other computers on the LAN are on the 192.168.1.X

network. The following route command can be used to update a Windows XP routing table so that it can access the smxNS system.

C:>route add 192.168.2.0 mask 255.255.255.0 192.168.1.100

In the same scenario, if the SLIP connected smxNS system needs to communicate with systems on the Internet, the default router should be configured with a static route to the smxNS system. If the default router is consumer device, this configuration may be possible through an advanced LAN configuration web page. The web GUI should take similar parameters.

To summarize, when a direct serial link connects an smxNS system to a Windows computer:

1. The smxNS system should be able to communicate with the directly connected Windows computer without any special configuration changes.
2. The smxNS system should be able to communicate with other systems on the LAN once IP forwarding is turned on in the directly connected Windows computer, and once the peer computer on the LAN has been updated with information on the route to the smxNS system.
3. The smxNS system should be able to communicate with the Internet once IP forwarding is turned on in the directly connected Windows computer, and once the default gateway has been updated with information on the route to the smxNS system.

MS-CHAP Authentication

This form of authentication is commonly performed when establishing a link with a Microsoft system. Note that the domain must be prepended to the user name. At the time this was implemented, one NT server had the domain USSOFTWARE so a person's user ID was "USSOFTWARE\\name". Authentication will fail without this. Of course, if smxNS is the authenticator rather than the authenticatee, the user ID is whatever smxNS says it is. Also, NT supports passwords of up to 256 Unicode characters, but we recommend that you not make use of this feature and use conventional passwords.

Routing and IP Addresses with PPP Interfaces

IPCP has the ability to change an interface's IP address. Many client-oriented interfaces will not know either their own IP address, the peer's IP address or both. The host address is changed if it is 0.0.0.0 and the remote host can provide a new one. In order to bring about the features mentioned, there are two cases to consider. Each case can be configured using the Portconfig() function.

CASE 1:

If smxNS is a server, an IP address to be assigned to the remote host can be specified by calling Portconfig() with the “PEER” attribute. When the peer suggests that its address be 0.0.0.0, smxNS will respond with a hint to use the one statically defined and the peer should take the hint.

CASE 2:

If smxNS is running as a client with a PPP interface, it is recommended that (unless the host IP address is known to not change) the host address be defined as 0.0.0.0. The subnet mask must be set to a value corresponding to what will be true of the future IPs and the rest of the network in general. This means that some information must be known about the remote network. For those situations where there is only one interface on your smxNS host, subnet masks do not matter because the remote host will always be the default router. If you wish to communicate with the peer, you can retrieve the IP address using the PPP ioctl() function as described below.

Another way to gain access to the IP addresses is through the PPP ioctl() function. The following options are available:

ussPPPHostAddressGetE — Get the host address
ussPPPHostAddressSetE — Set the host address
ussPPPRemoteAddressGetE — Get the remote address
ussPPPRemoteAddressSetE — Set the remote address

Note that the 'Get' ioctl() operations can only yield useful information after IP address negotiations (i.e. when PPP is open). The 'Set' ioclt() operations can only be used prior to link negotiations (i.e. when PPP is closed).

Renegotiation of IP Address

PPP always negotiates the host and peer IP addresses. In the case where the host address is unknown, PPP will request a 0.0.0.0 address and the peer will reply with a valid one to use instead. Once PPP is established, the new IP address is assigned to the host. If PPP goes down and then renegotiates a new link, the last negotiated host address will be requested. Usually the peer will assign a new address if the requested one is not valid, but it may be the case that the peer cannot handle this address and will either abort the link or let network layer operations fail by administering incorrect IP addresses. For cases such as this, the smxNS PPP must be configured to reset the host address to 0.0.0.0 between PPP negotiation sequences. The best way to do this is to assign the host address once LCP completes. The macro, PPPSIG_LCP_UP(netno), in pppsig.h can be defined to a value to modify the host IP address. The host address is stored in the nets[netno].haddr field. Here is an example macro definition:

```
#define PPPSIG_LCP_UP(netno) \
        do { \
           Iid id; \
           ussPPPTable.ioctl(&nets[netno], \
              ussPPPHostAddressSetE, &id, 4); \
        } while (0)
```

The PPPSIG_LCP_UP() macro is chosen because it always precedes an IPCP request.

The do { ... } while(0) phrase is used to encapsulate the command as a single expression.

Handling Loss of Carrier

Sometimes modems go off-line or a line breaks without warning. This situation can be dealt with in a variety of ways. The following cases examine a few examples and their relative advantages and disadvantages.

CASE 1 Application level timeout

Protocols such as TCP or other application level protocols often have timeout periods for expected data. If the PPP link becomes disestablished, there is a good chance that the upper-level protocols will be able to detect this error and restart the application when they notice that their data is not arriving in a timely manner.

Advantages

requires no initial engineering effort

requires no extra code

Disadvantages

requires a lot of time (depending on the application) to notice that data is not arriving.

CASE 2 PPP echo-request detect loss of link

PPP has a feature similar to the ping operation of IP level systems called the LCP echo-request packet. smxNS PPP can be configured to transmit this echo-request packet periodically. If a certain number of packets are sent in a row without any reply, the link is automatically terminated.

As an example, define the following in pppconf.h:

```
#define ECHO_TOUTMS 2500
#define ECHO_RETRIES 3
```

Given the above definitions and a broken link, PPP will recognize the failed link in a maximum of ECHO_TOUTMS * (ECHO_RETRIES + 1) milliseconds.

The application will probably need to be informed of the condition. The PPP_LINK_DOWN(netno) macro from pppsig.h can be used for this purpose.

Advantages

requires minimal engineering effort

relatively quick response time

Disadvantages

requires a timeout period with retransmission to detect a physical line break

does not detect looped-back condition

application must be signalled of link failure

CASE 3 PPP echo-request detect modem looped-back

If a Hayes modem is being used, the modem may enter echo mode automatically when it goes off-line. This feature enables PPP to depend on the magic number within the echo-request packet to detect a looped-back link. To use this feature, MAGICNUM, CHECK_LOOPED_BACK and ECHO_TOUTMS must all be enabled in pppconf.h. Once enabled, PPP will transmit echo-request packets periodically. It will notice that the packets contain the same magic number and will ignore them which will eventually force PPP to conclude that the peer is no longer available.

As an example, define the following in pppconf.h:

```
#define MAGICNUM 1
#define CHECK_LOOP_BACK 1
#define ECHO_TOUTMS 2500
#define ECHO_RETRIES 3
```

The above options will for PPP to send an echo-request packet every 2.5 seconds. If the modem goes off-line it will start echoing back all data. PPP will ignore the echo-request/reply packets and will force a link termination after 3 retransmissions of the echo-request packet. This will take a maximum of ECHO_TOUTMS * (ECHO_RETRIES + 1) milliseconds.

The application will probably need to be informed of the condition. The PPP_LINK_DOWN(netno) macro from pppsig.h can be used for this purpose.

Advantages

requires minimal engineering effort

relatively quick response time

discovers looped-back condition

Disadvantages

requires a timeout period with retransmission to detect a physical line break

application must be signalled of link failure

CASE 4 Application detect and handle link break

If the driver or application software can detect a change in line status corresponding to either modem loss of carrier or physical line break, the PPP link can be instantly forced shut.

Write the following code in a new module:

```
#include "smxns.h"
#include "ppp.h"
#include "dialapi.h"

static char ppp_kill_flag[NNETS];

void LOST_CARRIER_OR_LINE_BREAK_FOR_PPP(int netno)
{
        ppp_kill_flag[netno] = 1;
}

void nettask_ifkillflag(int netno) {
    if (ppp_kill_flag[netno]) {
        smx_TaskLock();
        nets[netno].state = PPPclsd;
#if DIALD
        MODEM_DIALIN(netno);
#endif
        pppDQ(netno); // Make non-static inside ppp.c!!!
        ppp_kill_flag[netno] = 0;
        smx_TaskUnlock();
    }
}
```

In net.c, insert the following:

```
void nettask_ifkillflag(int netno); // New prototype
void nettask(...)
{
        ...
        if (netp->protoc[0] == PPP) {
        pppTimeout(netno)
        nettask_ifkillflag(netno);  // New Call
   }
        ...
}
```

In the event that the application detects loss of carrier or a line break in an ISR, the application can call the LOST_CARRIER_OR_LINE_BREAK_FOR_PPP(netno) function. When nettask() runs next (within a second), the PPP link will reset itself to start again. The connection level application will probably need to be informed of the condition. This should be done by the ISR that initiates the transaction.

Advantages

requires less than a second from detection of loss of carrier to resolution of PPP link.

does not need to detect looped-back condition

Disadvantages

invasive into the stack

application must be signalled of link failure

PPP ioctl Routines

Description

The PPP ioctl() function allows an application to dynamically configure or manage various parts of the PPP protocol layer. The PPP ioctl() operations pertain to specific interfaces on the host and therefore each requires a handle to the network interface structure, nets.

Here is the function prototype.

```
int ussPPPTable.ioctl(
   void *netp,
   enum ioctlreq req,
   void *arg,
   size_t size);
```

If no error occurs, 0 is returned. Otherwise, applicable errors may be returned. At this time, only the ussErrInval return code is provided for cases when the ioctlreq option is invalid.

The netp parameter must be a smxNS (struct NET *) data type.

The arg parameter data type varies from on ioctl option to another.

Option Listing

These options are defined in net.h as part of the ioctlreq enumeration.

i) ussLinkIsUpE

Upon return, *(int *)arg will be true if the link is up and false if the link is down.

ii) ussLinkIsDownE

Upon return, *(int *)arg will be true if the link is down and false if the link is down. If the dialer is enabled, this means that the dialer is in the passive state waiting for an incoming call.

iii) ussLinkBringUpE

Attempt to force the link up for the time in seconds specified by the size parameter. If the link becomes established prior to the passage of the full amount of time, the function will return early. The value of the arg parameter is ignored in this option.

iv) ussLinkBringDownE

Attempt to force the link down for the time in seconds specified by the size parameter. If the link becomes fully disestablished prior to the passage of the full amount of time, the function will return early. The value of the arg paramter is ignored in this option.

v) ussPPPUserIdSetE

Set the host userid to be negotiated by authentication protocols (PAP, CHAP, MS-CHAP).

The userid is passed through the (char *)arg parameter.

Note that the function stores the pointer, not the actual data (NO strcpy()!). The userid must remain allocated for the entire PPP session.

The userid must be a null terminated string.

vi) ussPPPUserIdGetE

Get the host userid to be negotiated by authentication protocols (PAP, CHAP, MS-CHAP).

The userid is copied into the (char arg[15]) parameter.

Note that the function copies the actual data into the arg parameter so at least 15 bytes must be allocated to accomodate the potential maximum userid size.

vii) ussPPPPasswordSetE

Set the host password to be negotiated by authentication protocols (PAP, CHAP, MS-CHAP).

The password is passed through the (char *)arg parameter.

Note that the function stores the pointer, not the actual data (NO strcpy()!). The password must remain allocated for the entire PPP session.

The password must be a null terminated string.

viii) ussPPPPasswordGetE

Get the host password to be negotiated by authentication protocols (PAP, CHAP, MS-CHAP).

The password is copied into the (char arg[15]) parameter.

Note that the function copies the actual data into the arg parameter so at least 15 bytes must be allocated to accomodate the potential maximum userid size.

ix) ussPPPDialEnableE

Enable the dialer on the interface.

This is only valid if DIALD is defined in pppconf.h.

x) ussPPPDialDisableE

Disable the dialer on the interface.

This is only valid if DIALD is defined in pppconf.h.

xi) ussPPPHostAddressGetE

Get the Host IP address as negotiated by IPCP.

Before IPCP is completed, this value is the IP address that smxNS PPP will attempt to negotiate on the interface. After IPCP has completed, this value is the actual IP address smxNS is communicating as from the PPP interface. The arg data type is * Iid.

xii) ussPPPHostAddressSetE

Set the Host IP address to negotiate during IPCP.

This is only valid if it is performed prior to link establishment. Also, it does not guarantee that the IP address specified will be the one chosen because PPP must negotiate addresses. If the peer recommends that a different address be used, then smxNS will use that one instead. The arg data type is * Iid.

xiii) ussPPPRemoteAddressGetE

Get the Peer IP address as negotiated by IPCP.

Before IPCP is completed, this value is the IP address of the peer host that smxNS PPP was last connected to. After IPCP has completed, this value is the actual IP address of the peer host to which smxNS is directly connected to. The arg data type is * Iid.

xiv) ussPPPRemoteAddressSetE

Set the Peer IP address to negotiate during IPCP.

This is only valid if it is performed prior to link establishment. Also, it does not guarantee that the IP address specified will be the one chosen because PPP must negotiate addresses. If the peer recommends that a different address be used, then smxNS will use that one instead. The arg data type is * Iid.

Using PPP ioctl() routines

In all examples below, it is assumed that Ninit() and Portinit() have been called prior to the execution of any ioctl() procedure.

i) Forcing the link up

If your application requires the link layer to be up at a particular point in time, use the following:

```
#include "smxns.h"

void func(int netno)
{
    struct NET *netp;
    int i1;

    netp = &nets[netno];
    /*
    ** The last parameter to ioctl when using
    ** ussLinkBringUpE represents the time in seconds
    ** to block while waiting for the condition to
```

```
   ** occur.  The function will return early if the
   ** link conclusively fails or succeeds.
   */
   ussPPPTable.ioctl(netp, ussLinkBringUpE, 0, 30);
   ussPPPTable.ioctl(netp, ussLinkIsUpE, &i1, 0);
   if (!i1)
      DEBUG_MSG2_PAR0("Error: could not force PPP up!\n");
   else
      DEBUG_MSG3_PAR0("PPP is up\n");
}
```

ii) Forcing the link down

If your application requires the link layer to be down at a particular point in time, use the following:

```
#include "smxns.h"

void func(struct CONNECT *conp)
{
   struct NET *netp;
   int i1;

   netp = &nets[conp->netno];
   /*
   ** The last parameter to ioctl() when using
   ** ussLinkBringDownE represents the time in
   ** seconds to block while waiting for the
   ** condition to occur.  The function will return
   ** early if the link is closed.
   */
   ussPPPTable.ioctl(netp, ussLinkBringDownE, 0, 10);
   ussPPPTable.ioctl(netp, ussLinkIsDownE, &i1, 0);
   if (!i1)
      DEBUG_MSG2_PAR0("Error: couldn't force PPP down!\n");
   else
      DEBUG_MSG3_PAR0("PPP is down\n");
}
```

iii) Capturing the link status

If the status of the link must be known, use the following:

```
#include "smxns.h"

void func(int netno)
{
   struct NET *netp;
   int i1;

   netp = &nets[netno];
   ussPPPTable.ioctl(netp, ussLinkIsUpE, &i1, 0);
   if (i1)
      DEBUG_MSG3_PAR0("PPP is up\n");
   else {
      ussPPPTable.ioctl(netp, ussLinkIsDownE, &i1, 0);
      if (i1)
```

```
            DEBUG_MSG3_PAR0("PPP is down\n");
         else
            DEBUG_MSG3_PAR0("PPP is negotiating\n");
      }
   }
```

iv) Configuring the username and the password

If the host user ID or password must be set prior to link negotiations, use the following:

```
#include "smxns.h"

char *uid = "new userid", *pw = "new password";

void func(int netno)
{
   struct NET *netp;
   char tuid[15], tpw[15];

   netp = &nets[netno];
   ussPPPTable.ioctl(netp, ussPPPUserIdGetE, tuid, 0);
   ussPPPTable.ioctl(netp, ussPPPPasswordGetE, tpw, 0);

   /* Change userid if unmatched */
   if (!strcmp(tuid, uid))
      ussPPPTable.ioctl(netp, ussPPPUserIdSetE, uid, 0);

   /* Change password if unmatched */
   if (!strcmp(tpw, pw))
      ussPPPTable.ioctl(netp, ussPPPPasswordSetE, pw, 0);
}
```

v) Switching between modem and null modem links

If the host wants to connect through a null modem and a conventional modem with only one interface configured using Portinit(), use the following:

```
#include "nscfg.h"
#include "ppp.h"

extern int host_using_modem;

void func(int netno)
{
   struct NET *netp;

   netp = &nets[netno];
#if DIALD
   if (host_using_modem)
      ussPPPTable.ioctl(netp, ussPPPDialEnableE, 0, 0);
   else
      ussPPPTable.ioctl(netp, ussPPPDialDisableE, 0, 0);
#else
   DEBUG_MSG3_PAR0("No modem configured\n");
#endif
}
```

vi) Configuring and capturing the host and peer IP addresses

```
#include "nscfg.h"

static void getIP(Iid *id)
{
   int i1;
   Nprintf("Enter IP as four single typelings\n    > ");
   for (i1 = 0; i1 < 4; i1++) {
      id->c[i1] = Ngetchr();
      Nprintf(" %02x (%c)",
         id->c[i1], id->c[i1] > 0x19 && id->c[i1] < 0x7f ? id->c[i1] : '.');
   }
   Nprintf("\n");
}

void func(int netno)
{
   int i1;
   Iid hid, rid;
   /* Loop waiting for user request */
   for ( ; ; ) {
      SNS_YIELD();            /* Yield to smxNS and PPP */
      /* Check for user input */
      if (Nchkchr()) {
         /* Get user input */
         i1 = Ngetchr();
         if (i1 == 0x1b) {
            break;  /* Exit function */
         }
         else {
            /*
            ** Switch on the user request.
            **  1 -- Get host address
            **  2 -- Set host address
            **  3 -- Get peer address
            **  4 -- Set peer address
            **  ? -- Print addresses
            */
            switch (i1) {
            case '1':
               ussPPPTable.ioctl(&nets[netno],
                     ussPPPHostAddressGetE, &hid, 4);
            case '2':
               if (i1 == '2')
                  getIP(&hid);
               ussPPPTable.ioctl(&nets[netno],
                     ussPPPHostAddressSetE, &hid, 4);
            case '3':
               ussPPPTable.ioctl(&nets[netno],
                     ussPPPRemoteAddressGetE, &rid, 4);
            case '4':
               if (i1 == '4')
                  getIP(&rid);
```

```
                ussPPPTable.ioctl(&nets[netno],
                    ussPPPRemoteAddressSetE, &rid, 4);
                break;
              default:
                /* Force the link up (non-blocking) */
                ussPPPTable.ioctl(&nets[netno],
                    ussLinkBringUpE, 0, 0);
                continue;
              }
              Nprintf(" H %u.%u.%u.%u R %u.%u.%u.%u\n",
                  hid.c[0], hid.c[1], hid.c[2], hid.c[3], rid.c[0], rid.c[1], rid.c[2], rid.c[3]);
            }
          }
        }
        Nprintf("User terminated\n");
    }
```

PPP dialapi Routines

Description

The dial API routines are defined in include\dialapi.h. For the most part they are used within the PPP core module. In some cases, it may be beneficial to use them from the application to improve configurability. In other cases, it may be useful to modify the dial API to map onto an already defined dialing layer for smoother integration with smxNS PPP.

Definitions of API

i) MODEM_PROCESS(netno)
Execute modem operations in a non-blocking fashion for a particular interface.

ii) IS_MODEM_DONE(netp)
Boolean condition. True when the currently executing modem process associated with an interface structure is completed.

iii) IS_MODEM_NONE(netp)
Boolean condition. True when the currently executing modem process associated with an interface structure is ready for another operation.

iv) MODEM_DIALIN(netno)
Install or start the dial-in procedure for a particular interface.

v) IS_MODEM_DIALIN(netp)
Boolean condition. True when the currently executing modem process associated with an interface structure is the dial-in process.

vi) MODEM_SET_DIALIN(name)
This function is specific to the smxNS PPP implementation. It is used to define the dial-in script.

vii) MODEM_DIALOUT(netno)
Install or start the dial-out procedure for a particular interface.

viii) IS_MODEM_DIALOUT(netp)
Boolean condition. True when the currently executing modem process associated with an interface structure is the dial-out process.

ix) MODEM_SET_DIALOUT(name)
This function is specific to the smxNS PPP implementation. It is used to define the dial-out script.

x) MODEM_DIALDOWN(netno)
Install or start the dial-down procedure for a particular interface.

xi) IS_MODEM_DIALDOWN(netp)
Boolean condition. True when the currently executing modem process associated with an interface structure is the dial-down process.

xii) MODEM_SET_DIALDOWN(name)
This function is specific to the smxNS PPP implementation. It is used to define the dial-out script.

Dynamically Configuring smxNS PPP Dial Scripts

In order to change a dial script without recompiling, a few function calls can be made at run-time. At compile-time, the pppsrc\dial.mak file must be configured with the appropriate dial scripts.

It is not necessary, but is recommended, that PPP be not negotiating at the time of the change. This means that the state ought to be closed and the dialer ought to be inactive (dial-in).

Here is an example function that replaces the dial-in script with the ms-in.scr script already provided:

```
void install_ms_in(void)
{
   MODEM_SET_DIALIN(ms_in_scr);
   MODEM_DIALIN(netno);
}
```

Here is an example function that replaces both the dial-in and dial-out scripts using the Hayes modem type scripts:

```
void install_dial_in_and_dial_out(void)
{
   MODEM_SET_DIALIN(dial_in_scr);
   MODEM_SET_DIALOUT(dial_out_scr);
   MODEM_DIALIN(netno);
}
```

Note in the above case that MODEM_DIALIN() is called after installing the new script. This ensures that no ghost script is left executing.

PPP pppsig Routines

Description

The ppp signalling routines can be used to signal the application layer of events normally hidden within the PPP stack without the requirement of OS signal handlers. These events can be both informational and functional. They are defined in pppsig.h.

Definition of Signals Available

PPPSIG_PPP_UP(netno)
The entire PPP layer is up on the interface and is ready for network traffic.

PPPSIG_PPP_DOWN(netno)
The PPP layer is completely down on the interface and is ready for configuration or complete restart.

PPPSIG_IPCP_UP(netno)
The network layer is up (for IP) on the interface and is ready for traffic. This may also mean that IP addresses have been configured for perhaps the host and/or the peer.

PPPSIG_IPCP_DOWN(netno)
The network layer is down (for IP) on the interface and can no longer handle network traffic. This may also mean that IP addresses are no longer valid.

PPPSIG_HAUTH_UP(netno)
The host is authenticated to the peer on the interface.

PPPSIG_HAUTH_DOWN(netno)
The host failed authentication with the peer on the interface.

PPPSIG_PAUTH_UP(netno)
The peer passed authentication with the host on the interface.

PPPSIG_PAUTH_DOWN(netno)
The peer failed authentication with the host on the interface.

PPPSIG_LCP_UP(netno)
The link layer is established and ready for network level configuration and authentication on the interface.

PPPSIG_LCP_DOWN(netno)
The link layer is down completely and is ready for restart on the interface.

PPPSIG_DIALOUT_UP(netno)
The dial-out proceedure completed on the interface.

PPPSIG_DIALOUT_DOWN(netno)
The dial-out procedure failed on the interface.

PPPSIG_DIALIN_UP(netno)
The dial-in proceedure completed on the interface.

PPPSIG_DIALIN_DOWN(netno)
The dial-in procedure failed on the interface.

PPPSIG_DIALDOWN_UP(netno)
The dial-down proceedure completed on the interface.

PPPSIG_DIALOUT_DOWN(netno)
The dial-down procedure failed on the interface.

Note that this is not considered an error condition to the PPP layer so it will proceed to set up the dial-in procedure.

PPP_STATE_HOLD(netno) — From ppp.h
Stop PPP at the current state on the interface.

PPP_STATE_RELEASE(netno) — From ppp.h
Let PPP continue at the current state on the interfaces

Using PPP Signaling Routines

See the section on PPP ioctl Routines for a useful example of these macros.

It is important to remember that the signalling functions are meant to produce signals. The code that gets executed must not block and ought not to bloat the stack.

9. Simple Network Management Protocol (SNMP)

Introduction

This chapter describes the use of the Simple Network Management Protocol for smxNS®. smxNS SNMP provides support for integrating an SNMP agent into a real-time embedded system application. It is designed for use with SNMP Version 3 managers; however, it will also respond to Version 1 and 2 requests.

The reader ought to have a conceptual knowledge of SNMP in order to understand the terminology in this manual. There are several books available that explain more completely the terminology and function of SNMP systems, and some of these are listed in the Recommended Reading section of Chapter 1.

SNMP Overview

The Simple Network Management Protocol, SNMP, is used widely by industry to manage networks. On a network, a client in one host (a SNMP manager) communicates with a server in another host (an SNMP agent). The manager requests the agent to read or write information (objects) in a Management Information Base (MIB).

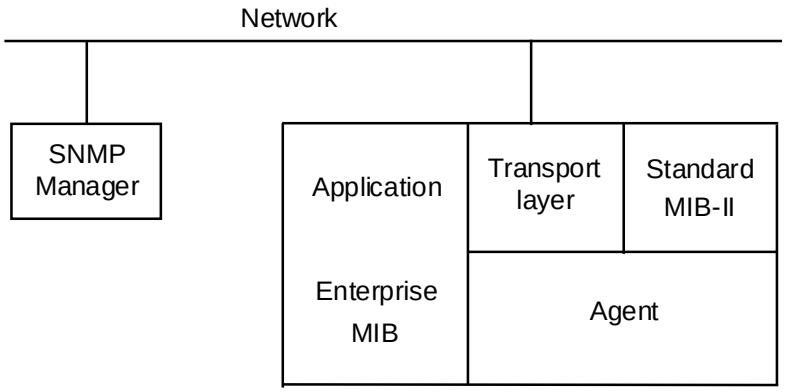


Figure 9-1: SNMP Agent on a network

Design of smxNS SNMP

The smxNS SNMP design includes these features:

ROMable

Compact

User-configurable

The agent is processor-independent. Almost any ANSI C compiler will do.

The agent is not tied to a particular transport layer. Any networking stack or other data communication layer can be used to transfer data to and from the agent. The code is ROMable in that all initialized data is type `const`, and there are no attempts to change code or constants at run-time.

The agent requires less than 30K code bytes and 12K RAM bytes on a typical compiler without optimization. If security is removed, the agent requires less than 20K code bytes and 4K RAM bytes. Actual code requirements also vary somewhat from processor to processor and compiler to compiler.

smxNS SNMP supports the same application interface and functionality across all processors. In other words, standard C code developed for one processor can be recompiled for another processor with minimal effort.

Custom MIBs can be created using the MIB compiler supplied with the smxNS SNMP agent. The application can add these new MIBs or remove old or unused MIBs with relative ease.

The following describes known limitations in the SNMPv3 configuration or functionality.

Version 3 Trap: Version 3 trap messages cannot be generated. They would require the handling of Report messages from management stations and possibly non-authoritative authentication.

Row creation/deletion: Row creation/deletion is not supported.

User Management: The usmUserTable cannot be accessed or modified through SNMP, but users can be added in application code.

VACM support: The vacmViewTable cannot be accessed or configured through SNMP, but views can be added in application code.

Building an Application

Build-time Configuration

The build-time configuration of the agent is performed in the **snmp.h, vacm.c** and **usm.c** files in the **snmpsrc** directory. In the file **snmp.h** there is a set of definitions used to configure the agent. These symbolic constants may require modification before compiling and linking the product. The View-based Access Control Model definitions are declared in **vacm.c**.

Constants

ENTERPRISE Constant

The ENTERPRISE value refers to the ENTERPRISE ID assigned by ICANN (formerly IANA). It is used to partly form the *snmpEngineID* for the agent. Information on obtaining a Private Enterprise Number (PEN) is available at the time of this writing at http://pen.iana.org/pen/.

```
ENTERPRISE = 991
```

This default value is an old number for U S Software, and it should be changed for a production release.

System Variable Constants

The MIB system group used by the agent provides a textual description of the agent and is required by SNMP. These strings can be modified, adding appropriate values for the particular agent application. These variables are shown in Table 9-1.

Table 9-1: System Variables

Variable	Description
SYSCONTACT	The value is stored in *system.sysContact*. Replace this value with the company name and phone number.
SYSLOCATION	The value is stored in *system.sysLocation*. Replace this with the company name and address.
SYSDESCR	The value is stored in *system.sysDescr*. Replace this string with a description for the agent

Note that these values should not be greater than 64 bytes each without changing the size of the arrays that hold them. See *sysContact*, *sysLocation*, and *sysDescr* in **snmpsrc\agent.c**.

```
SYSCONTACT = "MDI (714) 437-7333, support@smxrtos.com"

SYSLOCATION = "MDI  Costa Mesa, CA  USA"

SYSDESCR = "Embedded controller running smxNS"
```

ENABLEAUTHENTRAPSVAL Constant

The ENABLEAUTHENTRAPSVAL specifies the *snmpEnableAuthentTrapsVal* default value. Use 1 for enabled and 2 for disabled.

```
ENABLEAUTHENTRAPSVAL = 2
```

MAXOID Constant

MAXOID defines the maximum length of an object identifier in the MIB. The object identifier (OID) uniquely defines MIB variables. Be sure this is large enough to accommodate all objects within any application MIB.

MAXOID Example

```
#define MAXOID 12  /* maximum length of object ID */
static const struct
  {u8 nlen, name[MAXOID], key[16];}
  party[]={
    {11, {0x2b,6,1,6,3,3,1,3,10,11,12}, {0} },
    {11, {0x2b,6,1,6,3,3,1,3,10,11,13}, {0} },
    {11, {0x2b,6,1,6,3,3,1,3,10,11,14},
    {0x74,0x68,0x69,0x73,0x74,0x68,0x69,
     0x73,0x74,0x68,0x69,0x73,0x74,0x68,0x69,0x33} },
};
```

The name field in the above table stores SNMP object IDs, and MAXOID specifies the maximum size for this value. Note that the OIDs start with the value 0x2b, which is the BER encoding for .1.3.

```
MAXOID = 15
```

MAXKEY Constant

MAXKEY defines the maximum number of keys allowed. Keys form the index used to identify a MIB table entry. For example, the *tcpConnTable* has four keys: *tcpConnLocalAddress*, *tcpConnLocalPort*, *tcpConnRemAddress*, and *tcpConnRemPort*. No other table in the MIB-II has more than four, so MAXKEY can be set to 4.

```
MAXKEY = 4
```

MAXKLEN Constant

MAXKLEN defines the maximum length in bytes for an encoded index. An index is the encoding of the keys used to define a table entry. These keys may be one or more of nearly any fixed length data type such as *IpAddress* or *INTEGER*. For standard MIB-II objects, the largest possible index is potentially generated by the *tcpConnTable*. Its keys include two *IpAddresses* each up to 8 bytes encoded and two 16-bit unsigned integers each up to 3 bytes encoded. The result is 22 bytes.

```
MAXKLEN = 22
```

MAXVAR Constant

MAXVAR specifies the maximum number of variables allowed in a request. A request is a message sent by the manager to the agent for reading or setting values of one or more variables. MAXVAR sets the maximum number of variables that may be accessed in one request. Note that the number of total response variables for a response to a bulk request is limited by the packet size, not this constant.

```
MAXVAR = 16
```

SNMP_MAXSIZE Constant

SNMP_MAXSIZE specifies the maximum transport size in bytes. Note that this value represents the size of each of four SNMP message buffers used for the following purposes: Receiving requests, sending replies, sending traps, and performing security operations. RFC 3411 requires this value be at least 484 bytes.

```
SNMP_MAXSIZE = 1000
```

User-based Security Model Configuration

The smxNS SNMP Agent can respond to SNMPv1, SNMPv2c and SNMPv3 messages. These SNMP versions have different conventions for qualifying SNMP queries and providing for secure communication. The smxNS SNMP Agent adopts the framework used by SNMPv3 and adapts it so that the configuration information can also be applied to SNMPv1 and SNMPv2c.

SNMPv3 established strong security by adding the concepts of “groups” and “views”.

Under SNMPv3, access to MIB variables and the way the information is transferred is tied to a group, which can also be though of as a class of users. A view is a portion of a MIB tree that is visible. The view could consist of a collection of entire MIBs, or it may be limited to certain subsections. A given group might also use different contexts to access SNMP information, for example, the “poweruser” group may use “normal” context most of the time and “advanced” context for administrative operations that involve modifying variable states.

Security settings are defined at run time before starting the SNMP Agent. A detailed description of the functions follow later in the Application Interface section of this chapter, but for now here’s a simple example.

static const OID sys_oid = {6, {0x2b, 6, 1, 2, 1, 1}};

snmpViewAdd(“sys”, 0xffffffff, &sys_oid);

snmpAccessAdd(“user1”, “normal”, “sys”, 0, noAuthNoPriv);

snmpUserAdd(“user1”, 0, 0, 0, 0);

This code sets up a view named “sys” for the MIB-2 System group (given by the pointer to the OID &sys_oid). The second line sets an access policy that group “user1” when operating in context “normal” can have read access to the “sys” group with no authentication needed an no encryption of the response. The third line creates a group named “user1” with no authentication or privacy passwords. With this configuration, one could walk the system group using this Net-SNMP command

$ snmpwalk –u user1 –n normal –l noAuthNoPriv –v 3 10.0.1.100

SNMPv1 and SNMPv2c use a “community string” to identify the entity making an SNMP query. If the string matches the configured string, access is granted, otherwise the incoming query is ignored.

For SNMPv1 and SNMPv2c access, you can use the snmpAcessAdd() function with an empty group parameter and the context name set to the community string like this.

snmpAccessAdd(“”, “public”, “sys”, 0, noAuthNoPriv);

Note that this function call builds on the “sys” group that was added with the call to snmpViewAdd() we used earlier.

With this configuration one could use this command to perform a walk with SNMPv1

$ snmpwalk –c public –v 1 10.0.1.100

Note that snmpViewAdd() can be called multiple times with the same view name in order to create a collection of subtrees that are all included in the view. For example, here’s how to set up a view that includes the MIB-II group (.1.3.6.1.2) and the SNMPv2 group (.1.3.6.1.6).

```
static const OID mgmt_oid = {4, {0x2b, 6, 1, 2}};
static const OID snmp_oid = {4, {0x2b, 6, 1, 6}};

snmpViewAdd(“mib2”, 0xffffffff, &mgmt_oid);
snmpViewAdd(“mib2”, 0xffffffff, &snmp_oid);
```

SNMPv3 defines a method of security known as the User-based Security Model (USM). The definition in RFC 3414 encompasses both authentication and privacy. Authentication means the verification of host identity, usually through a user name and password. Privacy means the encryption of SNMP messages such that unauthorized hosts cannot interpret the data. The current agent supports authPriv (i.e. authentication with privacy), *authNoPriv* (i.e. authentication without privacy) and *noAuthNoPriv* (i.e. no authentication and no privacy) for security levels. Future versions may add new authentication and privacy protocols.

It would not be secure to transmit passwords over the network, so the authors of SNMPv3 came up with a scheme to hide passwords. This method is called password localization and is described in RFC 3414 in section A.2. It takes the password and the *snmpEngineID* as input and outputs a digest-specific key. A SNMP manager uses the key with each SNMP request message to form an authentication digest using HMAC-MD5 or HMAC-SHA, and transmits the message plus the new digest as an authenticated SNMP message. The agent checks each digest value with the digest it creates in the same fashion on each message. If the two match, the management station and agent must have used the same localized password for the request to be further processed. Otherwise, the request causes the agent to transmit a *usmStatsWrongDigests* report to the manager.

The *snmpEngineID* used by the agent concatenates the *ENTERPRISE* value and the transport layer IP address. The *ENTERPRISE* value must always be configured in **snmp.h**, but the IP address is retrieved at run time.

View-based Access Control Configuration

SNMPv3 defines a method of access control known as the View-based Access Control Model (VACM). It is defined in RFC 3415 as a means of restricting access to particular subsets of variables based on the identity of the manager and *securityLevel* used in the request.

A view is a group of MIB variables on the agent. The agent defines a view for each user based on the user identity and *securityLevel*. A *contextName* and a *securityName* define the user identity and the *securityLevel* is listed directly in each request. Note that if no security is used (i.e. *securityLevel* == *noAuthNoPriv*), the *securityName* can be undefined. Also, in order to provide compatibility with version 1 and 2c management stations, the *contextName* in each view entry may refer to either a *contextName* or a *community name*. The *securityLevel* would then be assumed to be *noAuthNoPriv*.

The general practice is that informational variables be accessible to all users with all security levels. Write access and read access to sensitive information are limited to selective users implementing

authentication and perhaps privacy. Generally, if a user uses greater security than is required by the access entry including a particular variable, access is allowed. The VACM module will search through each entry until it finds a valid entry for the variable. This way multiple entries can be defined for a single *securityName* given different combinations of *contextNames* and *securityLevels*.

Agent Use of Build-time Constants

Here are user configurable settings from **snmp.h**:

```
#define ENTERPRISE 991
#define SYSCONTACT "MDI (714) 437-7333, support@smxrtos.com"
#define SYSLOCATION "MDI  Costa Mesa, CA  USA"
#define SYSDESCR "Embedded controller running smxNS"
#define DEFAULT_CONTEXT_STR “public”
#define ENABLEAUTHENTRAPSVAL 2
#define MAXOID 15
#define MAXKEY 4
#define MAXKLEN 22
#define MAXVAR 16
#define SNMP_MAXSIZE 1000
```

There are also constants in vacm.c that establish limits

```
#define NVIEWS         4
#define NSUBTREES      4
#define NACCESSENTRIES 8
```

And in usm.c

```
#define NUSERS            3
#define PASSWORD_MAX_LEN 16  /* includes terminating NULL */
```

Application Interface

The application file defines the run-time environment in which the agent executes.

The first step in launching an SNMP agent is to configure security parameters. If these settings are not made, no incoming SNMP queries will be qualified, and the agent will be unresponsive.

snmpViewAdd

Creates or adds to an SNMP view.

```
int snmpViewAdd(const char *name, u32 mask, const OID *oid);
```

This function creates or adds to the view with the given name with the MIB subtree in oid. The mask is a bit mask with the least significant bit applied to the first subidentifier of the OID and so on. If a mask bit is not set, the corresponding subidentifier is not compared for a match. Although the mask could be used to make the view definition more flexible, in practice it is set to 0xffffffff.

Return Value

>= 0	Success. The value returned is the index of the view that was created or updated.
< 0	An error occurred. Check the log for details on the error.

Example

```
#include “snmp.h”
. . .
static const OID sys_oid = {6, {0x2b, 6, 1, 2, 1, 1}};
. . .
snmpViewAdd(“sys”, 0xffffffff, &sys_oid);
```

Additional subtrees can be included in a given view by calling snmpViewAdd() again with the same view name and a pointer to another OID.

snmpAccessAdd

Establishes an access level for a given group and context

```
int snmpAccessAdd(const char *group, const *context, const char
*readview, const char *writeview, uint level);
```

This function establishes what SNMP MIB access is permitted for a given group and context. The readview indicates which MIB view is available for read operations and the writeview indicates the MIB view for write operations. The level may be noAuthNoPriv, authNoPriv or authPriv, indicating if authorization and privacy protocols are used in communication.

Return Value

>= 0	Success.
< 0	An error occurred. Check the log for details on the error.

Example

```
#include “snmp.h”
. . .
snmpAccessAdd(“admin-md5”, “admin”, “admin”, “mib2”, authNoPriv);
```

In this example, the group admin-md5 when operating in the admin context is allowed read access to the admin view and read-write access to the mib2 view. Operations will apply the authentication protocol to confirm the identity of the entity making the queries before completing them.

This function is also used to set up the community string for use with SNMPv1 and SNMPv2. In that case the group field is left empty (“”), the context field provides the string, and the other fields are filled in as needed.

In order to indicate no read or no write access, the view name should be given as 0 in the corresponding field.

snmpUserAdd

Establishes a group that can access SNMP information.

```
int snmpUserAdd(const char *group, uint aproto, const char *auth_pw,
uint pproto, const char *priv_pw);
```

This function sets up an SNMP group and specifies the authentication protocol, the authentication password, the privacy protocol and the privacy password to be used with that group when needed.

The aproto field should be one of usmNoAuthProtocol, usmHMACMD5AuthProtocol, usmHMACSHAAuthProtocol or usmHMACSHA2AuthProtocol.

The pproto field should be one of usmNoPrivProtocol, usmDESPrivProtocol, usmAESPrivProtocol or usmAES2PrivProtocol.

The passwords that are used must be less than or equal to PASSWORD_MAX_LEN characters including a terminating 0. This constant is defined at the top of XNS/snmpsrc/usm.c and defaults to 16.

Return Value

>= 0	Success
< 0	An error occurred. Check log for details on error.

Example

```
#include “snmp.h”
. . .
snmpUserAdd(“admin-md5”, usmHMACMD5AuthProtocol, “secretpassword”,
usmDESPrivProtocol, “mylittlesecret”);
.
```

In this example, the group admin-md5 is set up to use HMAC-MD5 as the authentication protocol with the password “secretpassword” and use DES for the privacy protocol with the password “mylittlesecret”.

AGENT_CONTEXT Structure

```
typedef struct
{
    const MIB **mibs;          /* Array of pointers to host MIBs */
    uint16 nummibs;            /* Number of host MIBs */
    const TRAP_HOST **thosts;  /* Trap hosts */
    uint16 numthosts;          /* Number of trap hosts */
    uint16 trapv, trapt;       /* Trap version and startup type */
    const TRANSPORT_MAPPING *tm; /* Transport mapping */
} AGENT_CONTEXT;
```

The `mibs` field is the list of MIBs that managers may have access to. Note it is vital that the MIBs be listed in lexicographical order. If not, the agent will think certain variables do not exist within the MIB. The `nummibs` field specifies the number of MIBs available.

The `thosts` field specifies the hosts to which agent traps will be sent. The `TRAP_HOST` definition is simply '`typedef uint8 *TRAP_HOST;`' and each host should be acceptable to the transport layer. In other words, the transport layer needs to be able to open a connection to the entity specified by the trap host field. The `numthosts` field specifies the number of trap hosts available.

If the trap hosts or other properties of the AGENT_CONTEXT structure need to be modified after starting the SNMP Agent, the agent should be stopped and restarted with the new configuration.

The `trapv` field specifies the trap version to use during agent operations. The `trapt` field specifies the trap used by the agent during startup. Use –1 for none. Otherwise use one of these defined types from **snmp.h**:

```
COLDSTART
WARMSTART
LINKDOWN
LINKUP
AUTHENTICATIONFAILURE
EGPNEIGHBORLOSS
ENTERPRISESPECIFIC
```

The `tm` field specifies the transport mapping to be used by the agent. The `TRANSPORT_MAPPING` data structure is defined later.

Example

This is an example of a SNMP agent application taken from **nsdemo.c**.

A global structure is declared for the agent task to initialize from. In this example, the structure has been set up to request a SNMPv1 (0) `COLDSTART` trap be sent when the agent is started. The USNET DPI transport mapping is used for sending and receiving SNMP messages.

```
#include "snmp.h"

extern const MIB mib_if, mib_at, mib_ip, mib_icmp, mib_tcp, mib_udp;
extern const MIB mib_sys, mib_snmp, mib_engine;
extern const MIB mib_usm;

/* The following MIBs must be in lexicographical order */
static const MIB *mibs[] =
```

```
{
    &mib_sys,       /* system group */
    &mib_if,        /* interfaces group */
    &mib_at,        /* address translation group */
    &mib_ip,        /* IP group */
    &mib_icmp,      /* ICMP group */
    &mib_tcp,       /* TCP group */
    &mib_udp,       /* UDP group */
    &mib_snmp,      /* SNMP group */
    &mib_engine,    /* SNMPv3 engine group */
    &mib_usm        /* USM group */
};

static const TRAP_HOST primary = “192.168.1.30”;
static const TRAP_HOST secondary = “192.168.1.31”;
static const TRAP_HOST *thosts[] =
{
    &primary,
    &secondary
};

extern const TRANSPORT_MAPPING TM_DPI;

/* This structure is defined as external in SNMPAgentTask() */
const AGENT_CONTEXT snmp_ac =
{
    mibs, (sizeof(mibs) / sizeof(MIB *)),
    thosts, (sizeof(thosts) / sizeof(TRAP_HOST)), 0, COLDSTART,
    &TM_DPI
};
. . .
```

ussSNMPAgentInit

Initializes the agent.

```
sint16 ussSNMPAgentInit(const AGENT_CONTEXT *acp);
```

This function initializes the agent with the run-time environment defined by the value of the AGENT_CONTEXT parameter. The run-time environment that the agent uses is defined by the MIBs visible to the agent, the Trap hosts, and a transport mapping.

Return Value

>= 0	No error
< 0	An error

Example

```
#include “snmp.h”
. . .
extern const AGENT_CONTEXT snmp_ac;
. . .
i1 = ussSNMPAgentInit(&snmp_ac);
if (i1 < 0)
{
   DEBUG_MSG2_PAR1(“SNMPAgentTask: Initialization failed %d\n”, i1);
   return;
}
```

ussSNMPAgentCheck

Checks the status of the agent for pending requests, and responds as necessary.

```
sint16 ussSNMPAgentCheck(void);
```

This function checks the transport for incoming messages, and generates responses as necessary.

Return Value

>= 0	No error
< 0	An error

Example

```
#include “snmp.h”
. . .
/* Control loop for reading requests and
   forming/sending replies */
while (ussSNMPAgentCheck() >= 0)
    ;
```

ussSNMPAgentShut

Terminates the agent.

```
sint16 ussSNMPAgentShut(void);
```

This function performs any clean-up necessary to terminate all the layers of the Agent.

Return Value

>= 0	No error
< 0	An error

Example

```
#include “snmp.h”
. . .
ussSNMPAgentShut();
```

ussSNMPAgentTrap

Sends a trap to all configured trap hosts as defined in the AGENT_CONTEXT.

```
sint16 ussSNMPAgentTrap(uint8 type, uint8 spec,
                        const uint8 *contextName,
                        const uint8 *vbs, uint16 len);
```

type	the trap type
spec	trap-specific code
contextName	context or community name
vbs	pointer to a variable bindings for trap
len	the buffer length

The ***ussSNMPAgentTrap()*** function may be used from an agent application to send a trap to a manager. The ussSNMPAgentCheck() function may be run concurrently with the ussSNMPAgentTrap() function since they are designed to be thread safe with respect to each other. Trap types specified as 0 through 6 are shown in Table 9-2.

Table 9-2: SNMP Trap Types

Value	Trap Type	Description
0.	cold start	The agent network protocol has reinitialized, indicating that its configuration may have been altered.
1.	warm start	The agent network protocol has reinitialized; however, its configuration has not been altered.
2.	link down	A communication link has failed. The failing link is identified via the first variable within the variable bindings field of the PDU (protocol data unit). The PDU is, essentially, the data protocol used by SNMP. The variable bindings field is a list of MIB variables sent to the manager packaged within a PDU.
3.	link up	A communication link has come up. The affected link is identified as the first element within the variable bindings field.
4.	`AuthenticationFailure`	The agent could not resolve the authentication for an SNMP message received from the manager.
5.	`EgpNeighborLoss`	An EGP peer neighbor is down.
6.	`EnterpriseSpecific`	A nongeneric trap has occurred. This is specific to a particular enterprise. Use this for application-specific traps.

Return Value

The number of traps sent. This should be compared to the number of trap hosts configured in the AGENT_CONTEXT.

Example

To send a trap from an application, simply call ***ussSNMPAgentTrap()*** and pass in the trap type, the trap-specific code, the context/community name, a pointer to a buffer of variable data for the manager to process, and the length of the variable data. If the buffer is not needed 0 may be used. For example, to send a "warm start" trap with no variable data, use:

```
int rc;           /*return code */
rc = ussSNMPAgentTrap(WARMSTART,0, "public", 0, 0);
if (rc <= 0)
    <process error >
```

If a trap must pass variable data to the manager, declare a buffer, assign the variable binding data to it and pass it to ***ussSNMPAgentTrap()***.

```
#define VARBUFFERSIZE   <some constant value>
....
int rc;           /* return code */
u8 varbuffer[VARBUFFERSIZE];
....
varbuffer = <load the data into the buffer>;
....
rc = ussSNMPAgentTrap(WARM_START, 0, "public", varbuffer,
                VARBUFFERSIZE);
if (rc != 0)
   <process error>;
```

This function call is flexible in that the variable data may be passed in any format; however, it is constrained to what the manager can understand. Generally, this would be in the form of an SNMP variable bind list. Here is a more detailed example

```
static const u8 oid_snmptrapoid[] = {0x2b, 6, 1, 6, 3, 1, 1, 4, 1,
0};
static const u8 oid_test0[] = {0x2b, 6, 1, 4, 1, 16, 17};  /*
arbitrary enterprise OID tree starting with .16. */
static const u8 oid_test1[] = {0x2b, 6, 1, 4, 1, 16, 17, 18, 1};  /*
represents specific enterprise trap */

   u8  vbbuf[64];  /* size according to space occupied by var bindings */
   u8 *prevp;
   u8 *curp;
   u8 *startp;

     curp = vbbuf + sizeof(vbbuf);
     startp = curp;

     /* These will appear in reverse order */
     prevp = curp;
     snmpRWriteVal(&curp, "test", SNMP_STRING, strlen("test"));
     snmpRWriteVal(&curp, oid_test1, SNMP_IDENTIFIER, sizeof(oid_test1));
     snmpRWriteLength(&curp, SNMP_SEQUENCE, (s16)(prevp - curp));

     prevp = curp;
     snmpRWriteVal(&curp, oid_test0, SNMP_IDENTIFIER, sizeof(oid_test0));
```

```
        snmpRWriteVal(&curp, oid_snmptrapoid, SNMP_IDENTIFIER,
    sizeof(oid_snmptrapoid));
        snmpRWriteLength(&curp, SNMP_SEQUENCE, (s16)(prevp - curp));
        ussSNMPAgentTrap(ENTERPRISESPECIFIC, 0, (const u8 *)"public", curp,
    startp - curp);
```

Customizing the Agent

Configuring the Agent MIB

Standard MIBs are supplied with smxNS SNMP based on Internet standards defined by RFCs (request for comments, on the Internet) 1156 and 1213. The MIBs are the System Group, Interfaces Group, Address Translation Group, IP Group, ICMP Group, TCP Group, UDP Group, SNMP Group, snmpEngine Group and usmMIBBasicGroup. These RFCs have since been clarified in several updated RFCs modularized from the originals.

MIB Structure

Each MIB module must be molded into the MIB structure used by the agent.

```
typedef struct
{
    const MIBVAR *mvp;          /* MIB variables */
    sint16 (*numvars)(void);    /* Number of variables */
    const MIBTAB *mtp;          /* MIB tables */
    sint16 (*numtabs)(void);    /* Number of tables */
    void (*get)(sint16 varix, sint16 tabix, uint8 **vvptr);
    sint16 (*set)(sint16 varix, sint16 tabix);
    sint16 (*index)(sint16 varix, sint16 index);
    void (*init)(uint16 type);  /* Initialize the MIB */
} MIB;
```

MIBVAR and MIBTAB Structures

The MIBVAR and MIBTAB structures are the primary data structures, which define MIB data. Each MIB contains variables *mibvar* and *mibtab,* which are simply arrays of these structures. MIBVAR and MIBTAB are defined in **snmpv3.h** as follows:

```
typedef struct
{
    uint8 nlen, name[MAXOID];
} OID;

typedef struct
{
    OID oid;               /* Base OID of table */
    uint8 nix;             /* Number of indices for table */
    uint16 ix[MAXKEY];     /* Index values (offsets) */
    uint16 len;            /* Length of table */
} MIBTAB;
```

```
typedef struct
{
    OID oid;              /* Identifier name, length */
    uint8 opt;            /* Options */
    uint8 type;           /* Type of variable */
    sint16 len;           /* Length of pointer field */
    void *ptr;            /* Pointer to variable data */
} MIBVAR;
```

MIBVAR contains the definitions and values of all MIB variables. MIBTAB contains indices into the MIBVAR for accessing MIB table (SEQUENCE OF) entries. Most of these fields are used internally by the SNMP agent; however, some are useful to know. OID is used to uniquely define each record in the MIBVAR and MIBTAB. Also, for a given MIB table variable, the OID is the key value, which links MIBVAR and MIBTAB entries. The purpose of the MIBVAR is simply to store all MIB data; that is, scalar values and values within a MIB table. In the case of a MIB table, the mibtab.ix[i] values are used as indices to the appropriate records in the MIBVAR. An example of its use is provided in the 'MIB.index()' section.

Default Operation

When the SNMP agent receives a ***GetRequest*** PDU (protocol data unit), the entries in the MIBVAR array are reviewed to find an entry that matches the requested OID. The *ptr* field in the matching entry is then used to locate the memory location that contains the value that should be returned. For scalar variables, this location is read directly. For variables in tables, an offset is added to the pointer that corresponds to the index portion of the OID in the ***GetRequest*** PDU.

When the SNMP agent receives a ***SetRequest*** PDU, the corresponding entry is located as above, and the memory location based on the *ptr* field is overwritten with the value provided in the ***SetRequest*** PDU.

MIBVAR Record Options

Some of the variables in MIBVAR may not be well suited to the default operation of the SNMP agent. To support these needs, the *opt* field of the MIBVAR record allows for flags that will indicate that special processing is required.

IMMED — The variable value is stored directly in the *len* field, rather than being pointed to by the *ptr* field. The variable should be an 8-bit value. The value for *ptr* can be 0.

IMMED2 — The variable value is stored directly in the *type* and *len* fields, rather than being pointed to by the *ptr* field. The variable should be a 16-bit value. The value for *ptr* can be 0.

SCALAR — The variable is in a table, but should be looked up without adding an index to *ptr*. This allows a variable to be part of a table, but not accessed in the same manner as other variables in the table. If the value for a variable is known to be the same for every index in the table, then this technique can be used to reduce the size of the memory image that represents the contents of the table. This flag need not be specified for normal scalar variables.

W — The variable may be modified.

SX — The variable is the first item of a MIB table.

CAR	A read notification function may be called before returning the value of the variable.
CAW	A write notification function may be called after writing a new value to the variable.
CHOICE	A 'CHOICE' ASN.1 syntax element is required in the OID of this object. Note that it is only used to force the atTable to behave correctly and, if defined, code size will increase for all MIBs.

MIB.set() and MIB.get() Functions

These functions are written as part of each MIB and provide the actions to perform for read or write notification.

```
static sint16 set(sint16 varix, sint16 tabix);
void get(sint16 varix, sint16 tabix, uint8 **vvptr);
```

The first argument, *varix*, is an integer which acts as an index into the MIB identifying the variable to be accessed. If that MIB variable is a MIB table, the *tabix* parameter may be used as a 0-based index into the table. If *varix* is a scalar value or not a table entry, then no index is required and -1 is passed in for *tabix*. The ***vvptr* is passed to the ***get()*** function in case the MIB needs to replace the value pointer with a new address for the agent to operate upon.

The value returned by ***set()*** should be 0 if the function executes normally. In the case of an error situation, the value returned from these functions will be used as an error code in the response that the SNMP agent sends to the SNMP request.

The ***get()*** and ***set()*** functions are called indirectly from the function ***ussSNMPAgentCheck()*** in **agent.c** through the MIB structure in which the ***get()*** function pointer resides. The declaration below shows how the MIB structure is defined.

Example

```
#include “snmpv3.h”
. . .
static void get(sint16 varix, sint16 tabix, uint8 **vvptr)
{
    const MIBVAR *mvp = &mibvar[varix];
    uint8 *bytevp = *vvptr;

    /*
    ** If varix is 3, the variable is a 32-bit value
    ** that must be updated before being read by the agent.
    ** We set it here to a value that is determined by using
    ** a value in a table indexed by an array of index
    ** values.
    */

    if (varix == 3)    /* Fourth variable in MIB */
    {
        *(uint32 *)*vvptr = Barray[Aarray[tabix].nindex].value32;
    }

    /*
    ** If varix is 12, the first index is not stored in the
    ** table. The second and all subsequent indices are in
    ** the table, however.  We can simply point the value
```

```
    ** pointer to a new location.
    */

    if (varix == 12)   /* Thirteenth variable in MIB */
    {
        if (tabix == 0)
            *vvptr = &value;
        else
            *vvptr = &table[tabix].value;
    }
}

static sint16 set(sint16 varix, sint16 tabix)
{
    MIBVAR *mvp = &mibvar[varix];
    uint8 *bytevp = mvp->ptr;

    if (varix == 3)
    {
        if (*(uint32 *)bytevp == 0x1234567)
        {
            *(uint32 *)bytevp = 0;
            return badValue;
        }
    }

    return 0;
}
. . .

const MIB mib_example =
{
    mibvar,
    mibvarsize,
    mibtab,
    mibtabsize,
    get,
    set,
    index,
    init
};
```

The globally-accessible function pointer `mib_example.get` is assigned the ***get()*** function which is local to the current MIB module. The ***mib_example.get()*** function is only called if CAR is in the option field for the variable and the ***get()*** function pointer is valid (that is, not 0). Upon entry into the ***get()*** function, the variable *varix* is an index into the MIBVAR array for the current variable to be read. The *tabix* is assigned –1 if no table is being accessed. Otherwise, *tabix* is a zero-based index into the table to which the variable belongs.

MIB.index() Function

Determines size of tables in a MIB.

```
sint16 index(sint16 varix, sint16 index);
```

If tables exist in a MIB, the SNMP agent needs a mechanism to determine the size of the tables that have been added. The ***index()*** function indicates when the end of the table has been reached and also can be used to specify when a table entry should be skipped. Good examples of ***MIB index()*** functions can be found in **mib_if.index, mib_tcp.index, mib_udp.index**, etc.

The ***index()*** function is required to implement a table.

When the SNMP agent receives a get request or a get-next request that involves a MIB table and the ***index()*** function is defined, the agent will call the ***index()*** function while iterating through the table to determine if an entry should be included in the search for the variable. The MIB ***index()*** function is defined similarly to the MIB ***get()*** and ***set()*** functions.

Return Value:

1	Accept the record
0	Skip over the record
-1	End of table

Example

```
/* Index the IP MIB's tables */
static sint16 mibindex_ip(sint16 varix, sint16 tabix)
{
    uint8 *cp;
    uint16 us1;
    sint16 i1;

    cp = (uint8 *)mibvar_ip[varix].oid.name + 5;
    us1 = *cp++ << 8;
    us1 += *cp;

    switch (us1)
    {
    case 0x0416:            /* IP net to media table */
        if (nets[tabix].netstat == 0)
            break;
        for (i1 = 0; i1 < Eid_SZ; i1++)
            if (nets[tabix].Eaddr.c[i1])
                goto lab5;
        break;
    case 0x0414:            /* IP address table */
        if (tabix >= NNETS)
            goto lab7;
        if (nets[tabix].cfgflags & LOCALHOST)
            goto lab5;
        break;
```

```
    case 0x0415:            /* IP routing table */

        if (nets[tabix].netstat == 0)
            break;
        if (!(nets[tabix].cfgflags & LOCALHOST))
            goto lab5;
        break;
    default:                    /* any other */
        goto lab5;
    }
    return 0;
lab5:
    return 1;
lab7:
    return -1;
}
```

In this example, a section of the Object ID is used to identify the variable for which the index function is being called. The value of *index* could also be used for this purpose, but using a section of the OID allows a subtree of the MIB to easily be identified. At the beginning of the function, *cp* is set up to point to the interesting section of the OID, and then the next two bytes of the OID are stored in *us1*.

This is just one example of how an ***index()*** routine could be coded. Processing of `accept`, `skip`, or `end of table` is determined by checking values of smxNS data structures in the above case. The *index* may be used as an index into some of these structures. The `MIBTAB` values are simply used as flags to indicate which variable is to be processed. The actual value of the variable requires accessing of the smxNS data structures. Refer to the smxNS documentation and source code for explanations of values such as *NNETS*, and `nets[`*tabix*`]`.

Adding New MIBs

A particular application may require new MIBs in addition to those supplied as part of the MIB-II. If this is the case, use the ASN.1 (Abstract Syntax Notation) syntax to add the definitions of variables to a MIB file. Refer to a text on SNMP or the appropriate RFCs for definitions of this syntax. Then use **MIBTOC** to translate the ASN.1 definitions into C code understandable to the SNMP agent.

MIB Translation Overview

To use a new MIB with the smxNS SNMP agent, a file describing the MIB variables must be compiled into C source code. The program **MIBTOC**, performs this translation. It reads a description of the MIB variables in ASN.1 format, and produces two ANSI C-compatible files. In the following diagram, “MIB” represents the name of the MIB file.

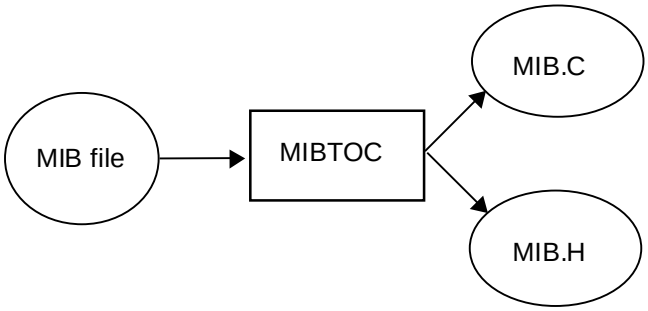


Figure 9-2: MIB Translation

The source files created by the MIB compiler may require additional hand coding to add features or supply information that can't be derived from the MIB. The application can compile and link the MIB with the agent so the agent can access the MIB database.

Building the MIB Translator

The translator is provided as source code and as a pre-compiled executable. The source is located in the **BIN\MIBTOC** directory. To build it by hand, simply use the included batch file. If the batch file isn't set up for your tools, pass the source file as an argument to a compiler/linker. For instance, if using the Borland compiler, run:

```
bcc BIN\MIBTOC\mibtoc.c
```

Or, if building from a UNIX environment, run:

```
cc BIN/MIBTOC/mibtoc.c
```

MIBTOC is ANSI-compatible and can be compiled by most commercially available compilers. Since the **MIBTOC** application uses a significant amount of stack space, the compiler or linker may need to be configured with an option to increase the stack space. The compiler is included in executable format for DOS and Windows platforms.

Running the MIB Translator

MIBTOC takes one or two arguments: The first argument is the name of the MIB file to be processed, and the optional second argument provides the base name for the output file. The syntax is:

```
MIBTOC mibfile [outfile]
```

If an output file name is not specified, the name for the output files will be derived from the base file name of the input file. For example, this command will generate the output files **toaster.c** and **toaster.h**:

```
MIBTOC toaster.mib
```

If the second parameter is provided, then the output file names are based on the second parameter. Given this command line, the translator will generate the output files **test.c** and **test.h**:

```
MIBTOC toaster.mib test
```

Watch the output of **MIBTOC** to be sure that no errors occurred in preparing the output files. A normal run will look like:

```
C:\usnet\snmpsrc>mibtoc rfc2571.txt

USNET MIB to C Translator 1.10
  Copyright (c) U S Software 1994, 1999, 2000.
Root: ccitt
Root: iso
Root: joint-iso-ccitt
Type 'No Access': org { iso 3 }
Type 'No Access': dod { org 6 }
Type 'No Access': internet { dod 1 }
Type 'No Access': mgmt { internet 2 }
Type 'No Access': experimental { internet 3 }
Type 'No Access': private { internet 4 }
Type 'No Access': security { internet 5 }
Type 'No Access': snmpV2 { internet 6 }
Type 'No Access': snmpDomains { snmpV2 1 }
Type 'No Access': snmpProxys { snmpV2 2 }
Type 'No Access': snmpModules { snmpV2 3 }
Type 'No Access': mib-2 { mgmt 1 }
Type 'No Access': transmission { mib-2 10 }
Type 'No Access': enterprises { private 1 }
Type 'No Access': snmpFrameworkMIB { snmpModules 10 }
TC: SnmpEngineID (OctetString)
TC: SnmpSecurityModel (Integer)
TC: SnmpMessageProcessingModel (Integer)
TC: SnmpSecurityLevel (Integer)
TC: SnmpAdminString (OctetString)
Type 'No Access': snmpFrameworkAdmin { snmpFrameworkMIB 1 }
Type 'No Access': snmpFrameworkMIBObjects { snmpFrameworkMIB 2 }
Type 'No Access': snmpFrameworkMIBConformance { snmpFrameworkMIB 3 }
Type 'No Access': snmpEngine { snmpFrameworkMIBObjects 1 }
Type 'OctetString': snmpEngineID { snmpEngine 1 }
Type 'Integer': snmpEngineBoots { snmpEngine 2 }
Type 'Integer': snmpEngineTime { snmpEngine 3 }
Type 'Integer': snmpEngineMaxMessageSize { snmpEngine 4 }
Type 'No Access': snmpAuthProtocols { snmpFrameworkAdmin 1 }
Type 'No Access': snmpPrivProtocols { snmpFrameworkAdmin 2 }
Type 'No Access': snmpFrameworkMIBCompliances {
snmpFrameworkMIBConformance 1 }
Type 'No Access': snmpFrameworkMIBGroups { snmpFrameworkMIBConformance
2 }
Type 'No Access': snmpEngineGroup { snmpFrameworkMIBGroups 1 }
2554 lines processed OK
```

If there is a problem in processing the file, the last line will not read "`... processed OK`" but rather will describe an error in processing the file. For example, if the definition for ***MAXOID*** in **mibtoc.c** is too small, then this message will be displayed:

```
L388 myTableIndex MAXOID too small
```

This indicates that in processing line 388 of the MIB file, it was discovered that there was not enough room to build the needed Object ID array. To correct this, the value for ***MAXOID*** should be increased in **mibtoc.c**, and **MIBTOC** should be rebuilt. Also ***MAXOID*** should be increased to the same value in **snmpconf.h**, because it will be used again when building the SNMP agent.

MIB Files

MIBTOC generates two files as output. Using the example of an ASN.1 input file named **toaster.mib**, the output files would be **toaster.c** and **toaster.h**. The SNMP agent uses the output files as follows:

toaster.h Defines external variable and symbol definitions to which the application and MIB module may wish to refer as "`extern`".

toaster.c Allocates MIB variable and table values statically and provides the global '`MIB mib_toaster`' structure declaration to provide global access to the MIB from the application.

Read/Write Notification

Each variable in a MIB may have read or write notification associated with it. This means that prior to a get operation or after a set operation, the agent will signal the MIB that its data is being operated upon.

For ***get***-, ***getNext***- or ***getBulk***-requests, the option field in the MIB variable is checked for read notification (`CAR` – Call Application Read). If this is set for the variable, the ***get()*** function for the MIB will be called with the index of the variable and a pointer to a pointer to the value of the variable. This is so that the MIB can update the value of the variable or dynamically redirect it to a new memory location.

For ***set***-requests, the option field in the MIB variable is checked for write notification (`CAW` – Call Application Write). If this is set for the variable, the MIB ***set()*** function will be called with the index of the variable. Special processing can be performed due to important changes in the value of the MIB variable.

To indicate to the agent that read or write notification is required on a given variable, add the `CAR` and/or `CAW` options to the `opt` field of the variable record within the MIB source file using the bitwise `OR` operator (i.e. '|').

Example

```
{8,{0x2b,6,1,2,1,1,6,0}, W | CAR | CAW, String,
        sizeof(syslocat), syslocat}, /* sysLocation */
```

This example shows a `MIBVAR` record (see the next section) which adds read and write notification to the MIB variable *`sysLocation`*. Before modification, the option field was simply `W`, indicating a variable that allows write access. The option field may be zero for no options or a combination of others. The possibilities are defined in **snmpv3.h** and are shown in Table 9-3 below.

```
#define IMMED   0x01 /* Immediate value in mvp->len */
#define IMMED2  0x02 /* Immediate value in mvp->type + len */
#define BASE1   0x03 /* Base 0 in data space, base 1 in MIB */
#define SCALAR  0x04 /* Table not indexed (no offset) */
#define W       0x80 /* Write allowed */
#define SX      0x40 /* Sequential table index inferred */
#define NWORDER 0x20 /* Network byte ordering for basic type */
#define CAR     0x10 /* Call application after read */
#define CAW     0x08 /* Call application before write */
```

Table 9-3: MIBVAR Record Options Field

Options Field	Description
IMMED	The variable value is stored directly in the *len* field (see below), rather than using the *ptr* field to store the address of the value.
IMMED2	Similar to IMMED except the variable value is stored directly in the type and *len* fields (see below).
BASE1	The variable index value is represented by SNMP starting at a base value of '1' even though the agent must deal with the actual data with a base '0'.
SCALAR	A scalar value. In other words, the value is not in a table even though its ASN.1 definition defines it as part of a table.
W	A variable that allows write access, i.e., the value may be modified.
SX	Indicates the first item of a MIB table, i.e., a SEQUENCE OF.
CAR	Use Read notification.
CAW	Use Write notification.

Summary of Adding a User-Defined MIB

1. Create the standard "out of the box" version of the SNMP agent, and confirm that the standard MIB-II variables are accessible from an SNMP manager.
2. Build the **MIBTOC** compiler, if it is not already built for the development platform.
3. Create the enterprise-specific MIB. This example presents the wt2000 remotely accessible weather station MIB, which uses the MIB called **weather.mib**. The MIB will be associated with a product of the fictional company "WeatherTek International" that makes devices that record weather conditions. These conditions can be retrieved from their instruments through an SNMP manager.

The first information to be included in the user-defined MIB will establish the path in the MIB hierarchy to the enterprise-specific MIB. If the enterprise code for WeatherTek International were 123, and the variables were those collected by the wt2000 model, then the following information might appear first in **weather.mib**:

```
—                   MIB DESCRIPTION
WEATHER-MIB DEFINITIONS ::= BEGIN
—
weathertek      OBJECT IDENTIFIER  ::= { enterprises 123 }
wt2000          OBJECT IDENTIFIER  ::= { weathertek 3 }
```

In this example, the weather station contains components that monitor conditions at a number of altitudes. Some of the variables in **weather.mib** concern the weather station as a whole, and some concern the conditions at each altitude. Let us say that a string is set up to hold the unit location, and the latitude and longitude of the installation are also stored.

This information might appear in **weather.mib** as follows:

```
—
— The wt2000 Group
—
location OBJECT-TYPE
   SYNTAX      DisplayString
   ACCESS      read-write
   STATUS      mandatory
   DESCRIPTION “The geographical name for the device location.”
   ::= { wt2000 1 }
latitude OBJECT-TYPE
   SYNTAX      INTEGER
   ACCESS      read-write
   STATUS      mandatory
   DESCRIPTION “The latitude at which the device is installed.”
   ::= { wt2000 2 }
longitude OBJECT-TYPE
   SYNTAX      INTEGER
   ACCESS      read-write
   STATUS      mandatory
   DESCRIPTION “The longitude at which the device is installed.”
   ::= { wt2000 3 }
```

Now a table can be introduced to hold the information that is collected for a number of altitudes. For this table, the altitude will act as an index, and temperature, humidity, wind speed and wind direction will be monitored. Here is how it might appear in **weather.mib**:

```
weatherTable OBJECT-TYPE
   SYNTAX      SEQUENCE OF weatherEntry
   ACCESS      not-accessible
   STATUS      mandatory
   DESCRIPTION “This table contains a tally of weather conditions”
   ::= { wt2000 4 }
weatherEntry OBJECT-TYPE
   SYNTAX      WeatherEntry
   ACCESS      not-accessible
   STATUS      mandatory
   DESCRIPTION “Each row represents conditions at a given altitude.”
   INDEX       { altitude }
   ::= { weatherTable 1 }
WeatherEntry ::= SEQUENCE {
   altitude      INTEGER,
   temperature   INTEGER,
   humidity      INTEGER,
   windSpeed     INTEGER,
   windDirection INTEGER { NORTH     (1),
                           NORTHEAST (2),
                           EAST      (3),
                           SOUTHEAST (4),
                           SOUTH     (5),
                           SOUTHWEST (6),
                           WEST      (7),
                           NORTHWEST (8)}}
altitude OBJECT-TYPE
   SYNTAX      INTEGER
   ACCESS      read-only
   STATUS      mandatory
   DESCRIPTION “Altitude in meters, used as an index.”
   ::= { weatherEntry 1 }
```

The definitions for *temperature*, *humidity*, *windSpeed*, and *windDirection* would appear similar to the definition for *altitude*.

Process the MIB with **MIBTOC** to create source code. Make sure that the compiler reports no errors. Using the **mibtoc.exe** utility in the **BIN** directory:

```
cd snmpsrc
BIN\mibtoc weather.mib
```

Add the files generated by **MIBTOC** to the project. So in this example, add the file weather.c to the project.

If there are any tables in the user-defined MIB, an ***index()*** function will have to be created in **snmpsrc\weather.c** and added to the MIB **mib_weather** declaration.

```
cd snmpsrc
edit weather.c
      const MIB mib_weather =
      {
          mibvar,
          mibvarsize,
          mibtab,
          mibtabsize,
          0,          /* get */
          0,          /* set */
          index,      /* <<< New >>> */
          0           /* init */
      };
```

Declare the program variables that are introduced in the user defined MIB. In this example, external declarations for the variables will be written into weather.h, but the variables will not be declared in any module. The names of the variables are based on the names appearing in the MIB definition, and can be found in weather.h, which is excerpted here:

```
extern char *location;
extern int latitude;
extern int longitude;
extern struct weatherTable weatherTable[];
```

These variables must be declared somewhere in the application, and for this example the declarations are made in a modified version of **weather.c**:

```
#define WTABSZ 3  /* number of entries in weather table */
char *location;
int latitude;
int longitude;
struct weatherTable weatherTable[WTABSZ];
```

Note that the size of the table is not apparent from the information in the MIB definition and may be variable. In this example, a constant has been defined to specify the size. *WTABSZ* represents the largest possible table size. This information should be used by the ***index()*** function.

Initialize the variables in the user-defined MIB. Any default values or fixed values can be set up before the SNMP agent is started. Also, any index fields in tables must be initialized before the agent is started.

Here is an example from the modified **weather.c**:

```
const char defaultlocation[] = “Portland, Oregon”;
#define DEFAULTLATITUDE 46
#define DEFAULTLONGITUDE 123

static void init(uint16 type)
{
    memset(weatherTable, 0, sizeof(weatherTable));
    location = defaultlocation;
    latitude = DEFAULTLATITUDE;
    longitude = DEFAULTLONGITUDE;

    for (i1 = 0; i1 < WTABSZ; i1++) {
        weatherTable[i1].altitude = i1 * 1000 + 1000;
        weatherTable[i1].windDirection = 1;
    }

}
```

In this example, default values for *location*, *latitude*, *longitude* and the *windDirection* field in *weatherTable* are initialized. The *altitude* index field in the table is initialized with the values 1000, 2000 and 3000.

If the value of a variable should be updated before being read, then the ***get()*** function should be implemented.

Likewise, if special action should be taken once a variable is written, then the ***set()*** function should be implemented, and if the number of rows in a table is variable then the ***index()*** function should be implemented.

The weather MIB structure will have to be updated to reflect any required get, set, index or init functions:

```
const MIB mib_weather =
{
    mibvar,
    mibvarsize,
    mibtab,
    mibtabsize,
    get,       /* <<< New >>> */
    set,       /* <<< New >>> */
    index,     /* <<< New >>> */
    init /* <<< New >>> */
};
```

Configuring the Transport Mapping

A Transport Mapping is a defined method of data transfer between SNMP hosts. RFC 3417 defines the use of SNMP over UDP/IP on Internet-based networks as well as many others. From this, a module was defined called TRANSPORT_MAPPING. Here is the structure definition that the smxNS SNMP agent uses:

```
typedef struct
{
   /* Initialize underlying transport framework */
    sint16 (*init)(uint8 *ip, uint32 *maxsize, uint8 *name);

   /* Open passively to receive SNMP messages */
    sint16 (*passive_open)(void);
    sint16 (*passive_read)(uint8 *buff, uint16 len);
    sint16 (*passive_write)(const uint8 *buff, uint16 len);
    sint16 (*passive_close)(void);

   /* Open actively to send SNMP messages */
    sint16 (*active_open)(const uint8 *rhost);
    sint16 (*active_write)(const uint8 *buff, uint16 len);
    sint16 (*active_read)(uint8 *buff, uint16 len);
    sint16 (*active_close)(void);

   /* The host's system time */
    uint32 (*time)(void);
} TRANSPORT_MAPPING;
```

The application is expected to perform basic initialization of the network or other media. Once that is completed, the agent may perform the following operations:

init()	Initialize the transport specific features required by the agent. Included are the IP address, maximum message size, and host name. If any of these is defined and does not conflict with the transport layer, they can remain the same.
passive_open()	Tell the transport that the agent is ready to receive data.
passive_read()	Get available data from the transport.
passive_write()	Transmit potential responses to ***passive_read()*** operations.
passive_close()	Tell the transport that the agent will no longer receive data.
active_open()	Tell the transport to create a data channel to a particular host for sending traps. Note that the `rhost` field is one of the trap hosts defined by the application.
active_write()	Transmit a message to the host to which an ***active_open()*** was performed.
active_read()	Receive data on the trap channel. This will not occur with SNMPv1 and v2c. However, SNMPv3 has the provision that an agent may have to authenticate itself to a management station. Version 3 trap messages are not supported at this time.
active_close()	Close the data channel for writing traps.
time()	Get the system time in tenths of a second.

Each of the above operations returns a signed 16-bit value, except ***time()*** which returns the current time as a 32-bit value. For ***passive_open(), passive_close(), active_open()***, and ***active_close()*** the return value should be >= 0 unless an error occurs. For ***passive_read(), passive_write(), active_read()***, and ***active_write()*** functions the return value should represent the number of bytes transmitted or received. Note that the agent cannot internally handle an error value when performing ***passive_open()***. Essentially, the agent is useless without its passive functions.

When the ***ussSNMPAgentTrap()*** function is called by the application or by the agent, the agent will actually iterate through each ***active_xxx()*** function for each trap host.

For example implementations, see the following:

snmpsrc\tm_bsd.c	smxNS BSD socket interface (smxNS, UNIX, and Windows)
snmpsrc\tm_dpi.c	smxNS DPI interface

Exercising the Agent

An SNMP agent traditionally services queries from an SNMP Manager, which is implemented by software such as HP OpenView. There are other freely available software packages that can perform SNMP Manager operations. This section discusses the use of Net-SNMP. The Net-SNMP package is currently available from http://net-snmp.sourceforge.net/, and binaries are available for a number of platforms, including Microsoft Windows.

The Net-SNMP package provides command line utilities that can perform operations on an SNMP Agent using SNMP Version 1, Version 2 or Version 3. Here are some example commands that demonstrate these functions. These examples use the default user configuration in usm.c.

Example: Dump the entire MIB tree using SNMP Version 1

```
C:>snmpwalk –c public –v 1 192.168.11.100
```

Here the community name is given as “public”, the version is specified as “1” and the smxNS SNMP agent is running on a system that has the IP address 192.168.11.100.

Example: Dump the entire MIB tree using SNMP Version 2C

```
C:>snmpwalk –c public –v 2c 192.168.11.100
```

Example: Dump the entire MIB tree using SNMP Version 3

```
C:>snmpwalk –l noAuthNoPriv –n public –u initial –v 3 192.168.11.100
```

Example: Dump the entire MIB tree using SNMP Version 3 with authentication. Note that Net-SNMP will only dump the MIB-II tree unless otherwise requested. This example command specifies that the walk start at .1 so that all MIBs are included.

```
C:>snmpwalk –a MD5 –A secretpassword –l authNoPriv –n admin –u
admin-md5 –v 3 192.168.11.100 .1
```

Example: Display the number of SNMP traps sent using SNMP Version 3 with MD5 authentication

```
C:>snmpget –a MD5 secretpassword –l authNoPriv –n admin –u admin-md5
–v 3 192.168.11.100 snmpOutTraps.0
```

Example: Display the number of SNMP traps sent using SNMP Version 3 with MD5 authentication and DES privacy

```
C:>snmpget –a MD5 –A secretpassword –l authPriv –n admin –u admin-md5
 –x DES –X mylittlesecret –v 3 192.168.11.100 snmpOutTraps.0
```

Example: Display the number of SNMP traps sent using SNMP Version 3 with MD5 authentication and AES privacy

```
C:>snmpget –a MD5 –A secretpassword –l authPriv –n admin –u admin-md5
 –x AES –X mylittlesecret –v 3 192.168.11.100 snmpOutTraps.0
```

Example: Display the number of SNMP traps sent using SNMP Version 3 with SHA authentication

```
C:>snmpget –a SHA –A mylittlesecret –l authNoPriv –n admin –u admin-sha
 –v 3 192.168.11.100 snmpOutTraps.0
```

Example: Display the number of SNMP traps sent using SNMP Version 3 with SHA-256 authentication and AES-256 privacy

```
C:>snmpget –a SHA-256 –A mylittlesecret –x AES256 –x secretpassword –l
authPriv –n admin –u admin-sha –v 3 192.168.11.100 snmpOutTraps.0
```

10. Web Server

Web Server Overview

The smxNS™ Web Server provides an HTML server framework with default modules, handlers, a server configuration file, and the **nsbldpg** utility to compile HTML. It also includes CGI system support routines and the USMETA programming interface. The developer does not have to create their own Web Server API, and the Web Server is customizable.

The smxNS Web Server supports any MIME file type that can be manipulated or displayed by your web browser. This includes audio and Java. The MIME types determine how the browser processes the information.

All source code discussed in this chapter is supplied with the smxNS Web Server unless stated otherwise.

The smxNS Web Server has a modular design, and can be easily modified to suit your application. Because existing web technology is page-oriented rather than object-oriented, full pages transfer from the server to the client. This limits the speed that data can be updated on the browser.

These are the general steps for creating and inserting web pages into the embedded Web Server:

1. Design and prototype your website using a standard web design tool (see *Recommended Reading* in Chapter 1).
2. Test your prototype HTML on any standard web server.
3. Move your prototype to the development system.
4. Change CGI programs to CGI functions (see *CGI Function Programming Interface* later in this chapter).
5. Configure the Web Server to work with your network by modifying the configuration file (see *Server Configuration File* later in this chapter).
6. Process your web pages through the **nsbldpg** utility to obtain a C file that is compiled into the embedded format (see *Using nsbldpg* later in this chapter).
7. Compile your application.
8. Test.

Though the smxNS Web Server is designed to be user-customizable, it probably will not need customization. If you do want to customize, design information and guidelines for modifications are included in this document.

Web Server Requirements

System Requirements:

For a typical Web Server configuration, a minimum of 6K RAM (data and stack), and 30K ROM. Since the Web Server is modular these sizes may vary depending on the application, processor, and compiler.

NOTE: The Web Server uses the program stack to hold temporary data, so make sure there is at least a 5K stack in your application.

Tools required to build the Web Server:

smxNS Web Server source, a compiler/linker for your target platform, and an editor.

Optional Tools:

A test Web Server for page design.

You can also use a web page design tool. Be sure that your tool produces only HTML without propriety extensions. Microsoft FrontPage contains proprietary extensions and will not work with the Web Server.

Example Web Server

NSDEMO is provided as a sample Web Server. Some of the terms listed below might be new (for definitions, see *Terminology* in Appendix A). They will be discussed throughout the manual. The example is placed here to show the powerful features available in the Web Server.

There are six web pages in the sample smxNS Web Server, NSDEMO. The main page, and the first two links demonstrate static pages with formatted text and graphics. The "Sample Form" page presents and accepts a web form. When the form is submitted, a sample CGI function is invoked to demonstrate the interpretation of the information submitted in the form.

Building the Example Web Server for Your Target

Edit the **buildpg.cfg** file, found in the **websrc** directory. The following lines might need to be modified to match your target configuration:

```
# Change ServerAdmin to be the email address of someone who
#  administers the target
ServerAdmin                   admin@yourcompany.com

# Change ServerName to the name associated with the IP
#  address of your target
ServerName            Target.yourcompany.com
```

These configuration variables are not used by the Web Server or test programs, but are available for use in your applications.

You may want to familiarize yourself with the other configuration files in the Web Server. More information on these files is given later in this chapter. New pages are added to the server by

specifying the pages in the file **pages.cfg**. If you want to access a variable via a META command, those variables are specified in the file **vartable.cfg**.

The NSDEMO example application includes code to launch the web server, so building the project that contains this file will create an example web server image. Note that when building with the CodeWarrior compiler, the nsbldpg utility must be invoked by hand prior to building the project. See the section “Using nsbldpg” for more detail.

Connecting to the Example Web Server

To connect to your Web Server from a browser such as Netscape Navigator or Internet Explorer, enter the following in the open dialog box:

`http://`*`xxx.xxx.xxx.xxx`*

Where *`xxx.xxx.xxx.xxx`* is the IP address of the target system running the Web Server.

Adding Web Pages Using a File System

The Web Server always includes a set of default web pages that are stored in ROM alongside the code for the web server itself. Some applications may find it useful to augment the set of web pages with information stored in a local file system. These pages could replace default web pages stored in ROM, or they could be additional pages not previously defined.

The Web Server has been set up to check the local file system if HTTPS_USE_LOCAL_FS is set to 1 in XNS\include\https.h. This support is designed for use with the smxNS file system, but it can be adapted to other non-volatile file systems. The capabilities needed are

- Check if a file exists
- Get file size
- Read from the file in chunks (typical size 256 bytes)

When the Web Server is configured for use with smxFS (FILE_SUPPORT set to 1), the path to a file is constructed from the "DocRoot" string concatenated with the path in the request. If no matching file is found at that location, the ROM file system is searched.

The DocRoot string is defined in DEMO\WEBPAGE\buildpg.cfg and has the identifier DocumentRoot

```
#ScriptAlias         /cgi-bin/   /
DirectoryIndex       index.html
Readme               ReadMe
DocumentRoot        A:\\htdocs\\
```

Using the Web Server

User Server Functions

These functions are described in this section:

Bwrite()	Performs a buffered write to the network.
doreq()	Processes incoming HTTP request.
GetEntry()	Finds and returns the *ENTRY* structure used to access the web page.
httpinit()	Sets up listening Web Server socket.
HTTPservinit()	Initalizes the Web Server and allocates space for all the structures.
httpterm()	Shuts down Web Server.
Neof()	Tests for the EOF indicator for the network stream.
waitreq()	Waits for incoming HTTP request.

Bwrite()

Performs a buffered write to the network.

```
int Bwrite(struct SERV_REC *recp,u8 *buf,u32 len)
```

recp a pointer to the request structure

buf a pointer to the output buffer

len the length of the buffer

Bwrite() writes out the buffer to the network. The output is buffered to minimize network traffic. To flush the buffer, use `NULL` for *buf*, or *len* of zero.

Return Value

<0	Error
0 or >0	Success

Example

```
Rslt = Bwrite(reqp,buf,len);     /* write buffer */
Rslt = Bwrite(reqp,NULL,0);      /* flush buffer */
```

doreq()

Processes an HTTP request.

```
int doreq(struct request_rec *reqp)
```

The ***doreq()*** function processes an incoming HTTP request. There is more information on how the request is qualified, broken down and the response is constructed in the "HTTP Server Request Structure" section that follows. Note that each incoming HTTP message that is processed is logged with the MODlog() function which by default will list the request line, associated file that was referenced and the return code for the request processing.

Return Value

0	request processed OK
<0	error in processing request

Example

```
while (web_server_enabled)
{
   reqp = waitreq(servp);
   if (reqp)
      doreq(reqp);
}
```

GetEntry()

Finds and returns the *ENTRY* structure if the web page is found. The *ENTRY* structure is used to access the web page.

```
ENTRY *GetEntry(REQUEST_REQ *reqp,const char *file,const char *path)
```

reqp	a pointer to the request structure
file	the name of the file, i.e., **index.html**
path	the absolute path after translation

The ***GetEntry()*** function searches the directory specified by *path* for the page *file*.. If the directory or file doesn't exist, a NULL is returned.

This is the ENTRY structure:

```
struct entry {
      const char  *name;
      const char  *path;
      s16         type;
      const char  *mime;
      char        *encoding;
      char        *lang;
      void        *offset;
      size_t      clen;
      size_t      ulen;
      u32         hits;
      ENTRYACCESS *access;
```

```
}
typedef struct entry ENTRY;
```

Return Value

Pointer to `ENTRY` structure if found

`NULL` if not found

Example

```
ENTRY *ep = GetEntry(reqp, "index.html",NULL);
```

httpinit()

Sets up listening Web Server socket.

```
struct server_rec *httpinit(struct server_rec *mysrv)
```

This function creates a socket for the Web Server to listen on. The port number is given in mysrv->port, which typically is 80. If the SSL library is part of the project and enabled (#define CSL_USSL 1), an additional socket will be set up listening on the HTTPS port (443). In order to only run the HTTPS server, set myserv->port to 0 before calling httpinit().

Return Value

Non-NULL Initialization OK, return value is pointer to server request structure

NULL Error

Example

```
extern SERV_REC server;

main()
{
   SERV_REC *servp;
   servp = &server;
   memset(servp, 0, sizeof(SERV_REC));
   memcpy(servp, &romserver, sizeof(SERV_REC));
   HTTPservinit(servp);
   httpinit(servp);
   …
}
```

HTTPservinit()

Initalizes the Web Server and allocates space for all the structures.

```
struct SERV_REC *HTTPservinit(struct SERV_REC *servp)
```

servp the server information for the Web Server

Use the ***HTTPservinit()*** function to initialize server information such as port or IP. The function is called only once per server.

Return Value

`struct SERV_REC` Filled-out server information

httpterm()

Shuts down Web Server.

```
int httpterm(struct SERV_REC *servp)
```

servp the server information for the Web Server

Use the ***httpterm()*** function to shut down the Web Server.

Return Value

0 Always returns 0

Neof()

Tests for the EOF indicator for the network stream.

```
int Neof(int stream)
```

stream the network file descriptor

Neof() tests the end-of-file indicator for the network stream pointed to by *stream*, returning non-zero if it is set.

Return Value:

0 More data available

!0 End of data

waitreq()

Waits for incoming HTTP request

```
struct request_rec *waitreq(struct server_rec *servp)
```

servp the server information for the Web Server

Use the ***waitreq()*** function to wait for the next incoming HTTP request so that it can be processed by the ***doreq()*** function. The function will time out after 1 second and can be called repeatedly.

Return Value:

Non-NULL pointer to updated request structure

NULL Error or timeout

HTTP Server Request Structure

The structure of the HTTP server is very modular, so modules can be added and removed at any time. This allows for additions of new features and control of code size without extensive changes.

The request structure is the heart of the server. The request structure is passed through a sequence of functions which process the request. By having a request filter through different modules, the processing of that request can be tailored to each application. It also allows for user-written processing without affecting other parts of the HTTP server, which reduces debugging.

The processing of the request structure occurs in the ***doreq()*** function.

```
int doreq (REQ_STRUCT *reqp)
```

reqp a pointer to the request structure

The pseudocode for ***doreq()*** is:

```
request processing
translate paths
check the URL
check the MIME type
check access
get user ID
authorize the user
handle the request
log the request
```

See also: *Request Structure,* later in this document

The following figure shows the process that each request to the embedded web server goes through.

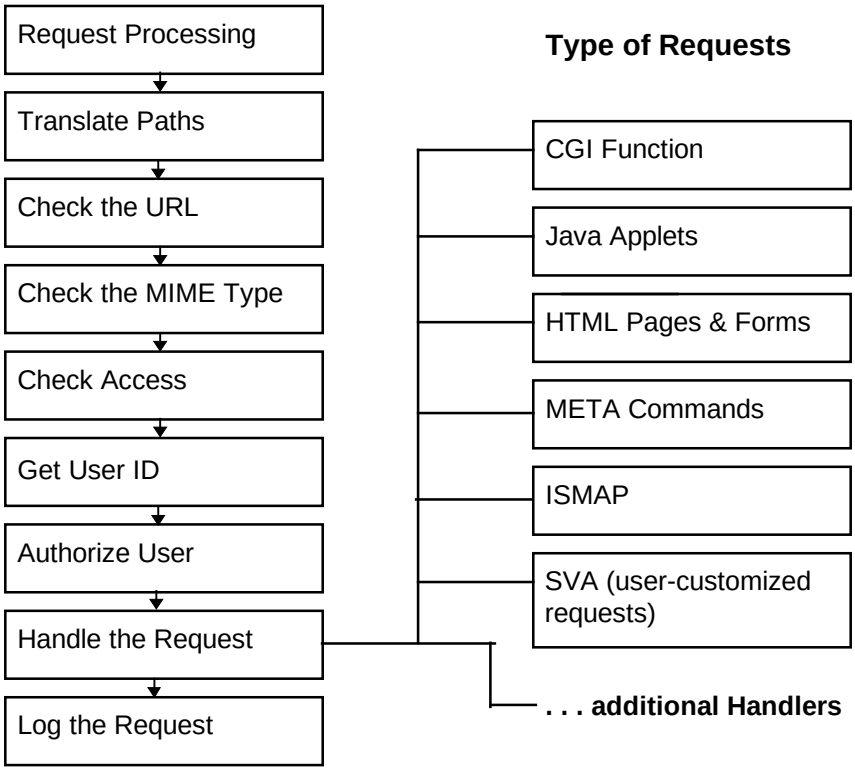


Figure 10-1: Process for Request to the Embedded Web Server

Return Value

<0 Error

Modules and Handlers

The structure of the HTTP server is very modular, so modules can be added and removed at any time. This allows for additions of new features and control of code size without extensive changes. New plug-in modules and increased functionality will be added in the future.

All data is passed through the modules by the request structure. The Web Server provides a framework and default modules for your use, and is designed so the user can customize it. To customize the modules, you must modify or replace the existing modules, using the existing modules as templates.

Each module has a function and modifies only certain parts of the request structure. Only the ***MODtranslate()*** and ***MODchkloc()*** functions are required; all others are optional. The module functions are described in alphabetical order, but are used in this sequence:

MODtranslate()	Parses and translates the URL.
MODchktype()	Determines the type and encoding of the document.
MODchkloc()	Checks for the existence of the file.
MODchkaccess()	Checks access privileges of the document.
MODgetuser()	Performs user authorization.
MODchkauth()	Finds the user in a database or file, and does the final authorization.
MODlog()	Logs errors and access.

Once the request has been processed by all the modules, the final display is the responsibility of the handler function. Each type of page has an associated handler. Each handler processes the page and sends the output to the browser. You can also add your own specialized handlers if needed for customization.

When the default web page type is set to 'text' (in buildpg.cfg), only the text handler is necessary. Additional handlers enhance the Web Server by allowing it to handle different page types.

These typical handlers are included with the smxNS Web Server:

`HNDtext`	Handles the standard HTML pages and text.
`HNDcgi`	Sets up the CGI environment and calls the function.
`HNDasis`	Sends the file to the browser without any processing.
`HNDmeta`	Handles server-side HTML parsing.
`HNDussnmp`	Comes with the U S Software SNMP package.

Module Function Descriptions

MODchkaccess()

Gets access privileges of the document.

```
int  MODchkaccess(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

This optional function checks the group flags or a directory file to determine the access permissions (security) for this page. Access parameters and page permissions are defined in access and page configuration files **access.cfg** and **pages.cfg**. This module sets the access group flag using the information specified in the access configuration file. ***MODchkauth()*** must be written so that the correct username/password returns a flag to match this access group flag.

The default ***MODchkaccess()*** module sets up two types of access checking:

None No checking done (anyone can access)

Group Checks a group flag associated with a user

The developer may implement other forms of access checking by modifying or replacing ***MODchkaccess().***

See also:*Request Structure*, in this document.
Using nsbldpg and *Page Configuration File*, in the *smxNS Web Server User's Guide*.

Return Value

< 0 Error

Otherwise modifies the structure.

Example

This is a pseudocode example for the authentication procecure:

```
MODchkaccess()
/* Checks access restiction of a given web page */
Check request structure for page protection
if (not protected) return 0
if protected
  initialize access information in request structure
  /* specifically, set access group flag */
  return 0
```

MODchkauth()

User-implemented routine to verify user authentication information.

```
int  MODchkauth(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

MODchkauth() is an optional routine that checks the authentication parameters obtained by ***MODgetuser()*** against a user-defined lookup. The default routine supplied with IAP sets the group to match the one specified in the access configuration file, if a preset username and password are entered. This routine must be modified by the developer to implement a site-specific lookup mechanism.

MODchkauth() does two types of access checking:

None	No checking done (anyone can access)
Group	Checks that the individual is within the group

If the developer has set up alternate checking methods in ***MODchkaccess()***, they must be implemented here.

See also:*Request Structure*, in this document.
Using nsbldpg and *Page Configuration File*, in the *smxNS Web Server User's Guide*.

Return Value

< 0	Error

Otherwise modifies the structure.

Example

MODchkauth()

```
/* Largely user-defined routine to authenticate user info */
if (no access restriction) return 0
match username/password to user-defined lookup
/* Default routine has a hard-coded username and password.
   When these are matched, a hard-coded group flag is
   returned. This group flag matches the one in the access
   configuration file, which was read into the request
   structure in MODchkaccess(). */
if (no match) return 401
if (match) return 0
```

MODchkloc()

Checks for the existence of the file.

```
int  MODchkloc(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

This required module finds the document and sets up a pointer to an embedded structure. If the page is not found, a result of `404` (not found) is returned to the requesting host.

If smxFS is included in the system, then this function looks for the file in the file system first, and then searches the files in ROM. The search starts at the location specified by DocRoot.

See also:*Request Structure*, in this document

Return Value

< 0 Error

Otherwise modifies the structure.

Example

See the file **modchklo.c** in your source code.

MODchktype()

Determines the type and encoding of the document.

```
int  MODchktype(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

This optional function checks the embedded type flags or the extension to determine the correct handler. This routine is appropriate when there is a file system in your embedded target.

See also:*Request Structure*, in this document

Return Value

< 0 Error

Otherwise modifies the structure.

Example

See the file **modchkty.c** in your source code.

MODgetuser()

Performs user authorization.

```
int  MODgetuser(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

MODgetuser() is an optional routine that gets authentication information from an end user. The routine extracts the username and password (commonly entered in a pop-up dialog from a browser) from the HTTP headers. This information is stored in the request structure and subsequently processed by ***MODchkauth()***. This routine decodes authentication information using either the basic or digest authentication schemes. Support for any other authentication scheme must be added by the developer.

See also:*Request Structure*, in this document
RFC 2069 and chapter 11 of RFC 2068

Return Value

< 0 Error

Otherwise modifies the structure.

Example

MODgetuser()

```
/* Checks user authorization information */
if (no access restriction) return 0
if (no “Authorization” in HTTP header)
  add “WWW-Authenticate” to HTTP header
  return 401 (Unauthorized)
  /* A browser receiving “WWW-Authenticate” will commonly
     pop up a username/password dialog. Entered parameters
     are sent to server as new request with “Authorization”
     in HTTP header. */

if (“Authorization” in header)
  if (not basic or digest authentication) return 401
  decode username and password from HTTP headers
  store username and password in request structure
  return 0
```

MODlog()

Logs errors and requests.

```
int  MODlog(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

MODlog() is an optional function that must be impelemented by the developer. This routine could log all requests and errors to a buffer, to a monitor, or to a file if a file system is present.

See also:*Request Structure*, in this document

Return Value

< 0 Error

Otherwise modifies the structure.

Example

See the file **httputil.c** in your source code.

MODtranslate()

Parses and translates the URL.

```
int  MODtranslate(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

MODtranslate() is a required module that parses the URL and translates its contents to a form usable by the Web Server. The path, file, and query information are parsed from the URL, and stored in the URI structure within the request structure. This information is used in the handler modules to take the appropriate action, such as displaying a page or executing a CGI function. This module supports HTML and CGI translation.

See also:*Request Structure*, in this document

Return Value

< 0 Error

Otherwise modifies the structure.

Example

See the file **modtrans.c** in your source code.

Request Structure

The request structure is the heart of the server. As an HTTP request is filtered through the modules, the request structure is filled in.

Since the structure is broken into stages, the user can customize each of the modules with little impact on the rest of the code. This also allows for future enhancements to be added easily.

The request structure is defined in the include file, **https.h**. An example of the `request_rec` structure is provided below:

```
struct request_rec {
    int           rslt;      /* result status */
    SERV_REC      *servp;    /* ptr to server rec */
    int           reqfd;     /* req sock descriptor */
    char          *ptr;      /* ptr for strng manip */
    uint          blen;      /* buf len left to read*/
    int           slen;      /* sz of sockadd struct*/
    struct sockaddr saddr;   /* sock addr structure */
    uint          flags;     /* keepalive and other flags */
    int           protonum;  /* protocol number */
    char          *protover; /* protocol version */
    s16           type;      /* type of HTTP req */
    const char    *method;   /* request method */
    s16           hostport;  /* listen port */
    char          *reqline;  /* request line */
    char          *status;   /* ptr to status line */
    char          *scheme;   /* GET, POST, (unused) */
    char          *hostname; /* where from */
    URI           uri;       /* text info */
    s16           headcnt;   /* num of HTTP headers*/
    struct headers *headers; /* HTTP headers */
    int            rplycnt;   /* num to HTTP reply */
    struct headers *rplyheads;/* reply headers */
    u8            *body;     /* ptr to body of POST */
    u32            bodylen;   /* how big? */
    struct entry  *fileinfo; /* after page is found,
                                ptr to the entry */
    char           *mime;    /* mime type */
    char           *encoding;/* the encoding */
    char           *lang;    /* the language */
    char           *accepth;
    char           *connecth;
    char           *from;

    struct cookie  *cookie;  /* cookie info */
    int           (*handler)(struct request_rec *req);
    ACCESS         *access;  /* access structure */
    u32            ldat;     /* undefined data */
    void           *data1;   /* now undefined ptr */
    void           *data2;   /* another undef ptr */
    char           *buff;    /* gen purpose buffer */
};
```

Using nsbldpg

The **nsbldpg** utility builds the web pages from your configuration files. To do this, it reads these files in this order:

- The server configuration file, named **buildpg.cfg**
- The MIME types file, named **mime.typ**
- The page configuration file, named **pages.cfg**
- The variable configuration file, named **vartable.cfg**

nsbldpg then takes the pages and turns them into C code, generating:

htpgtbl.c headers and tables, plus the server configuration and pages in binary format

htpgtbl.dat an included C file that contains source data for the web pages

These files are then compiled into your application.

When using the IAR compiler, the nsbldpg program is automatically invoked using the "Custom build" feature. Building the project will include the appropriate steps for building the generated C files.

When using the CodeWarrior compiler, the nsbldpg program must be invoked from the command line prior to building the NSDEMO project. To do this, invoke the utility and specify the buildpg.cfg configuration file, as in the following example:

```
C:\SMX\APP\DEMO\WEBPAGE>..\..\bin\nsbldpg buildpg.cfg
```

If the path to the configuration file contains spaces, use quotes around the path, for example:

```
C:\SMX\BIN> nsbldpg "c:\work\test area\smx\app\demo\webpage\buildpg.cfg"
```

The nsbldpg utility will try to locate the other configuration files (mime.typ, etc.) using the same path as that specified for buildpg.cfg. For convenience, the nsbldpg utility can be moved to a location that is in the executable search path.

Server Configuration File

The smxNS Web server's configuration is similar to the NCSA and Apache* web servers. **nsbldpg** uses the configuration file to build your web pages. There are five different areas of the server configuration, which can be seen in the example file on the next page:

- Other configuration files
- Application system information
- Server information
- Directory and file system information
- MIME information

This is an example of a typical **buildpg.cfg** file:

```
# This configuration file is read by the nsbldpg utility

# other configuration files
BuildDocRoot   .\
PageConfig     pages.cfg
VarConfig      vartable.cfg
TypesConfig    mime.typ
AccessConfig   access.cfg

# application system information
Processor      68EN302
HWdate         3 April 1951
HWversion      Release 35.1
HWconfig       WOM (Write Only Memory)
SWdate         11 Aug 1955
SWversion      1309.7.32
SWconfig       swodniW ultra light
TotalMem       32
SysMem         25
FreeMem        7

# server information
BindAddress    206.29.173.23
DefaultType    text/html
Port           80
ServerAdmin    admin@yourserver.company.com
ServerName     yourserver.company.com

# directory and file system information
Alias          /pages/     /
Alias          /other/     /
DirectoryIndex index.html
Readme         ReadMe

# mime information
AddEncoding    x-zip zip
AddEncoding    x-gzip      gz
AddType        application/x-us-snmp   smp
AddType        application/x-us-prog   uso
AddType        application/x-us-include      usi
```

Other Configuration Files

These variables provide information on where needed files are located. These files are described in detail later in this chapter.

Table 10-1: Other Configuration Files

Value	Description	Example
PageConfig	The name of the page configuration file. See also: *Page Configuration File*, in this chapter.	**pages.cfg**
VarConfig	The name of the variable configuration page. Each entry in the file has the format of: *Searchname, type, size, varname* An example is: VAR1, WEB_SHORT, sizeof(variable1), variable1 See also: *Variable Configuration File*, in this chapter.	**vartable.cfg**
TypesConfig	The name of the file that contains the file extension to MIME type mapping. See also: *MIME Information*, in this chapter.	**mime.typ**
AccessConfig	The name of the access configuration file. See also: *Access Configuration File*, in this chapter.	**access.cfg**

Application System Information

Application system information contains values that define more about the embedded system. The values are returned to the user when a META command is embedded into the HTML. These values can also be filled in at initialization time by the application. The values must be a string or a number, as specified in the following table, but they are not case-sensitive and can be in any format.

Table 10-2: Application System Information Variables

Value	Description	Example
BindAddress	Binds the listen connection to this address (eight 16-bit hex numbers).	0000:0000:0000:0000:0000:0000:C0A8:0101 (same as 192.168.1.1)
Port	The listen port.	80
ServerAdmin	The server administrator's e-mail address.	admin@*yourserver.company*.com
ServerName	The host name of the HTTP server.	*yourserver.company*.com
access_log	There are two different formats, depending on the logging method: • E-mail address -- the log is stored in RAM until it is mailed to this address. • File name -- the log information is saved to a file.	admin@*yourserver.company*.com

Server Information

These variables set the server and network environment.

Table 10-3: Server Information Variables

Value	Description	Example
BuildDocRoot	Defines the path used by **nsbldpg** to preprocess the pages.	**./pages**
DocRoot	On File System this would be the root where the search would start.	**C:/mypages**
DefaultType	If the system does not know what type a file is when handling a message, it will use this type.	**text/html**
Readme	Default name in directory for more information.	**ReadMe**
DirectoryIndex	Default file when no file is specified.	**index.html**
Alias	Changes the URL path, for instance, from /here/file to /there/file (the physical path would be C:/mypages/there/file).	**/here/ /there/**
ScriptAlias	Remaps the URL to a physical directory, and notifies the server that the file being accessed is code.	**/cgi-bin/ /**
ErrorAlias	If an error occurs, the output to the browser is changed from the standard error to this new page.	**404 notfound.html**

Directory and File System Information

These variables provide information on where needed files are located.

Table 10-4: Directory and File System Information Variables

Value	Type	Description	Example
Processor	string	Defines the processor type.	68EN302
Hwdate	string	Defines the hardware build data.	3 April 1951
Hwversion	string	Defines the hardware version.	Release 35.1
Hwconfig	string	Contains any special hardware configuration information.	WOM (Write Only Memory)
Swdate	string	Defines the software build date.	11 Aug 1955
Swversion	string	Defines the software version.	1309.7.32.8
Swconfig	string	Contains any special software configuration information.	swodniW ultra light
TotalMem	number	The total size of memory, in kilobytes.	32
SysMem	number	The amount of memory used by the system. Because the application defines what ‘system’ is, this could be anything.	25
FreeMem	number	The amount of free memory, in kilobytes.	7

MIME Information

MIME file types are defined by suffix (extension), and the MIME type controls how the server or browser will treat the defined files:

- If the file is server-specific, the MIME type tells the server how to handle it.
- If it is a browser file, the server adds the content type(s) to the header information for the browser's use.
- The MIME information also defines how to decode the data, and the **nsbldpg** program uses it for the encoding scheme.

There are two ways of defining MIME types for the smxNS Web Server: In the **mime.typ** file, or with the ***AddType*** command. The **mime.typ** file included in the smxNS Web Server distribution contains most of the standard definitions. The ***AddType*** command adds definitions to the server configuration file, allowing you to keep your **mime.typ** file general.

MIME Types File

This file lists the types of files the server is capable of sending. You can define multiple extensions for one file type.

This is an example portion of a **mime.typ** file:

```
# This is a comment. I love comments.

application/mac-binhex40  hqx
application/msword        doc
application/octet-stream  bin dms lha lzh exe class
application/pdf           pdf
application/postscript    ai eps ps
application/powerpoint    ppt
application/rtf           rtf
application/x-compress    Z
application/x-cpio        cpio
application/x-csh         csh
application/x-director    dcr dir dxr
application/x-gtar        gtar
application/x-gzip        gz
application/x-httpd-cgi   cgi
application/x-tar         tar
application/x-tcl         tcl
application/x-wais-source src
application/zip           zip
audio/basic         au snd
audio/mpeg          mpga mp2
audio/x-aiff        aif aiff aifc
audio/x-wav         wav
image/gif           gif
image/jpeg          jpeg jpg jpe
image/tiff          tiff tif
message/external-body
message/news
```

```
multipart/alternative
multipart/appledouble
multipart/digest
multipart/mixed
multipart/parallel
text/html           html htm
text/plain          txt
text/x-sgml         sgml sgm
video/mpeg          mpeg mpg mpe
video/quicktime           qt mov
video/x-msvideo           avi
```

AddType Command

Adds an additional MIME type to the Web Server.

`AddType application/`*`type`* *`extension`*

`type` the type of file

`extension` the extension for the file type

AddType helps define the file type when parsing. The new type goes into the server configuration file (not the **mime.typ** file) and functions like a command. Use ***AddType*** to add specialized MIME types to the Web Server rather than to your **mime.typ** file, thus keeping your **mime.typ** file general.

Example

```
AddType application/x-us-meta   usm
```

Page Configuration File

The page configuration file defines what local pages should be included in embedded web sites. Each page is defined by a line with a format of:

Buildname,webname,accessname,flags[,*maxsize, mime*]

Buildname — the name of the source file on your development system or the name of the CGI routine within the application program.

webname — the URL name.

accessname — a string used to associate authentication parameters with a web page. This variable is used by ***Modchkaccess()***. The authentication parameters associated with *accessname* are specified in **access.cfg**.

flags — define the processing this page needs. The flags are defined by 0xFFTT, where FF are bit flags and TT is a type number.

The flags are defined as:

0x01	RAM/ROM, if set move page to RAM and access it from RAM
0x02	If bit is set, the URL is executable (i.e., CGI function)
0x04	Undefined

The type is:

0,1	TEXT and HTML
2	CGI Function
3	ASIS, just send it out without parsing
4	USMETA, a HTML file with META commands
5	USSNMP, a UUUSMP file with META commands
255	QUIT, exit the server

maxsize — optional numeric variable used to reserve memory (a specified number of bytes) for the web page.

mime — rarely-used optional alpha variable that overrides the MIME definitions from the **mime.typ** file and ***AddType.***

This is a example of a typical **pages.cfg** file:

```
# format is
# build file name or link name
# page name
# accessname: string to define access parameters
# flags bits TYPE 0-7, ROM/RAM = 0x0100, DATA/LINK = 0x0200,
#  0,1 = TEXT
#  2 = CGI
#  3 = ASIS
#  4 = META
#  5 = USSNMP
#  255 = ABORT
# [maxsize] optional (0-9)
# [mime] optional (alpha)

# pages
index.htm,index.html,0,0
linktest.htm,linktest.htm,0,0
imagepag.htm,imagepag.htm,0,0
example3.htm,example3.htm,0,0
example4.htm,example4.htm,0,0
example5.htm,example5.htm,0,0
example6.htm,example6.htm,0,0
mailit.htm,mailit.htm,0,0

#images
example5.gif,example5.gif,0,3
image.jpg,image.jpg,0,3
lava_l.gif,lava_l.gif,0,3

#cgi functions
query_cgi,cgi-bin/query,0,0x0202
post_query_cgi,cgi-bin/post-query,0,0x0202
prntenv_cgi,cgi-bin/prntenv,0,0x0202
mailit_cgi,cgi-bin/mailit,0,0x0202
rainbow.cls,RainbowText.class,0,0x0003
```

Variable Configuration File

The variable configuration file defines the variables in the application that need to be accessed from the web pages. The file translates text strings into variables for access, and creates a table. The web pages can access the variables directly using META commands. You can use this to allow an end-user to access a variable within the application.

The format is:

web_name, web_type, sizeof(type), variable_name

web_name name used to access variable on web page

web_type one of:
WEB_INT, WEB_UINT, WEB_SHORT, WEB_USHORT, WEB_LONG, WEB_ULONG, WEB_CHAR, WEB_STRING

sizeof(type) sizeof(variable)

variable_name global variable in application

This is an example of a **vartable.cfg** file:

```
NAME,WEB_STRING,sizeof(name_var),name_var
pagecnt,WEB_LONG,sizeof(long),pagecnt_var
```

name_var and pagecnt_var are global variables in the application.

Example

```
NAME, WEB_STRING, sizeof(name_var), name_var
pagecnt, WEB_LONG, sizeof(long), pagecnt_var
```

Note that name_var and pagecnt_var are global variables in your application.

```
Your name is <!—-#ECHO FORMAT=”%s” VAR=”name”—-><BR>
```

Note that the variable name “name” is case-insensitive compared to the one in the configuration file.

```
There are <!--#ECHO FORMAT=“%d” VAR=”pagecnt”--> pages in this
document.
```

Note that the variable is declared as WEB_LONG in the configuration file -> type long, but is printed out in “%d” – int format. The format specified in the META Echo command supercedes the format specified in the variable configuration file. If the FORMAT is left out of the META Echo statement, then the one in the variable configuration file will be used.

Access Configuration File

The access configuration file defines parameters for page authentication. The file is typically named access.cfg and is located in the DEMO\WEBPAGE directory.

The format is:

name, check_type, auth_type, group, realm, [key, domain]

name	string used in pages.cfg to associate authentication parameters with a specific page. In the example which follows, "test_digest" and "test_basic" specify two different sets of authentication parameters. Using the access name string in the pages.cfg file will match a set of authentication parameters to a specific web page.
check_type	type of authorization: 0 = none, 1 = group, >1 other (not implemented)
auth_type	0 = basic, 1 = digest
group	an unsigned long flag which must match the flag returned by finduser(). The example below uses a flag of 0x7fffffff which is matched against the group set in finduser().
realm	defines protection space. In the following example, pages within the realm testing@smxrtos.com are protected. Once a request has been authenticated, all subsequent requests for pages in the same realm will be automatically authenticated.
key	a string used as the challenge key for digest authentication.
domain	the URL space to protect for digest authentication.

This is an example of an **access.cfg** file:

```
# This is an access file
# format for basic
#     name,checktype,authtype,groups,realm
#
# format for digest
#     name,checktype,authtype,groups,realm,key,domain
#
# string,int,int,long,string[,hexstring,string]

test_digest,1,1,0x7fffffff,testing@smxrtos.com,smxtest,smxrtos.com
test_basic,1,0,0x7fffffff,testing@smxrtos.com
```

The corresponding **pages.cfg** file would look like this:

```
# Basic authentication parameters test_basic required for access
# to web page ex0.htm:
ex0.htm,ex0.htm,test_basic,0
# Digest authentication parameters test_digest required for access
# to web page ex2.htm
ex2.htm,ex2.htm,test_digest,0
```

CGI Function Programming Interface

The heart of the interactive web is the Common Gateway Interface (CGI). The server needs to display different pages depending on the user's actions. CGI reads parameters from forms on the displayed web page to the server. The data is in the format of:

```
name1=value1, name2=value2
```

The smxNS Web Server supplies all needed support routines to manipulate CGI data. The HTTP server uses the standard CGI programming interface, but with a twist. The main difference is that the embedded HTTP server uses subroutines instead of programs.

ISMAP is supported via `argc` and `argv` passed into the CGI function. A mouse click would be passed in as `argv[1]` being x and `argv[2]` being y.

In UNIX the CGI programs are called like:

```
int main(int argc, char *argv[])
```

In the embedded world it would be:

```
int subname(int argc,char *argv[],REQ_STRUCT *reqp)
```

The CGI function can provide all of the components of the response to an incoming request, including the status code, response headers and the message body. Details on the format of an HTTP response are in Section 6 of RFC 2616.

There are a number of example CGI functions in XNS/websrc/CGI directory. The example in query.c is part of the smxNS Web Server demonstration. The query_get_cgi() function receives information submitted in a web form and stores it in global variables.

The code at the beginning of query_get_cgi() writes the HTTP status code, response headers and a CRLF delimiter that indicates the beginning of the message body for the response. This code can serve as a starting point for customization and as boilerplate for other CGI functions.

```
reqp->rslt = 200;
rplystatus(reqp);
reqp->rplycnt = addheader(reqp->rplycnt, reqp->rplyheads,
   "Content-type", "text/html");
reqp->rplycnt = addheaders(reqp->rplycnt, reqp->rplyheads,
   "Expires", "0");
rplyheaders(reqp, reqp->rplycnt, reqp->rplyheads);
PRINTF(reqp->, "\r\n");
```

The value set in reqp->rslt is sent as the HTTP status code when rplystatus() is called. The example code will generate the following

```
HTTP/1.1 200 OK
```

The calls to the addheader() function add HTTP headers to the response and the rplyheaders() function sends the headers. The example code will generate the following

```
Content-type: text/html
Expires: 0
```

To include more headers in the response, insert additional calls to addheader(). The first and second fields in the header are specified as the last two parameters in the addheader() call.

At the end of this sequence there is a call to PRINTF() to send the carriage-return line-feed characters that will mark the end of the response headers and the beginning of the message body.

The following sections includes descriptions of the CGI routines and the CGI system support routines.

System Support Routines

These routines are support routines for the application engineer to use for CGI functions such as exchanging information with the network. They are similar to standard CGI support routines, but tailored to the embedded environment.

These routines are described in this section:

findvar()	Searches the variable structure for a specified string.
getvar()	Searches the request structure for a variable.
Ngetenv()	Searches the environment structure for a specified string.
send_file()	Writes a file to the network.

findvar()

Searches the variable structure for a specified string.

```
VARENTRY *findvar(REQSTRUCT *reqp, char *name)
```

reqp a pointer to the request structure

name a pointer to the specified string

The ***findvar()*** function searches the variable structure for a string that matches the string pointed to by *name.* It is typically used for changing the variable structure. This allows *name* to be reassigned to a different pointer. This routine could be used to write a larger buffer for a pointer associated with the name.

See also:***getvar()***

Request Structure, in the *HTTP Server Request Structure* section.

Return Value

A pointer to the VARENTRY structure if found, NULL if not found.

Example

```
/* This program demonstrates the GET CGI routines */
/* the HTML is given a filename that is to be sent */

    typedef struct {
        char name[128];
        char val[128];
    } entry;

    static entry entries[10];

    int demo_cgi(int argc,char *argv[],REQUEST_REQ*reqp)

    {
          char *str, fname;
          int *pmaxetn;
          ENTRY *ep;
          VARENTRY *vp;
          reqp->rslt = 200;
          rplystatus(reqp);
          reqp->rplycnt = addheader(reqp->rplycnt, reqp->rplyheads,
             "Content-type", "text/html");
          reqp->rplycnt = addheader(reqp->rplycnt, reqp->rplyheads,
             "Expires", "0");
          rplyheaders(reqp->rplycnt, reqp->rplyheads);
          PRINTF(reqp, "\r\n");
          str = Ngetenv(reqp,"METHOD");  /* get the
                                            METHOD=XXXX */
          if(strccmp(str,"GET") != 0) {
                  /* compare str to "GET" case-insensitive */
             str = Ngetenv(reqp,"QUERY_STRING");
             if (str == NULL) {
                PRINTF();
                return 0;
             }
          } else if (strccmp() == 0) {
```

```
            char buff[8192];
            (reqp,buff,8192);
            str = buff;
         } else {
            PRINTF(reqp,"BAD METHOD");     /* bad method */
            return 0;
         }
         pmaxetn = (int*)getvar(reqp,"ENTRYSZ");
                              /* get a pointer to integer */
         for(x=0;cl[0] != '\0';x++) {
                        /* this section decodes the string
                              into an array for easy use */
            splitstr(entries[x].val,cl,'&');
                     /* get the whole "name=value" string */
            plustospace(entries[x].val);
                                 /* change any '+' to ' ' */
            unhex_str(entries[x].val);
                                    /* remove any nasties */
            splitstr(entries[x].name,entries[x].val,'=');
                /* split the entry into "name" and value" */
            if(x==*pmaxetn)            /* check if at max */
               break;
         }
      m=x;
      setvar(reqp,"THISENTRY",entries,0);
                     /* save the array to be used later */

                        /* usually the entries are in the
                           same order, but just in case */
      for(x=0;x<m;x++) {             /* loop through array */
         if(strcmp(entries[x].name,"SENDFILE") == 0){
            fname = entries[x].value;
            break;
         }
      }
      if(x==m) {
         PRINTF(reqp,"not found\n");
         return 0;
      }
      ep = GetEntry(reqp,fname,0);
      send_file(reqp, ep);
      vp = findvar(reqp, "THATVAR");

      if(vp == NULL) {
         PRINTF(reqp,"not found\n");
         return 0;
      }
      PRINTF(reqp,"name >%s, data pointer >%x\n",vp->name,
             vp->data);
      Bwrite(reqp,vp->data,vp->size);
      return 1;
   }
```

getvar()

Searches the request structure for a variable.

```
char * getvar(REQSTRUCT *reqp, char *name)
```

reqp a pointer to the request structure

name a pointer to the specified variable

The ***getvar()*** function searches the request structure for a variable that matches the variable pointed to by *name*. This function is used to access application variables from the CGI routine. The variable accessed is the same as if done from an HTML META command.

See also:***findvar()***
Request Structure, in the *HTTP Server Request Server* section

Return Value

Returns the pointer needed to access the variable specified by *name*, so the variable's value can be changed

Example

This is included in the example for ***findvar()***.

Ngetenv()

Searches the environment structure for a specified string.

```
const char *Ngetenv(struct request_rec *reqp, const char *str)
```

reqp a pointer to the request structure

str a pointer to the specified string

The ***Ngetenv()*** function searches the environment structure for a string that matches the string pointed to by *str*.

See also:*Request Structure*, in the *HTTP Server Request Structure* section

Return Value

A pointer to the value in the environment, or NULL if there is no match.

Example

This is included in the example for ***findvar()***.

send_file()

Writes a file to the network.

```
int send_file(REQSTRUCT *reqp, ENTRY *ep)
```

reqp a pointer to the request structure

ep pointer to the *ENTRY* structure, where *ENTRY* is a structure that contains a file or page description

The ***send_file()*** function writes the file in the *ENTRY *ep* to the network. This is a way to send out a file without processing it.

See also:***GetEntry()*** description, in this chapter, for a definition of the *ENTRY* structure
Request Structure, in the *HTTP Server Request Structure* section

Return Value

< 0 Error

0 or > 0 Success

Example

This is included in the example for ***findvar()***.

CGI Routines

These routines are described in this section:

escape_char()	Converts all 'nasty control characters' to '\x'.
hextochar()	Converts two hex values into an unsigned 8-bit value.
Nmakeword()	Parses a string and returns a pointer to the word that was matched.
plustospace()	Converts all '+' to spaces.
splitstr()	Parses a string.
subchar()	Substitutes one character for another in a string.
unhex_str()	Searches for %xx and terminates the string.

escape_char()

Converts all 'nasty control characters' to '\x'.

```
void escape_char(char *cmd)
```

cmd the string to convert

The ***escape_char()*** routine converts unwanted characters (which might blow up shells, be security holes, etc.) in the specified string to 'safe' characters. The 'nasty control characters' which are processed are:

```
& ; ` ' " | * ? ~~ < > ^ ( ) [ ] { } $ \ 0x0A
```

Return Value

None

Example

```
char *buf = "grep foo > x";
escape_char (buf);
  /* After execution of escape_char(),
     buf is "grep foo \> x". */
```

hextochar()

Converts two hex values into an unsigned 8-bit value.

```
char hextochar(char *what)
```

what the hexadecimal value to convert

The conversion is to characters or integers, depending on the hexadecimal number specified.

Return Value

The converted value.

Example

```
char *str=”AB”;
char num;
num = hextochar(str);      /* num = 0xab */
```

See the file **cgiutil.c** in your source code for another example.

Nmakeword()

Parses a string.

```
char * Nmakeword(struct request_rec *reqp, char stop, int *cl)
```

reqp pointer to request structure

stop character to stop at

cl length of string

Nmakeword() is like ***splitstr()*** but it returns a pointer to the word that was matched.

It receives a string one character at a time, and terminates the receive upon reception of the *stop* char or if there is an end-of-string or end-of-line. ***Nmakeword()*** returns a pointer to the *word*, and *reqp->ptr* is adjusted to point to the next character after the *stop* character.

See also:***splitstr()***

Return Value

A pointer to the word that was matched.

Example

See the file **cgiutil.c** in your source code for an example.

plustospace()

Converts all '+' to spaces

```
void plustospace(char *str)
```

str the string to convert

Return Value

None

Example

This is included in the examples for ***splitstr()***.

subchar()

substitutes one character for another in a string

```
void subchar(char *str, char oldchar, char newchar,)
```

str the string to modify

oldchar the character to be replaced

newchar the replacement character

Return Value

None

Example

plustospace() is implemented using ***subchar()***.

```
void plustospace(char *str) {
   subchar(str, '+', ' ');
}
```

splitstr()

Parses a string.

```
void splitstr(char *word, char *line, char stop)
```

word a pointer to buffer space

line the beginning of the string

stop the ending character

splitstr() parses the string pointed to by *line* until the *stop* char is matched or there is an end-of-string or end-of-line. ***splitstr()*** returns the contents of the buffer pointed to by *word*, and adjusts *line* to point to the next character after the *stop* character.

See also:***Nmakeword()***

Return Value

The contents of the buffer (the line up to the stop character) pointed to by *word*.

Example

This example determines whether to do GET or POST, and shows a GET routine and a POST routine. It includes ***splitstr()***, ***plustospace()***, and ***unhex_str()***.

```
#include httpd.h

extern int getcgi(int,char**,REQ_STRUCT*);
extern int postcgi(int,char**,REQ_STRUCT*);
#ifdef UNIX
int main(int argc,char *argv[])
#else
int cgiroutine(int argc,char *argv[], REQ_STRUCT *reqp)
#endif
{
   char *method = GETENV("REQUEST_METHOD");
   reqp->rslt = 200;
   rplystatus(reqp);
   reqp->rplycnt = addheader(reqp->rplycnt, reqp->rplyheads,
      "Content-type", "text/html");
   reqp->rplycnt = addheader(reqp->rplycnt, reqp->rplyheads,
      "Expires", "0");
   rplyheaders(reqp->rplycnt, reqp->rplyheads);
   PRINTF(reqp, "\r\n");
   if(strcmp(method,"GET”) == 0){
      return getcgi(argc,argv,reqp);
   }
   if(strcmp(method,"POST") == 0) {
      return postcgi(argc,argv,reqp)
   }
   return -1;                          /* bad request */
}

int getcgi(int argc,char* argv[],REQ_STRUCT *reqp);
{
   char *query;
   int m,x;
   query = GETENV("QUERY_STRING");
   if(query == NULL) {
```

```
        PRINTF(reqp,"No query information to decode.\n");
        EXIT(1);
    }

    for(x=0;query[0] != '\0';x++) {
        splitstr(entries[x].val,query,'&');
            /* get the whole name=value string */
        plustospace(entries[x].val);     /* convert '+' to ' ' */
        unhex_str(entries[x].val);
                            /* remove any nasty chars that might
                               blow up the system */
        splitstr(entries[x].name,entries[x].val,'=');
                                /* separate name from value */
    }

    m=x;
    PRINTF(reqp,"<H1>Query Results</H1>");
    PRINTF(reqp,"You submitted the following name/value
              pairs:<p>%c",10);
    PRINTF(reqp,"<ul>%c",10);
    for(x=0; x < m; x++)
        PRINTF(reqp,"<li> <code>%s = %s</code>%c",
           entries[x].name, entries[x].val,10);
    PRINTF(reqp,"</ul>%c",10);
    return 0;
}


int postcgi(int argc,char* argv[],REQ_STRUCT *reqp);
{
    char *body;
    int m,x,qlen;
    qlen = atoi(GETENV("CONTENT_LENGTH"));
                                    /* needed to buffer the input */
    body = getbody(reqp);
    for(x=0;!Neof(reqp);x++) {          /* read until no more */
        entries[x].val = Nmakeword(reqp,'&',&cl);
                    /* read input stream for full name=value */
        plustospace(entries[x].val);     /* convert '+' to ' ' */
        unhex_str(entries[x].val);
        entries[x].name = splitstr(entries[x].val,'=');
    }

    m=x;
    PRINTF(reqp,"<H1>Query Results</H1>");
    PRINTF(reqp,"You submitted the following name/value pairs:
        <p>%c",10);
    PRINTF(reqp,"<ul>%c",10);
    for(x=0; x <= m; x++)
        PRINTF(reqp,"<li> <code>%s = %s</code>%c",
           entries[x].name,entries[x].val,10);
    PRINTF(reqp,"</ul>%c",10);
    query = GETENV("QUERY_STRING");
    if(query == NULL) {
        PRINTF(reqp,"No query information to decode.\n");
        EXIT(1);
    }
```

```
   for(x=0;query[0] != '\0';x++) {
      splitstr(entries[x].val,query,'&');   /* get the whole
                                            name=value string */
      plustospace(entries[x].val);      /* convert '+' to ' ' */
      unhex_str(entries[x].val);    /* remove any nasty chars
                                 that might blow up the system */
      splitstr(entries[x].name,entries[x].val,'=');/* separate
                                            name from value */
   }

   m=x;
   PRINTF(reqp,"<H1>Query Results</H1>");
   PRINTF(reqp,"You submitted the following name/value
      pairs:<p>%c",10);
   PRINTF(reqp,"<ul>%c",10);
   for(x=0; x < m; x++)
      PRINTF(reqp,"<li> <code>%s = %s</code>%c",
         entries[x].name,entries[x].val,10);
   PRINTF(reqp,"</ul>%c",10);
   return 0;
}
```

unhex_str()

Unescapes %xx hex strings in URL.

```
void unhex_str(char *url)
```

url the URL to convert

The ***unhex_str()*** routine converts hex numbers to characters.

Return Value

None

Example

This is included in the example for ***splitstr()***.

CGI Environment Variables

When programming CGI, all the data about the world around you is passed by environment variables. Each environment variable has a different meaning.

Table 10-5: CGI Environment Variables

Variable	Description
SERVER_SOFTWARE	The name and version of the information server software answering the request (and running the gateway). Format: name/version
SERVER_NAME	The server's hostname, DNS alias, or IP address as it would appear in self-referencing URLs.
GATEWAY_INTERFACE	The revision of the CGI specification to which this server complies. Format: CGI/revision
SERVER_PROTOCOL	The name and revision of the information protocol this request came in with. Format: protocol/revision
SERVER_PORT	The port number to which the request was sent.
REQUEST_METHOD	The method with which the request was made. For HTTP, this is "GET", "HEAD", "POST", etc.

Table continued on next page.

Table 10-5: CGI Environment Variables (section 2 of 3)

PATH_INFO	The extra path information, as given by the client. In other words, scripts can be accessed by their virtual pathname, followed by extra information at the end of this path. The extra information is sent as PATH_INFO. If this information comes from a URL, the server should decode the information before it is passed to the CGI script.
PATH_TRANSLATED	The server provides a translated version of PATH_INFO, which takes the path and does any virtual-to-physical mapping to it.
SCRIPT_NAME	A virtual path to the script being executed, used for self-referencing URLs.
QUERY_STRING	The information which follows the ? in the URL which referenced this script. This is the query information, and it should not be decoded in any way. This variable should always be set when there is query information, regardless of command line decoding.
REMOTE_ADDR	The IP address of the remote host making the request.
AUTH_TYPE	If the server supports user authentication, and the script is protected, this is the protocol-specific authentication method used to validate the user.
REMOTE_USER	If the server supports user authentication, and the script is protected, this is the username they have authenticated as.

Table continued on next page.

Table 10-5: CGI Environment Variables (section 3 of 3)

REMOTE_IDENT	If the HTTP server supports RFC 931 identification, then this variable will be set to the remote user name retrieved from the server. Usage of this variable should be limited to logging only.
CONTENT_TYPE	For queries that have attached information, such as HTTP POST and PUT, this is the content type of the data.
CONTENT_LENGTH	The length of the content as given by the client.
HTTP_ACCEPT	The MIME types which the client will accept, as given by HTTP headers. Other protocols may need to get this information elsewhere. Commas as per the HTTP spec should separate each item in this list. Format: `type/subtype, type/subtype`
HTTP_USER_AGENT	The browser the client is using to send the request. General format: `software/version library/version`
DATE_GMT	The current date and time in Greenwich mean time.
DATE_LOCAL	The current date and time in the local time zone for the server.
DOCUMENT_NAME	The name of the file using this variable. Contains only the file name, not the location.
DOCUMENT_URI	The path to the file using this variable relative to the page root directory. Contains the directory location and the file name. For example: `/parsed_docs/myfile.shtml`
LAST_MODIFIED	The last modification date of the file using this variable.

USMETA Programming Interface

META commands are used to access predefined application system variables in the **vartable.cfg** file. They allow HTML access of the variables, which can be viewed while the application is running. You must define these variables and update them when necessary.

See also: *Variable Configuration File*, in this chapter

META commands are parsed by the server, and are stored as comments in the body of the HTML page. The commands have this format:

```
<!—-#command arg=″value″—->
```

Each command accepts different arguments. For example, this command includes a separate file within the page:

```
<!—-#include virtual+″../includes/header.txt″-—>
```

If the server cannot parse the command in the comment because of an error, it returns the unparsed comment to the browser.

The power of META commands is the ability to not only have access to the variable, but to format the variable.

For example, if you wanted to access an IP address, you can have it printed out in ether hex or decimal:

```
hex: <!—-#ECHO FORMAT=″%x″ VAR=″ipaddress″—->
dec: <!—-#ECHO FORMAT=″%d″ VAR=″ipaddress″—->
```

This is a command to print a string:

```
<!—-#ECHO FORMAT=″this is it %s″ VAR=″astring″—->
```

It would print out “this is a web page” if *`astring`* contains “`web page`”.

These HTML META tags are described in this section:

#echo	Prints a statement to the browser screen.
#exec	Runs a CGI function.
#include	Inserts the contents of a file.
#memory	Prints the memory size, in kilobytes.
#system	Prints information about the system.

#echo

Prints a statement to the browser screen.

The ***#echo*** command includes the value of one of the environment variables defined for CGI programs (see *CGI Environment Variables*) or uses SVA (Server Variable Access) to include one of the variables defined in the **vartable.cfg** file (see *Variable Configuration File*). By echoing a variable to the browser, the web page can dynamically update the page.

The only argument is *var*, whose value is the name of the variable you want to output.

Example 1

```
<!—-#echo var=”HTTP_USER_AGENT”—->
```

Example 2

```
<HTML>

<HEAD>

<TITLE> Meta Commands Examples </TITLE>

</HEAD><BODY>

This is an example of meta commands.

<!--#include file=”header.txt”-->  <!-- this would read the file
‘header.txt’ and send it out, then continue sending out this file-->

The number of widgets is <!--#echo var=”WIDGCNT”-->
  <!-- would look like The number of widgets is 5 -->

Total Memory is <!--#memory total-->
  <!-- Total Memory is 512K -->

</BODY>

</HTML>
```

#exec

Runs a CGI function.

The valid arguments are:

cgi runs the CGI function you specify and includes its output in the page.

The #exec META tag is useful when a web page should contain dynamically generated information that is best localized in a CGI function. For simple text insertions, #echo combined with server variables should be less complicated to implement.

The CGI function is implemented as discussed in the "CGI Function Programming Interface" section earlier in this document, and can process arguments. The server does not check to make sure your CGI program produces an output.

Example

```
<!—-#exec cgi="cgi-bin/fill_in"—->
```

#include

Inserts the contents of a file.

```
<!—-#include file="filename"—->
<!--#include virtual="path"—->
```

The ***#include*** command accepts either of the following arguments:

file gives a relative reference to the file you want to include. The path is relative to the directory containing the file that uses the ***#include*** command. You cannot use absolute paths with this argument. To keep your non-public directories secure, a page cannot use relative paths that traverse upward through the directory structure (that is, it cannot use paths that contain ../).

filename the filename in the physical file structure on the server machine

virtual gives the path to a file relative to the page root directory for the server. The double dash after the "!" is necessary.

path the file path as seen from the outside by those accessing the server

The ***#include*** command inserts the contents of the file you specify at the location of the ***#include*** command. The user must have read access to the file that gets included. If the file that is included has a file extension or location that causes it to be parsed by the server, that file can in turn include other files.

Make sure files you include contain only tags that are appropriate in the context of the files that include them. For example, don't use the <HTML>, <HEAD>, or <BODY> tags (or their end tags) in a file that will be included in another file that already contains these tags.

Examples

```
<!—-#include file="include.txt"—->
<!—-#include virtual="/doc/cust/include.txt"—->
```

See also:The second example for the ***#echo*** command.

#memory

Prints the memory size, in kilobytes.

The ***#memory*** command accepts these variables:

total	returns the total amount of memory in the system.
system	returns the amount of memory used by 'the system'. Because this is defined by the application, 'the system' is user-defined.
free	returns the amount of free memory.

This command returns information from the server configuration file's `TotalMem`, `SysMem`, or `FreeMem` field, where the application has earlier set these global variables.

See also: *Server Configuration File*, in this chapter

Examples

```
<!—-#memory total—->

<!—-#memory system—->

<!—-#memory free—->
```

See also:The second example for the ***#echo*** command.

#system

Prints information about the system.

The variables are stored in:

processor	returns the system processor string:	
	HWdate	hardware date
	HWversion	hardware version
	HWconfig	hardware configuration
	SWdate	software date
	SWversion	software version
	SWconfig	software configuration

Example

```
<!--#system HWdate-->
```

AJAX and jQuery

AJAX is a framework for creating interactive web pages by incorporating asynchronous requests from the web browser to the web server. This system was originally pieced together using JavaScript code and calls to the XMLHttpRequest() function. Use of this technique has been simplified over time by the introduction of JavaScript libraries, such as jQuery, which take care of cross browser compatibility and add a number of related interactive web page functions.

The key to making AJAX work, from the perspective of the web server, is to implement a function that interprets the XMLHttpRequest() function that is issued by the web browser. In the demonstration web page ajax.htm this comes together at the bottom of the file where there is a call

loadXMLDoc(url);

This, in turn, is defined earlier in the JavaScript in the file as a function specific to the type of browser that is in use. The result is that the browser will send an HTTP GET request of the form

GET /cgi-bin/updateAjax?q=1000

The value 'q=1000' here is an example of the value that the web browser is passing to the web server. In this case it means the "Blink Rate" slider is set to 1000 milliseconds.

On the web server side, the processing loop receives a GET request specifying a CGI function with a parameter. This is handled by the function mapped to "cgi-bin/updateAjax" in pages.cfg, which is updateAjax() defined in ajax.c. This function reads the passed parameter and updates a variable that controls the blink rate. updateAjax() sends a canned response back to the browser

<response><method>updateSlider</method><result>0</result></response>

to indicate that the request was handled successfully.

The jQuery example is similar. The HTML portion is implemented in jquery.htm, and uses the jQuery library's ajax function to issue a similar HTTP GET request

GET /cgi-bin/updatejQuery?q=591

The GET request is handled by the CGI function defined in jquery.c. The slider position is captured, and rather than send a canned response, the jQuery example sends back "live" data from the embedded system. A string of 5 values representing stack usage and network buffer usage is sent back as an ASCII string, for example

05 09 06 11 46

Back on the JavaScript side, the string is assigned to the variable "data" by the ajax function, and parsed into values that drive the progress bar UI elements that show percent stack usage.

In addition to being simpler to implement, using a popular JavaScript library may also allow you to load the library portion from a CDN server, reducing the amount of resources that need to be stored in your embedded system, and the amount of data needed to be transferred from the system. This is possible in systems that have access to the public internet, or which can host the JavaScript library elsewhere on the connected network. For reference, jQuery version 2.1.3 as a minified file is approximately 82K bytes.

From the project perspective, the list of what is built into the system image can also help illustrate the differences. In the AJAX demo, 7 files under the slider directory and ajax.htm are compiled into the htpgtbl.c "ROMed" file and the image also incorporates the ajax.c CGI function. The jQuery demo is implemented out of the JavaScript files jquery.min.js, jquery-ui.min.js, the jquery.htm HTML page and the jquery.c file that handles the GET request.

11. Device Drivers

Overview

A number of device drivers are already provided with smxNS for commonly used network controllers. These include Ethernet and serial connections. A list of supported controllers is provided in the file **driver.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 smxNS device drivers.

Support for interrupt handling is provided through smxBase. All you need to do is write the interrupt handler as a C function and pass it to **sb_ISRInstall()** as the interrupt handler for the device. smxNS provides several support functions which will assist with the interface between smxNS 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 a New Processor
- Error Codes
- Writing A Device Driver
- Character Drivers
- Block Drivers

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** and **support.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 */
struct MESSH *next;
u32 timems;
u32 target;
u16 mlen;               /* message length */
u8 netno;               /* network number */
u8 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 12-1.

Table 12-1: 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 nscfg.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: u8 ***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 smxNS. 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 */
    int netstat;
    NPTABLE *protoc[2];    /* link, driver protocol */
    ..
    u8 hwflags;            /* hardware level flags */
    MESS *bufbas;          /* input buffer base */
    MESS *bufbaso;         /* output buffer base */
    ..
    u32 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 12-2.

Table 12-2: 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 NPTABLES 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 and protoc[1] stores the device driver functions.
hwflags	Network controller hardware flags. This may be used if you need to get 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 only used with character drivers.
hw.goingc	Pointer to the function which transfers a byte from the link layer to the device driver. This is only used with character drivers.

Support Functions

The following smxBase macros and functions and other macros should be used when you are writing your device driver. They are intended to be used as an interface between smxNS and a driver. Use them within your driver code to separate device-dependent code from smxNS-dependent code.

Disable and Enable Interrupts

sb_INT_DISABLE() and **sb_INT_ENABLE()** are for disabling and enabling interrupts. This is done as follows:

```
sb_INT_DISABLE();
<< code that cannot be interrupted >>
sb_INT_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

sb_ISRInstall(irq, par, func, name) installs a new interrupt handler.

The interrupt handler is a C function you write within the device driver code.

When an interrupt occurs, the handler will be called with the instance as an argument. The instance is an index that can be mapped back to the smxNS data structure which defines the network interface to be used by the interrupt. This is described further in the *Data Structures* section in this chapter. sb_ISRInstall() automatically saves the old interrupt vector. Your device driver code should call sb_ISRInstall() from the initialization routine to install the interrupt handler. See the section on function ***init()*** later in this chapter for a specific example of its use.

Restore Interrupt Vector

sb_ISRRestore() restores the original interrupt vector which was removed by a previous call to **sb_ISRInstall()**.

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.

```
#define mapioadd(u32 flat) ((u8*)(flat))
```

It is only needed in segmented architectures that expect far pointers. See file **cs8900.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 smxNS 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 frame, your interrupt handler will need to add this new frame to the appropriate smxNS queue. To queue an arrived frame, use the macro:

```
QUEUE_IN(ptr, qname, mess)
```

ptr is a pointer to the network structure sns_Sys. The network structure contains fields which are pointers to message queues (see struct `HOSTINFO` in **support.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 frame.

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 omitted code which does not directly relate to this example.

```
static void irhan(int arg)
{
MESS *mess;
struct HOSTINFO *sysp;
struct NET *netp;
....
sysp = &sns_Sys;
netp = &nets[arg];     /* nets is a global smxNS table */
mess = << get message from controller's memory >>
....
QUEUE_IN(sysp, 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))
   DEBUG_MSG2_PAR0("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 arg)
{
MESS *mess;
struct NET *netp;
....
netp = &nets[arg];
....
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 arg)
{
MESS *mess;
struct NET *netp;
.....
netp = &nets[arg];
....
if (QUEUE_EMPTY(netp, depart))
   << process error >>
else
{
QUEUE_OUT(netp, depart, mess);
<<write message to network controller >>
}
.....
}
```

Interrupt Handling

The smxBase software ISR dispatcher already handles interrupts for you. Please read the smxBase User's Guide for detailed information about its ISR handling.

The network controller's interrupt handler code can be written as an ordinary processor-independent C function.

Interacting with an Ethernet PHY

Ethernet drivers for 10/100 Ethernet controllers need to interact with a PHY. A PHY often takes the form of a physically separate chip that handles signal level encoding and which negotiates the properties of the Ethernet link, such as duplex and speed.

The PHY presents registers to the system that are typically read and written with help from the Ethernet controller. Unfortunately, the details of PHY register access differ from controller to controller, so the interface to the PHY is a little complicated. The functions that read and write the PHY registers are included as part of the Ethernet controller driver. The logic for initializing the PHY and checking the current status of the PHY is common across Ethernet controllers, so these functions are implemented in a separate module, phy.c.

To initialize the PHY, the Ethernet driver should configure the controller for communication over the MDIO and MDC connections, and then call phy_init() to reset the PHY and start the autonegotiation process for the link.

The PHY status is polled as part of a periodic check function that NetTask() calls for all Ethernet controllers about once a second. The Ethernet driver in turn calls phy_check() in phy.c, which will determine the link state. If the state has changed since the previous call, phy_check() calls back into the Ethernet driver to report the change. Some Ethernet controllers need to coordinate with the PHY when the link speed or duplex change, so the callback function coordinates this.

Configuring a New Processor

If smxBase already provides support for your processor, you may ignore this section. Otherwise, refer to the smxBase User's Guide about how to port it to a new processor.

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**. 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 smxNS, 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 an NPTABLE. This is a table of pointers to your driver functions and is the mechanism smxNS uses to call them.

For example, the format for the driver is:

```
irhan(int netno)
{
   ....
}
init()
{
   ....
}
shut()
{
   ...
}
etc.
NPTABLE 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 smxNS expects the ***shut()*** function it will call the third function. Therefore, position within the table is crucial. Each function and the NPTABLE 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.

Starting with v2.90, smxNS uses drvsrc/sbuart.c as a driver shim to allow it to use the interrupt driven UART drivers in smxBase. The existing character drivers will continue to be used for the UARTs that they target, but using the sbuart shim will be the preferred way to add new character drivers.

Figure 12-1 shows how incoming data is handled within smxNS 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.

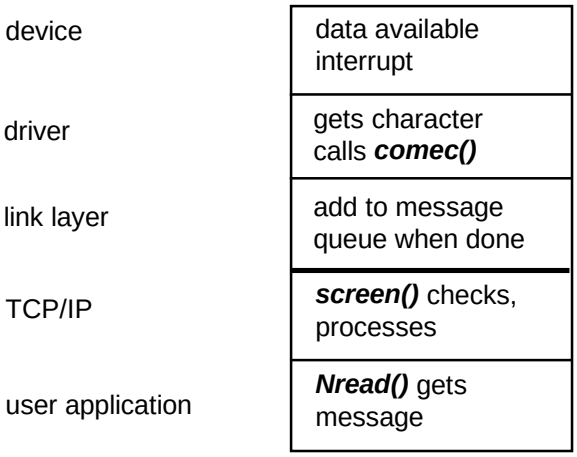


Figure 12-1: Incoming Data

Outgoing data shown in Figure 12-2 is handled according to the following diagram, again from top to bottom. The boxes below the wider line are done for each character.

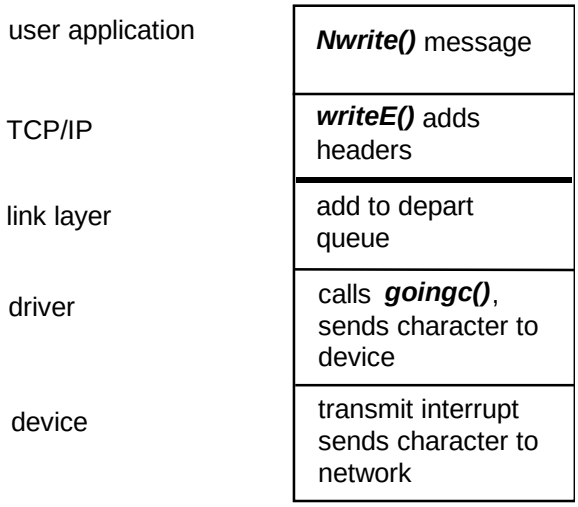


Figure 12-2: Outgoing Data

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. Higher rates than this require either a FIFO buffer in the serial port, or the use of DMA.

smxNS 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 shell which is triggered when a network I/O interrupt occurs. The argument is used to index to the network tables.

```
static int irhan(int inst)
```

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 shell.
   Returns to the interrupt shell.
*/
.
static int irhan(int inst)
{
int char;               /* character to be sent or received. */
uint tport;             /* device I/O port */
u8 status;              /* interrupt status */
struct NET *netp;       /* pointer to network structure */
UARTSTATEP  infop;      /* pointer to local UART state */
infop = uart_state + inst; /* index to state for this instance */
```

```
netp = uart_state[inst].netp; /* assign a ptr to current  */
                        /* network struct via index inst */
tport = infop->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 */
....

return 1;
} /* 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 smxNS during link layer 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 smxNS 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 */
u32 inst;         /* instance */
UARTSTATEP infop; /* local state */
(void)conno;      /* first parameter not needed here */
inst = netno_to_inst[mess->netno];
netp = &nets[mess->netno];
      /* get ptr to network required for this message */
infop = uart_state + inst;
tport = infop->port;     /* assign I/O port */
smx_TaskLock();    /* 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 */
smx_TaskUnlock();        /* 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 parameter for connection number is not used. Nevertheless, it is required for compatibility with the smxNS 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 openN(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, sets up the controller, and stores data into the network table.

```
static int init(int netno, const char *params)
```

netno is the network number

params are the device-initialization parameters

The initialization parameters are the same string you pass in a call to Portinit() (see Chapter 5, *Dynamic Protocol Interface*). Then it calls routine **sb_ISRInstall()** and **sb_IRQUnmask()** to store the interrupt address and enable the interrupt. This routine is called from smxNS 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, const char *params)

{
   int i1, tport;
   u32 l1;
   int irno;
   int port;
   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", &irno);
      else if (strcmp(par, "PORT") == 0)
         Nsscanf(val, "%i", &port);
      else if (strcmp(par, "CLOCK") == 0)
```

```
         Nsscanf(val, “%ld”, &l1);
      else if (strcmp(par, “BAUD”) == 0)
         Nsscanf(val, “%ld”, &netp->bps);
   }


   _outb(port+LCR, 0x80);     /* set baud reg access */
   i1 = l1 / netp->bps; /* set baud rate */
   _outb(port+BRDH, i1>>8);
   _outb(port+BRDL, i1);
   _outb(port+LCR, 0x03);           /* set LCR value */
   _outb(port+IER, 0x03);           /* set IER value */
   _outb(port+MCR, 0x0b);           /* set MCR value */
   i1 = (int)(char)_inb(port+LSR);
                       /* clear any line status int */

   if (i1 == -1)
     goto err2;
   (void)_inb(port+RDR);
                     /* clear any receive interrupt */
   (void)_inb(port+IIR);
                  /* clear any transmitter interrupt */
   (void)_inb(port+MSR);
                 /* clear any modem status interrupt */
   sb_ISRInstall(irno, fn, irhan, "i8250 ISR");
   DEBUG_MSG3_PAR3(“I8250 IR%d P%x BPS%ld\n”, irno, port, netp->bps);

   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. It also calls routine **sb_ISRRestore()** to restore original interrupt status that existed before smxNS was initialized.

```
static void shut(int netno)
```

netno is the network number

Shut() is called by smxNS 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;
```

```
        UARTSTATEP infop;
        inst = netno_to_fn[netno];
        infop = uart_state + inst;
        tport = infop->port;
        while (!(_inb(tport+LSR) & 0x40));
        _outb(tport+IER, 0);
        _outb(tport+MCR, 0);
        sb_ISRRestore(infop->irno);
        infop->inuse = 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.

Network Protocol Table

smxNS uses a *network protocol table* to call functions specific to a given protocol or device. An NPTABLE is defined as follows:

```
#define NPTABLE const struct P_tab
                  /* typedef caused trouble */
struct P_tab {    /* protocol table, end of each module */
    char name[10];            /* name of protocol */
    int (*init)(int, const 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 (*ioctl)(void *, enum ioctlreq, void *, size_t);
    uint Eprotoc;             /* external protocol num */
    u8 hdrsiz;                /* header size */
};
```

Here, you can see an NPTABLE is basically a structure of pointers to functions. smxNS uses this structure to call the protocol, link layer and device-specific functions when they are needed. In other words, smxNS will call your device driver functions by using pointers to them stored within a NPTABLE entry. Be sure you add your function names to the proper fields (see the example below). When smxNS 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 NPTABLE, otherwise your driver will not operate properly.

NPTABLE Example

```
NPTABLE I8250_T = {“I8250”, init, shut, 0, opeN, closE, 0, writE, 0, 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 (STM32ETH)* 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 12-3 shows how incoming data is handled within smxNS as the data is transferred between the network controller and the application. The logic flows from top to bottom.

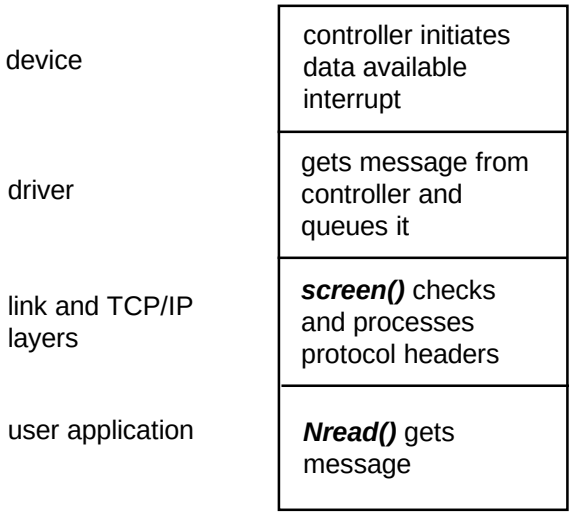


Figure 12-3: Block Driver Incoming Data

Outgoing data, as shown in Figure 12-4, is handled similarly; again, the sequence is from top to bottom.

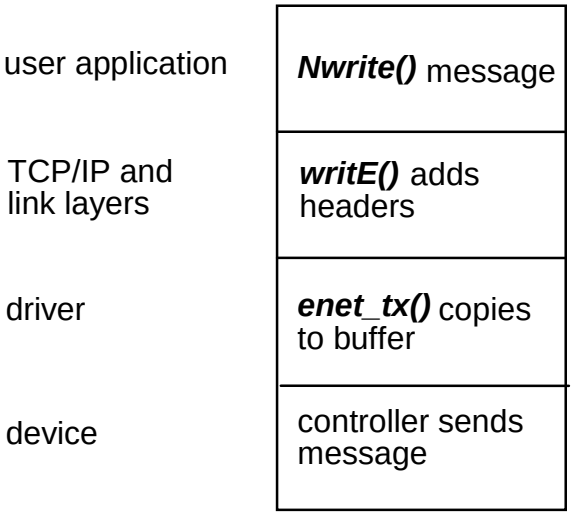


Figure 12-4: Block Driver Outgoing Data

smxNS block drivers are short and simple, and will operate in any protocol stack. A typical size would be 800 lines of code. The easiest way to produce a new driver is probably by editing the existing STM32ETH driver.

Recently written Ethernet controller drivers use a two part design since a large part of the logic is common across Ethernet controllers. If one thinks of the driver as being near the bottom of a TCP/IP stack, then the two part design uses a common upper layer that is implemented in XNS/drvsrc/ethctrl.c, and a lower layer that is specific to the Ethernet controller.

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. 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. 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 shell which is triggered when a network controller interrupt occurs.

`static int irhan(uint fn)`

fn is used to index to the instance of the network structure when multiple instances are supported

Interrupt Handler Example

The Ethernet controller interrupt handler should be short and efficient, capturing state information, performing any needed operations to clear the interrupt and returning as soon as possible. SMX link service routines (LSRs) can be used to defer processing related to the interrupt to make the ISR very short.

The STM32 interrupt handler and related functions appear next, followed by a detailed description.

Irhan Example

```
STATIC int st_enet_isr(uint arg)
{
   u32 status;
   interrupt_count++;
   status = eptr->DMASR;
   eptr->DMASR = (0x0001e7ff & status);  /* clear received flags */
   smx_LSR_INVOKE(st_enet_lsr, status);
   return 1;
}
```

```
STATIC void st_enet_lsr(u32 status)
{
   ....
   if (status & ((1 << 14) | (1 << 6)))  /* early rx or rx complete */
   {
      processing_incoming();
   }
   ....
   if (status & (1 << 0))               /* transmit complete */
   {
      process_outgoing();
   }
   ....
}

void process_incoming(void)
{
   void *fp;  /* pointer to frame buffer */
   u8   *bp;  /* pointer to frame data */
   u32   p;   /* pointer to newly allocated frame buffer */
   uint  pl;  /* frame length */

   while (1)
   {
      if (!(rbd[rx_index].RDES0 & 0x80000000))
      {
         rx_count++;
         p = (u32)NgetbufIR(0);
         if (p)
         {
            pl = (rbd[rx_index].RDES0 & 0x3fff0000) >> 16;
            bp = (u8 *)rbd[rx_index].RDES2;
            fp = update_frame_info(0, bp, pl);
            receive_frame((u32)fp);
            rbd[rx_index].RDES2 = p + MESSH_SZ;
         }
         _DSB();
         rbd[rx_index].RDES = 0x80000000;
         rx_index = (rx_index == NUM_RX_BD - 1) ? 0 : rx_index + 1;
      }
      else
      {
         break;
      }
   }
}

void process_outgoing(void)
{
   while (1)
   {
      if (!(tbd[tx_index2].TDES0 & 0x80000000) && tbd[tx_index2].TDES2)
      {
         tx_count++;
         transmit_done_lsr(tbd[tx_index2].TDES2);
         tbd[tx_index2].TDES2 = 0;
         tx_index2 = (tx_index2 == NUM_TX_BD - 1) ? 0 : tx_index2 + 1;
```

```
      }
      else
      {
         break;
      }
   }
}
```

st_enet_isr()

The interrupt service routine, st_enet_isr(), is about as straightforward as can be. The function captures the Ethernet controller status from a 32-bit register, masks the interrupt related bits and writes them back to the register to clear the interrupt(s), and then invokes an LSR to handle the rest of the processing.

In the case of this driver, the argument passed to the function is ignored. In the case of systems that support multiple instances of a given Ethernet controller, the arg parameter is used to specify which instance generated the interrupt. This can then be used to look up the block of registers specific to the instance.

Only the Ethernet controller specific interrupt acknowledgment needs to take place here. Since st_enet_isr() is called from an interrupt handler wrapper function, operations related to the interrupt controller itself will be handled by the wrapper.

The LSR is invoked by:

smx_LSR_INVOKE(st_enet_lsr, status);

and will run when the ISR exits.

st_enet_lsr()

The next function, st_enet_lsr(), tests the status bits to call helper functions for receiving frames or handling frame transmit complete.

This function and the helper functions run in an LSR context which has a priority lower than an ISR but higher than any task. LSRs are run in the order they are queued, so they don't interfere with each other. When considering concurrency issues, remember that an LSR can preempt a task, an interrupt can preempt an LSR, and there are mechanisms for blocking preemption so that one can set up critical sections where needed.

process_incoming()

The purpose of the process_incoming() function is to queue incoming frames for processing by the stack and reset any used buffer slots so that they can receive additional frames.

Before discussing the specifics of this function, it will be helpful to present some background on smxNS and Ethernet controllers.

In receiving and transmitting frames the device driver translates between the conventions of the Ethernet controller hardware and the frame buffers used by smxNS. For an Ethernet controller, the important attributes are usually a starting location in memory and the length of the frame. smxNS uses a frame buffer data structure to hold network frames, and the header of this structure includes other higher level attributes such as a time stamp and routing information.

The STM32 Ethernet controller uses arrays of buffer descriptors to maintain queues of incoming and outgoing frames. The buffer descriptors contain the start location and length of a frame as well as state attributes (available or in use) and information to maintain the queue.

Now to the specifics of the process_incoming() function.

The function is set up as a loop. The first step is to check to see if the next buffer descriptor contains a frame that is ready to be passed up to the stack. If the bit mask on RDES0 indicates no frame is present, then the function is done and we exit.

When a frame is received, the frame buffer containing the frame will be moved from the receive buffer descriptor queue to the smxNS "arrive" queue. process_incoming() first calls NgetbufIR() to allocate a new frame buffer to fill the slot. The pool of frame buffers contains a fixed number (NBUFFS) of buffers, so it possible that NgetbufIR() will fail, returning 0. In that case we skip the logic to queue the frame buffer (effectively dropping the received frame) and advance to the next receive buffer descriptor.

If we succeed in allocating a frame buffer, we retrieve the frame length and the pointer to the start of the frame and pass it to the function update_frame_info(). This function translates the information to the format used by smxNS's frame buffers and returns a pointer to the start of the frame buffer. The first parameter in this call is the instance and in this driver it is hard coded to 0.

All the information for the incoming frame is now captured in the frame buffer, so we call receive_frame() to queue the frame for processing by the stack. In doing this, we are transferring control of the frame buffer from the Ethernet controller to the stack, so we swap in a fresh one in the statement that follows that sets RDES2. We index from the pointer to the frame buffer to the start of the frame data using the constant MESSH_SZ, which stands for "message header size".

The following section marks the receive buffer descriptor empty and advances the index to the next receive buffer descriptor. The __DSB() memory barrier instruction is intended to handle transaction reordering issues on recent ARM processors, and it forces the operations on the registers to occur in the order of the C code statements.

To summarize the process of handling an incoming frame

Use update_frame_info(instance, ptr, len) to translate from frame pointer and length to an smxNS frame buffer and receive a pointer to that frame buffer.

Use receive_frame(fb) to queue the frame for processing by smxNS.

Use NgetbufIR(0) to allocate a fresh frame buffer, and use the expression (u8 *)fb + MESSH_SZ to generate a pointer to the start of the frame data within that frame buffer.

process_outgoing()

The overall structure of the process_outgoing() function is similar to process_incoming(). It iterates through the transmit buffer descriptors and releases frame buffers to the stack until it comes to an empty slot. Note that this function is called after the Ethernet controller indicates that a frame queued for transmission has completed transmission.

The TDES2 member of the the transmit buffer descriptor is a pointer to the start of the frame data. The function transmit_done_lsr(ptr) takes that pointer and releases the corresponding frame buffer to the stack.

Transmit Routine

This is called from smxNS when your application performs an ***Nwrite().***

```
BOOLEAN enet_tx(u32 fn, u8 *pp, u16 pl)
```

The arguments are: instance, pointer to start of frame data, frame length.

The basic strategy is

1. Set up the controller with the frame attributes.
2. Start the transmission is this is not already underway.
3. Return TRUE if the frame transmission was successfully set up, FALSE otherwise.

Transmit Routine Example

```
BOOLEAN enet_tx(u32 fn, u8 *pp, u16 pl)
{
   smx_TaskLock();  /* protect from reentrancy */
   smx_IRQMask(ENET_IRQ);    /* protect from ENET ISR/LSR */
   if (tbd[tx_index].TDES2)
   {
      /* No transmit buffer descriptors available */
      sb_IRQUnmask(ENET_IRQ);
      smx_TaskUnlock();
      return FALSE;
   }
   else
   {
      /* Put frame pointer in TX Descriptor, start TX */
      tbd[tx_index].TDES2 = (u32)pp;       /* set TX frame pointer */
      tbd[tx_index].TDES1 = pl;            /* set TX frame length */
      tbd[tx_index].TDES0 &= ~0xfcd1ffff;  /* don't clear TER */
      _DSB();
      tbd[tx_index].TDES0 |= 0xf0000000;   /* OWN, IC, LS, FS */
      _DSB();
      eptr->DMATPDR = 1;                   /* transmit poll demand */
      tx_index = tx_index == NUM_TX_BD - 1 ? 0 : tx_index + 1;
      sb_IRQUnmask(ENET_IRQ);
      smx_TaskUnlock();
      return TRUE;
   }
}
```

The enet_tx() function starts by setting up a critical section. smx_TaskLock() will prevent any other task from preempting this task within the critical section. smx_IRQMask() will also prevent the Ethernet controller ISR from being executed while inside the critical section.

The first check is to see if the next transmit buffer descriptor slot is available. Here we use the convention that a slot is available is the pointer to the frame data is 0, otherwise the slot is in use and since this is the next slot in the queue, the queue is full. If the queue ends up being full, this attempt to transmit the frame fails. We leave the critical section and return the value FALSE to indicate that the frame wasn't queued for transmission. Higher layers in the stack will later attempt to retransmit the frame if it contains TCP data, otherwise the outgoing frame is dropped.

If a slot is available, we set up the transmit buffer descriptor as required by the STM32 Ethernet controller. We store a pointer to the start of the frame data, set the frame length, and set other flags to describe the frame, including OWN (0x80000000) to indicate that this slot is now owned by the Ethernet controller.

Setting DMATPDR is a hint to the Ethernet controller that a new frame is available for transmission. Next we increment the transmit buffer descriptor slot so that we consider the next slot when we enter this function.

This completes the critical section, so we call sb_IRQUnmask() and smx_TaskUnlock() to return to normal, task level operation.

Critical Sections in Ethernet Controller Drivers

The critical section in the example enet_tx() function is designed specifically for the STM32 Ethernet controller, but it is useful for demonstrating the considerations one should take in designing the driver.

There are two variables that are important in this section. The first is tx_index, which points to the next available slot in the transmit buffer descriptor array that can be used to set up a frame for transmission. The second variable is the OWN bit flag in the TDES0 flags field of the descriptor.

Setting up the critical section for the tx_index variable is fairly straightforward. This variable is local to the STM32 driver and is initialized to 0 by the compiler and again in the enet_init() function.

The beginning of the Block Drivers section mentions that a network application that calls Nwrite() will end up calling enet_tx() to transmit a frame associated with the application level data. In this case, enet_tx() is running in the context of the network application. Since more than one network application may be running on a system, there is the possibility that one network application may preempt another if they are set to run at different priorities.

The NetTask() that handles background network processing also writes using the enet_tx() function, and by design it operates at a higher priority that network applications. So the processor may be in the middle of the enet_tx() function when it may be preempted by another task that will end up in enet_tx().

Since the tx_index variable is used at the beginning of the critical to check for an open slot and it is incremented at the end of the critical section, the whole section needs to be protected from being preempted by another task. If the section were not protected, one task could start setting up a slot to transmit a frame and another task could preempt and overwrite with the frame it is transmitting. smx_TaskLock() / smx_TaskUnlock() will take care of this, forcing competing tasks to serialize their access to the transmit buffer descriptor queue.

The critical section involving the OWN bit is more complex. In enet_tx(), the function sets OWN in TDES0 to tell the Ethernet controller hardware that the transmit buffer descriptor is completely set up and won't be modified by software. The controller owns the descriptor. Once the frame has been transmitted, the controller hardware clears the OWN bit to indicate it no longer owns the descriptor. Since this is performed with hardware, there is no code that shows this operation. Also, once the frame is transmitted, the transmit complete interrupt is set, and the process_outgoing() function is called in LSR context.

process_outgoing() uses the following check to identify transmit buffer descriptors slots for frames that have been sent and are ready to be released.

if (!(tbd[tx_index2].TDES0 & 0x80000000) && tbd[tx_index2].TDES2)

If TDES2 is non-zero, then the slot has been set up for transmission, and if the OWN bit is clear (the TDES0 check), the controller is finished sending the frame.

Since process_outgoing() uses its own tx_index2 to walk the tail end of the transmit queue, it might appear that enet_tx() and process_outgoing() operate independently and there is no chance for concurrency issues. However, there is this scenario.

A network application calls enet_tx() with Frame 1.

A network application calls enet_tx() with Frame 2.

Frame 1 completes transmission and the transmit complete interrupt arrives as enet_tx() is setting up Frame 2. Specifically, the interrupt arrives just after setting the pointer to the start of Frame 2 and before setting the OWN bit

tbd[tx_index].TDES2 = (u32)pp;
*** interrupt arrives here ***
....
tbd[tx_index].TDES0 |= 0xf0000000;

The interrupt calls process_outgoing() which first releases Frame 1 and then moves to the slot containing Frame 2. From looking at the structure information, it appears that the controller has completed transmitting Frame 2, but in fact it hasn't been set up yet. The problem is that the state of a slot involves the combination of the OWN bit and the frame data pointer, and at this point in enet_tx() the structure is in an inconsistent state.

To move from the specific details here to more abstract guidelines, one should consider where preemptions are possible and which data structures may be in an inconsistent state when preempted. The enet_tx() function should always be implemented keeping in mind that another task may preempt attempting to write a different frame. The mechanism that serializes access to the outgoing frame queue at the Ethernet controller level can use a critical section to keep the variables associated with the queue consistent. The ISR may also interact with the enet_tx() function, and the interaction of any shared variables should be carefully reviewed.

Configure and Start Up

The ***enet_init()*** routine sets up the hardware.

`static int enet_init(u32 fn, uint irno, u32 *addr)`

fn	is the instance number
irno	is the interrupt number
addr	is the MAC address

Configure and Start Up Example

```
static PHYSTATE phystate = {0, 0, 0, 0, 0, &pread, &pwrite,
                            &psetspeed};


/* * * * * * * * * * * * * * * * * * *
** Initialize STM32 MAC
 * * * * * * * * * * * * * * * * * * */
int enet_init(u32 fn, uint irno, u32 *addr)
{
   ....
   eptr = (struct st32enet *)0x40028000;
   ....
   enet_set_buffer();
   stm32_reset(&macaddr[0]);

   /* Init MII PHY */
   if (!phy_init(&phy_state))
   {
      return NE_HWERR;
```

```
   }

   if (!sb_ISRInstall(ENET_IRQ, 0, st_enet_isr, "ENET ISR"))
   {
      return NE_CFGERR;
   }
   sb_IRQUnmask(ENET_IRQ);

   eptr->DMAIER = 0x00014045;  /* enable rx and tx interrupts */

   /* Start transmission, start receive */
   eptr->DMA0MR |= ((1 << 13) | (1 << 1));

   return 0;
}
```

The STM32 only uses one parameter that is passed in the call to enet_init() -- the MAC address. Another device might need different parameters.

The enet_set_buf() function sets up the arrays of buffer descriptors that the STM32 uses to maintain receive and transmit queues. For drivers that use buffer descriptors, smxNS drivers typically set up the constants NUM_RX_BD and NUM_TX_BD to size the arrays. Sometimes the hardware has constraints on the arrays, such as a range of sizes that can be supported.

From an ideal performance standpoint, there should be enough receive buffer descriptors to allow for worst case system latency when responding to a frame received interrupt. On the transmit side, consider that any number of network applications could attempt to queue outgoing data.

Tracking high water marks or frames dropped in the driver under heavy network load is one way to tune these sizes. Also note that even though dropping frames is not good behavior, higher level protocols are designed considering the possibility of lost frames and reliable delivery is not guaranteed at the network level.

A call to phy_init() is required for all modern Ethernet controllers. This function probes for the PHY, initializes it, and starts autonegotiation on the link. It can be called once the registers that control the MDC and MDIO signals to the PHY are configured. phy_init() passes pointers to the PHY access functions that should also be implemented in the driver as well. These functions are described in the next section.

The call to sb_ISRInstall() sets up the interrupt. The parameters, in order, are (1st) interrupt number, (2nd) value that is passed on to the ISR when it is invoked, (3rd) the ISR function, (4th) text string for debugging.

The call to sb_IRQUnmask(ENET_IRQ) enables the Ethernet controller interrupt at the interrupt controller, but no interrupts will occur until the next two lines execute, which enable specific interrupt conditions for the Ethernet controller, and then start up the transmit and receive subsystems.

PHY Support Functions

```
/* Ethernet controller specific function for reading PHY register */
STATIC BOOLEAN pread(PHYSTATE *ctxt, uint reg, uint *val)
{
   int retry;

   eptr->MACMIIAR = (eptr->MACMIIAR & 0x1c) |
                     0x01 | (ctxt->phyaddr << 11) | (reg << 6);
```

```
   /* Wait for response from PHY. */
   for (retry = 0; retry < MII_FRM_TRY; retry++)
   {
      sb_DelayUsec(MII_FRM_DLY);
      if ((eptr->MACMIIAR & 0x1) == 0)
      {
         *val = eptr->MACMIIDR & 0x0000ffff;
         return TRUE;
      }
   }
   return FALSE;
}

/* Ethernet controller specific function for reading PHY register */
STATIC BOOLEAN pread(PHYSTATE *ctxt, uint reg, uint *val)
{
   int retry;

   eptr->MACMIIAR = (eptr->MACMIIAR & 0x1c) |
                    0x01 | (ctxt->phyaddr << 11) | (reg << 6);

   /* Wait for response from PHY. */
   for (retry = 0; retry < MII_FRM_TRY; retry++)
   {
      sb_DelayUsec(MII_FRM_DLY);
      if ((eptr->MACMIIAR & 0x1) == 0)
      {
         *val = eptr->MACMIIDR & 0x0000ffff;
         return TRUE;
      }
   }
   return FALSE;
}

/* Ethernet controller specific function for change in PHY state */
STATIC void psetspeed(PHYSTATE *ctxt)
{
   u32 maccr_copy;

   if (ctxt->state & PHY_STATE_ANEG)
   {
      /* Clear FES | DM */
      maccr_copy = eptr->MACCR & ~((1 << 14) | (1 << 11));

      if (ctxt->state & PHY_STATE_100M)
      {
         maccr_copy |= 1 << 14;  /* FES, Fast Ethernet Speed */
         report_speed(ctxt->fn, 100000000L);
      }
      else
      {
         report_speed(ctxt->fn, 10000000L);
      }
      if (ctxt->state & PHY_STATE_FDUP)
         maccr_copy |= (1 << 11);  /* DM, Duplex Mode */
```

```
      eptr->MACCR = maccr_copy;
   }
   else
   {
      report_speed(ctxt->fn, 0);
   }
}
```

The PHY read and write functions are similar. The PHY register offset and value are passed as parameters. The PHY address is needed to set up the management bus transaction and can be retrieved from the context parameter.

The read and write functions return TRUE for a successful operation and FALSE otherwise.

The PHY driver calls the psetspeed() function on a change of PHY state. The state conditions that can change are stored in the state member of the ctxt parameter.

(state & PHY_STATE_ANEG) indicates that the PHY has established a link and can transmit and receive data.

(state & PHY_STATE_100M) indicates the PHY is set for 100 Megabit speed, otherwise the speed is 10 Megabit.

(state & PHY_STATE_FDUP) indicates the PHY is in full duplex mode, otherwise it is in half duplex mode.

After coming out of initialization, the state of the PHY is no link, so this function will be called as soon as autonegotiation completes, passing the details on what was negotiated.

Depending on the Ethernet controller, some registers may need to be adjusted to match these PHY properties, so this is the place to implement that support.

The psetspeed() function calls the report_speed() function to communicate interface status to the higher layers in the stack. This logic should follow that example here. The actual speed of the link is not used directly by the higher layers, but it can be helpful diagnostic information. The stack does check to see if the link speed is zero or non-zero to determine if a link is up.

Polling

The NetTask() function periodically calls a polling function every FASTTIMERDELAY milliseconds (typical value 200) in each active device driver for housekeeping. For Ethernet controllers, this is currently used to check the PHY state. In a two layer Ethernet controller driver, the entry point is enet_check().

```
void enet_check(u32 fn)
```

fn the Ethernet controller instance

Polling Example

```
/* * * * * * * * * * * * * * * * * *
** ST Check
**
** Called by ioctl() in ethctrl.c.
 * * * * * * * * * * * * * * * * * */
void enet_check(u32 fn)
{
```

```
   (void)fn;

   phy_check(&phystate);
}
```

The implementation of this function should call the phy_check() function in the PHY driver, passing the PHYSTATE structure.

The polling function is also useful for debugging purposes, since it is predictably called while the driver is running. Other entry points such as enet_tx() and the interrupt handler depend on network application activity or Ethernet activity. Here the execution is in task context and not being called at a high rate, so it's a dependable place to capture state information.

Shut Down

This turns off the controller.

`void enet_term(u32 fn)`

fn the Ethernet controller instance

It calls routine **sb_ISRRestore()** to restore the original interrupt.

Shutdown Example

```
/* * * * * * * * *
** Shut down EMAC
 * * * * * * * * */
void enet_term(u32 inst)
{
   int i1;

   eptr->DMAIER = 0x00000000;  /* All interrupts disabled */

   for (i1 = 0; i1 < NUM_RX_BD; i1++)
   {
      if (rbd[i1].RDES2)
      {
         release_pkt((u8 *)rbd[i1].RDES2);
      }
   }
   sb_ISRRestore(ENET_IRQ);
}
```

The goal of the shut down function is to return the Ethernet controller to the state that existed before the controller was initialized. The hardware is put into an idle state, any frame buffers that were reserved for storing incoming frames are returned to system and the interrupt vector is returned to its previous state.

Protocol Table

See the *Character Driver* section of this chapter for an explanation of the protocol table. It is identical for block drivers. Since the STM32 example uses a two part driver, the protocol table is located in the upper layer in ethctrl.c.

NPTABLE Example

```
GLOBALCONST NPTABLE ussEthCtrlTable = {“ETHCTRL”, init, shut, 0,
openN, closE, 0, writE, ioctl, 0, MESSH_SZ};
```

The field “ETHCTRL” is the name of the driver; all others are normally fixed. Zeros may be used for functions which are not used.

12. Technical Background

Overview

smxNS 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.

smxNS 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:

- smxNS 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.
- smxNS 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 smxNS.

smxNS 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 13-1.

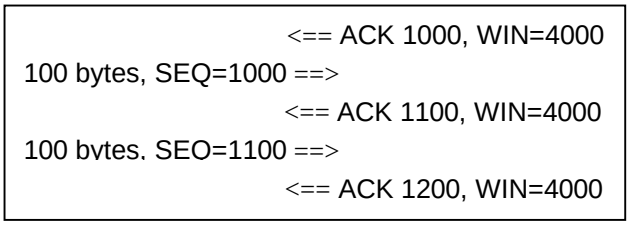


Figure 13-1: Packet Exchange

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:

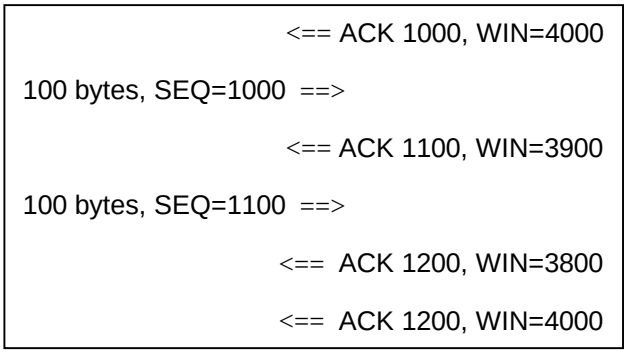


Figure 13-2: Delayed ACK

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:

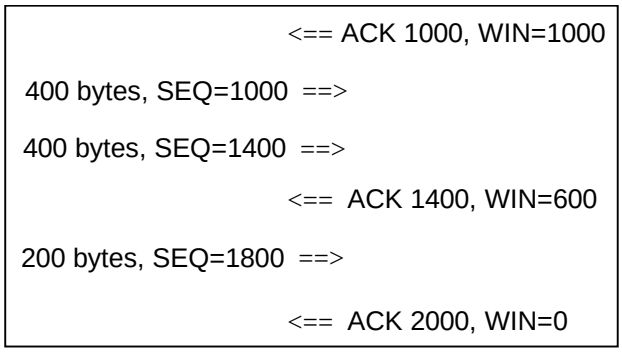


Figure 13-3: Exhausted Window

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.

smxNS uses a 1-byte data probe as described in RFC 1122, and as used by the UNIX-based implementations.

TCP Delayed ACK

smxNS 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. smxNS 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.

smxNS 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.

TCP Window Probe

There is a known limitation in the smxNS TCP/IP stack that the TCP layer will not perform a window probe in non-blocking mode. What this means to the application is that, if the remote host's TCP window ever closes (or becomes smaller than the packet we are trying to write), our local non-blocking host may never be able to write that packet. The remote host never needs to notify the local host that its window is larger. This is the purpose of the window probe - we must probe the remote host to check when its window becomes larger. (Note that some hosts, including smxNS will inform the other host when the window opens, so this problem will not always occur.)

Address Conflict Detection

If two hosts on a network attempt to use the same IP address, it is likely that both systems will have trouble with their communication. In order to avoid this condition, smxNS tests the availability of an IP address before using it, and watches for attempts by other systems to use the same address. RFC 5227 recommends practices to implement Address Conflict Detection, and smxNS follows these recommendations.

If a conflict is detected when probing a candidate address for a conflict when first bringing up an interface, the network interface is moved to a state of "address not configured". If the start up testing shows no conflicts, the network interface moves to a state of "address announced", which is the normal state.

For systems with a single network interface, system start up should include a call to Portinit() to bring up the interface, followed by a poll of the ACD state, to make sure that an IP address has been established with no conflicts. Here is an example taken from nsdemo.c.

```
if (Portinit("enet", “”) < 0)
{
  DEBUG_MSG2_PAR0("smxNS Portinit for enet Failed\n");
}
/* Wait for network interface to be ready before launching network apps */
while (nets[0].acd_state != IP_ADDR_ANNOUNCED)
  smx_DelayMsec(500);
```

If a conflict is detected once smxNS has started using an IP address, a level 1 error message will be logged and a counter will record the event. The smxNS system will continue to use the original address in this case. If the conflict was with an address probe packet from another system that has implemented Address Conflict Detection, then the other system should abandon the address and a conflict should be avoided.

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.

smxNS 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

CGI	Common Gateway Interface. CGI reads parameters from forms on the displayed web page to the server, so the server can display different pages depending on the user's actions.
CHAP	Challenge Handshake Authentication Protocol. A user and password authentication method used by a PPP connection. Both the user name and password are encrypted.
DNS	Domain Name Server. This is a machine which tells remote hosts what IP address corresponds to a host name and vice versa.
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.
DPI	Dynamic Protocol Interface. This is smxNS'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.
HTML META commands	Commands embedded in the HTML that return predefined system information to the user.
HTTP	Hypertext Transfer Protocol, a simple application- level protocol used to access hypermedia documents. The protocol is stateless and generic, which allows it to be used for many tasks.
ISMAP	An HTML tag which returns position coordinates within the page image.
Link Layer	The protocol used over the physical connection between two hosts. smxNS supports Ethernet, PPP, and SLIP.
MIME	Multipurpose Internet Mail Extensions, which defines how to encode and decode multipart messages and non-ASCII character sets.
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.
POP	Post Office Protocol, a minor variation of SMTP that allows a client to retrieve mail from a remote server mailbox.
RTOS	Real Time Operating System, such as SMX®.
SMTP	Simple Mail Transfer Protocol, a protocol for transferring mail.
SVA	Server Variable Access, a mechanism for accessing static global variables within an embedded application via HTML.

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. Debugging Techniques

Overview

Debugging a network application can be complicated because there are many places where the flow of information could be interrupted. Your initial symptom may be that an FTP "get" operation failed, but walking that failure down to the root cause may require a good understanding of TCP/IP networking and details on hardware and software configuration.

Many network link layers, such as Ethernet, are "unreliable". This means that there is no guarantee that a frame sent on one end of a connection will be successfully received at the destination end. These rules of behavior extend into the device drivers. Thus, the smxNS Ethernet controller driver may find that there is no memory available to receive an incoming frame, and so the frame will be dropped.

Upper layers in the network are designed to take these shortcomings into account and work around them with techniques such as retransmission. This can also make debugging difficult, because upper layers may mask problems at lower layers, and not all errors in the system indicate a problem.

TCP/IP networking is standards-based, and there are freely available tools that can help you review and analyze network traffic. In addition, smxNS includes features to flag unusual events, log activity, and display status. This allows you to see "inside" the stack as it is running or walk through a period of network activity to see at what point things went wrong.

By using smxNS-specific tools together with tools that present a third-party view of the network, you can compare the different accounts and determine the exact point at which a problem occurred. For example, if a network analyzer shows that an ARP request is being sent out on the network, but the smxNS log indicates that the request wasn't received, you can narrow the search down to somewhere between the Ethernet jack and the smxNS ARP module.

This section describes smxNS specific features as well as software you can run on a desktop PC that will help you uncover the root cause of a network problem.

Displaying Trace Data

Throughout the smxNS source code there are lines that look like the following.

```
DEBUG_MSG2_PAR1("FQ queued SQ%08lx\n", UL1.l);
```

This example comes from the module tcp.c, and it logs the sequence number of a TCP segment that is being placed in the future queue. Normally, TCP segments arrive in sequence, so this is an unusual event.

The macro SNS_DEBUG_LEVEL is defined in XNS\include\nscfg.h, and it allows you to control the level of trace data output. The trace statements flag error conditions and also log normal activity. You may change the value of SNS_DEBUG_LEVEL to any value between 0 and 6, with 0 representing no trace and increasing numbers representing an increasing amount of trace output. At installation SNS_DEBUG_LEVEL is set to 3.

Here are examples of the type of information that is logged at different SNS_DEBUG_LEVEL settings.

0: No logging. Can be used in a shipping product to reduce memory use.

1: Only output fatal error information

2: Output additional warning information

3: Output additional status information

4: Output additional device change information

5: Output additional data transfer information

6: Output interrupt information

The DEBUG_MSG macro formats and prints the log information using the Nprintf() function. This function is similar to the C library printf() function, but since it is implemented as part of smxNS, the way the resulting string is handled can be customized.

The debug macro is of the form DEBUG_MSGd_PARp, where d is the debug level from 1 to 6, and p is the number of parameters in the format string from 0 to 10.

Most hardware platforms support sending network stack trace information through an RS232 serial port. Systems that are built with smxAware can also display the trace information by selecting smxAware | smx Objects | Print, from the IDE.

Even with no special support, the trace strings are directed to a buffer that can be dumped under the debugger. Display the contents of sns_Log.buf under the debugger to see the trace information in raw format.

Below is a TCP trace fragment captured from a target while executing a file transfer. 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
```

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.

Debug over Telnet

The default set of smxNS demo programs in nsdemo.c include a Telnet server that can display smxNS data structures while the system is running. This feature can be incorporated into other applications by using similar code to start a Telnet server task and calling the sns_DebugCli() function to process Telnet command lines.

This debug version of the Telnet server will accept simple commands from a Telnet client that display state information about the network as it runs. The downside of this feature is that information is not available if the system becomes unresponsive.

Here are the Telnet debug commands

arpstat: Dump the ARP Table

```
>arpstat
ARP Status for Net 0
ARP Status for Net 1
```

```
  4 192.168.001.126  00:80:c8:39:7b:b1
ARP Status for Net 2
>
```

In this example, the ARP tables for three networks are dumped, but only Net 1 has an ARP table entry. The line displaying the entry shows the index in the ARP table, the IP address and the ARP address.

bufstat: Display Details for Frame Buffers

Example:

```
smxNS skeleton Telnet server
>bufstat
NBUFFS=15  MAXBUF=1536  MESSH_SZ=32 Nbufbase=0x20000000
 Buffers Alloc OK=25859 Fail=0 Free OK=25853 Low Water=5
Nfirstbuf (first free buffer) =  4
 # nx   time        target     mlen netno offset conno  id   Queue
 1 00  2744780 192.168.001.126   87     1 RLEASD     0 ALOC  0x0000
 2 00  2744890 192.168.001.126   87     1 RLEASD     0 ALOC  0x0000
 3 11  2738650 192.168.001.126 1536     1 RLEASD     0 FREE  0x0003
 4  6  2738670 192.168.001.126   86     1 RLEASD     0 FREE  0x0001
 5 00  2744640 192.168.001.126   88     1     86    58 ALOC  0x1000
 6  7  2738650 192.168.001.126   86     1 RLEASD     1 FREE  0x0003
 7  3  2738650 192.168.001.126 1536     1 RLEASD     0 FREE  0x0003
 8 00  2745180 192.168.001.126   88     1 TXDONE     0 WACK  0x0200
 9 00  2738650 192.168.001.126  928     1     86    58 ALOC  0x0800
10 00  2744110 192.168.001.126   86     1 RLEASD    58 ALOC  0x0000
11 12  2600420 192.168.001.126 1536     1 RLEASD     0 FREE  0x0003
12 13        0 000.000.000.000    0     0      0     0 FREE  0x0003
13 14        0 000.000.000.000    0     0      0     0 FREE  0x0003
14 15        0 000.000.000.000    0     0      0     0 FREE  0x0003
15 00        0 000.000.000.000    0     0      0     0 FREE  0x0003
>
```

The bufstat display starts with some compile time constants that are give a convenient picture of the layout of the frame buffers in memory. The buffers are implemented as an array in memory that start at Nbufbase.

The next line provides statistics on the buffer allocation function. If the buffer allocation fail count is greater than zero, then some attempts to allocate a buffer failed due to low space, but this doesn't necessarily mean that the pool was completely exhausted. The value "Low Water" indicates the lowest number of free buffers available since starting networking support. Ideally, this value should fall to 1 after the system had been tested under the most stressful network traffic load.

The columns in the tabular section of the display are as follows

the index of the buffer in the array.

nx free buffers are in a singly linked list. This is the next buffer in the list. The first buffer in the list is given by Nfirstbuf in the heading.

time a timestamp for the buffer, in milliseconds. One place that this timestamp is written is when an outgoing TCP segment is first constructed. It is also written at other times, or it may be an old value from the last time the buffer was used.

target the IP address of the host to which this frame is being sent.

mlen the number of bytes in the network frame held in the buffer.

netno the index of the network interface associated with the frame.

offset for frames that are in the process of being processed, this points to the location at which new information is being accessed. For frames that are complete, this may contain a flag to indicate if it is queued for transmission, or if it has been sent.

conno the index of the connection data structure (connblo) associated with the buffer.

id a flag indicating the disposition of the buffer. In order to be responsive, the system should always have at least one buffer in the FREE state.

queue a bit-encoded tally of the queues that reference this buffer. The significance of the bits follows.

0x01	Free buffer list
0x02	Waiting for ARP reply
0x04	Network interface arrive queue
0x08	Network interface depart queue
0x10	IP layer fragment queue
0x20	IP layer fragment head
0x40	PPP partial frame inbound
0x80	PPP partial frame outbound
0x100	Connection arrive queue
0x200	TCP waiting for ACK queue
0x400	TCP future queue
0x800	Application layer output buffer
0x1000	Application layer input buffer

ifstat: Display Network Interface State

```
>ifstat
Network Interface enet index 0
Bits/sec 100M  Frame size  1280  MAC 00:01:02:03:04:05
Link layer: Ethernet  Driver: ETHCTRL  State: Ready
          Frames  Bytes  Discards  Errors
Inbound                          0       0

Outbound                         0       0
IPv4 10.0.1.100  Mask 255.255.255.0
IPv6 fe80::201:2ff:fe03:405
```

In this example, details for network interface "enet" are displayed. State: Ready indicates that the Ethernet link is established and the interface is ready to transmit and receive frames. The Inbound and Outbound rows show a count of frames that were discarded or contained errors. There are columns for displaying frame and byte counts, but support for displaying this information isn't implemented yet.

logdump: Display smxNS Log

```
>logdump
192.168.1.126
1 FTtest OK
FTP.getput: TX 9360 bytes in 10 ms = 936000 bytes/sec
FTP.getput: RX 9360 bytes in 10 ms = 936000 bytes/sec
2 FTtest OK
TN-TX IAC 253 24
TN-RX IAC 251 24
TN-TX IAC 250 24 1
TN-RX IAC 251 31
TN-RX IAC 250 24
TN-RX 0 41 4e 53 49 ff f0
>
```

The log captures the information that is written with the DEBUG_MSG() macros used throughout the code. The buffer wraps, and the logdump command shows the information most recently written into the log.

memdump: Display Memory

```
>memdump 20004fb4
20004fb0  6e 2d 00 00>8b 53 00 20 0f 23 00 00 36 30 2f 36
20004fc0  34 33 37 36 20 53 51 30 30 30 30 32 38 34 30 20
20004fd0  41 4b 34 63 36 34 34 30 36 65 20 31 38 0a 51 75
20004fe0  65 75 65 64 20 57 72 69 74 65 20 50 61 63 6b 65
20004ff0  74 0a 53 43 20 30 30 30 30 31 39 38 39 30 20 43
20005000  30 33 2f 32 33 20 20 20 20 53 54 30 31 20 44 4c
20005010  30 30 30 30 20 57 34 39 36 30 2f 36 34 33 37 35
20005020  20 41 4b 30 30 30 30 32 38 34 31 20 53 51 34 63
20005030  36 34 34 30 36 65 20 31 30 0a 53 43 20 30 30 30
```

In this example, a memory dump starting at address 0x20004fb4 is displayed. The display starts on a 16-byte aligned location and highlights the requested location with ">".

The number of lines displayed is fixed with the setting of DUMP_LINE_COUNT in XNS/netsrc/debug.c. The memdump command is disabled by default since dumping at a protected location can lead to a memory access violation fault. memdump can be enabled by setting ENABLE_MEMDUMP to 1 at the top of debug.c.

netstat: Display Connection Status

```
>netstat
Dropped incoming connection attempts: 0
 # bs Type Rx Tx Local Address         Remote Address        State
 0  1 UDP   0  0 000.000.000.000:00161 255.255.255.255:00000    16
 1  1 TCP   0  0 000.000.000.000:08123 255.255.255.255:00000    LN
 2  1 TCP   0  2 010.000.001.100:00023 010.000.001.061:52312    ES
 3  0
 4  0
 5  0
 6  0
```

```
 7  0
 8  0
```

The netstat command displays information about network connections. The first line shows the number of dropped incoming connection attempts tracked by the global variable sns_TcpSynDrops. If this appears as a value greater than zero, you should consider increasing the setting of NCONNS in nscfg.h so that incoming connections attempts are not dropped due to lack of available connections.

The meaning of the columns in the following lines are explained below

the index of the network connection.

bs the state (blockstat) of the connection.

Type UDP or TCP

Rx the number of incoming segments waiting to be processed.

Tx the number of outgoing segments waiting for acknowledgement.

Local the local IP address and port associated with the socket.

Remote the remote IP address and port associated with the socket

State the connection state.

nqstat: Show the State of Connections

```
>nqstat
   <-CONNBLO-> <--- NCONNS QUEUES ---> | <- NNETS ->
 # bs nx ic St iQ:fb wQ:fb oS:fb iS:Fb | aH:aT dH:dT
 0  1  0  2 ES  0: 0  1: 6  1: 3  1: 4 |  0: 0  0: 0
 1  0  0  0 TW  0: 0  0: 0  0: 0  0: 0 |  0: 0  0: 0
 2  1  0  0 LN  0: 0  0: 0  0: 0  0: 0 |  0: 0  0: 0
 3  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
 4  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
 5  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
 6  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
 7  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
 8  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
 9  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
10  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
11  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
12  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
13  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
14  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
15  0  0  0  0  0: 0  0: 0  0: 0  0: 0 |
>
```

The netstat display provides information about two data structures: network connections (connblo) and network interfaces (nets). Information about queues is in the format qn:fb, where qn is a mnemonic for the queue name, and fb is the index of the first buffer in that queue. The meaning of the columns are as follows.

the index in the structure.

bs the state (blockstat) of the connection.

nx the index of a linked connection if one exists.

ic the listening connection that spawned this connection.

St	the TCP connection state.
iQ	the number of buffers queued for reception on this connection.
wQ	the number of buffers in the "wait of ACK" queue for this connection
oS	the number of buffers in the Sockets API output stream for this connection. This value should not be greater than 1.
iS	the number of buffers in the Sockets API input stream for this connection. This value should not be greater than 1.
aH	the number of network frames queue for input on the network interface with this index.
dH	the number of network frames queued for transmission on the network interface with this index.

routestat: Display Routing Information

```
>routestat
 Default router 192.168.1.1
>
```

The routestat command displays the default router.

Other Commands

"help" displays a list of available commands.

"quit" closes the Telnet session.

Network Analyzers

A network analyzer is a tool that can capture and intelligently display network traffic. Dedicated hardware boxes are sometimes used for this purpose, but there are also software packages that can do the job very well.

Since most Ethernet hubs today provide switched circuits for network traffic, a software analyzer plugged into a hub will normally not be able to capture the traffic between two other systems. In order to see all the traffic, you can use an older 10BASE-T repeater hub, or you can run the analyzer on the computer that is communicating with the system running smxNS. There are also more sophisticated switches that can mirror traffic to a designated port on the device.

The **Wireshark** protocol analyzer is very capable and also freely available. It is available from http://www.wireshark.org.

Windows Utilities

There are a number of useful command line network utilities that are available on Windows computers. Typing the command name by itself will usually display brief usage information.

arp Display or modify ARP table information. This can be used to see which MAC address a Windows system has associated with an smxNS system.

ftp Connect to an FTP server. This can be used with the FTP server in the nsdemo.c demo.

ipconfig Displays information about the Windows system's network interfaces. If the computer receives its IP address via DHCP, this is a quick way of finding that address.

ping Send ICMP echo request and report the response. This is a useful first test when bringing up a system.

route Display or modify routing tables.

telnet Establish a telnet session with a system. This can be used with the Telnet server in the nstels.c or nsdemo.c demos.

tftp Connect to a TFTP server. This can be used with the TFTP server in the nsdemo.c demo.

Web Servers

A web server must be available in order to exercise the HTTPget() function that retrieves web pages. If the server is to be located on local LAN, you may consider the TinyWeb server freely available here.

http://www.ritlabs.com/en/products/tinyweb/

This implementation is very small and simple, and is freely available for commercial use at the time of this writing. It runs from the command line, and will serve pages from a specified directory.

Of course, if you have the smxNS Web Server package, you could use that too!

Verification Testing

Micro Digital runs smxNS through a verification test as part of the release process. Some components of the test are included in the nsdemo.c application, for example the TTCP performance test. Tests include:

- PHY monitoring
- ARP subsystem
- DHCP client including address renewal
- ISIC random traffic test, checks integrity when receiving invalid frames
- sockstress denial of service testing
- IPv6 testing
- Long duration FTP client transfer

- Web Server
- Telnet Server
- TTCP performance test
- mDNS Responder test
- SNMP Agent test

C. Dynamic Configuration

Overview

Dynamic configuration utilities are implemented in the file **confupd.c**. Dynamic configuration supports "on-the-fly" modification of host properties.

Configuration Functions

The following functions are discussed in this section:

SetDefaultRouter() sets the default router.

SetDefaultRouter

Sets the default router.

```
int SetDefaultRouter(const char *ip, const char *ifname)
```

ip Default router IP address, specified in dotted decimal format

ifname Interface that should be used to reach the default router

SetDefaultRouter sets the default router for the system. IP traffic will be directed to the default router if an outgoing datagram cannot be directly reached on an attached network. This function can be called at any time, and it can be called multiple times to change the default router setting.

Return Value

This function will return –1 if the specified network interface cannot be found, otherwise it will update the setting and return 0.

Example

SetDefaultRouter("192.168.1.2", "eth0");

D. Driver-Specific Information

ACT10100

About the device:

type	Ethernet
chip	Actel 10100 Ethernet controller (IP)
card	N/A
buffer memory	on-chip, configurable with IP
data transfer	DMA
interrupts	DMA, Ethernet event

Configuration

Interrupt number and port address are fixed. The MAC address must be supplied when initializing the driver.

```
Portconfig("eth0", "MAC", "00:01:02:03:04:05");
Portconfig("eth0", "DRIVER", "ETHCTRL");
```

AT91

About the device:

type	Ethernet
chip	Atmel AT91 ARM processors
card	N/A
buffer memory	host memory
data transfer	DMA
interrupts	single interrupt

This is a driver for the Atmel AT91 on-chip EMAC. Drivers for AT91SAM7X, AT91RM9200, AT91SAM9260, and AT91SAM9263.

Configuration

Interrupt number and port address are fixed. The MAC address must be supplied when initializing the driver.

```
Portconfig("eth0", "MAC", "00:01:02:03:04:05");
Portconfig("eth0", "DRIVER", "ETHCTRL");
```

CFFEC

About the device:

type	Ethernet
chip	Freescale ColdFire Fast Ethernet Controller drivers
card	N/A
buffer memory	host memory
data transfer	DMA
interrupts	RX, TX, DMA (548x/7x) and several exception conditions

This is a driver for the Freescale ColdFire on-chip Fast Ethernet Controller. See the comment at the top of cf5282.c for a list of ColdFires supported. The Coldfire 548x/7x also have an on-chip Ethernet controller known as a FEC, but this design has significant differences. The CF548x/7x FEC is supported by the driver cf548x.c.

Configuration

Interrupt number and port address are fixed, so only the MAC address needs to be configured:

```
Portconfig("eth0", "MAC", "00:01:02:03:04:05");
Portconfig("eth0", "DRIVER", "ETHCTRL");
```

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.

Configuration

The hardware parameters are configured automatically by the PCI services. The call to Portconfig() defines PCI as the driver:

```
Portconfig(“eth0”, “DRIVER”, “PCI”);
```

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 smxNS 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 smxNS by changing *CPU* back to 0. However, you might want to look for a better PC. Note: BENCH was a standalone test program in USNet.

EP93XX

About the device:

type	Ethernet
chip	Cirrus Logic EP93xx ARM family of processors
card	N/A
buffer memory	host memory
data transfer	DMA
interrupts	single interrupt

This is a driver for the Cirrus Logic EP93xx on-chip Ethernet MAC.

Configuration

Interrupt number and port address are fixed. The MAC address must be supplied when initializing the driver.

```
Portconfig("eth0", "MAC", "00:01:02:03:04:05");
Portconfig("eth0", "DRIVER", "ETHCTRL");
```

I8255X

About the device:

type	Ethernet
chip	Intel 8255x family.
card	EtherExpress PRO/100
buffer memory	host memory
data transfer	DMA
interrupts	RX, TX

This is a driver for the Intel EtherExpress PRO/100. This driver has been tested on the i82559 and the i82550. Note that more recent versions in this chip family have lower numbers than the initial versions, i.e., the numbering went from i82559 to i82550. The more recent chips have a superset of the features of the earlier versions, and are register level compatible.

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 call to Portconfig()is simply:

```
Portconfig("eth0", "DRIVER", "PCI");
```

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.
2. Adds the new buffer to the tail of the receive list.

LAN91CXXX

About the device:

type	Ethernet
chip	SMSC LAN91C90, LAN91C92, LAN91C94, LAN91C111
card	SMSC91C92
buffer memory	on-chip, amount varies
data transfer	8 or 16-bit, input/output
interrupts	RX, TX, allocation

This is a driver for the Standard Microsystems Corporation LAN91CXXX family of Ethernet adapters.

Configuration

Interrupt number and port address are needed, for instance:

```
Portinit(“eth0”, "IRNO=5 PORT=0x340");
```

See the SMC booklet on how to configure the board. The actual board configuration must match the parameters specified in the call to Portinit(). An incorrect configuration will most likely cause a crash or a hang.

The LAN91C111 has an integrated PHY. The driver initializes this in autodetect mode.

Interrupt Handling

The driver clears the interrupt by masking off all LAN91CXXX 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 smxNS 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 smxNS buffers are available.

LM3S

About the device:

type	Ethernet
chip	Luminary Micro Stellaris LM3Sxxxx ARM family of processors
card	N/A
buffer memory	dedicated FIFO
data transfer	PIO
interrupts	single interrupt

This is a driver for the Luminary Micro LM3Sxxxx on-chip Ethernet MAC.

Configuration

Interrupt number and port address are fixed. The MAC address must be supplied when initializing the driver.

```
Portconfig("eth0", "MAC", "00:01:02:03:04:05");
Portconfig("eth0", "DRIVER", "ETHCTRL);
```

LPC2XXX

About the device:

type	Ethernet
chip	NXP LPC2xxx ARM family of processors
card	N/A
buffer memory	host memory
data transfer	32-bit input/output
interrupts	single interrupt

This is a driver for the NXP LPC2xxx on-chip Ethernet MAC.

Configuration

Interrupt number and port address are fixed. The MAC address must be supplied when initializing the driver.

```
Portconfig("eth0", "MAC", "00:01:02:03:04:05");
Portconfig("eth0", "DRIVER", "ETHCTRL");
```

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 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.

Configuration

Interrupt number and port address are needed, for instance:

```
Portinit(“eth0”, "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 parameters specified in the call to Portinit(). 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. smxNS 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 smxNS buffers are available.

RTL8139

About the device:

type	Ethernet
chip	Realtek RTL8139
card	Commonly found in commodity PCI cards circa 2008.
buffer memory	host memory
data transfer	DMA to/from host memory, then copied into frame buffers
interrupts	RX, TX

This is a driver for the Realtek RTL8139 Ethernet controller with integrated PHY.

Configuration

The driver needs an interrupt number (parameter IRNO) and the port address. These are normally supplied by the PCI BIOS support, so typical configuration through Portconfig() is simply:

```
Portconfig("eth0", "DRIVER", "PCI");
```

The RTL8139 has an integrated PHY. The driver initializes this in autodetect mode. The RTL8139 implements a register set that mimics standard PHY registers in the controller, but it is not a complete implementation. There are no PHY ID registers, so the RTL8139 driver returns the value "deadbeef" for these registers so that the RTL8139 can be recognized.

Sending

The RTL8139 controller provides 4 buffer descriptors for outgoing frames. The driver uses this set of descriptors for outbound queueing. If no buffers are available, the driver increments *ifOutDiscards* for the network interface.

The transmit threshold size setting is adjusted upward when the transmit FIFO underrun condition is detected.

Receiving

Since the RTL8139 controller requires incoming information to be directed to a single large buffer, the incoming frames are copied from this single buffer to individual smxNS frame buffers in the receive interrupt.

STRXXX

About the device:

type	Ethernet
chip	STMicro STRxxx ARM family of processors
card	N/A
buffer memory	host memory
data transfer	DMA
interrupts	single interrupt

This is a driver for the STMicro STRxxx on-chip Ethernet MAC. Tested on STR912.

Configuration

Interrupt number and port address are fixed. The MAC address must be supplied when initializing the driver.

```
Portconfig("eth0", "DRIVER", "ETHCTRL");
```

USBD

About the device:

type	Ethernet
chip	N/A
card	N/A
buffer memory	host memory
data transfer	memcpy()
interrupts	none

This is a driver for USB device stack to Remote NDIS Ethernet. This allows the system running smxNS to be connected to a desktop host via USB and appear as an RNDIS network adapter.

Configuration

This is an interface between software layers. No configuration is needed.

```
Portconfig("eth0", "DRIVER", "USBD");
```

Sending

The USBD driver uses the API to the USB device stack to write frames using the RNDIS layer.

Receiving

The USBD driver provides a received data callback function that is called from a task in the USB device stack. The USBD driver registers the callback function once the USB stack reports the port is connected.

State Information

When the USB interface is connected, the value of nets[n].bps is non-zero, otherwise it is 0. Here n is the index of the network interface.

USBH

About the device:

type	Ethernet
chip	N/A
card	N/A
buffer memory	host memory
data transfer	memcpy()
interrupts	none

This is a driver for USB host stack to Ethernet converter. It allows the smxNS system to use a USB to Ethernet adapter (dongle) to connect to an Ethernet cable and network.

Configuration

This is an interface between software layers. No configuration is needed. Note that the MAC address for the Ethernet interface is supplied by the external hardware.

```
Portconfig("eth0", "DRIVER", "USBH");
```

Sending

The USBH driver uses the API to the USB host stack to write frames.

Receiving

The USBH driver provides a received data callback function that is called from a task in the USB host stack. The USBH driver registers the callback function once the USB stack reports the Ethernet interface is up

State Information

When the USB interface is connected, the value of nets[n].bps is non-zero, otherwise it is 0. Here n is the index of the network interface.

WiFi

About the device:

type	Ethernet
chip	N/A
card	N/A
buffer memory	host memory
data transfer	memcpy()
interrupts	none

This is a driver for 802.11 devices. This driver interfaces with the smxWiFi MAC stack and typically uses the smxUSBH host stack.

Configuration

Configurable parameters may be modified using the device driver ioctl() function.. Following the call to Portinit() that performs basic initialization on the WiFi interface, the interface is in an idle state and it will not attempt to make a connection. During this time, the SSID and security parameters may be set.

Set SSID

```
#define SSID “MDIWireless”
nets[WFNET].protoc[ussDriverIndex]->ioctl(&nets[WFNET],
ussWifiSsidSetE, (void *)SSID, 0);
```

Set WPA Pre-Shared Key authentication

```
struct AUTHMODE tests;
tests.iAuthMode = SWF_AUTH_MODE_WPAPSK;
tests.iEncrypt = SWF_ENCRYP_AES;
nets[WFNET].protoc[ussDriverIndex]->ioctl(&nets[WFNET],
ussWifiAuthModeSetE, (void *)&tests, 0);
swf_GenerateWPAKey("1234567890", SSID, WPAKey);
```

```
nets[WFNET].protoc[ussDriverIndex]->ioctl(&nets[WFNET],
ussWifiWPAKeySetE, (void *)WPAKey, 0);
```

Once these are in place, an explicit ioctl() command brings up the WiFi interface as follows.

```
nets[WFNET].protoc[ussDriverIndex]->ioctl(&nets[WFNET],
ussInterfaceBringUpE, (void *)0, 0);
```

Note that the MAC address for the Ethernet interface is supplied by the external hardware.

Here is a sample configuration line for the WiFi interface:

```
Portconfig("eth0", "DRIVER", "WIFINET");
```

Sending

The WiFi driver uses the API to the smxWiFi MAC stack to write frames.

Receiving

The WiFi driver provides a received data callback function that is called from a task in the smxWiFi MAC stack. The WiFi driver registers the callback function once the smxWiFi stack reports the interface is in the inserted state.

State Information

When the WiFi link is up, the value of nets[n].bps is non-zero, otherwise it is 0. Here n is the index of the network interface. When link speed information is available, the value is passed through to the bps field, otherwise a minimum value of 1 Mbit is stored when the link is up.

E. Serialized MAC Addresses

When a system goes into production, each Ethernet interface needs to have a unique MAC address. The MAC address space is administered by the IEEE, which assigns Organizationally Unique Identifiers (OUI) to companies as needed. The OUI is used for the first three bytes of the MAC address, and the following three bytes can be serialized. If your organization does not already have a block of addresses that can be applied to network products, you can contact the IEEE to obtain a new block of addresses. The IEEE web site also has a page where you can do an online search for your organization to see if you already have a block of addresses, and who the organization contact person is for this assignment.

smxNS is set up to use a dummy MAC address for development. This address is specified by calling the Portconfig() function with the "MAC" property. Once development has reached a certain stage, the value for the MAC address should be made unique for each network interface by providing a unique string to the Portconfig() call. This can be based on a serialized value that is found somewhere in the memory map or is read in a platform-specific way. You will need to implement this feature earlier in development if you will be running multiple smxNS systems on the same network, since duplicate MAC addresses will make the ARP mapping between MAC addresses and IP addresses unreliable.

F. Memory Usage and Performance

Memory Usage (KB)

Component	ARM Thumb		ARM		ColdFire	
	RAM	ROM	RAM	ROM	RAM	ROM
Core Files	2.0 + 10.0	25.6	2.0 + 10.0	37.1	2.4 + 11.4	48.8
DPI API	0.0	2.2	0.0	3.3	0.1	3.3
Socket API	0.1	4.4	0.1	6.4	0.3	6.3
DHCP c	0.1	3.2	0.1	4.5	0.1	5.0
DHCP s	2.0 + 0.1	5.7	2.0 + 0.1	8.2	2.4 + 0.3	8.9
FTP c	1.0 + 0.6	3.4	1.0 + 0.6	4.5	2.4 + 0.7	7.0
FTP s	2.7 + 0.1	3.2	2.7 + 0.1	4.2	2.7 + 0.3	4.7
HTTP c	1.0 + 1.5	3.2	1.0 + 1.5	3.8	2.4 + 1.5	5.6
IGMP	0.4	2.9	0.4	3.8	0.8	4.8
NAT	0.4	4.0	0.4	8.7	0.5	3.5
POP c	tbd + 0.0	1.6	tbd + 0.0	2.3	tbd + 0.1	4.0
PPP	8.0	15.4	8.0	25.1	2.6	22.4
PPPoE c	8.5	18.6	8.5	29.7	3.0	27.3
PPPoE s	8.4	18.8	8.4	29.9	2.9	27.4
SMTP c	tbd + 0.0	2.1	tbd + 0.0	2.8	tbd + 0.1	4.4
SMTP s	tbd + 0.3	1.8	tbd + 0.3	2.5	tbd + 0.4	4.3
SNMP v2	2.7 + 4.0	15.2	2.7 + 4.0	22.4	2.7 + 4.7	18.4
SNMP v3	2.7 + 9.5	25.8	2.7 + 9.5	39.0	2.7 + 10.4	30.2
Telnet Server	0.6 + 0.0	0.7	0.6 + 0.0	0.7	1.2 + 0.0	1.3
Web Server	2.7 + 7.0	12.3	2.7 + 7.0	16.9	3.0 + 8.0	19.3

Notes

- In the RAM columns, the first number in xx + yy is the approximate stack size for a multitasking system. Otherwise, in a non-multitasking system, ignore those and assume about 3 to 4KB extra stack depth. Some applications, such as the web server, have extra deep stack needs if features such as web form processing are used. tbd indicates we have not yet measured the size.
- These memory requirements are typical for a system that services one active TCP session at a time.
- Core Files includes support for TCP, UDP, IP, ICMP, and an Ethernet driver.
- Socket API and DHCP support are commonly used but listed separately.
- Support for IP fragmentation and reassembly is included.
- Support for IP Options Headers is not included.
- PPPoE client and server values include PPP
- For each additional active session, smxNS should be configured with NCONNS increased by 1, NCONFIGS by 1, and NBUFFS by 5. So each active session (client or server) adds about 8KB to the RAM requirement.

Performance

Family	Processor	MHz	Memory	Ethernet	TCP S	TCP R
ARM7	AT91SAM7X256	48	SRAM	on-chip	1315	827
ARM7	LPC2468	72	SRAM	on-chip	966	725
ARM9	AT91SAM9260	210	SDRAM	on-chip	857	532
CF	MCF5282	64	SDRAM	on-chip	1131	845
CF	MCF5329	240	SRAM	on-chip	1768	3276
X86	VIA C3	800	SDRAM	Intel i825xx	6687	6375

Notes

- MHz is the clock speed we tested, not necessarily the rated speed of the processor.
- SRAM is internal memory on the processor, and SDRAM is external memory.
- VIA C3 may be equivalent to 266MHz Pentium II.
- Benchmarking performed using nuttcp version 5.3.1.

In order to run the TTCP test on an smxNS system, follow these steps:

1. To enable the TTCP server in smxNS, set
   #define TEST_TTCP_SERVER 1
   in nsdemo.c
2. Obtain the latest TTCP Windows application at http://www.nuttcp.net/
3. Build and run the smxNS application, then launch the Windows TTCP application

Sample command lines follow. The IP address is that of the smxNS system. Use the -r flag to have the smxNS system source the traffic, leave it out to have the smxNS receive the traffic. The –l flag specifies the application level write size. For good performance, this can be set to the TCP MSS (typically 1460 bytes).

```
C:\bin\ttcp\nuttcp-5.5.5.win32>nuttcp-5.5.5 -l1460 10.1.1.100
9.7020 MB /  10.12 sec =    7.6600 Mbps 5 %TX 0 %RX

C:\bin\ttcp\nuttcp-5.5.5.win32>nuttcp-5.5.5 -r -l1460 10.1.1.100
9.4611 MB /  10.00 sec =    7.9334 Mbps 0 %TX 3 %RX
```

Index

#

#echo META command
  description, 241
  example, 241
#exec META command
  description, 242
  example, 242
#include META command
  description, 243
  examples, 243
#memory META command
  description, 244
  examples, 244
#system META command
  description, 244

_

_inb() macro
  and character drivers, 257
  and init(), 261
  and shut(), 262
_outb() macro
  and character drivers, 257
  and init(), 261
  and shut(), 262

A

accept record, 185
accept() BSD function, 75
  example, 75
access configuration file
  example, 224
access, restricting, 207
access.cfg file
  example, 224
ACT10100 controller, driver info, 297
address conflict detection, 281
AddType command
  description, 220
  example, 220
agent
  definition, 165
  design of, 166
  running, 174
AGENT_CONTEXT structure, 174
AJAX, 245
application
  beginning, 17
  developing, 17
application development, 31
architecture, segmented, 251
ARP, 108
ARP cache, 282
ARP caching, 282
ARP table, 282
arrive queue, 251
AT91 controller, driver info, 297
authenticating user, 39
authentication, 170
authentication of user, 208, 210
AutoIP, 110

B

baud rate
  for I8250, 260
bind() BSD function, 76
  example, 76
block drivers
  description, 263
broadcasting
  example, 67
browser
  printing to, 241
BSD, 17
BSD functions
  accept(), 75
  bind(), 76
  closesocket(), 77

connect(), 78
fcntlsocket(), 79
for connectionless protocol, 74
freeddrinfo(), 79
gai_strerror(), 80
getaddrinfo(), 81
getpeername(), 83
getsockname(), 84
getsockopt(), 85
inet_ntop(), 87
inet_pton(), 88
ioctlsocket(), 89
listen(), 90
readsocket(), 91
recv(), 92
recvfrom(), 94
recvmsg(), 95
return values, 74
selectsocket(), 96
send(), 98
sendmsg(), 100
sendto(), 101
shutdown(), 102
socket(), 103
typical calling sequences, 73
writesocket(), 104
BSD socket interface, 71
BSD sockets
writing new code, 71
BSD Sockets vs. Dynamic Protocol Interface, 43
buffer space, 3
buffers
code for checking, 18
code for constructing, 17
code for server.c, 21
setting number available, 36
building
example Web Server for target, 198
buildpg.cfg file
editing, 198
example, 214
bulk request, 169
Bwrite() user server function, 200

C

CAget() function, 182
example code, 182
CAindex() function
example code, 184
CAR MIBVAR option, 182, 188
CAW MIBVAR option, 182, 188
CFFEC controller, driver info, 298
CGI
definition, 225, 283
CGI environment variables, 237
CGI function programming interface, 225
CGI programs
running, 242
CGI routines, 231
escape_char(), 231
hextochar(), 232
Nmakeword(), 232, 236
plustospace(), 233
splitstr(), 234
subchar(), 233
summary list, 231
unhex_str(), 236
CGI support routines
calling, 225
findvar(), 227
general description, 225
getvar(), 229
Ngetenv(), 229
send_file(), 230
summary list, 226
CHAP, 138
CHAP, definition, 283
character drivers
description, 255, 257
chksum_INASM Macro, 38
CHOICE MIBVAR option, 182
client
data collection loop, 23
defining, 18
FTP, 117
required features, 19
role of, 18
slow start, 281
Telnet, 135

terminating smxNS, 23
client.c file
compiling, 24
structure, 23
closE() routine
description, 260
closesocket() BSD function, 77
code
reentrant, 4
ROMmable, 4
source, 4
code requirements, 166
code size, 166
comec() routine
accessing, 258
and irhan() function, 257
for reading data, 256
compiler, 166
compiling
application, 24
ConfDel() routing table function
description, 295
configuration, 33
build settings, 34
start up example, 260
configuration parameters, 33
configuring, 213
congestion control, 280
connect() BSD function, 78
example, 77, 78
connecting from browser, 199
connection
establishing, 20
connections
accepting on sockets, 75
active open, 51, 54
example, 55
closing, 56
general description, 51
initiating on a socket, 78
listening for, 90
opening, 54
passive open, 51, 54, 67
example, 55
receiving messages from, 57, 94
shutting down, 102
writing messages to, 58
constants
ENABLEAUTHENTRAPSVAL, 168
ENTERPRISE, 167
MAXKEY, 168
MAXOID, 168
MAXVAR, 169
control parameters
setting for socket, 89
CS8900 controller
block driver, 263

D

data
incoming, and block driver, 263
initialized, 166
outgoing, and block driver, 263
reading, and character drivers, 256
sending, and character drivers, 256
transfer between controller and application, 256, 263
transfer to and from agent, 166
data collection loop, 23
data structures, 247
fd_set, 96
include files needed for, 72
MESSH, 247, 248
MIBTAB, 180
MIBVAR, 180
msghdr, 95, 100
NET, 247, 249
request_rec, 212
sockaddr, 72
sockaddr_in, 72
timeval, 96
DC21140 controller, driver info, 298
Debug over Telnet, 287
arpstat, 287
bufstat, 288
ifstat, 289
logdump, 290
memdump, 290
netstat, 290
nqstat, 291
routestat, 292
debugging techniques, 285

departure queue, 251, 258
design considerations, 277
developing first application, 17
development
  application, 31
device driver macros
  QUEUE_FULL(), 252
  QUEUE_IN(), 251
  QUEUE_OUT(), 253
  SNS_DISABLE(), 250
  SNS_ENABLE(), 250
device drivers, 4, 247, 297
  bad parameters, 255
  called from NPTABLE, 262
  code you write, 256
  format, 255
  interface, 247
  restoring interrupt, 251
  support functions, 250
  using struct NET, 249
  writing your own, 255
DHCP, 109
  definition, 283
  description, 109
DHCP client configuration, 109
DHCP lease time, 110
DHCP server configuration, 111
DHCP testing, 113
DHCPget() routine, 109
DHCPrelease() routine, 109
dial on demand, 146
directory structure, 10
DMA, 257
DNS, 116
  definition, 283
DNS macro, 38
DNSresolve() function
  example, 117
DNSresolve() routine, 116
documentation, 10
documents
  determining encoding of, 209
  determining type, 209
  finding, 209
domain name, getting, 116
DPI, 43
  definition, 283
DPI (Dynamic Protocol Interface), 17
driver.txt file, 247
dynamic protocol functions
  Nclose(), 56
  Ninit(), 44
  Nopen(), 54
  Nread(), 57
  Nterm(), 45
  Nwrite(), 58
  Portinit(), 45, 46, 49
  Portstate(), 50
  Portterm(), 51
Dynamic Protocol Interface, 43
  blocking mode, 44
  non-blocking mode, 44
  overview, 43
Dynamic Protocol Interface macros
  SOCKET_BLOCK(), 60
  SOCKET_CANSEND(), 61
  SOCKET_FIN(), 63, 64
  SOCKET_HASDATA(), 60
  SOCKET_ISFATAL(), 61
  SOCKET_ISOPEN(), 60
  SOCKET_ISSENDING(), 61
  SOCKET_LOCADDR(), 62
  SOCKET_LOCLINKADDR6(), 64
  SOCKET_LOCPORT(), 63
  SOCKET_LOCSITEADDR6(), 64
  SOCKET_MAXDAT(), 62
  SOCKET_NOBLOCK(), 60
  SOCKET_PUSH(), 63
  SOCKET_REMADDR(), 62
  SOCKET_REMADDR6(), 64
  SOCKET_REMPORT(), 63
  SOCKET_RXTOUT(), 62
  SOCKET_TESTFIN(), 61
  summary list, 59
Dynamic Protocol Interface vs. BSD Sockets, 43

E

Email test, 14, 15
embedded web server
  including pages, 221
  request process, 205

encoding, determining type, 209
end of table, 185
ENTRY structure
  definition, 201
  finding and returning, 201
EOF for network stream, 203
EP93xx controller, driver info, 300
errno
  and BSD functions, 74
error codes, 255
errors, logging, 211
escape_char() CGI routine
  description, 231
Ethernet
  drivers provided, 247
  using block driver, 263

F

fcntlsocket() BSD function, 79
fd_set structure, 96
file transfer
  example, 67
FILE_SUPPORT macro, 39
files
  receiving, 118
  sending, 118
  writing to network, 230
files, checking for, 209
findvar() CGI support routine
  description, 227
  example, 228
firstapp.h file, 19
flow control, 51, 278
  for congestion, 280
fragmentation, 37
FRAGMENTATION macro, 37
freeddrinfo() BSD function, 79
FTP
  definition, 283
  description, 117
  example, 118
FTP Client test, 12
FTP Server test, 13
FTPget() routine, 118
  examples, 119
FTPput() routine, 118
FTPserv() routine, 117
functions
  MIB.index(), 184
  MIB.set(), 182
  ussSNMPAgentCheck, 176
  ussSNMPAgentCheck(), 182
  ussSNMPAgentInit, 176
  ussSNMPAgentShut, 177
  ussSNMPAgentTrap, 177

G

gai_strerror() BSD function, 80
getaddrinfo() BSD function, 81
  example, 82
GetEntry() user server function, 201
gethostbyname_r() BSD function
  example, 79
getpeername() BSD function, 83
  example, 80, 83
getsockname() BSD function, 84
  example, 84
getsockopt() BSD function, 85
  example of retrieving errno, 74
getvar() CGI support routine
  description, 229
getword() CGI routine
  example, 234
goingc() routine
  and irhan() function, 257
  for sending data, 256

H

handlers
  included, summary list, 206
hardware
  configuring, 39
  parameters, 260
header files, including, 19
hextochar() CGI routine
  description, 232
host name, 39
hosts, 174
HTML META commands
  definition, 283
HTTP client, 15
HTTP server

modules, 206
structure, 204
HTTP, definition, 283
HTTPdisplay() routine, 120
HTTPget() routine, 120, 132
httpinit() user server function, 202
HTTPservinit() user server function, 202
httpterm() user server function, 203

I

I/O
mapping addresses, 251
I386 processor
interrupt handling capacity, 257
I8250 processor
initialization parameters, 260
NPTABLE example, 262
I8255X controller, driver info, 301
ICMP protocol, 55
identifying user, 39
IGMP, 120, 121, 122
BSD API, 105
DPI API, 65
implementation considerations, 277
include files, 19, 44
inet_ntop() BSD function, 87
example, 87
inet_pton() BSD function, 88
example, 88
init() routine
description, 260
init_char_driver, 270
initialization
and Ninit(), 44
initializing, 202
initializing smxNS, 19
functions required, 19
Internet standard MIBs, 180
interrupt addresses
character drivers, 260
interrupt handler, 247, 254
example for character drivers, 257
example with block driver, 264
example without transmit interrupt, 264
installing, 251
irhan() description, 257
with writE(), 258
interrupt number
for I8250, 260
interrupt shells
for block drivers, 264
for character drivers, 257
interrupt vectors
installing, 251
restoring, 251
interrupts
support, 247
ioctlsocket() BSD function, 89
IP address, getting, 116
IP_MC_DFLT_NETO macro, 37
IPOPTIONS macro, 37
IPv6, 121
irhan() function
and block drivers, 264
and character drivers, 257
example, 264
example for character drivers, 257
IRinstall() function, 251
and init(), 260
description, 251
IRrestore() function
and shut(), 261, 274
description, 251
ISMAP, 225
definition, 283

J

jQuery, 245

K

KEEPALIVETIME macro, 38
key, 168
maximum length, 168

L

LAN91CXXX controller, driver info, 302
LCP Phase, 138
link layer
called from NPTABLE, 262
link local address, 110

listen() BSD function, 90
  example, 90
LM3S controller, driver info, 303
local address
  getting for socket, 84
local parameters, 34
LOCALHOSTNAME macro, 39
logging errors and access, 211
Loopback test, 13
LPC2xxx controller, driver info, 304
LTEST, 13
  goals, 14
  pass indicators, 14

M

MAC address, serialized, 310
macros
  Dynamic Protocol Interface, 59
  for device drivers, 250
manager, definition, 165
manuals, 10
mapioadd() routine, 251
Maxbuf parameter, 57
MAXKEY() constant, 168
MAXKLEN() constant, 168
MAXOID() constant, 168
MAXVAR() constant, 169
mDNS Responder, 122
memory
  printing size of, 244
memory usage, 311
message buffers, 248
messages
  adding to a queue, 251
  broadcasting, 67
  reading from a connection, 57
  receiving, 92, 95
  receiving from connection, 94
  receiving from socket, 91
  removing from a queue, 253
  sending, 98, 100, 101
  sending to socket, 104
  writing to a connection, 58
MESSH structure
  uses, 247
META commands
  #echo, 241
  #exec, 242
  #include, 243
  #memory, 244
  #system, 244
  arguments accepted, 240
  format of, 240
  general description, 240
  summary list, 240
MIB
  application-specific variables, 185
  custom, 166
  data, 180
  definition, 165
  standard, 180
  supplied, 180
  translation, 185
  user-defined, example, 189
MIB files, 188
MIB structure, 180
MIB table, 181
  end of, 185
MIB translator
  building, 186
  overview, 185
  running, 186
MIB.index() function, 184
MIB.set() function, 182
MIBTAB structure, 180
MIBTOC
  and adding variables, 185
  and MIB translation, 185
  arguments, 186
  building MIB translator, 186
  output files, 187, 188
  running MIB translator, 186
MIBVAR structure, 180
  read/write notification, 188
  record options, 181
MIME types
  adding to server, 220
MIME types file, 219
  example, 219
MIME, definition, 283
MODchkaccess() module/function, 207
modularity, 4

mouse click, 225
MS-CHAP, 150
msghdr structure, 100
  definition, 95
MTU macro, 36
multicast, 120, 121, 122
  BSD API, 105
  DPI API, 65
Multicast, 120, 121

N

names
  binding to sockets, 76
NAPT, 126
NAT, 126
NAT configuration, 126
NBUFFS macro, 36
Nclose() function, 22
  and closE(), 260
  description, 56
  example, 56
NCONNS macro, 36
NC-SI, 128
NDNSS macro, 38
NE_HWERR error code, 255
NE_PARAM error code, 255
NE2000 controller, driver info, 304
Neof() user server function, 203
NET structure
  code example, 249
  description, 249
  uses, 247
net.h file
  contents, 19
netdata[] table
  and initialization, 19
network
  buffered write to, 200
  initialization, 44
  initializing interfaces, 45, 46, 49
  shutting down, 45
  shutting down interfaces, 50, 51
  turning off, 274
  writing files to, 230
Network Address Translation, 126
network analyzers, 292
network applications, 107
network configuration table, 39
network controller
  drivers provided, 247
  interrupt, 264
  turning off, 261
  using for hostname, 39
network interfaces
  initializing, 45, 46, 49
  shutting down, 50, 51
network stream
  finding EOF, 203
networking application routines
  DHCP, 109
  FTP and TFTP, 117
  HTTP, 119
  SLIP, 131
  SMTP, 132
  SNTP, 134
  summary list, 107
  Telnet, 135
networking stack, 166
Ngetenv() CGI support routine
  description, 229
  example, 227
Ninit() function
  and initialization, 19
  description, 44
  example, 45
Nmakeword() CGI routine
  description, 232, 236
NNETISRS macro, 40
NNETS macro, 40
non-blocking operations
  example, 68
Nopen() function, 20
  and opeN(), 259
  description, 54
  examples, 55
  parameters, 21
Nportno() function, 23
NPTABLE, 255
  description, 262
  example, 262
  structure definition, 262
Nread() function, 20, 21, 57

description, 57
example, 57
parameters, 22
nsbldpg utility
files generated, 213
files read, 213
using, 197
nscfg.h, 34
configuration options, 34
nscfg.h file, 11
and protocol selection, 33, 40, 41
contents, 19
contents and location, 33
nsclient.h, 28
nscs.h, 24
nsdemo, 2, 11
nsdemo.c, 11
nsserver.c, 25
nstels.c, 3
Nterm() function, 22
description, 45
example, 45, 46, 48
null modem, 147
Nwrite() function, 20, 21, 58, 267
and writE(), 258, 267
description, 58
example, 58
parameters, 22

O

object identifier (OID), 168, 181
open
active, 51
passive, 51, 75, 90
opeN() routine, 259
options, 188

P

packets
exchanging, 278
short, 280
page configuration file
description, 221
example, 222
pages.cfg file
example, 222
PAP, 138
parsing URLs, 211
passive open, 20, 75, 90
definition, 283
PASSWD macro, 39
passwords, 170
performance, 311, 312
PHY, 254
plustospace() CGI routine
description, 233
example, 234
POP, definition, 283
port address
device, 261
for I8250, 260
port numbers, 51
example, 55
Portinit() function
and init(), 260
and initialization, 19
description, 45, 46, 49
examples, 49
Portstate() function
description, 50
Portterm() function, 22
and shut(), 261
description, 51
examples, 50, 51
PPP
configuration, 140, 161
dialapi, 160
ioctl, 154
NCP phase, 138
pppsig, 162
routing, 150
scripting, 143
PPPoE, 129
PPPoE configuration, 129
processor-independent agent, 166
processors, 166
protocol stack, 52
and opening connections, 52
with block drivers, 264
with character drivers, 257
protocol table, 275
structure definition, 262

protocols, 107
  link-level, 4
  selecting, 40, 41
Proxy ARP, 108

Q

QUEUE_EMPTY() macro
  description, 253
  example, 254
QUEUE_FULL() macro, 259
  description, 252
  example, 253
QUEUE_IN() macro, 259
  description, 251
  examples, 252
QUEUE_OUT() macro, 253
  description, 253
  example, 253
queues
  adding messages to, 251
  removing messages from, 253
  testing if empty, 253
  testing if full, 252

R

RAM, 166
  fixed, 3
read notification, 182
reaD(), 262
Read/Write Notification, 188
readsocket() BSD function, 91
recv() BSD function, 92
  example, 93
recvfrom() BSD function, 94
  example, 94
recvmsg() BSD function, 95
RELAYING macro, 38
remote address
  getting for a socket, 83
request structure
  description, 204
  example, 212
  loop, 204
  searching for variables, 229
request to web server
  process, 205
request_rec structure, 212
requirements, 198
RFC 5227, 281
ROM, 4
routing table configuration functions
  SetLocalIP, 295
RTL8139 controller, driver info, 306

S

screen(), 262
security, 170
selectsocket() BSD function, 96
  example, 97
send() BSD function, 98
  example, 98
send_file() CGI support routine
  description, 230
  example, 228
sendmsg() BSD function, 100
sendto() BSD function, 101
  example, 101
SEQUENCE OF, 181
serial drivers
  and character drivers, 255
  provided, 247
serial FIFO buffer, 257
server
  defining, 18
  required features, 19
  role of, 18
  terminating smxNS, 22
server configuration
  application system information, 216
  directory and file system variables, 218
  MIME types, 219
  other files, 215
  page configuration file, 221
  server information variables, 217
server configuration file
  contents, 213
  example, 214
server.c file
  and include files, 19
  code to add, 21
  compiling, 24

servers
  FTP, 117
  starting, 117
  Telnet, 135
SetDNS() routine, 116
setsockopt() BSD function
  example, 86
shut() routine, 274
  description, 261, 274
  example, 261, 274
shutdown() BSD function, 102
silly window syndrome, 281
sizes, 311
skip, 185
sliding window, 278
  for flow control, 278
SLIP, 131
  and Windows, 131
SLIP program
  description, 131
SMTP, 132
SMTP, definition, 283
smxNS
  design, 3
  overview, 1
smxns.h, 19
SNMP
  configuration, build-time, 167
  constants, 167
  design, 166
  introduction, 165
SNMP Agent
  customizing, 180
SNMP Agent test, 15
SNMP Manager, 195
snmp.h file, 180, 188
SNMPAgent
  MIB
    configuring, 180
snmpconf.h file, 171
SNS_BUFFS_IN_SRAM macro, 35
SNS_CPU_CACHE_DATA macro, 35
SNS_DEBUG_LEVEL macro, 9, 40
  and application development, 31
  and testing, 11
SNS_DISABLE() macro
  description, 250
SNS_ENABLE() macro
  description, 250
SNS_HW_RX_CHECKSUM macro, 35
SNS_HW_TX_CHECKSUM macro, 35
SNS_MIN_RAM macro, 35
sns_SntpGet() routine, 134
SNTP, 134
SNTP program
  description, 134
sockaddr structure, 72
sockaddr_in structure, 72
socket interface, 71
socket() BSD function, 103
  example, 103
SOCKET_BLOCK() macro
  description, 60
SOCKET_CANSEND() macro
  description, 61
SOCKET_FIN() macro
  description, 63, 64
SOCKET_HASDATA() macro
  description, 60
SOCKET_ISFATAL() macro
  description, 61
SOCKET_ISOPEN() macro
  description, 60
SOCKET_ISSENDING() macro
  description, 61
SOCKET_LOCADDR() macro
  description, 62
SOCKET_LOCLINKADDR6() macro
  description, 64
SOCKET_LOCPORT() macro
  description, 63
SOCKET_LOCSITEADDR6() macro
  description, 64
SOCKET_MAXDAT() macro
  description, 62
SOCKET_NOBLOCK() macro
  description, 60
SOCKET_PUSH() macro
  description, 63
SOCKET_REMADDR() macro
  description, 62
SOCKET_REMADDR6() macro

description, 64
SOCKET_REMPORT() macro
description, 63
SOCKET_RXTOUT() macro
description, 62
SOCKET_TESTFIN() macro
description, 61
sockets
accepting connections on, 75
binding names to, 76
blocking, 60
closing, 77
controlling flags, 79
creating, 103
getting local address for, 84
getting options, 85
getting remote address for, 83
initiating a connection on, 78
non-blocking, 60, 77, 79, 99
receiving messages, 91
sending messages to, 104
setting control parameters for, 89
setting options, 85
waiting for activity on, 96
splitstr() CGI routine
description, 234
standard MIB, 180
strings
parsing, 232, 234
searching for, 229
structures
AGENT_CONTEXT, 174
allocating space for, 202
ENTRY, 201, 230
MIB, 180
request, 204
STRxxx controller, driver info, 306
subchar() CGI routine
description, 233
support.h file, contents of, 19
SVA
and #echo command, 241
definition, 283
SYSCONTACT variable, 167
SYSDESCR variable, 167
SYSLOCATION variable, 167
system group, 167
system information
printing, 244

T

target system, 2
design, 9
TCP
and flow control, 51
compared with UDP, 51
definition, 284
delayed ACKs, 280
file transfer example, 67
flow control, 278
retransmission, 277
timeout, 277
TCP delayed ACK, 280
TCP vs. UDP, 17, 40
TCP window probe, 281
TCP/IP, 277
and relaying, 38
embedded, 277
protocol relationships, 2
protocols supported, 1
size, 3
user interface, 71
TCP_SACK macro, 39
Telnet
description, 135
programs, 135
Telnet Server test, 15
terminating smxNS, 22
terminology, 283
test programs, 11
TFTP
definition, 284
description, 117
example, 118
TFTPget() routine, 118
TFTPput() routine, 118
TFTPserv() routine, 117
timeval structure, 96
trace
and LTEST, 13
and LTEST results, 14
displaying output, 285

field definitions, 286
trace output for file transfer, 286
translating URLs, 211
transmission
timeout, 277
transmit routine
description, 267
examples, 259, 268
writeE(), 258
transmitter empty, 257, 258
transmitting
routines for, 267
transport layer, 166
Transport Mapping, 193
traps, 174
sending, 189
sending data, 179
types, 177, 179
TTCP test, 312

U

UDP
compared with TCP, 51
definition, 284
UDP vs. TCP, 17, 40
unescape_url() CGI routine
example, 234
unhex_str () CGI routine
description, 236
UNIX
sockets, 71
URL
translating and parsing, 211
USBD, driver info, 307
USBH, driver info, 307
user
authenticating, 39
identifying, 39
user authentication, 208, 210
user server functions
Bwrite(), 200
GetEntry(), 201
httpinit(), 202
HTTPservinit(), 202
httpterm(), 203
Neof(), 203
summary list, 200
waitreq(), 203
User-based Security Model, 169
USERID macro, 39
USMETA programming interface, 240
USNET, 185
USS_IP_MC_LEVEL macro, 37
USS_PROXYARP macro, 39
USSBUFALIGN macro, 36
ussHostGroupJoin, 65
ussHostGroupLeave, 65
ussSNMPAgentCheck() function, 176
ussSNMPAgentInit() function, 176
ussSNMPAgentShut() function, 177
ussSNMPAgentTrap() function, 177

V

variable bindings, 179
variable configuration file
description, 223
example, 223
variable structure
searching for string, 227
variables
maximum number, 169
writable, 188
vartable.cfg file
example, 223
Verification Testing, 293
version, 10
View-based Access Control
Configuration, 170

W

waitreq() user server function, 203
WD8003 controller
receiving messages, 259
web pages
inserting into web server, 197
steps for creating, 197
web server, 293
web server modules
description, 206
web server modules/functions
MODchkaccess(), 207
sequence of use, 206

summary list, 206
Web Server test, 16
WiFi, driver info, 308
window
exhausted, 279
for flow control, 278
silly window syndrome, 281
windows
utilities, 293
Wireshark, 292
writable variable, 188
write notification, 182
writE() routine
and block drivers, 267
description, 258
example, 259
write, buffered, 200
writesocket() BSD function, 104