NicheStack Reference Manual - IPv4 and IPv6

Interniche Legacy Document

Version 1.00

Date: 18-May-2017 10:49

All rights reserved. This document and the associated software are the sole property of HCC Embedded. Reproduction or duplication by any means of any portion of this document without the prior written consent of HCC Embedded is expressly forbidden.

HCC Embedded reserves the right to make changes to this document and to the related software at any time and without notice. The information in this document has been carefully checked for its accuracy; however, HCC Embedded makes no warranty relating to the correctness of this document.

Table of Contents

Product Overview 10
Terms and Conventions 10
What a Port is 11
Requirements 12
Memory Requirements 12
CPU Requirements 12
Operating System Requirements 13
TCP/IP Stack Source Code Organization 14
Portable Files vs. Port Dependent Files 14
Target System Independent Directories 14
Target System Dependent Directories 15
Build Directories 15
System Architecture 16
Operating System Interface - porting macros 16
Memory Organization and Buffer Management 16
Buffer management functions 16
int pk_init(int len, int num); 16
PK_ALLOC(pkt, n) 17
PK_ALLOC_DONOTLOCK(pkt, int size) 17
PK_CONTIG(pkt, n) 17
PK_CONTIG_DONOTLOCK (PACKET pkt, int size) 17
PK_FREE(PACKET pkt) 17
PK_FREE_DONOTLOCK (PACKET) 17
System Initialization 18
Modules 18
The 'prep' function 18
The 'init' function 18
The 'start' function 19
The 'close' function 19
The 'menu' function 19
Task initialization 20
userdata.c 20
NET_DEVICE 20
NET_MODULES 20
NET_BUFQ 20
userdata.c support functions 21
Module Initialization 21
_rc files 23
Ethernet Device Driver 24
Device Driver API 24
Handling Packets within a Device Driver 27
Before Contacting Support 28

- Porting Guide 29
  - The InterNiche Source Code Tree 29
    - Target-Specific System Directory 29
    - Windows Drivers 30
    - Makefiles 30
    - System Configuration Files 31
    - ipport.h 31
    - Target Specific System-level Macros and Definitions 32
    - Standard Macros and Definitions 32
    - System-Level Size definitions 33
    - Mutual Exclusion 34
      - Critical Section Method 35
      - Net Resource Method 36
        - Net Resource Method, Walkthrough 37
    - Debugging Aids 38
    - Timers and Multitasking 39
    - Stack Features and Options 41
    - Package Options 41
      - Configuration Example 42
    - Error Codes 43
  - The 'glue' Layer 44
  - Task Control 44
    - The SuperLoop Method 44
      - TK_YIELD() 45
      - Design Rules 45
    - Multitasking 46
  - Data Structures 47
    - The netbuf Structure and the Packet Queues 47
    - The nets[ ] Array and the netlist 50
    - The IfMib Structure 53
  - Initialization 54
    - Initialization of net Structure IP Addressing Fields 54
    - Initialization of the Packet Buffer Queue Sizes 55
    - Initialization of the IP and TCP Layers: ip_startup 55
    - IPv4 address configuration protocols: DHCP and auto-configuration 55
    - Initialization of Application Servers 57
  - Servicing the Stack 58
  - Applications and Testing 59
- NicheTask Multitasking Scheduler 60
  - Source Files 60
  - osport.h Data Types 60
  - Task Stacks 60
  - The Task Control Structure 61
  - General Task Behavior 61
  - Interrupts and Tasks 62
  - TK_ Macro Definitions 63

TK_ Task Control Definitions 63
Tasking System Designs - Spinning vs. Event-blocking 63
Priorities 64
The TK Macros 65
create_device 66
eth_prep 67
get_pticks 68
TK_BLOCK 69
TK_CREATE 70
TK_DELETE 71
TK_RESUME, TK_WAKE 72
TK_SIGNAL 73
TK_SIGNAL_ISR 74
TK_SIGWAIT 75
TK_SLEEP 76
TK_SUSPEND 77
TK_YIELD, tk_yield 78
User Tasking Functions 79
tk_block () 80
tk_del() 81
tk_exit () 82
tk_init_os 83
tk_kill 84
tk_new 85
tk_res_lock 87
tk_res_unlock 88
tk_sleep() 89
tk_start_os 90
tk_stats 91
tk_suspend() 92
tk_wake 93
Low-Level Routines 94
tk_frame 95
tk_getsp 96
tk_getspbase 97
tk_switch 98
Porting Engineer Provided Functions 99
General Functions 100
npalloc, npfree 100
tcp_sleep 102
SignalPktDemux 103
dtrap 104
dprintf 105
console_only 106
dputchar 107
kbhit 108

getch 109
ENTER_CRIT_SECTION, EXIT_CRIT_SECTION 110
LOCK_NET_RESOURCE, UNLOCK_NET_RESOURCE 111
cksum 112
panic 113
sysuptime 114
Network Interfaces 115
n_close 116
n_init 117
n_refill 119
n_reg_type 120
n_setstate 121
n_stats 122
netbuf Manipulation 123
netbuf Allocation 123
netbuf Initialization 123
Copy Received Data to netbuf Packet Buffer 124
Enqueuing netbuf structure to rcvdq 124
Signal Packet Demultiplexor 124
pkt_send 125
raw_send 128
rcvdq 130
TCP Sleep 131
Internal Functions 132
Chained Buffers 133
pk_free 134
pk_init 135
pk_copy 136
pk_gather 137
ARP 138
make_arp_entry 138
IP 139
add_route 140
ip_mymach 141
iproute 142
parse_ipad 143
print_ipad 144
ICMP 145
icmpEcho (Copy) 146
UDP 147
udp_alloc 148
udp6_alloc 149
udp_free 150
udp_open 151
udp6_open 152
udp_send 153

- udp6_send 154
- udp_close 155
- misclib 156
- DNS Client 156
- DNS Client API 157
- gethostbyname 158
- gethostbyname2 159
- nslookupr 160
- getaddrinfo 161
- freeaddrinfo 162
- getnameinfo 163
- in46_reshost 164
- dns_update 165
- inet_ntop (Copy) 166
- inet_pton (Copy) 167
- Syslog Client 168
- Integration Notes 168
- Interoperability Notes 168
- Usability Notes 168
- Syslog API 170
- syslog, openlog, closelog, setlogmask 172
- openlogaddr, closelogfac 175
- Semaphores 176
- tk_sem_pend 177
- tk_sem_post 178
- Mutexes 179
- tk_mutex_pend 180
- tk_mutex_post 181
- Introduction to NicheStack IPv6 182
- Bigger IP address 182
- Header Layers 183
- Hardware Limits 183
- Datagram Size 183
- PMTU 183
- Routers Do Not Fragment 183
- Multicast Addressing 184
- Pseudo Checksum 184
- Interface Addresses 185
- Neighbor Discovery 185
- IPv6 Addresses 186
- IPv6 Interfaces 186
- Address Notation 187
- Address passing and storage 187
- Address classes 188
- Link Local vs. Global addressing 188
- Link Local Addresses and ScopeID 188

- Unique Local Addresses 189
- Node Local 189
- Predefined Addresses 190
  - Loopback - ::1 190
  - All nodes multicast - FF02::1 190
  - All routers multicast - FF02::2 190
  - Solicited node multicast 190
  - Unspecified address 190
- Neighbor cache 191
- More notes on IPv6 Addressing 191
- IPv6 Routing 192
  - The interfaces 192
  - Controlling IPv6 router behavior 194
    - ip6cfg 196
    - ip6tbl 197
    - rt6add 198
    - rt6del 199
    - rt6list 200
    - rt6man 201
    - rt6prfx 202
- IPv6 over IPv4 Networks: 6TO4 203
  - 6TO4 Relay Router 204
- DHCPv6 205
  - What the DHCPv6 Client Does 205
  - Porting Considerations 205
  - DHCPv6 Menu Commands 207
    - dhcpv6 lease 207
    - dhcpv6 netstat 208

Sockets API 209
- Overview 209
- Sockets Errors 210
- Quick List for Sockets Prototypes 211
- Quick List for Socket Options 212
- Sockets API Calls Reference 214
  - t_socket 216
  - t_listen 219
  - t_connect 220
  - t_socketclose 221
  - t_select 223
  - t_recv, t_recvfrom 225
  - t_send, t_sendto 228
  - t_accept 230
  - t_bind 231
  - t_shutdown 232
  - t_getpeername 233
  - t_getsockname 234

t_getsockopt, t_setsockopt 236
tcp_sleep, tcp_wakeup 240
Utility Functions 241
inet_ntop 242
inet_pton 243
IPv6 Sockets 244
socket6.h 244
Socket Creation 244
Connecting 245
FTP and IP addressing in the data stream 246
Socket domain field 246
ICMPv6 callbacks 247
IP6EQ and IP6CPY 247
Address Identification Macros 248
TCP Zero-Copy 249
Overview 249
Sending Data with the TCP Zero-Copy API 250
Receiving Data with the TCP Zero-Copy API 251
Writing a Callback Function 251
Reading the Data 252
TCP Zero-Copy API Reference 252
tcp_pktalloc 253
tcp_pktfree 254
tcp_xout 255
MAC Drivers 256
Scatter gather - performance 256
nb_tlen length - primarily on sends 256
Multicast is required 257
GIO - Generic IO 258
GIO Contexts 258
GIO Context Structure 259
GIO API 262
gio_dev 263
gio_done 264
gio_in 265
gio_out 266
gio_pop 267
gio_printf 268
gio_push 269
GIO Example 271
NicheTool 273
The Menu System 273
Menu Structures 274
Commands and Parameters 275
User-defined Menus and Commands 276
Command Line Parsing 276

Command Execution 277
Virtual File System - VFS 280
VFS API Overview 280
VFS Implementation 280
Source Files that Constitute the VFS 281
VFS Configuration Options 281
VFS_FILES 281
HT_RWVFS 282
HT_EXTDEV 282
HT_LOCALFS 282
FILENAMEMAX 282
VFS_MAX_TOTAL_RW_SPACE 282
VFS_MAX_DYNA_FILES 283
VFS_MAX_OPEN_FILES 283
Detailed Description of VFS API 283
vclearerr 284
vfclose 285
vferror 286
vfopen 287
vfread 289
vfseek 290
vftell 291
vfwrite 292
vgetc 293
vunlink 294
Internal Data Structures 295
vfs_file Structure 295
Bits of the Flags Field 296
vfs_open Structure 297
Porting Engineer Provided VFS Functions 298
VFS_VFS_FILE_ALLOC 298
VFS_VFS_FILE_FREE 298
VFS_VFS_OPEN_ALLOC 298
VFS_VFS_OPEN_FREE 299
vfs_lock() and vfs_unlock() 299
VFS NicheTool Commands 299
vfs attribute 300
vfs delete 301
vfs directory 302
vfs read 303
Local File Systems 304
External File Systems 305
VFSCOMP - External Utility for VFS Configuration 306
VFSComp - VFS Filesystem Compiler 307
VFS Compiler Use for Web Pages 308

1 Product Overview

This Technical reference is provided with the InterNiche NicheStack TCP/IP protocol stack sources. The purpose of this document is to provide enough information so that a moderately experienced "C" programmer with a reasonable understanding of TCP/IP protocols can port NicheStack to a new environment.

It is assumed that the InterNiche "w32_nichetask_vs" or "w32_superloop_vs" sources are available as a reference. These sources can be compiled and linked to produce a Windows "Console Application" that contains the NicheStack and a simple user interface that allows the user to exercise the functionality of the stack. Depending on what other InterNiche software products have been licensed, this reference port will also include other modules that use the stack to communicate with other IP hosts. Examples of these other modules are an HTTP Server, SNMP agent, FTP Server, and a Telnet Server. These modules provide an implementation of NicheStack and related servers on a PC in a way that closely approximates the environment of embedded systems.

1.1 Terms and Conventions

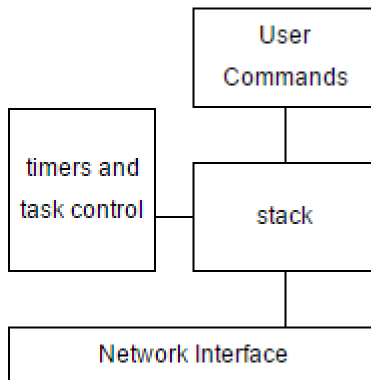
In this document the term "stack", when used without further qualification, means the NicheStack software and related code as ported to an embedded system. "System" refers to your embedded system. "Sockets" refers to the TCP API developed for UNIX at U.C. Berkeley. "Porting engineer" refers to the engineer who is porting the InterNiche software to an embedded system. A "user" or "end user" refers to the person who ultimately ends up using the product containing the ported InterNiche software. "FCS" is an acronym for "First Customer Ship", the point in the software development cycle when the product is declared ready to ship. A "packet" is a sequence of bytes sent on network hardware, also known as a "frame" or a datagram".

Names of files, C structures and C routines are displayed as follows: c_routine().

Samples of source code from C programs are displayed in these boxes:

```
/* C source file - yet another 'hello' program . */
main()
{
   printf("hello world.\n");
}
```

1.2 What a Port is



Above is a simplified diagram of the events which drive a typical embedded networking stack and the responses it makes. An event occurs (e.g. a user enters a command, a packet is received or a timer goes off) and in response a call is made to the stack to handle the event. The stack will in turn make calls to the system: sending network packets, returning data or status information to the user and setting more timers. In an ideal situation, these calls to the system map directly - for example, the stack's external call to send a packet has the exact same syntax as the network interface's exported send call.

In the world of portable stacks the stack designer does not know what multitasking system, user applications or interfaces will be supported in the target system. A "portable" stack is one that is designed with simple, generic interfaces and a "glue layer" which maps the generic interfaces to the specific interfaces available on the target system. Using the example of sending a packet, the stack would be designed to call a generic `send_packet()` routine and the porting engineer would code a "glue" routine to send the packet on the target system's network interface.

Making a stack portable involves minimizing the number of glue routines and keeping the glue routines simple and therefore easy to implement. The glue routines also need to be well documented. The interfaces to the InterNiche stack have developed through many years of porting to a variety of processors, network media and multitasking systems. Wherever possible we have used standard interfaces (e.g. Sockets, ANSI C library) or included example glue routines to illustrate their use.

The bulk of the work in porting a stack is understanding and implementing these glue routines. The InterNiche stack has the following categories of glue routines:

- Application API (e.g. Sockets)
- Memory management (e.g. malloc(), free())
- Network hardware interface
- Timer and multitasking interface

1.3 Requirements

Before beginning a port, the porting engineer should ensure that the necessary resources are available in the target environment. There must be a processor (with some spare CPU power) with some sort of operating system or monitor, some RAM memory and some sort of network interface. The exact amounts of these resources will vary depending on which features are to be implemented, what kind of performance is required and how many simultaneous users are to be supported.

Here is a brief summary of the services NicheStack needs from the system:

- At least one Network interface device (may be polled or interrupt driven)
- A timer which ticks at least once a second
- Memory and processing power as described below
- Suitable compiler and debugger

Memory Requirements

There is no easy way to determine the exact memory sizes required, however a rough idea can be obtained by examining the "w32_nichetask_vs" reference port. The numbers presented in the table below were taken from Microsoft C "map" files for a compile for the Intel x86 processor with Microsoft C/C++, using options to optimize code size ("`/EHs-c- /Oi /GS- /c /O1 /W4`"). Most of the `ipport.h` build options have not been selected, some of which would result in larger, and some in smaller code sizes. These numbers, therefore, represent an untuned "starting point".

These numbers, from a recent release of InterNiche NicheStack, are subject to change and configuration specifics but should provide a good feeling for what you can expect in your particular build. Please contact InterNiche Support for more up-to-date values.

Bytes	Use
16,507	TCP code space
5,554	IP module code space
10,644	PPP code space
22,627	IPSec code space
5,861	SNMPv1 code space

These sizes will change as the code is ported to different CPU architectures and compilers.

CPU Requirements

The most reasonable approach to assessing processing power requirements is to consider similar systems and proceed from there. For embedded agents, the InterNiche "w32_nichetask_vs" reference port is intended to provide a starting point. Protocols can be tested under load and multiple console applications can be run simultaneously, each with their own IP address.

Operating System Requirements

The stack also requires a few basic services from the target system. These are listed here:

clock tick	A clock tick counter needs to be incremented at regular intervals. See the Timers and Multitasking discussion.
memory access	The standard `calloc()` and `free()` library calls are ideal and can be replaced with a static or partition-based scheme.
multitasking	The stack needs to obtain CPU cycles to process received packets (and handle timeouts) on a timely basis. See Servicing the Stack.

1.4 TCP/IP Stack Source Code Organization

The source code that a customer receives when they purchase NicheStack is organized into several directories. This section describes this directory structure and the files contained therein.

Portable Files vs. Port Dependent Files

The NicheStack source code files can be categorized according to their degree of portability. "Portable" files are those which should be compiled and used on any target system without modification. "Port dependent" files are those which might need to be modified or replaced for different target systems.

The organization of NicheStack files is designed to clearly distinguish the port-dependent files from portable files. At the system level, each delivery of products licensed from InterNiche will contain the target specific reference port system directory, `ReferencePorts/w32_nichetask_vs`, and possibly one or more target-specific system directories for other platforms. These directories contain the system-level port-specific code, drivers, libraries, defines, and configuration files. The files in these directories will be described in detail in Porting Guide.

Most of the module directories will contain one file that holds all the port specific code, e.g., `ftpport.c` for FTP, `httpport.c` for HTTP, `ppp_port.c` for PPP, etc. The likelihood that these port dependent files will need to be modified for a particular target system is related to how much that target system varies from the target system for which the `XXX.port.c` file was created.

In general the other NicheStack source code files are portable and should require little or no modification. However, our classification of whether a particular file is portable or port dependent is very much a judgment call on our part. When we say that a file is portable, what we mean is that we don't think that it will need to be modified during the porting process, but there is always the possibility that the requirements of a particular target system or application will require that a portable file be modified. Likewise, when we say that a file is port dependent, it does not necessarily mean that the file will need to be modified.

Target System Independent Directories

Some directories contain code that is relatively target system independent. They contain the implementation of the TCP and IP protocols and the simple user interface. The expectation is that the source code contained in these directories should function with little or no modification on any target system.

These directories are briefly described below:

tcp	TCP and Sockets source files
ip	IP and UDP source files
net	network support
misclib	user interface, IP address parsing code, other similar extra functions

Target System Dependent Directories

Certain directories contain code that is relatively target system dependent. They contain code that allows the system independent code to run on a particular class of target systems, effectively providing the "glue layer" for several popular target systems. A few examples of these directories are listed below. Note that this is not a complete list and some of these target boards are no longer being manufactured, but these reference ports are available as examples to the porting engineer. The directory names can be decoded as a combination of the target's processor, its operating environment and the toolchain used to produce the final project. For example, the `mcf5282_nichetask_cw` and `mcf5282_mqx_cw` ports both use CodeWarrior tools to create an executable which runs on a Freescale MCF5282, but one runs InterNiche's NicheTask operating system and the other assumes ARC's MQX. InterNiche's standard reference port is `w32_superloop_vs` (Windows console application, InterNiche OS, Microsoft "Visual Studio (express)"). Since InterNiche products are designed to be portable, all application development and testing which are not strictly related to the final target environment can be done on the developer's desktop using either the "w32_nichetask_vs" or "w32_superloop_vs" port.

Check with InterNiche to see if your target system is closely related to one for which a port already exists, for if it does then much of the job of porting the stack will have already been performed. Documentation describing the contents of these target system directories is typically contained in a "README" file found in each target directory.

Build Directories

Some directories do not contain product source code per se, but are useful in the process of compiling and linking the source code to produce executable programs.

h	The target system include file directory described in the next section.
xx_yy_zz	Contains the 'glue' code described earlier for architecture 'xx', operating environment 'yy' and toolchain 'zz'.

2 System Architecture

2.1 Operating System Interface - porting macros

2.2 Memory Organization and Buffer Management

Socket data is maintained internally in packets. A packet consists of a linked list of packet buffers. A packet is contiguous if has only one packet buffer. A packet is chained if more than one packet buffer is used to contain the packet's headers and data. When data is written to a socket, the Stack allocates a packet large enough to hold the data, copies the data into the packet buffer(s), and appends the packet to the socket's "send" queue. When data is read from a socket, the packet is removed from the socket's receive queue, the data is copied into the application's buffer and the packet is freed. The application writer does not need to be familiar with the internal organization of a packet.

An alternate socket API is the TCP Zero-Copy API which exposes the internals of a packet. This allows programmers to allocate packets and write directly to packet buffers, thereby eliminating the data copy step. Received data is passed in packets directly to an application via a callback mechanism. The TCP Zero-Copy section of this manual describes this API is further detail.

Unused packet buffers are kept in free queues, sorted by packet buffer size. When a packet allocation request is made, one or more packet buffers are removed from the queue(s) and linked together to form a chained packet of sufficient length. The number of free queues and the size of packet buffers in each queue is configured by the porting engineer. The minimum packet buffer size is 128 bytes. The maximum packet buffer size will depend upon available target memory, device driver requirements, and the protocols being used.

2.3 Buffer management functions

int pk_init(int len, int num);

Creates a new packet buffer free queue. The queue will be initialized with 'num' packet buffers, each of length 'len' bytes. A return code of 0 indicates success. A non-zero return code indicates an error condition; probably not enough memory for the number of packet buffers requested. It is also an error to attempt to create two queues with the same packet buffer length.

Packet buffer queues can be created at any time. As a convenience, the userdata.c file contains a NET_BUFQ structure which describes the packet buffer queues to be created during NicheStack initialization.

PK_ALLOC(pkt, n)

This macro obtains the NET_RESID lock and allocates a chained packet of size 'n' bytes. On return 'pkt' points to the first netbuf structure of the packet or NULL if the packet could not be allocated. Releases the NET_RESID lock.

The variable 'pkt' is of type PACKET which has the typedef, "struct netbuf *" See The netbuf Structure and the Packet Queues for more details.

PK_ALLOC_DONOTLOCK(pkt, int size)

Same as PK_ALLOC except it does not obtain the lock. Used when the calling function already holds the NET_RESID lock

PK_CONTIG(pkt, n)

This macro is similar to PK_ALLOC(), except that the returned packet must be a contiguous packet.

PK_CONTIG_DONOTLOCK (PACKET pkt, int size)

Same as PK_CONTIG except it does not obtain the lock. Used when the calling function already holds the NET_RESID lock PK_FREE(pkt)

PK_FREE(PACKET pkt)

Obtains the NET_RESID lock and returns a contiguous or chained packet to the packet buffer free queue(s). Each packet buffer in a chained packet is returned to its free queue.

PK_FREE_DONOTLOCK (PACKET)

Same as PK_FREE except it does not obtain the lock. Used when the calling function already holds the NET_RESID lock.

Note: On systems that do not need locks (e.g., SUPERLOOP), there is no penalty for calling the locking versions of the macros because LOCK_NET_RESOURCE(XXX) can be defined as nothing.

2.4 System Initialization

Modules

A module encapsulates one or more components that make up the InterNiche stack and its protocols. Some examples of modules are Telnet, FTP, and TCP/IPv6. Some modules, such as Telnet and FTP, include tasks which are managed by the operating system. Modules are independent of one another and have a well-defined interface, which makes it easy to add or remove modules from the core NicheStack. Adding a new module to an existing NicheStack port, should require little effort beyond adding a reference to the module's `NET_MODULE` structure to the array of existing `NET_MODULE` structures.

A modules is described by its `NET_MODULE` structure. It includes informational fields, such as the module's name and version, as well as a set of function pointers which are called during the stages of NicheStack initialization to allow all modules to come up in an orderly manner. A function pointer can be set to `NULL` if a module does not have any work to do during that initialization stage. The `NET_MODULE` structure is defined in `userdata.h`, and has the following fields:

prep_func	function to prepare the module
init_func	function to initialize the module
start_func	function to start the module
close_func	function to shutdown the module
menu_func	function to calculate the module's menu resources
task_func	function to calculate the module's task resources
name	module's name string
version	module's version string

The 'prep' function

```
int (*prep_func)(void)
```

A module's "prep" function contains code which is intended to be called once, before the module is initialized. This could include initializing variables or flags that control the later initialization stage of the module.

The "prep" function returns zero if it was successful. Otherwise a non-zero error code is returned.

The 'init' function

```
int (*init_func)(void)
```

A module's "init" function initializes all of the variables and structures associated with the module. Any tasks are created and any CLI menus are registered. The initial state of a task is "not running". This eliminates the possibility of a race condition, should a task be scheduled to run before the NicheStack is fully initialized.

The "init" function returns zero if it was successful. Otherwise a non-zero error code is returned.

The 'start' function

```
int (*start_func)(uint32_t flags)
```

A module's "start" function is used to move any tasks from the "no running" state to the "ready to run" state. If a module does not include any tasks, the "start" function can be set to NULL.

The "flags" parameter is currently unused.

The "start" function returns zero if it was successful. Otherwise a non-zero error code is returned.

The 'close' function

```
int (*close_func)(uint32_t flags)
```

A module's "close" function is used to shutdown a module. This will include terminating any tasks, uninstalling any CLI menus, freeing all of the mdoule's allocated resources, and resetting any variables. After the "close" function has finished, the module should be in a state where it can be restarted by calls to the module's "init" and "start" functions.

The "flags" parameter is currently unused.

The "close" function returns zero if it was successful. Otherwise a non-zero error code is returned.

The 'menu' function

```
void (*menu_func)(int32_t *num_menus, int32_t *num_params)
```

The CLI module calls the "menu" function to query the module for its menu resource requirements. The module sets the "num_menus" variable to the maximum number of command menus that it may register. The "`num_params`" variable is set to the maximum number of parameters used by any of the module's CLI commands. The `cli_calc_args()`function can be used to perform the latter calculation:

```
struct cli_menu snmp_nt = { /* CLI menu structure */ };

*num_params = cli_calc_args(&snmp_nt);
```

If the module does not have any command menus, this function pointer can be set to NULL.

Task initialization

```
void (*task_func)(int32_t *num_tasks, int32_t *stack_size)
```

This function is called by the NicheTask operating system to query each module for its task resource requirements. The operating system will use this information to allocate internal OS structures and task stacks. The module sets the "`num_tasks`" variable to the maximum number of tasks that it will create. This number should include any tasks that might be created dynamically after NicheStack is up and running. The "`stack_size`" variable is set to the number of bytes required for the stacks of all of the module's tasks.

If the module does not create any tasks, this function pointer can be set to NULL.

userdata.c

Modules are "tied" into the core NicheStack through an array of `NET_MODULE` references in the file, `userdata.c`. The `userdata.c` file is intended to be the focal point for porting engineers to add any code required to customize the initialization and configuration of their NicheStack port. `userdata.c` includes the following tables:

NET_DEVICE

The "`in_devices[ ]`" array is an array of network interface device structures, i.e. `NET_DEVICE`s. There is one entry in the array for each device that is created and initialized during NicheStack initialization. Each `NET_DEVICE` has a pointer to the device's "`prep`" routine (described in Device Driver API), the device's IPv4 address, subnet mask, and network gateway IPv4 address. The "flags" field is a bitwise-OR of the `NF_xxx` values (defined in `net.h`). If the device obtains its network address from a remote DHCP server, the user would set the `NF_DHCPC` and `NF_AUTOIP` bits in the "`flags`" field. The IPv4 address field then becomes the default address if a DHCP server is not available.

NET_MODULES

The "`in_modules[ ]`" array contains pointers to the `NET_MODULE` structures that are part of the port. Users can add modules to the port by adding a pointer to each module's `NET_MODULE` structure to the end of the array.

NET_BUFQ

The "`in_bufq[ ]`" array is used to statically define the numbers of packet buffers and their sizes to be created during NicheStack initialization.

userdata.c support functions

`userdata.c` includes the following functions. These functions are intended to be modified by the porting engineer to support the desired system configuration:

```
int user_pre_setup(void)
```

This function is called at the end of the `pre_task_setup()` function, and is intended to give porting engineers an opportunity to perform and system initialization that is not part the normal NicheStack port. This could include initializing devices, such as USB or video displays which are part of the developer's product, but not used by NicheStack directly.

```
int user_post_setup(void)
```

This function is called at the end of the `post_task_setup()` function. Porting engineers can add code here to further configure the NicheStack environment; set debug flags, add entries to tables that were not or could not be initialized via CLI scripts, or perform any other non-standard module configuration.

Module Initialization

At each stage of NicheStack initialization, each module's function is called to perform its initialization. When all modules have successfully completed a stage, initialization progresses to the next stage.

There are several boolean variables that can be tested to monitor the stages of the initialization process. These variable are set to `FALSE` when initialization begins:

`iniche_init_done`	Set to TRUE during post_task_setup() processing. Indicates that all NicheStack resources have been created and initialized.
`iniche_net_ready`	Set to TRUE after the "iniche_rc" script is executed. Indicates that the network is up and that tasks can make calls into the Stack. Network interfaces may not be available yet.
`iniche_os_done`	Set to TRUE within the TK_OS_START() macro. Used by NicheTask to inform the scheduler that task initialization is done and task scheduling may begin.

The sequence of steps that happen during initialization is:

1. `clock_init()`	Initialize the 'cticks' timer.
2. `pre_task_setup()`	Perform any initial NicheStack preparation. Call the `user_pre_setup()` function to execute any additional user-supplied initialization code.
3. `prep_modules()`	Call each module's "prep" function.
4 `prep_devices()`	Call the "prep" function for each NET_DEVICE device in the `in_devices[ ]` array. The DHCP address processing is started here, but may not be completed.
5. boot script	Call the CLI module to execute each of the CLI commands in the "boot_rc" file. These commands are a small subset of the CLI command set, and are intended for configuring NicheStack resources.
6. `init_packets()`	Create the initial pools of packets and packet buffers.
7. `init_devices()`	For each device created in the "prep_devices" stage, call the device's "init" function.
8. `init_modules()`	Call each module's "init" function.
9. `post_task_setup()`	• Perform any post-initialization processing prior to starting NicheStack. • "iniche_init_done" is set to TRUE. • Finally, the `user_post_setup()` function is called to execute any additional user-supplied post-initialization code.
10. iniche script	Call the CLI module to execute each of the CLI commands in the "iniche_rc" file. The complete set of CLI commands is available at this point in the initialization process.
11. `iniche_net_ready`	is set to TRUE.
12. (finally)	Call each module's "start" function.

_rc files

If your target defines `INCLUDE_CLI`, the system will read 2 script files during the boot and initialization phase.

boot_rc	Script of configuration commands called before the initialization routines
iniche_rc	Commands to configure, initialize or start modules or devices.

The `boot_rc` script is executed by `nichestack_init()` after the module and device preparation routines have been called, but before the initialization of packets, devices, and modules. Only a limited number of configuration commands would be appropriate at this stage in initialization. Examples include: setip() to set the IP address for an interface, config() to set the screen prompt, and cbadd() to add a buffer queue.

Note: NicheStack does not use flags or any other means for enforcing which commands can be called from `boot_rc`. The porting engineer should ensure that only appropriate commands are included in this script.

The second script, iniche_rc is executed after the initialization of devices and modules and after `post_task_setup()`, but before the variable `iniche_net_ready` is set and before the modules are actually started by `start_modules()`. Any command can be included in this script unless the command depends on something that does not occur until a module's start routine has been executed. Examples include adding entries to the user table, specifying the DNS servers, and module configuration commands.

If your target will not be using the command line interface and menus, then the porting engineer will have to develop another means for calling any required initialization functions.

2.5 Ethernet Device Driver

The Ethernet device driver implements an API for moving packets between NicheStack and the phyical network. Network devices are described in the `NET_DEVICE` array in `userdata.c`. Each entry include a pointer to a user-defined "prep" function which is called to initialize the device's `NET` structure. The `NET` structure includes an array of function pointers which are called by NicheStack to manage the device driver. The "prep" function and the functions in the `NET` structure comprise the Device Driver API.

2.6 Device Driver API

The NicheStack Device Driver API consists of the following functions:

- `int prep(int iface);`
  This is a user-defined function which initializes a `NET` structure in the `nets` array. '`iface`' is the 0-based index into the `nets` array. Upon entry, the `NET` structure is filled with zeros. The function performs the following steps:
  - Initializes any per-device local variables. The '`n_local`' field in the `NET` structure can be use to point to a per-device private data area.
  - Initializes the MIB fields associated with the interface
  - Sets the bits in the '`n_flags`' field in the `NET` structure to the device's characteristics. Flags that are normally set are:

`NF_BCAST`	device supports receiving broadcast packets
`NF_MCAST`	device supports receiving multicast packets
`NF_NBPROT`	device will set the nb_prot field in received packets
`NF_GATHER`	device supports chained packets
`NF_IEEE48`	device has a 48-bit MAC address
`NF_IPV6`	device supports IPv6 protocol (requires NF_MCAST)

  - Initializes the device API function pointers

  The function returns the number of interfaces that were "prepped". A return code of 0 indicates an error. Normally, the return value is 1, but the "prep" function could be written to initialize multiple devices.

- `int (*n_init)(int iface);`
  The 'n_init' field in the device's NET structure points to the init function. This is a user-defined function which initializes the device's hardware and software. Typical steps include:
  - Reset the device hardware.
  - Configure the device hardware.
  - Auto-negotiate the link characteristics.
  - Setup any device interrupt handler(s).
  - Set the device's MIB status to "UP".
  - Allocate and initialize any buffer descriptor arrays.
  - Enable transmit and receive of packets.

  The function returns 0 if successful, and non-zero if there was an error.

- `int (*pkt_send)(PACKET pkt);`
  The 'pkt_send' field in the device's NET structure points to the packet sending function. Packets should be sent in the order in which they are passed to the driver. The driver is responsible for freeing a packet after the data has been sent. The NF_GATHER bit should be set in the 'n_flags' field of the device's NET structure if the device driver can accept chained packets from the Stack. If the bit is not set, chained packets will be converted by the Stack into contiguous packets before calling the pkt_send() function. This may impact Stack performance.
  The packet to be sent consists of a chain of packet buffers. The following fields in each packet buffer describe the data to be sent:

nb_tlen	The total number of bytes in the packet (including ETHHDR_BIAS). (This field is only valid in the first packet buffer.)
nb_plen	The number of bytes in the packet buffer. (ETHHDR_BIAS must be subtracted from the length in the first packet buffer.)
nb_prot	Points to the first data byte in the packet buffer. (ETHHDR_BIAS must be added for the first packet buffer.)
pk_next	Pointer to the next packet buffer in the chained packet.

  If the device is busy, the driver should queue the packet internally for transmission at a later time. The device driver is responsible for freeing the packet after it has been sent. If the packet data must be copied from the packet buffer(s) to an internal memory space, the data copy functions described in TCP Zero-Copy may be of use to the porting engineer.
  A return code of 0 indicates that the packet has been successfully sent (or queued). A non-zero return code indicates an error.
- `int (*raw_send)(NET ifp, char *data, unsigned len);`
  Sends 'len' bytes of contiguous data from the buffer pointed to by 'data'. The data is sent to the device associated with NET structure 'ifp'. The function returns after the data has been sent (or copied into an internal buffer). This function is deprecated in favor of the 'pkt_send' function. If a 'pkt_send' function is implemented, this function pointer should be set to NULL.

- `int (*n_close)(int iface);`
  This function closes the device associated with `NET` structure '`nets[iface]`'. The device hardware should be reset to an idle state, any memory resources should be freed, and the network interface status should be set to "DOWN". The device's '`n_init`' function must be called before the device may be used again.
- `int (*n_reg_type)(unshort regtype, NET ifp);`
  Some systems, such as `WinPcap` on Windows are capable of filtering incoming packets based on packet type. The '`n_reg_type`' function provides a hook to allow the Stack to register interest in specific types of packets. This function is not normally implemented for embedded devices.
- `int (*n_stats)(int iface, void *stats)`
  Copies the statistics of the device associated with `NET` structure '`nets[iface]`' into the structure pointed to by '`stats`'.
- `void (*n_refill)(int iface)`
  Replenish the driver's internal packet buffer pool from the NicheStack free packet queue(s). Additional packet buffers are allocated from the NicheStack free packet queues until the total pool size equals or exceeds the pool threshhold or the NicheStack queues are exhausted. The 'iface' parameter specifies which interface is being replenished. A single refill function can be used for all interfaces or each interface can have its own refill function.
  In a multitasking OS environment, the `n_refill()` function is called every time the main NicheStack task is scheduled.

2.7 Handling Packets within a Device Driver

The sending of a packet is addressed in the 'pkt_send' function description. When data is received by the Device Driver, the data must be stored in a packet (either chained or contiguous), the packet placed on the Stack's receive queue, and the packet processing task must be notified. If the data must be copied from an internal memory space to a chained packet, the copy functions described in TCP Zero-Copy may be of use to the porting engineer.

Due to the overhead involved, packets should not be allocated or freed within an interrupt service routine, but doing so may be unavoidable in some cases.

When storing data in a chained packet, the data in the first packet buffer should begin at offset ETHHDR_BIAS from the beginning of the buffer. For all other packet buffers, the data begins at the beginning of the packet buffer. After the incoming data has been stored in the packet buffer(s), The ' nb_tlen' field is set to the total length of the packet data (including all headers), the 'nb_prot' fields point to the first data byte in each packet buffer, and the 'nb_plen' fields are set to the number of bytes of data in each packet buffer. The following code snippet illustrates the remaining packet initialization and queuing logic:

```
/* pkt = pointer to the packet */
/* iface = network interface index */

struct ethhdr *et = (struct ethhdr *)(pkt->nb_buff);

pkt->net->n_mib->ifInOctets += pkt->nb_tlen;    /* MIB statistics */
pkt->net = nets[iface];

/* nb_prot is adjusted to point to the first byte after the 14-byte
   ethernet header */
pkt->nb_prot = pkt->nb_buff + ETH_HDR_LEN;
pkt->nb_tstamp = cticks;                        /* timestamp */
pkt->type = et->e_type;                         /* 16-bit packet type */

/* queue the packet and signal the Stack task */
putq(&rcvdq, pkt);   /* incoming packets go on the 'rcvdq' queue */
SignalPktDemux();    /* wake Interniche netmain task */
```

3 Before Contacting Support

Unless available through your source code distributor or otherwise specified in the terms of your InterNiche license, Technical Support is available via email at Support@HCC-Embedded.com. Please include your "Contract ID" in the Subject of your email. It can be found in the top few lines of every `.c` file in your source code distribution.

4 Porting Guide

This section describes the steps needed to port NicheStack to a new environment. The discussions below generally assume that the stack is being ported to a small or embedded system with a network interface and that a minimal ANSI C library is available.

The recommended steps to getting the InterNiche stack working on your target system are summarized below:

1. Create an InterNiche stack source code tree in your development environment.
2. Code your versions of the port-dependent system files, including configuration files and glue layers.
3. Code any glue layers required in the individual modules directories
4. Build a target system image, test and debug.

4.1 The InterNiche Source Code Tree

The InterNiche sources are typically distributed as a .zip file, which should be unzipped with an appropriate utility in such a way as to preserve the underlying directory structure, described in TCP/IP Stack Source Code Organization.

Target-Specific System Directory

InterNiche source code distributions typically include at least one reference port, `w32_in_vc`, and possibly one or more target-specific system directories for other platforms. These directories contain the system-level port-specific code, drivers, libraries, defines, and configuration files.

The port-specific files in `w32_in_vc` reference port were designed for a product developed with Microsoft's Visual Studio Express to run on a Windows platform. In this section, we will describe the files and functions in the `w32_in_vc` directory. They should be considered as an example. Other target platforms will require many of the equivalent macro definitions, configurations and functions, but the implantations will be port-specific and the organization may be different.

The following is a list and brief description of the files found in the v4.0 release of `w32_in_vc`. Many of these files will be described in more detail below.

Windows drivers	`windrv`, `windrv_pcap`, `winuart` files
Top-level makefile and program executable	`makefile`, `port.mk`, `os.mk`, `tools.mk`, `iniche.exe`
System configuration files	`ipport.h`
Target-specific defines, macros and libraries	`ipport.h`, `libport.h, osport.h, toolport.h`
Port-specific system files	`inport.c`, `intimer.c`, `inutil.c`, `inppp.c`, `sysinit.c`, `osporttk.c`, `uartutil.c`, `sum386.c`, `tk_386.c`
`main()` and system initialization	`inmain.c`, `ininit.c`, `userdata.c`
Example script files	`boot_rc`, `iniche_rc`

Windows Drivers

The windows drivers require no modification, assuming that you will be using Windows for development and testing, but not as the actual target.

`WinPcap` is an application that works in coordination with the Windows driver. The installation of this file is described in the readme.txt file in the `w32_in_vc` directory. Once installed, iniche.exe will run on the Windows desktop, and it will behave as though it were running on a target platform. Data can be read from, and written, to any of the system Ethernet, wireless or serial interfaces. There are, however, some differences.

As it appears to NicheStack, the Windows driver is not interrupt driven. Rather it is called in a loop. Code to turn interrupts on and off will have no effect.

It is important to note that WinPcap was not developed and is not supported by InterNiche. At the time of this writing, it can be freely obtained from http://www.WinPcap.org.

Makefiles

ReferencePorts/w32_nichetask_vs contains a makefile for building the entire NicheStack executable, Your development system may use a different mechanism, but will need to perform similar functions, Each subdirectory under the top of tree contains a makefile.inc which will be included into the make process. In the w32_nichetask_vs directory there is a cflags.vs that contains locations and compiler options and is intended to isolate the rest of the makefiles from tool specific requirements. There are versions for each supported toolchain.

The w32_nichetask_vs makefile redirects to the makefile in top of tree which:

- reads port.mk/os.mk/tools.mk for port specific options.

- >reads cflags.mk to sort out the toolchain paths
- Sets up the path to the build directory
- Includes each of the module makefiles. Modules not present are skipped automatically.
- Sets the default build target to the ports binary
- Lets make resolve dependencies and compile/link the target

The process is written to use gmake and a copy of the windows binary is provided in /utils/bin/gmake. No other tools are required for the build to succeed.

System Configuration Files

If your project includes any of InterNiche's security protocols, the `crypt_port.h` file will contain configuration macros that define which portions of the security protocols will be compiled with the build.

ipport.h

A large section of `ipport.h` consists of defines that control which portions of the code will be compiled both for the general system and for each of the individual modules. These definitions are divided into sections for each module, and each of these contains a section surrounded by the define `#ifdef NOT_USED`. Each NicheStack file starts with `#include "ipport.h"`, and with the exception of files that must be included in all builds, one of the next lines will be an `#ifdef XXX` that controls whether or not the code within the file will be compiled. This means that even if the library for a module is included in the build, the code for that module can be eliminated from the executable simply by moving its controlling defines into a `NOT_USED` section. Conversely, if the controlling defines for a module are in `ipport.h` and not within an `#ifdef NOT_USED` section, but the modules library is not included in the top-level makefile, then numerous compile and/or link errors will appear during the build.

Note that InterNiche code looks for the *existence* of a `#define` and not its value, so redefining a `#defined` value from a `1` to a `0` will not have the desired effect and the results may be difficult to debug.

The system-level configuration defines are described in the section System Configuration Defines below. The configuration defines for individual protocol modules are described in their respective protocol reference manuals.

Target Specific System-level Macros and Definitions

Three header files in the w32_in_vc directory contain target-specific defines and macros.

ipport.h	Macros and definitions needed by NicheStack
osport.h	OS related macros, definitions, and prototypes: • Those related to mutual exclusion and the scheduling of tasks See manual sections "Task Control" and "NicheTask Multitasking Scheduler" • Defines that determine the stack sizes for each module in the system. • Time macros. See manual sections "Timers and Time Macros"
toolport.h	Macros that can be used where the equivalent C library functions are not available. Most are used for copying and string manipulation.
libport.h	Macros and definitions needed by NicheStack based on port hardware.

Standard Macros and Definitions

The InterNiche stack expects `TRUE`, `FALSE`, and `NULL` to be defined within the scope of `ipport.h`. The best way to do this is usually to include the standard C library file `stdio.h` inside of `ipport.h`. If stdio.h is impractical to use or not available on your development system, the examples below will work for almost every C environment:

```
#ifndef TRUE
#define TRUE  1
#define FALSE 0
#endif
#ifndef NULL
#define NULL (void*)0
#endif
```

Four common macros are used from Berkeley UNIX for doing byte order conversions between the representation used in a particular CPU and the CPU independent "network" order. These are `htons()`, `htonl()`, `ntohs()`, and `ntohl()`. They may be either macros or functions. They accept 16 and 32 bit quantities as shown and convert them between network order ("big-endian") and the local CPU format, often referred to as host order. Most "big-endian processors", such as Motorola 68K, Power PC and ARM can just return the variable passed, as in this example:

```
#define htonl(long_var)   (long_var)
#define htons(short_var)  (short_var)
#define ntohl(long_var)   (long_var)
#define ntohs(short_var)  (short_var)
```

The Intel 8086 and its descendants require the byte order in the word or long to be swapped ("little-endian"). The standard InterNiche stack source code distribution which works for Intel processors in 16 bit real mode implements `htons()` and `ntohs()` as macros, whereas `htonl()` and `ntohl()` are implemented as function calls to the assembly language function `lswap()`, the implementation of which is contained in the file `cksum1.asm`.

```
#define htonl(long_var)   lswap(long_var)
#define ntohl(long_var)   lswap(long_var)
#define htons(short_var)  (((u_short)(short_var) >> 8) | \
                              ((u_short)(short_var) << 8))
#define ntohs(short_var)  htons(short_var)
```

Depending on your C compiler, it may be more efficient to define inline C macros or inline assembly language implementations of these macros.

```
#define   LITTLE_ENDIAN   1234
#define   BIG_ENDIAN      4321
#define   BYTE_ORDER      LITTLE_ENDIAN
```

In addition to the byte order conversion functions described above, it is necessary to set the value of the defined constant `BYTE_ORDER` to either `LITTLE_ENDIAN` or `BIG_ENDIAN` in order to indicate the byte ordering of the target system processor.

```
#define  ALIGN_TYPE   2     /* 16 bit alignment */
```

Some processors will access memory more efficiently if the addresses of the addressed data are evenly divisible by 2 or 4. If the target system processor is of this variety, set the defined constant `ALIGN_TYPE` to either 2 or 4, respectively. `ALIGN_TYPE` affects the memory alignment of allocated packet buffers.

System-Level Size definitions

`ipport.h` also contains the following system-level size definitions:

MAXNUMFREEQS	maximum number of queues for buffers of different lengths
MAXPACKETS	Maximum number of packet buffers per size
MINCHAINEDPKTSZ	Maximum buffer size that can be allocated
TCP_SEND_SPACE	Maximum size of TCP send window
TCP_RCV_SPACE	Maximum size of TCP receive window
MAX_USERLENGTH	Maximum bytes in user name
MAX_PASSLENGTH	Maximum bytes in password
MAX_SECRETLENGTH	Maximum bytes in secret used for
RT_TABS	Maximum entries in IP routing table

Mutual Exclusion

There are several data structures in the InterNiche stack for which it is necessary to ensure that access is serialized. By serialized we mean that once access to the data structure is started by one thread of execution, then that thread of execution must complete its access to the data structure before another thread of execution accesses it.

In software applications that are implemented as a single polling loop with no interrupts, there is only one thread of execution and therefore serialization of access to shared data structures is inherent to the system. However, in systems that use interrupts or multitasking operating systems, serialization may need to be performed explicitly. This explicit serialization of access to shared data structures is referred to as mutual exclusion.

The InterNiche stack makes use of one of two different methods of mutual exclusion that are referred to as the critical section method and the net resource method. Generally, the critical section method is used farther down the function call tree or at a lower level than the net resource method.

Usually the critical section method is appropriate for embedded systems that lack a multitasking operating system, and the net resource method is appropriate for systems with a multitasking OS.

Failure to provide a correct implementation of mutual exclusion can result in the most intermittent, difficult to find, types of bugs. It is well worth the porting engineer's effort to desk check his implementations of mutual exclusion carefully since improper implementations can easily result in a system that works 99.9% of the time yet still crashes on occasion.

Critical Section Method

The stack calls two entry points, ENTER_CRIT_SECTION and EXIT_CRIT_SECTION, when using the critical section method of mutual exclusion. Basically, any code which needs to serialize access to a shared data structure calls ENTER_CRIT_SECTION() before it starts to access the data structure and EXIT_CRIT_SECTION() after it completes its access to the data structure An example is shown in the code fragment below:

```
thread1()
{
   ...
   ENTER_CRIT_SECTION();
   queue an element to the head of a shared queue structure named q;
   EXIT_CRIT_SECTION();
   ...
}
thread2()
{
   ...
   ENTER_CRIT_SECTION();
   dequeue an element from the tail of a shared queue structure named q;
   EXIT_CRIT_SECTION();
   ...
}
```

When a given thread of execution returns from a call to ENTER_CRIT_SECTION(), other threads of execution are prevented from accessing the shared data structure until the first thread calls EXIT_CRIT_SECTION() to release its exclusive access to the data structure. In the example shown above, if thread1() were to call ENTER_CRIT_SECTION() first then thread2() would be prevented from accessing the queue named q until thread1() called EXIT_CRIT_SECTION().

It is the responsibility of the porting engineer to provide implementations of ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION() that are appropriate for his target system and application. The porting engineer should consider the following issues before deciding how to implement these entry points:

On systems without a multitasking operating system and in which interrupt service routines (ISRs) never access shared data structures, explicit mutual exclusion is not needed. In these cases ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION() can be no-ops.

On systems in which both ISRs and non-ISR code access shared data structures, ENTER_CRIT_SECTION() should be implemented as that saves the current interrupt state and disables interrupts. EXIT_CRIT_SECTION() should be implemented as code restores the interrupt state to the state that existed before the matching call to ENTER_CRIT_SECTION(). Note that it is not sufficient to simply disable interrupts in ENTER_CRIT_SECTION() and enable them in EXIT_CRIT_SECTION() because calls to ENTER_CRIT_SECTION() can be nested. This interrupt based implementation is the simplest for most target systems.

For systems that have hard real time requirements, disabling interrupts to implement these macros could present a problem. In cases like these, the porting engineer could implement his system such that the InterNiche stack shared data structures are never accessed by ISRs and in which `ENTER_CRIT_SECTION()` and `EXIT_CRIT_SECTION()` are implemented as functions which acquire and release a single operating system semaphore or mutex, respectively, keeping in mind the fact that the calls can be nested in the same thread of execution. Using this method, the OS tasks that access the InterNiche shared data structures can be executed at low priority and can be pre-empted by higher priority tasks and ISRs that support the system's hard real time requirements.

Net Resource Method

The net resource method is usually a concern for target systems that have pre-emptive multitasking operating systems. For systems without multitasking, `LOCK_NET_RESOURCE()` and `UNLOCK_NET_RESOURCE()` can in most instances be implemented as no-ops.

Once a task has made a call into the stack, it must not be pre-empted by another task making a call into the stack until the first task exits from the stack, either by returning from its call or blocking in the `tcp_sleep()` function (described later). To facilitate this requirement all API calls where an application may call into the stack begin with a call to `LOCK_NET_RESOURCE(NET_RESID)` and end with a call to `UNLOCK_NET_RESOURCE(NET_RESID)`. Also, a proper implementation of `tcp_sleep()` on a multitasking system will call `UNLOCK_NET_RESOURCE(NET_RESID)` before blocking the caller and will call `LOCK_NET_RESOURCE(NET_RESID)` upon returning from the block. Tasks making calls to the stack may be pre-empted by other higher priority tasks so long as the higher priority tasks do not make calls into Sockets, thus it is not necessary nor desirable to implement `LOCK_NET_RESOURCE()` by disabling context switching. The most natural implementation of `LOCK_NET_RESOURCE()` on a multitasking system is to have `LOCK_NET_RESOURCE()` pend on a properly initialized semaphore or mutex and have `UNLOCK_NET_RESOURCE()` post to it.

Examples of resource identifiers that can be passed to these functions:

```
enum {
   NET_RESID,
   RXQ_RESID,
   FREEQ_RESID,
   ...
}
```

`NET_RESID` is passed to serialize access to the Sockets, TCP, UDP, and IP layers of the stack. `RXQ_RESID` is passed to serialize access to the received-packet queue structure, `rcvdq`. For example, the portable function `pktdemux()` locks `RXQ_RESID` when it dequeues packets from `rcvdq`. `FREEQ_RESID` is passed to serialize access to the free packet buffer queue structures via the portable `pk_alloc()` and `pk_free()` functions.

The porting engineer should consider the following issues before deciding how to implement these functions:

- If a thread of execution calls LOCK_NET_RESOURCE() with a given resource identifier as the parameter, another call to LOCK_NET_RESOURCE() with the same resource identifier performed by another thread of execution should block execution of the latter thread of execution until the former thread calls UNLOCK_NET_RESOURCE() with the same resource identifier.
- Calls to lock NET_RESID should not have an effect on calls to lock RXQ_RESID.
- Calls to lock NET_RESID should not have an effect on calls to lock FREEQ_RESID.
- If NET_RESID and FREEQ_RESID must both be locked from the same thread of execution (as will happen when the stack wants to allocate or free a packet), NET_RESID will be locked before and unlocked after FREEQ_RESID.
- If RXQ_RESID and FREEQ_RESID must both be locked from the same thread of execution (as may happen from a driver attempting to get a packet, fill it in, and place it on the received queue), RXQ_RESID will be locked before and unlocked after FREEQ_RESID.
- Calls to LOCK_NET_RESOURCE() with a given parameter value are never nested in the same thread of execution as are calls to ENTER_CRIT_SECTION().
- Given the requirements of the first two bullets, it would imply that the code which calls LOCK_NET_RESOURCE() should never be called from an interrupt service routine. However, in practice an efficient way to implement LOCK_NET_RESOURCE() for RXQ_RESID and FREEQ_RESID is to disable interrupts by saving the interrupt controller mask register state. Likewise, for the matching LOCK_NET_RESOURCE() call to free the resources, restore the interrupt state to enable interrupts.

Net Resource Method, Walkthrough

The diagrams below illustrate how InterNiche's TCP/IP stack can serialize the access of many processes to the stack with a single Mutex or Semaphore.

The first diagram illustrates the portions of the Interniche code which are protected by the "NET_RESID" Object (the Mutex or Semaphore). Any thread which enters this code (for example, by making a sockets call) must acquire the NET_RESID mutex before entering. If the NET_RESID mutex cannot be acquired, the thread blocks; as per the definition of LOCK_NET_RESOURCE().

The second picture illustrates the FTP application making a sockets call. The process had to acquire the NET_RESID mutex early in the socket call to proceed into the protected code. The code inside the "Protected by" oval will never block or busy-wait.

The third picture illustrates what happens if the socket is not ready for return to the application, for example, the application called recv(), on a blocking socket, but no received data is ready. The TCP code calls the OS dependent call tcp_sleep(), which releases the NET_RESID mutex before suspending the thread.

While the FTP thread is suspended, the telnet thread initiates a socket call. This Thread may enter the stack (acquire NET_RESID) since the FTP thread released NET_RESID prior to suspending.

While the telnet thread is in the stack, the scenario we are protecting against occurs: A datagram arrives for the suspended FTP thread. The OS suspends the telnet thread and wakes the FTP thread. The FTP thread, however cannot acquire the NET_RESID mutex, since it's still owned by the telnet thread. It blocks inside the call to LOCK_NET_RESOURCE(NET_RESID) and is unable to return from tcp_sleep().

The OS resumes the telnet thread. In this example, it blocks (perhaps also waiting for received data) by calling tcp_sleep() - which frees the NET_RESID mutex. The FTP thread now acquires the NET_RESID mutex and returns from tcp_sleep().

The FTP thread is able to copy the received TCP data and return to the FTP application. From the application's perspective the thread was blocked "inside" the stack the whole time.

Similar mechanisms, each with its own mutex, are used to protect the queue of free packet buffers and the queue of received packet buffers. When the code inside the TCP stack uses buffers it will often hold two mutexes (NET_RESID and one of the buffer mutexes) at the same time, however the buffer queue mutex is only held for a few cycles. This allows the buffer queue mutexes to synchronize with ISR code by disabling hardware interrupts around accesses to the buffer queues.

Debugging Aids

`dtrap()` is a macro called by the stack code whenever it detects a situation which should not be occurring. The intention is for the `dtrap()` routine or macro to try to trap to whatever debugger may be in use by the porting engineer. Think of it as an embedded break point. A macro is available to insert a break instruction into this function, the below is for x86 processors:

```
#define TRAP   _asm( int 3 )
```

The stack code will generally continue executing after a `dtrap()`, but the `dtrap()` usually indicates that something is wrong with the port. **NO PRODUCT BASED ON THIS CODE SHOULD BE SHIPPED UNTIL THE CAUSES OF ALL CALLS TO `dtrap()` HAVE BEEN ELIMINATED!** When it comes time to ship code, the `dtrap()`s can be redefined to a null function to slightly reduce code size.

The next two primitives have the same function and syntax as `printf()`. They have separate names so that they can have their output redirected or be completely disabled.

In most ports, these can be mapped to `printf()` as shown while the product is under development. Note: This example works on Microsoft C, but some compilers will complain about this syntax since it ignores the fact that these names have parameters. You may have to experiment.

```
#define dprintf   printf   /* same parms as printf, called during run time */
```

For some products, it may make sense to define these away before FCS.

```
#define dprintf(...)      /* define to nothing */
```

The last debugging tool in `ipport.h` is the `#define NPDEBUG`. Defining this will cause the debug code to be compiled into the build. This code does things like check for valid parameters and sensible configurations during runtime. It frequently invokes `dtrap()` or `dprintf()` to inform the programmer of detected problems. The porting engineer will want to make sure this flag is defined during development. Unless PROM space is tight, it is OK to leave it defined for FCS - there will be no noticeable performance degradation from this code.

```
#define NPDEBUG   1   /* enable debug checks */
```

Timers and Multitasking

IP stacks require a clock tick for such things as TCP retry, ARP cache time-outs, etc. The InterNiche stack depends the macro "`CTICKS`". The macro `CTICKS` should return an unsigned long that represents the number of timer ticks since NicheStack was booted. CTICKS should be regularly incremented by the port code between 5-100 times per second. The define TPS (ticks per second) should be set in ipport.h to represent the number of times per second that the `CTICKS` counter is incremented.

`CTICKS` is a 32 bit value that wraps back to `0` after reaching `0xFFFFFFFF`. The `TIME` macros in the table below were designed to handle the problems associated with the wrapping of `CTICKS`. These `TIME` macros should be used for all arithmetic and comparisons involving `CTICKS`. They are defined in osport.h.

Macro	Description
`TIME_ADD(t1,t2) ((t1) + (t2))`	Add times Usage: `expiration_time = start_time +`

`TIME_SUB(t1,t2) ((long)(((t1)) - (t2)))`	Subtract time
	Usage:
	`elapsed_time = now - start_time`
	`start_time = now - duration`

`TIME_CMP(t1,t2) (int)((t1) - (t2))`	Compare times Usage: `if ((t1 - t2) > duration` Result: positive integer if t1 > t2 negative integer if t1 < t2 0 if t1 = t2;
`TIME_EXP(e,n) (bool_t)(((int)((e) - (n))) <= 0)`	Determine if a timer as expired e = unsigned long int (expiration_ n = unsigned long int (now (i.e., c system clock)) Result: 0 timer not expired 1 timer expired

The `TIME` macros will only work for durations that are less than `0x7FFFFFFF` ticks. The term “duration” here refers to the values added or subtracted from CTICKS, or the value that results from the comparison of to CTICKS values. Any duration longer 0x7FFFFFFF ticks should use the macro INFINITE_DELAY, defined in `osport.h` as `0x7FFFFFFF`.

Stack Features and Options

The stack assumes you have at least one device for sending and receiving network packets. These are usually hardware devices such as Ethernet or serial ports, but they also may be logical devices such as loopback drivers or inter-process communication software. Many IP stacks on embedded systems support only one device (most commonly Ethernet), however devices like routers need two or more. The InterNiche stack supports multiple logical devices and has been used with up to three. The structures to manage these devices are statically allocated at compile time, so the maximum number of devices the system will use at runtime must be set in `ipport.h` via the constant `MAXNETS`.

```
/* define the maximum number of hardware interfaces */
#define MAXNETS   2   /* maximum entries of nets[ ] array */
```

The stack supports a fixed size IP routing table. The size of the routing table can be defined via the constant `RT_TABS`.

```
#define RT_TABS   16   /* number of entries in IP routing table */
```

Package Options

Near the beginning of `ipport.h` are a number of defined constants that are bounded by comments as shown below.

```
/* options to trade off features for size. Do not enable options
 * for modules you don't have or your link will get unresolved
 * externals.
 */

#define INCLUDE_ARP  1   /* use ethernet ARP */
...
/* end of option list */
```

These defined constants specify which features and applications are to be enabled in the stack. For example, if the constant `INCLUDE_ARP` is defined then the stack will use the ARP protocol for physical layer address resolution on interfaces that support the use of ARP. Each InterNiche customer receives a different set of enabled options in the version of `ipport.h` that is shipped in the target system dependent directory that the customer has selected that is dependent on which features and products he has ordered. The porting engineer can disable an option that is not needed by his application by moving the defined macro to the `#define NOT_USED` section. **Note:** Setting the value of the macro to `0` does NOT disable the feature, as the Interniche code does NOT test the value of a macro, instead it checks if the macro has been defined or not. Disabling options can result in reduced target system memory requirements.

Configuration Example

By making changes with `ipport.h`, it is possible to greatly affect the footprint requirements of the executable image. Features can be disabled and in many cases entire subsystems can be eliminated from the build. For example, it is entirely realistic to assume that a "production version" of the stack would have no use for a Menu System or Command Line Interpreter (CLI).

The following `#define`s should be removed from the build. This is most easily accomplished by simply moving them to within "`NOT_USED`" blocks of `ipport.h`:

- `SCRIPT_CMDS`
- `INCLUDE_CLI`

Also be sure that the Telnet and HTTP Servers are not present in your build, as each of these modules requires the Menu System.

As always, if you are having configuration difficulties or have special requirements, feel free to contact Support@HCC-Embedded.com.

Error Codes

The following negative error codes are used internally by NicheStack routines. Generally, full success is `0`; definite errors are negative numbers and indeterminate conditions are positive number. The negative internal error codes are converted to positive socket error numbers in the sockets interface. These error codes are provided in `in_errors.h` so that they can be changed if they conflict with error codes used by other parts of the target system. However the values **must remain negative** or the stack will not work.

```
#define SUCCESS              0 /* whatever the call was, it worked */
#define ESUCCESS             0 /* whatever the call was, it worked */
#define EFAILURE            -1 /* whatever the call was, it failed */

/* programming errors */
#define ENP_PARAM           -10 /* bad parameter */
#define ENP_LOGIC           -11 /* sequence of events that shouldn't happen */
#define ENP_NOCIPHER        -12 /* No corresponding cipher found for the cipher id */
#define ENP_NOT_ALLOWED     -13 /* Operation not allowed */

/* system errors */
#define ENP_NOMEM           -20 /* malloc or calloc failed */
#define ENP_NOBUFFER        -21 /* ran out of free packets */
#define ENP_RESOURCE        -22 /* ran out of other queue-able resource */
#define SEND_DROPPED        ENP_RESOURCE /* full queue or lack of resource */
#define ENP_BAD_STATE       -23 /* TCP layer error */
#define ENP_TIMEOUT         -24 /* TCP layer error */

#define ENP_NOFILE          -25 /* expected file was missing */
#define ENP_FILEIO          -26 /* file IO error */
#define ENP_NOOBJ           -27 /* Specified object does not exist */

/* net errors */
#define ENP_SENDERR         -30 /* send to net failed at low layer */
#define ENP_NOARPREP        -31 /* no ARP for a given host */
#define ENP_BAD_HEADER      -32 /* bad header at upper layer (for upcalls) */
#define ENP_NO_ROUTE        -33 /* can't find a reasonable next IP hop */
#define ENP_NO_IFACE        -34 /* can't find a reasonable interface */
#define ENP_HARDWARE        -35 /* detected hardware failure */
#define ENP_CBRD_FAILED     -36 /* cb_read() function failed */
#define ENP_CBWR_FAILED     -37 /* cb_write() function failed */
#define ENP_COALESCE_FAILED -38 /* a packet coalesce operation failed */
#define ENP_DUPLICATE       -39 /* duplicate detected */
#define ENP_BAD_TRANSACTION -40 /* unexpected transaction identifier (DHCP) */
#define ENP_CBTRUNC_FAILED  -41 /* cb_truncxxx() function failed */
#define ENP_IFNOTREADY      -42 /* Interface not yet ready for requested operation */

/* conditions that are not really fatal OR success: */
#define ENP_SEND_PENDING     1 /* packet queued pending an ARP reply */
#define ENP_NOT_MINE         2 /* packet was not of interest (upcall reply) */
#define ENP_ONGOING          3 /* initialization in progress */
```

4.2 The 'glue' Layer

Once the `ipport.h` file has been implemented as described in the previous section, the next step is to code the glue layers. These are the routines which map the generic service requests that the target system independent code makes to specific services your target system provides. Many may have already been handled through `#define` mapping in `ipport.h`. Ideally many more will have been implemented in the code contained in the target system dependent directory that best fits the description of your target system.

Usually the most complex part of the glue layers is the network hardware interface, described in Network Interfaces. ipport.c should use a routine named prep_ifaces() which initializes the pre-allocated network structure(s) to point to the interface routines, fills in hardware specific parameters, and sets up MIB-II structures. This routine calls the function pointed to by `port_prep` for initializing the interface specific to the target.

Most of the glue layer is described in Porting Engineer Provided Functions. Every function in that section should be either coded or `#define`d to a system function in `ipport.h`.

4.3 Task Control

NicheStack needs to obtain CPU cycles to process received packets and handle time-outs on a timely basis. The stack (and all Internet applications) support two methods of doing this; the "SuperLoop" method (discussed below), which involves regularly polling a central routine, and the `sleep()`/`wake()` method (preferred by multitasking systems) where a network task is blocked (put to sleep) until its services are required (usually because packets are received). The porting engineer will need to choose the method which best fits with the target system and implement the appropriate logic, generally just a few lines of code in `main.c` (or equivalent) file and possibly in `tcpport.c`.

The SuperLoop Method

Many embedded systems do not require a multitasking operating system. They simple execute an infinite loop that invokes each task in succession. Each task runs to completion and then returns to the main loop. InterNiche provides a similar environment called "SuperLoop" because it is more than just a simple "while" loop.

Advantages of a "no os" architecture include:

- There is only one task with a single call-stack. This greatly reduces RAM requirements
- Switching between "tasks" is a simple function call. This is far faster and more efficient than a context switch.
- Locking mechanisms are not needed because no module will ever be interrupted by another.
- Note: Because device drivers may interrupt task execution, CRITICAL_SECTIONS are still required.

Disadvantages of a "SuperLoop" architecture include

- By rapidly looping through tasks looking for one to execute, big-loop systems waste some CPU cycles polling routines when they have no work to do.

TK_YIELD()

TK_YIELD is a macro resolving to the function tk_yield() which is called with no parameters. `tk_yield()` *IS* the scheduler for SuperLoop. It contains the infinite "while" loop. Each time around the loop, it simply goes down the task list and calls each runnable task in succession.

When a task is called by `tk_yield()`, it begins execution from its main entry point. When this task returns, tk_yield() calls the next task. If a task calls tk_yield(), tk_yield() will go around its main loop calling all other tasks that are ready to run and then it will return to the task that called it. That task will resume from the next instruction following its call to tk_yield().

Design Rules

Basic principles for running a SuperLoop module:

- SuperLoop modules should use polling to determine if there is any work to be done or if it is time to do work. If there is no work to be done immediately, it should return. If there is work, it should do that work and then return.
- If a SuperLoop module needs to do something after a period of time (e.g., send a message), it should remember the time and then return. Each time the module is called, it should check to see if it is time to do the action.
- SuperLoop applications should NOT make any blocking calls. This means that `BLOCKING_APPS` should not be defined for the stack (`ipport.h`).
- For the same reason blocking calls to `t_connect()` should NOT be used when SUPERLOOP is defined. A blocking `t_connect` contains an internal while loop that waits for the connection to be made.
- A SuperLoop module should NOT contain an infinite internal loop. That is, it should NOT be constructed in the form:

```
while (1)           /* Do Not Do This */
{                   /* Do Not Do This */
   do work          /* Do Not Do This */
   tk_yield();      /* Do Not Do This */
}                   /* Do Not Do This */
```

- The preferred structure of a SuperLoop task is:

```
main_entry_point() {
   do work
   return
}
```

- While it is not strictly wrong for a module to call tk_yield(), in general, modules should avoid this. A module that calls tk_yield will be scheduled less often than other modules.
- With SuperLoop, TK_SLEEP() should ALWAYS be followed by a return. TK_SLEEP() should only be used when a modules has finished its current work and does not want to be called again for a period of time.
- New in v4.1: `TK_SLEEP()` should ALWAYS be followed by a `return`

For more detailed information, additional examples and elaboration on SuperLoop please contact Support@HCC_Embedded.com.

Multitasking

If the target system has a multitasking operating system (OS), the preferred method of processing received packets and timer events is to create a "network" task at boot time whose job will be to process received packets and network timer ticks. This task's code will look similar to the `main()` routine in the DOS demo, however the loop which calls `tk_yield()` will be replaced with a call to an operating system function that blocks on some sort of event, followed by a call to `pktdemux()`. The code which implements network device drivers should then post the event on which the network task has blocked whenever it has enqueued a received packet into the queue `rcvdq`. For an example of how to enqueue the packet and post the event, see the use of the `SignalPktDemux()` macro in `net/macloop.c`. Timer events may also unblock the task so it can make the required calls to the various timer routines.

InterNiche has sample "main task" software for several popular commercial embedded RTOS systems. Call us for example source code. If we don't have a port for your exact OS, we probably have something quite like it.

4.4 Data Structures

This section describes various data structures the contents of which are important to understand in order to do a port.

The netbuf Structure and the Packet Queues

The `netbuf` structure is used to define a packet that is to be transmitted or that has been received. `PACKET` is typed to be a pointer to a netbuf structure as a convenience. A packet is said to be "chained" if multiple `netbuf` structures are used to define a packet. The packet is "chained" using the `pk_next` and `pk_prev` fields in the `netbuf` structures, and the packet data is stored in the packet buffers pointed to by the `nb_buff` fields.

```
struct netbuf
{
   struct netbuf *next;     /* queue link */
   char      *nb_buff;      /* beginning of raw buffer */
   uint32_t nb_blen;        /* length of raw buffer */
   char      *nb_prot;      /* beginning of protocol/data */
   uint32_t nb_tlen;        /* total length of nb_prots in pk_next list */
   uint32_t nb_plen;        /* length of protocol/data */
   u_long    nb_tstamp;     /* packet timestamp */
   struct net *net;         /* the interface (net) it came in on, 0-n */
   ip_addr   fhost;         /* IP address asociated with packet */
   uint16_t type;           /* IP type set by receiver(rx) or net layer.(tx) */
   uint16_t inuse;          /* use count, for cloning buffer */
   uint16_t flags;          /* bitmask of the PKF_ defines */

#ifdef IP_MULTICAST
   struct ip_moptions *imo;    /* IP multicast options */
#endif

   /* support for chained buffers - scatter/gather - required for IPv6, tunneling, et.al. */
   struct netbuf *pk_prev;     /* previous pkt in chain */
   struct netbuf *pk_next;     /* next ptk in chain */
#ifdef IP_V6
   struct ipv6 *ip6_hdr;       /* "Current" IPv6 header */
   struct in6_addr *nexthop;   /* for pass to ipv6_send() */
   int       nb_pmtu;          /* Path MTU for sends */
#endif   /* IP_V6 */
   struct ip_socopts *soxopts; /* socket options */
};

typedef struct netbuf * PACKET; /* struct netbuf in netbuf.h */
```

`netbuf` structures are allocated and freed dynamically by the network stack via the macros `PK_ALLOC()`, `PK_CONTIG()`, and `PK_FREE()`

The Stack maintains pools of `netbuf` structures and their packet data buffers. The Stack may contain multiple pools of free `netbufs`. All of the packet data buffers in a pool are of the same size. Having multiple pools of different sized packet buffers allows the Stack to create chained packets which more closely match the requested sizes. Each pool of free packet buffers is initialized with a call to `pk_init()`.

The remainder of this section describes the various fields of the `netbuf` structure. Most of this will be informational to the typical porting engineer, though some of the fields are significant to those who are writing network interface code or are using the lightweight UDP interface.

The next field is used to create a linked list of netbuf structures. This linked list is used to implement the bigfreeq and lilfreeq free queues. The next field is not significant between the time a netbuf structure is allocated with `pk_alloc()` and freed with `pk_free()`.

```
struct netbuf * next;   /* queue link */
```

The `pk_prev` and `pk_next` fields are used to implement a packet buffer chain. A `PACKET` is a chained packet if the `pk_next` field in the first `netbuf` structure is not equal to NULL. Otherwise, it is a contiguous `PACKET` and all of the packet data is contained in a single contiguous packet buffer.

```
struct netbuf *pk_prev;    /* previous pkt in chain */
struct netbuf *pk_next;    /* next ptk in chain */
```

`nb_buff` contains the address of the beginning of a data buffer that is used to store data that is to be transmitted or that has been received. `nb_blen` contains the length of this buffer in bytes. Their values should not be modified by any function other than the `pk_init()` function that creates the free queues.

```
char *nb_buff;      /* beginning of raw buffer */
unsigned nb_blen;   /* length of raw buffer */
```

The `nb_tlen` field is only significant in the first `netbuf` structure of a chained packet. Its value is the total amount of protocol data in the chained packet, and is equal to the sum of all of the `nb_plen` fields. The value of this field should be adjusted any time the value of one of the `nb_plen` fields is adjusted.

```
unsinged nb_tlen;       /* total length of the protocol data */
```

`nb_prot` and `nb_plen` are used by the stack to support encapsulation on packet transmission and de-multiplexing on packet reception. For example, when a UDP packet is to be sent and a `netbuf` structure is allocated to contain it, the `nb_prot` field is initially set to point to an offset into the data buffer at which the application constructs the UDP data. The offset chosen is large enough that lower layers in the stack can prefix the data with the UDP, IP and link layer headers. The `nb_plen` field is initially set to the length of the UDP data. As the packet is processed by succeeding lower layers of the stack, `nb_prot` is decreased to point to the beginning of the UDP header, the IP header and eventually the link layer header. Likewise, `nb_plen` is increased to include the sizes of these headers. On packet reception a similar process occurs in reverse. `nb_prot` is initially set to the beginning of the received link layer frame and `nb_plen` is set to the length of the entire received frame. As the packet is processed by succeeding upper layers of the stack, `nb_prot` is increased and `nb_plen` is decreased.

```
char *nb_prot;       /* beginning of protocol data */
unsigned nb_plen;    /* length of protocol data */
```

The porting engineer needs to concern himself with these fields in the following circumstances:

- If the target system application uses the `udp_send()` function, `nb_plen` must be set to the length of the UDP data to be transmitted before `udp_send()` is called.
- If the target system requires the implementation of a network interface, both fields need to be inspected in order to determine what data needs to be transmitted in the `pkt_send()` function. Also, these fields must be properly initialized in netbuf structures that are allocated for received packets.

```
long nb_tstamp;       /* packet timestamp */
```

When received packets are placed into a `netbuf` structure, the current value of `cticks` is stored in `nb_tstamp`. The field is not otherwise used, but can be useful during debugging.

```
struct net *net       /* the interface (net) it came in on, 0-n */
```

`net` contains a pointer to the `net` structure that is associated with the network interface on which a packet is to be transmitted or on which a packet has been received. Applications above the Sockets layer would normally not set this field as this is performed by the stack's IP routing function. Network interface implementations need to set this field to point to the net structure that is associated with the network interface when packets are received.

```
ip_addr fhost;       /* IP address associated with packet */
```

The stack uses the `fhost` field to store the IP address of a packet's "foreign host" where the foreign host is the destination address on transmitted packets and the source address on received packets. Porting engineers would normally not modify a `netbuf` structure's `fhost` field. An exception to this occurs when using the lightweight UDP API in which case the `fhost` field should be set to the destination IP address before the call to `udp_send()`.

```
unsigned short type;     /*i.e. 0800 for IP, filled in by receiver(rx)
                           or net layer.(tx)*/
```

Network interface implementations need to set this field to indicate the link layer protocol type of a received packet. One of two values should be assigned; `ARPTP` for received ARP packets and `IPTP` for received IP packets.

The nets[ ] Array and the netlist

The "net" structure is defined in the file `net.h`. The stack maintains a queue of active `net` structures, `netlist`. For compatibility with previous releases and stack-related code that uses interface indexes to access information about network interfaces, it also maintains an array of pointers to these structures, `nets [ ]`, which is sized by the `MAXNETS` constant that is defined in `ipport.h`. `MAXNETS` therefore sets an upper limit on the number of network interfaces that the stack can support.

The stack also declares storage for the `net` structures that are to be used to represent static network interfaces. Static network interfaces are defined at stack startup time, initialized from the `prep_ifaces()` function, and will persist for the lifetime of the stack. This array, `netstatic[ ]`, can be loaded with default IP address configuration before `ip_startup()` is called. This array is sized by the `STATIC_NETS` constant that can be defined in `ipport.h`. `STATIC_NETS` therefore sets an upper limit on the number of static network interfaces. Note that if `STATIC_NETS` is not defined, it will default to `MAXNETS` for compatibility with previous releases of the stack.

These arrays are shown below:

```
struct net netstatic[STATIC_NETS];
struct net *nets[MAXNETS];
queue netlist;
```

Most of the fields in the net structure are handled internally by the NicheStack code. However, some fields need to be initialized by the driver(s) or by customer defined applications. The porting engineer does not need to deal with these fields. The stack expects the rest of these fields to be initialized by the code that implements the network layers. The porting engineer should make sure that these fields are properly initialized with values that are appropriate for the target system. The function `prep_ifaces()` is called early on during stack initialization. It contains calls to functions that initialize various standard network devices that are supplied with the InterNiche stack. It also calls `port_prep` (if `port_prep` is initialized). It is expected that the porting engineer will either use the standard devices on his target system or initialize `port_prep`, so that a similar interface preparation function gets called, the purpose of which is to initialize the necessary fields of the net structure that is assigned to the interface.

The fields of a `net` structure and their required initialization are described below:

```
char name[IF_NAMELEN]; /* device ID name */
```

`name` should contain a short printable null-terminated character string that identifies the network interface. The stack and network interface drivers (e.g. the `n_stats` function pointer, described below) can be used to provide status and configuration information for the interface.

```
int (*n_init)(int);          /* net initialization routine */
```

`n_init` should point to a network interface initialization function that the stack will call once after all the net structures have been prepared. The purpose of this function is described in Network Interfaces.

```
int (*raw_send)(struct net *, char*, unsigned); /* put raw data on media */ int

        netbuf *); /* send packet on media */
```

raw_send or pkt_send should be set to point to a function which transmits packets over the network interface. One or the other should be chosen with the other set equal to NULL. When the stack needs to transmit a packet on the network interface it will call whichever function is not NULL to do so. The decision of which field to use for a particular port depends on the nature of the network interface hardware. When the stack calls the pkt_send() function, the stack leaves it up to the function to free the PACKET structure when it is done with it (PACKET structures, which describe a packet to be transmitted, are described later). When the stack calls the raw_send() function, the stack frees the PACKET structure once the call returns. Implementations of new interface drivers on modern hardware should implement a pkt_send() function. The raw_send() function is a legacy interface that was used with older network interfaces. The functional requirements of these functions are described in Network Interfaces.

```
int   (*n_close)(int)      /* net close routine */
```

n_close should point to network interface shutdown function that the stack will call once during target system software termination. On embedded targets in which the software never terminates, there is no need for the function addressed by n_close to do anything.

```
int (*n_reg_type)(unshort, struct net*);
```

With some types of standard network interface device drivers (Ethernet device drivers in particular) it is necessary to register the link layer protocol types that will be used with the device. The InterNiche stack currently registers two types: IP_TYPE (0x800) and ET_ARP (0x806). If your target system will be using a standard network interface device driver in which this registration is required, then the n_reg_type field should point to a function that does the necessary registration of the type with the driver.

```
int (*n_stats)(int iface, void *stats);
```

The user interface provided with the stack allows the user to get statistics that are useful during debugging and testing. n_stats should point to a function that returns whatever statistics are deemed pertinent for this purpose.

```
int n_lnh;      /* net's local net header size */
```

n_lnh should contain the length in bytes of the network interface's link layer header. For example, with standard (non-IEEE 802.2/802.3) Ethernet devices n_lnh is set to ETHHDR_SIZE. Ethernet devices use the macro ETHHDR_SIZE, which is usually defined to 14 in the ipport.h file. On systems that require alignment, one mechanism of handling alignment is to make the ETHHDR_SIZE be 16 bytes, which includes an ETHHDR_BIAS of 2 bytes, and on received packets locate data at this ETHHDR_BIAS so that the IP header starts at a 4 byte boundary.

```
int n_mtu; /* net's largest legal buffer size */
```

n_mtu should contain the length in bytes of the network interface's Maximum Transmission Unit or MTU. The MTU defines the maximum size of link layer frame that can be transmitted on the network interface.

The size specified n_mtu should include the length of any link or MAC layer header that is used to encapsulate the IP packets that are transmitted.

```
ip_addr   n_ipaddr;       /* interface's internet address */
```

If a network interface's IP address is to be determined statically (that is no IP address resolution protocol such as DHCP is to be used) then n_ipaddr should be set to the interface's IP address expressed in network order (for a description of network order, see CPU Requirements). This should be performed before the call to ip_startup() to initialize the IP stack. If DHCP is to be used to dynamically determine the interface's IP address, n_ipaddr should be initialized to 0. See IPv4 address configuration protocols: DHCP and auto-configuration for a description of how to use DHCP to determine an interface's IP address.

```
ip_addr snmask; /* interface's subnet mask */ ip_addr n_defgw; /* the default ga

                 this net */
```

If a network interface's IP address is to be determined statically then snmask and n_defgw should be set to the interface's subnet mask and default gateway expressed in network order. This should be performed before the call to ip_startup() to initialize the IP stack. If DHCP is to be used to dynamically determine the interface's IP address, then these fields can be left unassigned because they will be set by the DHCP protocol.

Each target system should have at most a single default route (a.k.a. default gateway) specified. If the target has a default route then that route should be defined in the n_defgw field of the net structure associated with the network interface that is connected to the host that serves as the default gateway and the n_defgw fields of all other net structures should be set to 0. If the target has no default route then all n_defgw fields should be set to 0.

```
unsigned n_hal; /* Hardware address length */
```

n_hal should be set to the length in bytes of the network interface's MAC address. For example, with Ethernet n_hal is set equal to 6.

```
IFMIB n_mib; /* pointer to interface(if) mib structure */
```

IFMIB is typed to point to an IfMib structure. The stack's initialization code sets n_mib to point to an IfMib structure that is associated with the interface. The IfMib structure that is used to store SNMP MIB information that is associated with the interface is discussed in The IfMib Structure.

```
void *n_local; /* pointer to custom info, null if unused */
```

n_local is available for use by porting engineers who need to implement network interfaces. The stack code does not use it.

```
void *n_flags;       /* mask of the NF_ bits below */
```

n_flags is a mask of bit flags used by the stack and by network interface drivers to indicate various per-interface attributes. NF_DYNIF is set by the stack, and indicates that the network interface is a dynamic network interface created by the ni_create() API function. The NF_NBPROT flag is set by the network

interface driver. NF_BCAST and NF_MCAST should be set by network interface drivers if the device can support broadcast and multicast operations, respectively.

```
int (*n_setstate)(struct net *, int); /* set link state up/down */
```

n_setstate is available if the stack has been built with the DYNAMIC_IFACES option defined, and may be set by the network interface driver to point to a driver-provided function that sets the link state for the interface.

```
int (*n_mcastlist)(struct in_multi *); /* register a multicast list */
```

n_mcastlist is available if the stack is built with the IP_MULTICAST option defined, and may be set by the network interface driver to point to a driver-provided function that accepts a list of multicast addresses for the interface.

The IfMib Structure

The IfMib structure is used to maintain the "Interface" MIB as defined in RFC1156. The structure contains a set of fields whose names and purposes match the Interface MIB definition that is contained in the RFC. The stack allocates an IfMib structure for each network interface that is accessible via the n_mib field of the interface's net structure. Since the RFC is readily available to define the contents of the MIB, the contents of the IfMib structure are not further defined in this document. It is the responsibility of a network interface implementation to maintain the various fields that are defined in the MIB, particularly if the target system is to support SNMP agent functionality.

4.5 Initialization

This section discusses various issues that have to do with runtime stack initialization. The steps required to initialize the stack are summarized in the list below:

- Initialization of the network interfaces' net structure IP addressing fields.
- Initialization of the packet buffer queue sizes.
- Initialization of the IP and TCP layers (`ip_startup()`).
- Dynamic IP address resolution (optional).
- Initialization of application servers (optional).

Following this initialization the stack imposes requirements on the target system for various services to be performed, the nature of which will depend on the software architecture of the rest of the target system.

Initialization of net Structure IP Addressing Fields

The net structures and the fields they contain are described in The nets Array and the netlist. The InterNiche stack provides implementations of several network interfaces that can be useful on some embedded systems. These implementations include interface preparation functions that when called will initialize most of the fields of the interface's net structure. These provided preparation functions do not however do the whole job of net structure initialization because some of these fields define target system IP addressing information that must be unique for each instance of the target system.

The initialization of the following IP addressing fields of a target's net structures is, by convention, performed before the interface preparation functions are called, usually in the `main()` function of the target system before the call to `ip_startup()` (we want to emphasize that this is just a convention, not a requirement).

n_ipaddr	interface's IP address.
snmask	interface's subnet mask.
n_defgw	target system's default route.

How the values for these fields are to be determined will depend on the target system. Some target systems will include some sort of non-volatile storage (EPROM, FLASH, disk, etc.) into which these values can be stored and retrieved during system initialization. We refer to this as "static" IP address resolution. Other target systems will use a protocol such as BOOTP or DHCP to allow the network to configure these addresses. We refer to this as "dynamic" IP address resolution.

If a target system is to use static IP address resolution, then these fields should be assigned for each of the target system's interfaces. Note that this information should be stored in these fields in network order. If a target system is to use dynamic IP address resolution, then these fields should be set to `0`. Dynamic IP address resolution is described in IPv4 address configuration protocols: DHCP and auto-configuration.

Initialization of the Packet Buffer Queue Sizes

The packet buffer queues are described in The netbuf Structure and the Packet Queues. The global variables that define the sizes of these queues and the sizes of the big and little packets need to be assigned. There are no hard and fast rules to determine what queue sizes should be used on a given target system. The porting engineer will likely find that the best way to determine these queue sizes is to do a little experimentation with the target system under whatever is considered to be heavy network load for the application.

Initialization of the IP and TCP Layers: ip_startup

The next step is to call the function `ip_startup()`. The function prototype for `ip_startup()` is shown below:

```
char *ip_startup(void);
```

`ip_startup()` returns NULL if successful, otherwise it returns an ASCII string that is descriptive of the error that it encountered that caused it to be unsuccessful. In a properly ported system, `ip_startup()` should always succeed. An unsuccessful return indicates a bug someplace.

The following list summarizes what ip_startup() does:

- Sets the `nets[ ]` array pointers equal to the `netstatic[ ]` array elements, and puts the `netstatic[ ]` array elements on the `netlist` queue, for the first `STATIC_NETS` pointers and elements. See The nets Array and the netlist for a description of these data structures.
- Calls the function `prep_ifaces()`. `prep_ifaces()` in turn calls the interface preparation functions that initialize the target system interfaces' net structures. If a target system port requires that a network interface be written, `port_prep` should be initialized to the interface preparation function before a call to `prep_ifaces()`. `pre_task_setup()` is usually a good place to initialize `port_prep`.
- Initializes the receive queue, `rcvdq`.
- Initializes the big and little packet buffer queues.
- Calls the `n_init()` function for each network interface structure in the `nets[ ]` array. If a target system port requires that a network interface be written, this is where the interface's `n_init()` function will be called.
- Calls a series of stack initialization functions to initialize the various layers of the stack like ARP, IP, and TCP. If additional InterNiche products have been purchased (like the Web Server, NAT router, etc.) many of the functions that initialize these modules will be called at this point.

IPv4 address configuration protocols: DHCP and auto-configuration

The '`flags`' field in each `NET_DEVICE` structure in the `in_devices[ ]` array (defined in `<target directory>/userdata.c`) can be set to a combination of `NF_DHCPC` and/or `NF_AUTOIP` flags to enable the desired set of IPv4 address configuration protocols (DHCP and/or auto-configuration). Three sample configurations are listed below:

```
NET_DEVICE in_devices[ ] =
{
 { &wd_prep, 0x09000072, 0xffffff00, 0x09000001, 0 },
 { &wd_prep, 0x0a000072, 0xffffff00, 0x0a000001, NF_DHCPC },
 { &wd_prep, 0x0b000072, 0xffffff00, 0x0b000001, NF_DHCPC | NF_AUTOIP },
 ...
}
```

When the '`flags`' field is set to 0, the system uses the static IPv4 addressing information in the `in_devices[ ]` entry to update the corresponding network interface structure.

If both DHCP and auto-configuration are enabled on a link, the system first attempts to obtain an address via DHCP. If that fails, it acquires an address via auto-configuration. If DHCP is enabled but hasn't acquired an IPv4 address, it will periodically reattempt to acquire an address.

The `dhc_main_ipset()` function is invoked when the DHCP client successfully acquires an IPv4 address. It displays the IPv4 address, subnet mask, and gateway information that has been acquired from the DHCP server.

The DHCP client can also be used to retrieve name server addresses. This information in stored in the dhc_states[<interface>].dnsrv[ ] array.

The user can also utilize the '`setip`' CLI command to dynamically reconfigure the set of IPv4 address configuration protocols that are used on the link. Please consult the man page for '`setip`' for additional information.

When enabling the DHCP client via the `DHCP_CLIENT` #define, the user must also enable `USE_UPNP`. Auto-configuration is enabled via the `USE_AUTOIP` #define.

Initialization of Application Servers

The final step of initialization on most targets will be to call functions which initialize various application level services. Most of these application level services are provided as separate InterNiche products, the initialization of which is described in the documentation that accompanies those products. If you are using these other InterNiche products with NicheStack, this is the point at which to call their initialization functions.

There are however a few application level services that are part of NicheStack. Their initialization functions are described below. These functions all accept no parameters and return `0` when successful, unless specified otherwise:

ping_init()	Initializes the user interface ping application so that it is possible to initiate pings (ICMP echo requests) from the target system. The call is necessary if it is desired to send ping requests. The stack will respond to ping requests from other hosts with or without the ping application being initialized.
udp_echo_init()	Initializes a UDP echo server in the target that will listen on the standard UDP echo port and respond to UDP datagrams sent to it from other hosts.
tcp_echo_init()	Initializes a TCP echo server in the target that will listen on the standard TCP echo port and respond to connections made to it from other hosts.

It is recommended that the above application services be enabled and initialized on the target system, if only for the duration of target system debugging and testing. They are invaluable tools for debugging and validating a target system and port. The TCP echo server in particular has proven to be good at turning up problems related to heavy network load and throughput that do not surface otherwise.

4.6 Servicing the Stack

After initialization is complete, the stack needs to be serviced in order to run properly. The section Task Control provides some conceptual background for this topic. Whichever tasking method is chosen for a given target, SuperLoop or multitasking, the following functions need to be called on a regular basis in order for the stack and its various application services to function properly. It can be useful to inspect the implementations of the `tk_yield()` function in the various target system dependent directories in order to better understand these stack servicing functions. The functions listed below all take no parameters and return no useful value.

kbdio()	Periodic calls to this function drive the character orient user interface that comes with the stack. The function polls the system console, whatever device that has been configured to be, for user input and when an entire command has been entered, executes the command.
pktdemux()	When this function is called the packet receive queue, rcvdq, is read and the packets contained in it are demultiplexed and passed up to higher layers of the stack. On SuperLoop based systems, this function is usually called in every iteration of the SuperLoop so that received packets are processed in a timely manner. On multitasking systems, this function often gets called from its own task in response to a call to `SignalPktDemux()`.
ping_check()	Calls to this function drive a state machine that handles responses to ICMP echo requests (pings) that originate on the target system. If it is desirable to be able to ping other hosts from the target system using the user interface, `ping_check()` should be called on a regular basis to handle the responses to these pings.
tcp_tick()	The TCP protocol relies on timers. A call to `tcp_tick()` should be made at least once every system tick in order to service the TCP timers.
udp_echo_poll()	If the UDP echo server is configured to run, `udp_echo_poll()` should be called on a regular basis to give this application service some CPU cycles in which to run.
tcp_echo_poll()	If the TCP echo server is configured to run, `tcp_echo_poll()` should be called on a regular basis to give this application service some CPU cycles in which to run.
ip_frag_check()	Periodic calls to this function allow the IP stack to discard partially reassembled IP packets in which some but not all IP fragments have been received within a time out period.
dhc_second()	Periodic calls to this function drive the DHCP state machine described in IPv4 address configuration protocols: DHCP and auto-configuration. It does not need to be called more often than about once per second.
dns_check()	Periodic calls to this function are necessary for the DNS lookup state machine to function properly. It does not need to be called more often than about once per second.

4.7 Applications and Testing

Once your `ipport.h` file is set up and your glue layers are coded, you are ready to test your stack. The traditional first test of most IP stacks is ping, the popular term for ICMP echo packets. All client stacks support this; just go to one of your workstations and try it. At a DOS or UNIX shell prompt, the usual command is "`ping 207.45.67.9`" (the number is an example IP address, use the one you assigned to your stack's network interface). The ping program will send a network packet to the InterNiche stack, which will echo it back. This indicates that the packet got from the net media to your interface, that your interface's IP information is properly configured (at least to some extent), that your IP layer is receiving from the interface and that the ICMP layer is attached to IP. It also indicates the reverse of all this works: ICMP can send to IP, which can send to the interface, which can send on the physical net. If you are using Ethernet, ARP has probably worked too. If ping works, then most of your port is done!

What other tests you can run depends on your product. InterNiche sells FTP servers, SNMP agents, and Web Servers for embedded systems. If you have these you can refer to the manuals that came with them for implementation and testing. If not, then the TCP and UDP echo servers can be used to test the TCP and UDP layers of the stack and to test the target's handling of volume network traffic. To use these servers, you will likely need to write some client test applications on a PC or workstation that can be used to connect to and interact with the servers. The nature of the echo protocol can make the implementation of these clients very simple.

In any case you should now have a working TCP/IP stack. Hopefully your porting was fast, easy, and fun. If you have any suggestions as to how this manual or NicheStack could be easier to understand or port, please contact us with them.

5 NicheTask Multitasking Scheduler

This chapter describes NicheTask, the multitasking scheduler software provided with the stack's default reference port. Tasks are scheduled in a round-robin manner and are not preempted. Once it gains control, each task runs until it voluntarily blocks. It is up to the programmer who uses this package to ensure that tasks do not run for an infinite amounts of time by periodically relinquishing the CPU via calls to one of the blocking `TK_` macros described below.

The NicheTask API is designed so that it can be easily mapped to a more sophisticated (i.e. real time) system by #defining task calls to the RTOS calls. The `TK_` macros are defined in TK_ Macro Definitions. This mechanism allows projects to begin development on a NicheTask and "graduate" to a RTOS if it turns out to be required.

Tasks can be created and deleted dynamically by calls to the tasking API. Each task has a stack and a task control structure. Tasks are created via the `TK_CREATE` macro. The ID of the task is stored in a variable passed to the `TK_CREATE` macro. Subsequent task operations will use the task ID to specify the task on which to operate.

The bulk of the tasking system is written in portable ANSI C code. Four low-level routines are typically required to port NicheTask to a new architecture.

Since NicheTask is being available as Open Source by InterNiche Technologies, Inc., this document makes frequent reference to InterNiche communications protocol and application products. There is no requirement that InterNiche code be used in conjunction with NicheTask, and these references exist only to provide examples of how the tasking system can be used.

5.1 Source Files

The entire C version is implemented in two C language files: `osporttk.c` and `osport.h`.

5.2 osport.h Data Types

To call the tasking package from within a source file, you should include `osport.h` which defines the structures, types ane macros used in the tasking package. There is a task control structure associated with each task instance, and a defined type `TASK`, which is a synonym of that structure. Each task structure contains a pointer to the tasks stack space (described below). This task structure and its stack area are allocated as separate blocks via the `TK_ALLOC()` and `STK_ALLOC()` macros. Generally these are implemented by `calloc()` and `free()`.

5.3 Task Stacks

Each task structure contains a pointer to the task's stack. The pointer is a defined type `stack_t`, usually typedef'fed to an integer; thus each task's stack is an array of integers in memory. Note that some CPU /compiler combinations use stack space starting with the lowest address and go up, i.e. a `push` increments

the stack pointer, and other systems start at the highest address of stack space and move down. The former are referred to as "bottom up" stacks and the later as "top down". You will need to indicate one of these options by #defining either STK_TOPDOWN or STK_BOTTOMUP.

5.4 The Task Control Structure

```
struct task
{
   struct task  * tk_next;      /* pointer to next task */
   stack_t      * tk_fp;        /* task's current frame ptr */
   char         * tk_name;      /* the task's name */
   int            tk_flags;     /* flag set if task is scheduled */
   u_long         tk_count;     /* number of wakeups */
   stack_t      * tk_guard;     /* pointer to lowest guardword */
   unsigned       tk_size;      /* stack size */
   stack_t      * tk_stack;     /* base of task's stack */
   void         * tk_event;     /* event to wake blocked task */
   u_long         tk_waketick;  /* tick to wake up sleeping task */
   IN_SEM       * tk_semaphore; /* per-task semaphore */
   struct task  * tk_waitq;     /* event wait queue, linked list */
}
```

A pointer to a task control structure is used as a process ID in this system. Task control structures form a circular list, chained by the tk_next member of the structure. The scheduler is a round robin scheduler; it simply loops through the circular list of tasks until it finds one which is runnable (that is, a task whose TF_AWAKE is TRUE) and then switches to that task. A context switch merely consists of saving a small amount of state on the stack and calling the routine tk_switch() (which is called by tk_block() more about which you'll read later). When the task later runs again, it resumes execution by returning from the call to tk_switch().

When a task stack is allocated, it is filled with guardwords - a predefined constant used to track stack usage. The tk_guard field points to the last (lowest on STK_TOPDOWN systems) word on the stack. On every context switch, this word is checked to verify that it still contains a guardword. If it doesn't, the tasking package assumes that the task had a stack overflow and aborts the system with a call to the panic() routine. Guardwords are also used by the tk_stats() function to determine how much of the stack has been used.

Finally, there is a global variable, tk_cur, which is a pointer to the task control structure of the task which is currently running. For portability, it is recommended that you should only access tk_cur via macros and not read or set its members directly.

5.5 General Task Behavior

New tasks are created in a suspended state and must be started via the TK_RESUME() macro. Once a task is started, it is not expected to return. Tasks which have finished their job and wish to self-destruct should do so by calling the TK_DELETE macro. Generally, task wakeups should be treated only as hints. When a

task runs, it should try to discover why it was woken up and do the right thing. This includes being able to cope with a seemingly reasonless wakeup. Since tasks can be started before NicheStack is fully ready, this is the suggested logic for a task routine:

```
void task_routine(void *param)
{
   /* declare local variables */

   /* wait for the stack to initialize */
   while (!iniche_net_ready)
      TK_SLEEP(1);    /* delay a short while */

   while (1)
   {
      /* see if there is work to do */
      if (work_to_be_done)
      {
         /* do ongoing work */
      }
      TK_YIELD();    /* let rest of system run */
   }
}
```

There is a global variable `in_tdebug` which is normally set to `FALSE`. When `NPDEBUG` is enabled in `ipport.h` and `in_tdebug` is set to `TRUE`, a message is printed on the console each time a task runs or blocks. Of course, this makes it hard to see anything else which might be going on, but it can be useful to identify which task is crashing the program, what task runs when, and things of this nature.

5.6 Interrupts and Tasks

Interrupt handlers (ISRs) should never call any of the tasking macros other than `TK_SIGNAL` or `TK_RESUME` . In general the only interaction between ISRs and tasks should be the ISR setting or clearing a task's "runnable" flag via `TK_RESUME` or as a result of `TK_SIGNAL`.

5.7 TK_ Macro Definitions

If a task-oriented application will potentially be ported to other multitasking systems, then it is obviously desirable to not tie its design or source code directly to NicheTask's (or any other's) multitasking API. The macros, referred to for obvious reasons as the `TK_` macros, are described in this section. By way of example, all of InterNiche's network protocols and applications are written using a set of macros that can be mapped to virtually any OS or RTOS.

Many of the `TK_` macros map directly to NicheTask calls as well as calls from other popular embedded operating systems.

In InterNiche's networking code, the `TK_` macros are defined in a file named `osport.h`. Projects which support multiple RTOS will have either multiple osport.h files available, or a single `osport.h` file with `#ifdefs` for the different systems. This `osport.h` file should be included in any file using the `TK_` macros. In InterNiche applications this is usually done by including `osport.h` after the inclusion of `ipport.h`; which is ultimately included in nearly all InterNiche source files.

TK_ Task Control Definitions

The `TK_` macros that map to procedure calls fall into two sub-classes - those that create & delete tasks, and those which transfer program control. In tasking systems which do not support dynamic task creation and deletion, the "create" macros may simply finish construction of task management objects or mark a task as runnable - they don't necessarily have to do the actual creation. In applications which do not delete tasks, the "delete" macros may be simply #defined away.

The procedural `TK_` macros are listed here and described in detail in the next section. This example is for the NicheTask system. Some aspects of these definitions are explained below.

```
#define TK_CREATE(code, name, stack, param, prio, taskp)     \
                              tk_new((code), (stack), (name), (prio), 0, (void *)(param),
(taskp))
#define TK_YIELD()            tk_yield()
#define TK_SLEEP(ticks)       tk_sleep(ticks)
#define TK_WAKE(tk)           tk_wake(tk)
#define TK_BLOCK()            tk_block(tk_cur)
#define TK_SUSPEND(tk)        tk_suspend(tk)
#define TK_RESUME(tk)         tk_wake(tk)
#define TK_SIGNAL(s)          tk_sem_port((s))
#define TK_SIGWAIT(s, n)      tk_sem_pend((IN_SEM)(s), (n)
#define TK_DELETE(tk)         tk_del(tk)
```

Tasking System Designs - Spinning vs. Event-blocking

Before describing the `TK_` macros in detail, it is worth explaining the underlying assumptions about how they will be used. The main design goal of these macros is to support InterNiche networking code on a wide variety of systems without any code modifications other than the macro implementations themselves. As with any such system there are some simplicity vs. performance tradeoffs which warrant explaining.

The simplest task model is one where each task remains always ready to run. The tasks run for a reasonable period of time (as determined by the porting programmer) or, more often, until there is no more work to be done. The task then blocks, giving other tasks the opportunity to run in round-robin fashion. Each task has a chance to run in the same fashion. This OS design is often referred to as a "Super Loop."

The advantage of this approach is simplicity - there are no decisions to be made about how and when to suspend or wake up tasks. Each task essentially makes the decision for itself, blocking briefly when there is no work and running when there is. The disadvantage is the inefficiency of waking every task on every pass through the scheduler. This inefficiency is not as bad as it sounds. The InterNiche networking tasks, for example, are written to quickly determine if they should run or not, and return immediately if not. The C code when well optimized on a RISC processor can do the call-test-return sequence in as little as three CPU instructions. This means that this "infinite spinning" system is an excellent choice for most applications.

Some systems, however, require that the tasks not be runnable when there is no work to be done. One example where this is important is a battery powered device which wishes to be able to shut down the CPU to save power, and only does so when all tasks are suspended. An application implementing the `TK_` macros can work on such event based blocking systems if the underlying tasking package is capable of supporting tasks which block pending an event. NicheTask has this capability.

The basic philosophy of InterNiche networking tasks is that each task services a queue or linked list of structures, where each list item represents some aspect of the network activity which may shortly require CPU cycles. The most common examples of these items are TCP connections with received data in the socket structures, and network packets received from hardware devices. When no such circumstance is pending, all tasks suspend themselves with the `TK_SUSPEND()` or the `TK_SIGWAIT()` macros, and the system can be powered down until incoming network traffic (or a user command) awakens it.

Priorities

NicheTask does not support task priorities, instead scheduling each "ready" task to run in a round-robin manner.

5.8 The TK Macros

This section contains detailed descriptions of the `TK_` macros. The syntax illustrated in the synopsis section assumed the macro is treated like a function.

create_device

API Name

`create_device()`

Syntax

```
int create_device(NET ifp, void * bindinfo);
```

Parameters

```
NET ifp /* interface descriptor (pointer to struct net) */
```

```
void * bindinfo /* driver-specific binding information for the device */
```

Description

A `create_device()` function must be supplied for drivers that are written to support dynamic network interfaces. This function must be passed to the stack's `ni_create()` function when creating a network interface; `ni_create()` will call this function so that the driver can bind or attach to a device and complete the initialization of the struct net for the device.

When the stack calls the driver's `create_device()` function, the ifp argument will be a pointer to the newly-created network interface's `struct net`, and the `bindinfo` argument will be the `bindinfo` argument that was passed to `ni_create()`. The `create_device()` function may use the bindinfo argument to locate any addressing or binding information that it needs to initialize the network interface device, and should perform initialization of the supplied struct net, as would need to be done by `prep_ifaces()` and `n_init()` for static network interfaces that are initialized at startup time.

Returns

Returns `0` if OK, else one of the `ENP_` codes.

eth_prep

API Name

`eth_prep() - Setup ethernet nets structure`

Syntax

```
int eth_prep(int index)
```

Parameters

`index`	interface number in the `nets` array

Description

Prepare the ethernet interface structure for this device, including populating the interface function pointers and flags for operation.

Returns

success(1) or failure(0)

get_pticks

API Name

```
get_pticks() - Get fast timer tick count
```

Syntax

```
uint32_t get_pticks(void)
```

Parameters

None.

Description

Counts time since power up. Frequency is PPS (default 100Hz)

Returns

Returns 32 bit tick count of time since power up.

TK_BLOCK

API Name

`TK_BLOCK - Relinquish the CPU to another task`

Syntax

```
void TK_BLOCK();
```

Parameters

None.

Description

The macro is called by the current task when it has no more immediate work to do. In most systems, this macro is equivalent to TK_YIELD(). Execution of the current task is stopped and execution of the next task that is ready to run is started.

Notes/Status

See TK_YIELD() for further discussion.

Returns

Nothing

TK_CREATE

API Name

`TK_CREATE - Create a task`

Syntax

```
int TK_CREATE(void (*code)(void *), char *name, int stack, void *param, unsigne
```

Parameters

code	Pointer to the function to be called when the task is started. This function's prototype is: `void code(void *param)` In a tasking environment, this function should never return.
name	`NUL`-terminated string that is the name of the task.
stack	An integer specifying the size of the task's stack in bytes.
param	The parameter passed to the '`code`' function when the task is started.
prio	An unsigned integer specifying the task's execution priority. If the RTOS does not support task priorities, '`prio`' can be zero.
taskp	A pointer to a variable of type `TASK *`. If '`taskp`' is not NULL, the ID of the created task will be stored in '`taskp`'.

Description

`TK_CREATE()` is called to create a task. This may include allocating a task structure, a stack, or other resources for the task. Note that in some tasking systems the task structures and stack memory are statically declared and only need to be activated, while in others, such as NicheTask, the tasks and stacks are allocated from the heap. After the task is created, it is left in the `SUSPENDED` state; `TK_RESUME()` must be called to set the task to run.

Tasks may be started by the system at any time after creation. Tasks should be coded to test for any required resources or conditions as they start executing. An example of this is the netmain_mod.c application tasks, which test the global variable `iniche_net_ready` before commencing network I/O.

Returns

`TK_CREATE` returns `ESUCCESS` if the task was successfully created, and `EFAILURE` otherwise.

TK_DELETE

API Name

`TK_DELETE - Delete a task`

Syntax

```
void TK_DELETE(TASK tk);
```

Parameters

tk	Task ID of the task to delete. A value of `TK_THIS` is equivalent to the ID of the calling task.

Description

Terminates execution of the specified task and deletes the task's control structure and stack. If the task is self-destructing, execution will continue with the next task that is ready to run.

Returns

Nothing

TK_RESUME, TK_WAKE

API Name

`TK_RESUME - Resume execution of a suspended task`

`TK_WAKE - Resume execution of a suspended task`

Syntax

```
void TK_RESUME(TASK tk);

void TK_WAKE(TASK tk);
```

Parameters

tk	Task ID of the task to be woken up.

Description

This is called to awaken a task which has been suspended via `TK_SUSPEND()` or to start a task that has just been created. The task is marked ready to run. The task will not resume execution immediately unless it is of a higher priority than the current task.

Returns

Nothing

TK_SIGNAL

API Name

TK_SIGNAL - Signal a task from another task

Syntax

```
int TK_SIGNAL(IN_SEM s);
```

Parameters

s	s is a semaphore object.

Description

When a semaphore is signaled, all tasks that are waiting for the signal (see TK_SIGWAIT()) are set ready to be run. TK_SIGNAL() is similar to TK_RESUME() except that a signal can occur before a task is ready to wait for it. In that case, the signal is recorded, and the task's call to TK_SIGWAIT() will return immediately.

Interrupt handlers should use TK_SIGNAL_ISR() rather than TK_SIGNAL() to wake up a task due to the asynchronous timing between the calls to TK_SIGNAL() and TK_SIGWAIT().

Returns

ESUCCESS if the signal was recorded successfully and EFAILURE if there is an error recording the signal.

TK_SIGNAL_ISR

API Name

`TK_SIGNAL_ISR - Signal a task from an interrupt routine`

Syntax

```
int TK_SIGNAL_ISR(IN_SEM s);
```

Parameters

s	Semaphore to be signaled.

Description

The semaphore is signaled. If a task is waiting for the semaphore via a call to `TK_SIGWAIT()`, the task will be set ready to run. If no task is waiting, then its next call to `TK_SIGWAIT()` will return immediately becasue the signal will already be present.

On preemptive systems, if the priority of the waiting task is higher than the priority of the interrupted task, a task switch may occur when the interrupt handler returns.

Returns

`TRUE` if a task switch will occur and `FALSE` if a task switch will not occur.

TK_SIGWAIT

API Name

`TK_SIGWAIT - Wait for a signal`

Syntax

```
int TK_SIGWAIT(IN_SEM s, long timeout);
```

Parameters

s	A semaphore object.
timeout	The number of cticks to wait for the signal to occur.

Description

Suspends the calling task until the specified semaphore is signaled or until the specified number of cticks has elapsed. If the semaphore has already been signaled, `TK_SIGWAIT()` returns immediately.

Returns

`ESUCCESS` if the signal was received before the timeout elapsed, `EFAILURE` if there was an error in signaling the semaphore and `TK_TIMEOUT` if the timeout elapsed before the signal was received.

TK_SLEEP

API Name

`TK_SLEEP - Pause a task for a period of time`

Syntax

```
void TK_SLEEP(unsigned long ticks);
```

Parameters

ticks	Number of CTICKs to wait before being scheduled.

Description

Execution of the calling task is suspended for the specified number of system clock ticks (`CTICK`s). On InterNiche networking systems, clock ticks are tracked by the variable cticks, and the frequency is defined by `TPS` (ticks per second).

Tasks put to sleep with this call may be awakened before the indicated time by a call to `TK_RESUME()`.

Returns

Nothing

TK_SUSPEND

API Name

`TK_SUSPEND - Suspend execution of a task`

Syntax

```
void TK_SUSPEND(TASK tk);
```

Parameters

tk	Task ID of the task to be suspended. A task ID value of TK_THIS refers to the current task.

Description

When a task is suspended, the task flags are set to not ready. If the task being suspended is the current task, it is as if the current task called `TK_BLOCK()`. The task will not be run again until another task or interrupt handler calls `TK_RESUME()` with the suspended task's ID.

Returns

Nothing

TK_YIELD, tk_yield

API Name

`TK_YIELD () - Relinquish the CPU to another tasktk_yield() - Relinquish the CPU to another task`

Syntax

```
void TK_YIELD(void);
```

Parameters

None

Description

`TK_YIELD()` is called when the task code wants to wait for something to occur - a situation often referred to as a "busy wait". The `TK_YIELD()` primitive must give other tasks a chance to run, yet resume the calling task in a short interval. On a round-robin system like NicheTask this is easy - you simply mark to current task as runnable an call the round-robin scheduled.

On an RTOS where tasks have priorities, this can be somewhat trickier to implement. These systems sometimes support a call which will let tasks of equal or greater priority run, by not lower priority tasks. A task spinning on such a `TK_YIELD()` macro would never allow a lower priority task to run.

One remedy for this is to code the `TK_YIELD()` macro to put the task to sleep for a single clock tick. This will force it to wait a reasonable interval during which lower priority tasks may potentially get some cycles. The draw back is that even when the system has nothing else to do, the task spinning on will never be able to utilize all the CPUs power - it will always spend a certain amount of time gratuitously blocked.

Notes

The `tk_yield()` macro (same name in lower case) is identical to the uppercase version. It is supported for historical reasons.

Returns

Nothing

5.9 User Tasking Functions

Descriptions of functions in the tasking system follow.

tk_block ()

API Name

`tk_block ()`

Syntax

```
void tk_block(void);
```

Parameters

None

Description

`tk_block()` takes no arguments and returns no value. It blocks the currently running task. Since the tasking system is non-preemptive, this is the only way for another task to gain control of the processor. Other tasking system entry points which swap tasks do this by first setting a wake condition and then calling `tk_block()`. This routine returns the next time the task which blocked runs.

`tk_block()` contains the heart of the scheduler. It runs through the circular list of tasks until it finds a runnable task and then performs a context switch to that task, which then sees its last call to `tk_block()` return. `tk_block()` calls `tk_switch()` to perform the actual context switch.

Returns

Nothing

tk_del()

tk_del() - Delete a task

API Name

```
tk_del() - Delete a task
```

Syntax

```
void tk_del(TASK task);
```

Parameters

`task`	task to delete; a value of `TK_THIS` means delete the current task

Description

The specified task is removed from the task queue and its resources are freed. If the task is the calling task, `tk_del()` does not return, and the next ready task is scheduled for execution.

Returns

Nothing

tk_exit ()

tk_exit ()

API Name

`tk_exit ()`

Syntax

```
void tk_exit(void);
```

Parameters

None

Description

This function causes the current task to die. When another task becomes runnable, this task's task control structure and stack will be deallocated. This routine should not be called from interrupt level and no further references should be made to this task's task control structure after this call is made.

Returns

`k_exit()` never returns.

tk_init_os

tk_init_os() - Initialize the operating system

API Name

`tk_init_os() - Initialize the operating system`

Syntax

```
void tk_init_os(void);
```

Parameters

None

Description

Implements the `TK_INIT_OS()` macro. This is a placeholder function to allow the porting engineer to perform any operating system initialization steps prior to calling the NicheStack initialization function. These steps might include initializing global task structures or creating semaphores and mutexes.

Returns

Nothing

tk_kill

tk_kill ()

API Name

`tk_kill ()`

Syntax

```
void tk_kill(task * tk);
```

Parameters

tk	Pointer to the task to be killed

Description

This function kills a task. The task is immediately removed from the list of tasks and its stack is deallocated. Tasks should not call `tk_kill` with their own task pointer, they should use `tk_exit()` instead.

Returns

Nothing

tk_new

Syntax

```
int tk_new(void (*start)(void *),
           int stksiz,
           char *name,
           u_int priority,
           uint32_t flags,
           void *arg,
           TASK **taskp);
```

Parameters

start	Address of the task function
stksiz	Size of task's stack in bytes
name	Name for the new task
priority	Task priority
flags	Tasking flags
arg	Argument for the task function
taskp	Pointer to created task structure

Description

This call creates a new task, setting up a task control structure and a stack for it. When a task is created, it is created in the `SUSPENDED` state. A call to `TK_RESUME` is required to start a task executing.

`tk_new()` builds a stack frame for this function by calling the lower level function `tk_frame()`, so that the first time the task runs, it can utilize its stack. The task runs by simulating a call to its '`start`' function with the parameter '`arg`'. The '`start`' function should never return. When a task has completed execution, it should call `TK_DELETE` to remove the task from the system.

This is an example of 'start' for a task which prints a message and then loops forever, printing further messages.

```
void
task1_entry_point()
{
   printf("Hello world. \n");
   printf("task1: starting up. \n");

   while(1)
   {
      printf("- still running -");
      TK_YIELD();
   }
}
```

Note: It is generally good form to enclose the main body of the task in a 'while' or 'for' loop.

If the task was created successfully, and 'taskp' is not NULL, a pointer to the task control block is stored in the variable pointed to by 'taskp'. This value should be used to reference the task in subsequent task functions.

Returns

ESUCCESS or an error code.task

tk_res_lock

tk_res_lock() - Request ownership of a NicheStack resource

API Name

```
tk_res_lock() - Request ownership of a NicheStack resource
```

Syntax

```
int tk_res_lock(int resid, int32_t timeout);
```

Parameters

`resid`	index of the NicheStack resource mutex
`timeout`	number of CTICKs to wait before timing out

Description

NicheStack uses a number of mutexes to control access to the shared NicheStack resources. This function implements the `LOCK_NET_RESOURCE()` macro. The macro specifies a timeout value of `INFINITE_DELAY`, so the function will only return after the resource as been acquired.

Returns

ESUCCESS or an error code

tk_res_unlock

tk_res_unlock() - Release ownership of a NicheStack resource

API Name

```
tk_res_unlock() - Release ownership of a NicheStack resource
```

Syntax

```
void tk_res_unlock(int resid);
```

Parameters

`resid`	index of the NicheStack resource mutex

Description

Implements the `UNLOCK_NET_RESOURCE` macro.

Returns

Nothing

tk_sleep()

tk_sleep() - Delay the current task for a period of time

API Name

```
tk_sleep() - Delay the current task for a period of time
```

Syntax

```
void tk_sleep(uint32_t ticks);
```

Parameters

ticks	number of CTICKs to delay the current task

Description

Execution of the current task is suspended until the specified number of CTICKs has elapsed.

Returns

Nothing

tk_start_os

tk_start_os() - Start the OS task scheduler

API Name

```
tk_start_os() - Start the OS task scheduler
```

Syntax

```
void tk_start_os(void);
```

Parameters

None

Description

Implements the `TK_START_OS()` macro. This function sets the global '`iniche_os_ready`' variable to TRUE and begins execution of the first task. If the OS was already running, then the `iniche_os_ready` variable could be set in the `TK_START_OS()` macro. If the OS is not running, `tk_block()` is called to schedule the first task.

Returns

Nothing

tk_stats

tk_stats() - Show status for all threads

API Name

```
tk_stats() - Show status for all threads
```

Syntax

```
void tk_statsl(void *gio);
```

Parameters

`gio`	GIO to write output to.

Description

Write status of each task to the provided GIO, information on each task should include things like run state, stack usage, and anything else that may be interesting to system debug.

Returns

Nothing

tk_suspend()

tk_suspend() - Suspend execution of a task

API Name

```
tk_suspend() - Suspend execution of a task
```

Syntax

```
void tk_suspend(TASK *task);
```

Parameters

`task`	task to suspend; a value of `TK_THIS` means suspend the current task

Description

Execution of the specified task is stopped. A call to `tk_wake()` by another task or interrupt routine is required to resume execution of this task. When execution is resumed, the task will return from where it was suspended.

Returns

Nothing

tk_wake

tk_wake() - Suspend execution of a task

API Name

```
tk_wake() - Suspend execution of a task
```

Syntax

```
void tk_wake(TASK *task);
```

Parameters

task	task to resume

Description

The suspended task is marked ready to run. If the task has a higher priority than the task that called `tk_wake()`, execution of the suspended task begins immediately.

Returns

Nothing

5.10 Low-Level Routines

The following functions are internal to the tasking system only and should not be called from outside of it. These routines are generally not portable across processors (or sometimes even assemblers), and as such need to be implemented for each target system. These are usually written in machine (Assembly) language.

tk_frame

tk_frame ()

API Name

```
tk_frame ()
```

Syntax

```
tk_frame(task *tk, int (*entry)(), int arg);
```

Parameters

tk	Address of the new task's control stucture.
entry	Address of the new task's entry point
arg	Paramter passed to the new task's entry point when run for the first time.

Description

This routine builds the actual stack frame for a task. The task is not set to runnable by `tk_frame()`; this must be done by the caller if desired.

This is called by `tk_new()`.

Returns

Nothing

tk_getsp

tk_getsp()

API Name

`tk_getsp()`

Syntax

```
stack_t * tk_getsp(void);
```

Parameters

None

Description

This returns the current stack pointer. It should just place the current stack pointer into a return register as expected by the C compiler. The value of the system's stack may change when this call returns, but the calling code allows for this.

This is called from `tk_init()` to find the main task's stack.

Returns

Returns the current stack pointer.

tk_getspbase

tk_getspbase() - Get the address of the system stack

API Name

```
tk_getspbase() - Get the address of the system stack
```

Syntax

```
stack_t *tk_getspbase(void);
```

Parameters

None

Description

Returns the address of the stack used by the embedded device prior to starting the operating system. This stack becomes the task stack for the main NicheStack task. This function is called within `tk_new()`.

Returns

A pointer to the start of the system stack.

tk_switch

tk_switch ()

API Name

`tk_switch ()`

Syntax

```
tk_switch(task *tk);
```

Parameters

tk	Address of the task which will be "switched-in"

Description

This routine performs the actual context switch to the task whose control block is pointed to by `tk`. When the current task runs again, the call that was made to `tk_switch()` will return.

This is called from inside `tk_block()`.

Returns

Nothing

6 Porting Engineer Provided Functions

The functions described in this section must be provided by the porting engineer as part of the NicheStack port. The `w32_in_vc` reference port can be referenced for examples. Further functions will be required as described in Network Interfaces to implement a network hardware driver.

In the demo packages these functions are either mapped directly to system calls via MACROS in `ipport.h` or they are implemented in the port files.

6.1 General Functions

npalloc, npfree

API Name

`npalloc`

`npfree`

Syntax

```
void *npalloc(int size);
```

```
void *npfree(void *);
```

Description

All the IP stack's dynamic memory is allocated by calls to `npalloc()` and released by calls to `npfree()`. The syntax for these is exactly the same as the standard C library calls `malloc()` and `free()`, with the exception that buffers returned from `npalloc()` are assumed to be pre-initialized to all zeros. In this respect `npalloc()` is like `calloc()`.

If the target system already supports standard `calloc()` and `free()` calls, all that is necessary is to add the following lines to `ipport.h`:

```
#define npalloc(size) calloc(1, size) #define npfree(ptr) free(ptr)
```

In the event your target system does not support `calloc()` and `free()`, you will need to implement them. An exhaustive description of how these functions work and sample code is available in "The C Programming Language" by Kernighan and Ritchie.

The great majority of the calls to `npalloc()` are made at initialization time. Only the UDP and TCP layers require these calls during runtime. If your system has severe memory shortages, then these layers can be modified to use pre-allocated blocks of static memory rather than implement a fully functional `npalloc()` and `npfree()`, but this is invariably more work and puts limits on the number of simultaneous connections which can be supported.

One issue that must be dealt with on some target processors is memory alignment. Some processors will generate faults if instructions to read or write more than one byte of memory at a time from odd numbered addresses are executed. Even if the target processor supports 2 or 4 byte reads and writes to odd numbered addresses, instructions of this sort usually execute more slowly than accesses to addresses that are an integer multiple of the number of bytes accessed by the instruction. System performance can suffer. If the target system supports the standard `calloc()` and `free()` calls then the C library vendors have probably already made sure that the buffers returned by `calloc()` are properly aligned for the target processor. However if you need to implement `npalloc()` and `npfree()` without the standard C library memory allocation functions then you should implement these functions such that the memory blocks are aligned on addresses that are a multiple of the data bus width of the target processor. If you don't know the data bus width of the target processor, 4 is usually a safe guess.

Returns

`npalloc()` returns a pointer to the block allocated or `NULL` if memory is unavailable.

tcp_sleep

tcp_sleep()

API Name

`tcp_sleep()`

`tcp_wakeup()`

Syntax

```
void tcp_sleep(void *address);

void tcp_wakeup(void *address);
```

Description

These functions provide a mechanism by which the InterNiche TCP code can yield control of the target processor while waiting for one or more events to occur. The functions' address parameters provide a mechanism by which the source of the events can be synchronized.

See the detailed description of this in the TCP Sleep section of this document.

SignalPktDemux

API Name

`SignalPktDemux()`

Syntax

```
void SignalPktDemux(void);
```

Description

`SignalPktDemux()` is called by network interface code to indicate to the InterNiche IP layer that received packets have been enqueued to `rcvdq`. A call to `SignalPktDemux()` should result in the target system calling the portable `pktdemux()` function to dequeue `rcvdq`.

The implementation of `SignalPktDemux()` is dependent upon whether the target system has a multitasking OS or is implemented as a superloop. If the target system is implemented as a superloop, as is the case in the `w32_in_vc` reference port, `SignalPktDemux()` can be a no-op so long as the superloop calls `pktdemux()` on a regular basis. No explicit notification is necessary in this case.

If the target system has a multitasking OS, `SignalPktDemux()` could still be a no-op so long as a task was created that periodically called `pktdemux()` to empty `rcvdq`. This approach could be made to work but is less than optimal in systems with a multitasking OS. The preferred approach in this case is to create a task that is constructed as a loop in which, on each pass through the loop, the task blocked on some OS dependent event. Upon return from the event block, the task would call `pktdemux()`. In this case, the function of `SignalPktDemux()` would be to post the event on which the task was blocked so as to cause the task to call `pktdemux()`. This approach is illustrated below:

Example

```
void receiveTask()
{
   for ( ; ; )
   {
      block on event X;    /* this call will be dependent on the OS */
      pktdemux();          /* portable function to empty rcvdq */
   }
}

void SignalPktDemux()
{
   post event X;           /* this call will be dependent on the OS */
}
```

dtrap

dtrap()

API Name

`dtrap()`

Syntax

```
void dtrap(void);
```

Description

This primitive is intended to hook a debugger whenever it is called.

See the detailed description in the Debugging Aids section.

Returns

Usually nothing, depends on porting engineer modifications.

dprintf

dprintf()

API Name

`dprintf()`

`initmsg()`

Syntax

```
void dprintf(char *, ...);

void initmsg(char *, ...);
```

Description

These two routines are functionally the same as the standard C library `printf()` function and are called by the stack code to inform the porting engineer or end user of system status. `initmsg()` prints status messages at initialization time. `dprintf()` prints error warnings during runtime.

InterNiche provides an version of `printf()` in `misclib/ttyio.c`. This implementation does not support floating point formats, but is otherwise consistent with standard specifications for the function. The compile-time macro `NATIVE_PRINTF` (`ipport.h`) determines whether this code, or a user/library implementation is to be used.

See the detailed description in Debugging Aids.

console_only

console_only() - Stop all threads except console

API Name

```
console_only() - Stop all threads except console
```

Syntax

```
void console_only(void *pio, bool_t dumpsystem)
```

Parameters

pio	Handle for output. If NULL, output goes to stdout.
dumpsystem	Non-zero means call the `dumpsysinfo` API before suspending tasks.

Description

Suspends all tasks, except the console. If dumpsystem is non-zero, it will call the `dumpsysinfo` API before suspending tasks. During debugging, an engineer could call this API when a special condition occurs, e.g., a dtrap. This API is only available when `NPDEBUG` is defined.

NOTE: The system cannot be returned to a normal state following this API.

Returns

- None

dputchar

dputchar() - Send a character to the console

API Name

```
dputchar() - Send a character to the console
```

Syntax

```
void dputchar(int chr);
```

Description

The InterNiche CLI routines call `dputchar()` in order to display a character on the target system display or monitor port. If such output is not desired, `dputchar()` can be implemented as a no-op. Its parameter is an ASCII character that should be displayed on the target system display or monitor device.

`dputchar()` should perform newline expansion. If the value of `chr` is an ASCII newline character (`0xa`) then a newline followed by a carriage return should be output to the display device.

Returns

Nothing.

kbhit

kbhit() - Pool for character ready from console

API Name

```
kbhit() - Pool for character ready from console
```

Syntax

```
int kbhit(void);
```

Description

`kbhit()` should return a non-zero value if a keystroke has been entered by a user at the CLI of the target system. It should not dequeue the character itself from the input device, rather the return value from `kbhit()` should simply poll the device to determine if a character is present. The entered character is retrieved using the `getch()` function.

Returns

`0` if no character had been entered at the input monitor device, non-zero if at least one character is available.

getch

getch() - Get character from console

API Name

```
getch() - Get character from console
```

Syntax

```
int getch(void);
```

Description

`kbhit()` and `getch()` are used together to effect CLI input. The stack code calls `kbhit()` to determine if a character is available and then if a character is available, calls `getch()` to return the value of the character. **`getch()` should never block for user input**.

Returns

If a character is available at the CLI or system monitor device, `getch()` returns the ASCII value of that character. Its return value is undefined if no character is available.

ENTER_CRIT_SECTION, EXIT_CRIT_SECTION

API Name

`ENTER_CRIT_SECTION() - Enter critical section`

`EXIT_CRIT_SECTION() - Leave critical section`

Syntax

```
#define ENTER_CRIT_SECTION
```

```
#define EXIT_CRIT_SECTION
```

Parameters

None

Description

These two entry points should be designed to be paired around sections of code that must not be interrupted or pre-empted. Usually, `ENTER_CRIT_SECTION()` should save the processor interrupt state and disable interrupts, whereas `EXIT_CRIT_SECTION()` should restore the processor interrupt state to what it was before the most recent call to `ENTER_CRIT_SECTION()`.

See the detailed discussion of these macros see Critical Section Method.

Returns

These return no meaningful value.

Note

Nested calls to ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION() **must** be supported.

LOCK_NET_RESOURCE, UNLOCK_NET_RESOURCE

API Name

`LOCK_NET_RESOURCE() - Resource access lock`

`UNLOCK_NET_RESOURCE() - Resource access unlock`

Syntax

```
void LOCK_NET_RESOURCE(int resID);

void UNLOCK_NET_RESOURCE(int resID);

void WAIT_NET_RESOURCE(int resID, int timeout);
```

Parameters

Any of the `xxx_RESID` constants.

Description

See description of Net Resource Method.

Returns

Nothing.

cksum

API Name

```
cksum() - Calculate buffer checksum
```

Syntax

```
unsigned short cksum (char *buffer, unsigned word_count);
```

Parameters

```
char *buffer /* pointer to buffer to checksum */

unsigned word_count /* number of 16 bit words in buffer */
```

Description

Returns 16 bit Internet checksum of buffer. Algorithms for this are described in RFC1071.

NOTE: A portable C language version of this routine is provided with the demo packages, however TCP implementations can spend a significant portion of their CPU cycles in the checksum routine. This routine is described here to encourage porting engineers to optimize their ports by implementing their checksum routines in assembly language. An Intel x86 assembly language checksum routine is also included which can be used on Intel processors as is. Versions for other CPUs are widely available - contact us if you need help finding one.

Returns

The 16 bit checksum.

panic

panic()

API Name

```
panic()
```

Syntax

```
void panic(char *msg);
```

Parameters

```
char *msg /* short test message describing the fault */
```

Description

`panic()` is called if the InterNiche stack software detects a fatal system error. `msg` is a string describing problem. What this should do varies with the implementation. In a testing or development environment it should print messages, hook debuggers, etc. In an embedded controller, it should try to restart (i.e. warm boot) the system.

Sample for a DOS application is shown below.

Returns

Generally there is no return from this routine, however it is sometimes useful to allow a return under control of a debugger.

Example

```
void
panic (msg)
   char *msg;
{
   dprintf("panic: %s\n", msg);
   dtrap();      /* try to hook debugger */
   netexit(1);   /* try to clean up */
}
```

sysuptime

API Name

```
sysuptime()
```

Syntax

```
unsigned long sysuptime(void);
```

Parameters

None

Description

Returns the number of 1/100ths of a second that have elapsed since the target system was last booted.

Returns

See above.

Example

```
unsigned long sysuptime()
{
   return ((cticks/TPS)*100);   /* 100ths of a sec since boot time */
}
```

6.2 Network Interfaces

The stack allocates a `net` structure for each network interface. This structure is described in The nets Array and the netlist. Each net structure includes pointers to functions that implement the network interface. The porting engineer will need to write the routines listed below for each hardware/device driver to be used in the target system or use one of the standard devices provided with the stack.

Network hardware interface routines:

```
/* net initialization routine */
   int (*n_init)(int);

/* send data on media */
   int (*raw_send)(struct net *, char *, unsigned);
/* OR */
   int (*pkt_send)(PACKET pkt);

/* net close routine */
   int (*n_close)(int);

/* register a MAC type, i.e. 0x0800 for IP */
   int (*n_reg_type)(unshort, struct net*);

/* per net statistics, in addition to n_mib */
   void (*n_stats)(int, void *);

#ifdef DYNAMIC_IFACES
/* set link state up/down */
int (*n_setstate)(struct net *, int);
#endif /* DYNAMIC_IFACES */
```

The porting engineer must also provide a packet receive mechanism which takes received packets and places them in the queue `rcvdq`. This receive routine should obtain its data buffers from `PACKET` structures via calls to `pk_alloc()`.

The remainder of this section describes these routines in detail.

n_close

n_close() - Accessor to shutdown network interface

API Name

```
n_close() - Accessor to shutdown network interface
```

Syntax

```
int n_close(int if_number);
```

Parameters

```
int if_number /* index into nets[ ] for NET to close */
```

Description

Does whatever is necessary to restore the device and its associated driver software prior to exiting the application. This function may not be required to do anything on embedded systems which start their devices at power up and don't have any reason to shut them down. If packet types (i.e.: `0x0800` for IP and `0x0806` for ARP) have been accessed in a lower layer driver, they should be released here.

Returns

Returns `0` if OK, else one of the `ENP_` codes.

n_init

n_init() - Accessor to startup network interface

API Name

```
n_init() - Accessor to startup network interface
```

Syntax

```
int n_init(int if_number);
```

Parameters

```
int if_number /* interface number, for indexing nets[ ] */
```

Description

This routine is responsible for preparing the device to send and receive packets. It is called during system startup time after `prep_ifaces()` has been called, but before any of the other network interface's routines are invoked. When this routine returns, the device should be set up as follows:

- Net hardware ready to send and receive packets.
- All required fields of the net structure are filled in.
- Interface's MIB-II structure filled in as show below.
- IP addressing information should be set before this returns unless DHCP or BOOTP is to be used. See the section titled **"Initialization of net Structure IP Addressing Fields**".

This will usually include hardware operations such as initializing the device and enabling interrupts. It does not include setting protocol types. This is handled later (see the section **n_reg_type**. Upon returning from this routine it is safe for your hardware's interrupt or receive routines to start enqueuing received packets in the `rcvdq`. Packets which are not IP or ARP will be discarded by the stack.

The `nets[ ]` structure array element that is indexed by `if_number` should be completely filled in when this function returns. Note that the work of filling this structure is shared between `prep_ifaces()` and this function, so if all `nets[ ]` structure setup was done in `prep_ifaces()` (see **The 'glue' Layer**) there may be nothing to do here.

Below is an example of code that can be used for setting up the MIB structure for a 10 Mbps Ethernet interface. The `n_mib` field of the `nets[ ]` structure points to a structure that is used to contain the MIB information which has already been statically allocated by the calling code. See RFC1213 for detailed descriptions of the MIB fields. Most of the MIB fields are used only for debugging and statistical information, and are not critical unless your device is managed by SNMP. The `ifPhysAddress` field is an exception. It is used by ARP to obtain the hardware's MAC address and MUST be set up correctly for the IP stack to work over Ethernet. Note that although `ifPhysAddress` is a pointer, it does not point to valid memory when the MIB structure is created. The porting engineer

should make sure it points to a static buffer containing the MAC address before this function returns. The size of this address is determined by the media (6 bytes for Ethernet) and should be set in the `nets[ ]` structure member `n_hal` (hardware address length).

```
u_char macaddress[6];   /* should contain interface's MAC address */

nets[if_number]->n_mib->ifDescr = "Ethernet Packet Driver";
nets[if_number]->n_mib->ifType = ETHERNET; /* SNMP Ethernet type */
nets[if_number]->n_mib->ifMtu = ET_MAXLEN;
nets[if_number]->n_mib->ifSpeed = 10000000; /* 10 megabits per second */
nets[if_number]->n_mib->ifAdminStatus = 1;
nets[if_number]->n_mib->ifOperStatus = 1;
nets[if_number]->n_mib->ifPhysAddress = ...macaddress[0]; /* example */
```

Returns

Returns `0` if OK, else one of the `ENP_` codes.

n_refill

n_refill - Replenish device driver's internal resources

API Name

```
n_refill - Replenish device driver's internal resources.
```

Syntax

```
void (*n_refill)(int iface);
```

Parameters

iface	interface number of device to refill

Description

Refills the device's internal packet buffer pool by calling PK_ALLOC() or PK_CONTIG to obtain packet buffers from the Stack's free packet buffer queues. The 'iface' parameter specifies the index of the device in the 'nets[iface]' array. The FREEQ_RESID resource should be locked within the 'n_refill' function prior to allocating the packets. The number of packets and their sizes is dependent upon the design of the driver. If there are multiple devices in the system, the developer can implement a single 'n_refill' function for all of the devices or a separate 'n_refill' function for each device.

The 'n_refill' function is called in the pktdemux() function which is normally part of the main NicheStack task. The 'n_refill' function should no consider it an error if the device's internal packet buffer pool cannot be completely refilled.

Returns

Nothing

n_reg_type

n_reg_type()

API Name

`n_reg_type()`

Syntax

```
int n_reg_type(unshort type, NET net);
```

Parameters

`unshort type NET net`

Description

Register with any lower level drivers to receive a MAC type, i.e. `0x0800` for IP and `0x0806` for ARP. On most embedded systems with Ethernet, where the InterNiche stack does not share the hardware with other network stacks, no action is required. Since the InterNiche stack gets all the packets anyway, `n_reg_type()` can simple return an `OK` code without doing anything. The porting engineer should be sure, however, that all received packets will be passed to the stack. Note that on some driver subsystems a type must be registered with the driver informing it that we are interested in the packets.

On SLIP links, all packets are IP, so nothing has to be done in `n_reg_type()`.

On PPP links, PPP will sort out the packets, so again, nothing has to done in `n_reg_type()`.

Returns

Returns `0` if OK, else one of the `ENP_` codes.

n_setstate

n_setstate()

API Name

`n_setstate()`

Syntax

```
int n_setstate(struct net * ifp, int opcode);
```

Parameters

```
struct net * ifp /* pointer to net structure whose state is to be set */
```

```
int opcode /* operation code */
```

Description

`n_setstate()` allows the driver to receive requests to mark the interface up or down when the application calls the stack's `ni_set_state()` API function. This function is optional: it is not used if the stack is built without the `DYNAMIC_IFACES` option defined and so does not support dynamic network interfaces, and if it is not supplied, the `n_setstate` member of struct net may be set to `NULL` to inform the stack that it is not to be used.

When `n_setstate()` is called, the ifp argument is set to the struct net for the network interface whose state is to be set, and the opcode argument will be set to `NI_UP` if the network interface is to be marked as "up", or `NI_DOWN` if it is to be marked as "down".

Returns

Returns `0` if OK, else one of the `ENP_` codes.

n_stats

n_stats() - Accessor to get network interface statistics

API Name

```
n_stats() - Accessor to get network interface statistics
```

Syntax

```
int (*n_stats)(int iface, void *stats);
```

Parameters

```
int iface /* interface number to dump statistics for */

void * stats /* pointer to a user defined structure */
```

Description

OPTIONAL: `n_stats()` enables the driver to provide hardware specific information which is not included in the generic MIB-II interface group. This information might include hardware specific error counters, such as the number of collisions on an Ethernet link; or internal resource information, such as the status and number of current buffers available on a ring-buffer device. The definition of the '`stats`' structure and its contents is left to the driver writer. An example is the `enet_stats` structure in `h/ether.h`.

Returns

The function returns `ESUCCESS` if it is successful and `EFAILURE` if an error, such as a parameter value out of range, is encountered.

netbuf Manipulation

netbuf Allocation

pk_alloc() returns a pointer to a netbuf structure that contains a pointer to a packet buffer and other fields that describe the packet to be enqueued. (Note that the source code shows pk_alloc() returning a value of type PACKET. PACKET is typed to be a pointer to a netbuf structure.) The parameter to pk_alloc() specifies the length in bytes of the data to be stored in the packet buffer. The length that should be specified in the call to pk_alloc() should be the length of the received packet, less the length of the MAC layer header, plus the value containing the global integer MaxLnh. In the example shown below, suppose that the local variable frameLen contained the length of a received Ethernet frame. Given this supposition, the example illustrates the correct call to allocate the netbuf structure to be used to enqueue the frame.

```
$body
```

```
int frameLen;
PACKET pkt;

frameLen = length of received Ethernet frame including Ethernet MAC header;
pkt = pk_alloc(frameLen - 14 + MaxLnh);    /* Ethernet MAC header is 14 bytes */
```

When pk_alloc() succeeds in allocating a netbuf structure, it returns a pointer to that structure with an associated packet buffer. If pk_alloc() fails due to a lack of buffers, it returns NULL. When this happens, the packet should be discarded.

netbuf Initialization

Once a netbuf structure has been allocated for the received packet, it needs to be initialized to describe the packet. The code fragments shown below illustrate how the various fields of the netbuf structure need to be initialized.

```
pkt->nb_prot = pkt->nb_buff + MaxLnh;  /* point to start of IP header */
```

nb_buff points to the beginning of the allocated packet data buffer. nb_prot points to the received data less any MAC layer header. For received IP packets, nb_prot ends up pointing to the beginning of the IP header. If any MAC header bias was used to align the start of the IP header, then nb_prot must point past the ETHHDR_BIAS offset. If used in the port the ETHHDR_BIAS is usually set to 2 for Ethernet Devices.

Drivers developed for the 1.8 and later releases, and which have set the NF_NBPROT flag in the device's struct net n_flags field, must set nb_prot to point to the start of the IP header. This can be wherever is appropriate for the device and target so long as nb_prot points into the buffer starting at nb_buff. (Note: Some targets may have alignment requirements that force the IP header to start on a word-aligned boundary. If so, the driver should make sure that nb_prot is set so that it is aligned according to ALIGN_TYPE.

NOTE: *Drivers developed for releases of NicheStack earlier than 1.8 must set `nb_prot` so that it is offset from the beginning of the packet data buffer by the number stored in MaxLnh.*

```
pkt->nb_tstamp = cticks;
```

`nb_tstamp` is time stamped with the current clock tick.

```
pkt->type = IPTP or ARPTP;
```

`type` is used to indicate to the IP layer whether the received packet is an IP packet or an ARP packet. The network interface code must make this determination which will be dependent on the nature of the interface. `IPTP` and `ARPTP` are defined constants in the file `ip.h`.

```
pkt->net = appropriate element of nets[ ] array;
```

`net` should contain a pointer to the `nets[ ]` array element associated with the interface.

```
$body
```

```
int length = frameLength - length of MAC layer header;
pkt->net->n_mib->ifInOctets += frameLength;    /* maintain MIB counter */
pkt->nb_plen = length;
```

`nb_plen` should be set to indicate the length of the received data less any MAC layer header. This value should be added to the MIB counter.

Copy Received Data to netbuf Packet Buffer

The next step is to copy the received packet, less any MAC layer header, to the packet buffer offset stored in `nb_prot`, as illustrated below:

```
MEMCPY(pkt->nb_prot, data, length);
```

Enqueuing netbuf structure to rcvdq

The next step is to enqueue the `netbuf` structure to `rcvdq`. This is performed via a single function call, as illustrated below:

```
putq(...rcvdq, (q_elt) pkt);
```

Signal Packet Demultiplexor

The last step is to signal the packet demultiplexor to allow the portable `pktdemux()` function to be called in order to allow the IP layer to demultiplex the received packets that have been enqueued to `rcvdq`. The network interface software provided by the InterNiche stack calls the port dependent function `SignalPktDemux()` in order to do this. Network interface code written by the porting engineer for a given target system should do the same. See the discussion of SignalPktDemux.

pkt_send

pkt_send() - Insert frame into network driver queue

API Name

```
pkt_send() - Insert frame into network driver queue
```

Syntax

```
int pkt_send(PACKET pkt);
```

Parameters

```
typedef struct netbuf *PACKET

PACKET pkt /* pointer to netbuf structure containing frame to send */
```

Description

This routine is responsible for sending the data described by the passed `pkt` parameter and queuing the `pkt` parameter for later release by the device driver. If the MAC hardware is idle the actual transmission of the packet should be started by this routine, else it should be scheduled to be sent later (usually by an "end of transmit" interrupt (EOT) from the hardware).

The `PACKET` type is described in the section titled **The netbuf Structure and the Packet Queues**". All the information needed to send the packet is filled into the structure addressed by this type before this call is made. Some of the important fields are:

```
pkt->nb_prot;    /* pointer to data to send. */
pkt->nb_plen;    /* length of data to send */
pkt->net;       /* nets[ ]structure for posting statistics */
```

The data addressed by `pkt->nb_prot` may or may not have already been prefixed with a MAC layer header depending on how the `nets[ ]` structure associated with the interface (`pkt->net`) has been configured. The rule for determining whether the MAC layer header is present or not can be expressed with the following pseudocode fragment.

```
if ((pkt->net->n_mibifType == SLIP)
      || (pkt->net->n_mib->ifType == PPP)
      || (pkt->net->n_lnh== 0))
   the packet at pkt->nb_prot is not encapsulated with a MAC header;
else
   the packet at pkt->nb_prot is encapsulated with a MAC header;
```

If the if statement in the above pseudocode evaluates to `TRUE` then the packet at `nb_prot` is not encapsulated with a MAC header and it is up to the network interface code to transmit the MAC header

that is appropriate for the network medium (if any). On the other hand, if the "if" statement evaluates to `FALSE` then appropriate MAC headers for media such as Ethernet or Token Ring will have been placed at the head of the buffer passed by the calling routine and are not the responsibility of this routine; however some drivers may have to access, strip or modify the MAC header if they are layered on top of complex lower layers. The ODI `pkt_send()` routine is an example of this (see `doslib/odi.c`).

Regardless of whether it is the responsibility of the network interface layer to transmit the MAC header, it is necessary for the network interface to transmit the `nb_plen` bytes starting at `nb_prot` plus "any" MAC header bias that was used to align the start of the IP header. For Ethernet devices, the macro `ETHHDR_BIAS` is sometimes defined to 2 bytes, to align the IP header at a 4 byte boundary. Likewise, the number of bytes to transmit in this case would be `(nb_plen - ETHHDR_BIAS)`, if `ETHHDR_BIAS` was defined to non-zero. When all the bytes are sent, the structure addressed by the `PACKET` type should be returned to the free queue by a call to `pk_free()`, which may be called at interrupt time. Do not free the packet before it has been entirely sent by the hardware, since it may be reused (and its buffer altered) by the IP stack.

The simplest way to implement this routine is to block (busy-wait) until the data is sent. This allows for fast prototyping of new drivers, but will generally hurt performance. The usual design followed by InterNiche in the example drivers is to put the packet in an awaiting_send queue, check to see if the hardware is idle, and then call a send_next_from_q routine to dequeue the packet at the head of the send queue and begin sending it. The "end of transmit" ISR (EOT) frees the just sent packet and again calls the send_next_from_q routine. By moving all the PACKETs through the awaiting_send queue we ensure that they are sent in FIFO order, which significantly improves TCP and application performance.

If your hardware (or lower layer driver) does not have an end of transmit (EOT) interrupt or any analogous mechanism, you may need to use the `raw_send()` alternative to this function.

Slow devices (such as serial links), and hardware which DMAs data directly out of predefined memory areas, may copy the passed buffer into driver managed memory buffers, free the `PACKET` and return immediately; however they should be prepared to be called with more packets before transmission is complete.

Interface transmit routines should also maintain system statistics about packet transmissions. These are kept in the `IfMib` structure that is addressed by the `n_mib` field in each `nets[ ]` entry. Exact definitions of all these counters are available in RFC1213. At a minimum you should maintain packet byte and error counts since these can aid greatly with debugging your product during development and isolating configuration problems in field. Statistics keeping is best done at EOT time, but can be approximated in this call. The following fragment of code is a generic example:

```
/* compile statistics about completed transmit */
eth = (struct ethhdr *)pkt->nb_prot;   /* get ether header */
ifc = pkt->net;
if(send_status == SUCCESSFUL)   /* send_status set by hardware EOT */
{
    if(eth->e_dst[0] ... 0x01)      /* see if multicast bit is on */
        ifc->n_mib->ifOutNUcastPkts++;
     else
```

```
        ifc->n_mib->ifOutUcastPkts++;

    ifc->n_mib->ifOutOctets +=pkt->nb_plen;
}
else   /* error sending packet */
{
    ifc->n_mib->ifOutErrors++;
}
```

Returns

Returns `0` if OK, else one of the `ENP_` codes. Since this routine may not be waiting for the packet transmission to complete, it is permissible to return a 0 if the packet has been successfully queued for send or the send is in progress. Error (non-zero) codes should only be returned if a distinct hardware (or lower layer) failure is detected. There is no mechanism to report errors detected in previous packets or during the EOT. Upper layers like TCP will retry the packet when it is not acknowledged.

See Also

raw_send

raw_send

raw_send()

API Name

`raw_send()`

Syntax

```
int raw_send(NET net, char * data, unsigned data_bytes);
```

Parameters

```
typedef struct net *NET
```

```
NET net /* pointer to net structure to send it on */
```

```
char *data /* pointer to data buffer to send */
```

```
unsigned data_bytes /* number of bytes to send (length of data) */
```

Description

This routine should transmit the data as indicated on the device corresponding to the `net` parameter passed. A MAC header may or may not have been prefixed to the IP data depending on how the `nets[ ]` structure addressed by the net parameter has been configured. See the description of MAC headers in the description of the pkt_send function. This routine should not return until it is through with the data in the passed buffer, as the buffer may be reused (thus corrupting the data) immediately upon return.

The `pkt_send()` routine should be used instead of this one if there is an end of transmit interrupt (EOT) available on the hardware. This routine was designed for old DOS "packet driver" specification drivers which did not support EOT and should generally not be used on modern designs.

Slow devices (such as serial links), and hardware which DMAs data directly out of predefined memory areas may copy the passed buffer into driver managed memory and return immediately; however they should be prepared to be called with more buffers before transmission is complete.

Interface transmit routines should also maintain system statistics about packet transmissions. These are kept in the `n_mib` structure attached to each `nets[ ]` entry. Exact definitions of all these counters are available in RFC1213. At a minimum you should maintain packet byte and error counts since these can aid greatly with debugging your product during development and isolating configuration problems in the field. Statistics keeping is best done at EOT time, but can be approximated in this call. The following fragment of code is an example that works for Ethernet devices:

```
/* compile statistics about completed transmit */
eth = (struct ethhdr *)pkt->nb_prot;   /* get ether header */
```

```
if(send_status == SUCCESSFUL)   /* send_status set by hardware EOT */
{
   if(eth->e_dst[0] ... 0x01)         /* see if multicast bit is on */
      ifc->n_mib->ifOutNUcastPkts++;
   else
      ifc->n_mib->ifOutUcastPkts++;

   ifc->n_mib->ifOutOctets +=pkt->nb_plen;
}
else         /* error sending packet */
{
   ifc->n_mib->ifOutErrors++;
}
```

Returns

Returns `0` if OK, else one of the `ENP_` codes.

See Also

pkt_send

rcvdq

Name

`rcvdq`

Syntax

```
queue rcvdq;
```

Description

`rcvdq` is a global structure to which packets received at the MAC layer should be enqueued. For each target system network interface, the porting engineer will need to write code to enqueue packets received on that interface to `rcvdq`. The steps involved in enqueuing received packets to rcvdq are summarized below:

- Allocate a `netbuf` structure to contain the received data by calling `pk_alloc()`.

1. Initialize the fields of the `netbuf` structure to properly describe the received data. See the section titled "The netbuf Structure and the Packet Queues" for a description of the `netbuf` structure fields.
2. Copy the received data to the buffer associated with the `netbuf` structure.
3. Call the `putq()` function to enqueue the `netbuf` structure to `rcvdq`.
4. Call the `SignalPktDemux()` function to signal the IP layer to process the enqueued packet.

The details of each of these steps are described below.

TCP Sleep

Once the `ip` and `net` directory sources have compiled successfully and the glue routines coded, there is usually one more pair of related functions to write. The TCP portion of the stack needs a mechanism to wait briefly if resources (usually free buffers) run short or the other host on a connection runs too slowly. The mechanism for this are the functions `tcp_sleep()` and `tcp_wakeup()`. If TCP is not being used in the target system, the porting engineer doesn't need to implement these. Otherwise he will need to map these into whatever temporary block or sleep function the target system OS provides.

Programmers familiar with the UNIX kernel `sleep()` and `wakeup()` will notice `tcp_sleep()` and `tcp_wakeup()` can be mapped directly to the UNIX `sleep()` and `wakeup()`, respectively.

`tcp_sleep()` and `tcp_wakeup()` both take as a parameter a pointer to some location in memory (the parameter is typed as a `void *`, a pointer to unstructured data). The contents of the memory addressed by the pointer are not significant to the functions and should not be modified by them. What is significant is the address itself. The semantics of these functions in a multitasking environment are such that:

The period of time between making a call to `tcp_sleep()` and that call's return can be up to, or slightly after a `tcp_wakeup()` call is made to the same address.

It is useful to elaborate on some of the fine points of the above semantic definition. The InterNiche stack code interprets a return from `tcp_sleep()` to mean that some significant event (e.g. a packet has been received or a timer has expired) MIGHT have occurred to cause the calling task to continue execution. The calling task always checks other internal variables to determine which, if any, events actually occurred and, if it determines that no significant event occurred, the call to `tcp_sleep()` is repeated. This means it is perfectly acceptable for `tcp_sleep()` to return before a corresponding call to `tcp_wakeup()` is executed. Thus, a simple implementation of `tcp_sleep()` could be to simply delay the calling task for an OS system clock tick. `tcp_wakeup()` in this case would be a no-op. This simple implementation would result in some wasted CPU cycles, but if the network task is executed at low priority, this would not significantly affect overall system performance.

If minimizing CPU cycles in the target system is a requirement, a more sophisticated implementation of `tcp_sleep()` and `tcp_wakeup()` would include an algorithm that mapped the addresses passed as parameters to these functions to OS events or other IPC objects. A call to `tcp_sleep()` would result in a block on an OS event and a call to `tcp_wakeup()` with the same parameter would result in the posting or generation of the same OS event.

On the `w32_in_vc` reference port, `tcp_sleep()` calls our basic "superloop" function, `tk_yield()`, and `tcp_wakeup()` is a no-op. This is the simplest possible of round robin processes, yet it gives excellent TCP performance. See the source code for details. On Windows, `tcp_sleep()` can be a simple windows message loop, as long as `packet_demux()` has a chance to receive incoming packets.

7 Internal Functions

The following pages contain a list of InterNiche internal routines which may be useful to programmers writing customized applications with the InterNiche stack. These functions are a subset of the routines in the libraries that are deemed to be of interest to a TCP/IP application or network interface writer.

7.1 Chained Buffers

pk_free

API Name

`pk_free()`

Syntax

```
void pk_free(PACKET pkt);
```

Parameters

PACKET pkt	Pointer to netbuf structure previously allocated by pk_alloc()

File

`net/pktalloc.c`

Description

`pk_free()` is used to return a previously allocated `netbuf` structure to the pool of such structures that is maintained by the InterNiche stack. When a packet buffer is passed to NicheStack, it is the responsibility of the lowest layer that handles the packet to free it. Normally, this would be the interface driver. However, it may be another layer such as TCP or IP if, because of a timeout or error, it does not pass the packet to a lower layer. The porting engineer should include a call to `pk_free()` in his network interface code in order to return a `netbuf` structure and its associated packet buffer to the free pool after the packet has been transmitted by the network device. For a description of how this is performed, see the description of pkt_send.

Note that if you happen to be implementing Mutual Exclusion using the Net Resource Method, then the FREEQ_RESID resource would need to be locked and unlocked while making calls to `pk_free()`.

Returns

Nothing.

pk_init

API Name

`pk_init()`

Syntax

```
int pk_init(int len, int num);
```

Parameters

len	Length in bytes to be allocated.
num	Number of buffers to allocate.

Description

pk_init() allocates num buffers of length len in a free buffer queue. It will typically be called several times to allocate the various queues of buffers needed by the port. The function will fail if:

- len < 0
- too small: len < (6 * MaxLnh)
- too large: len > MAXCHAINDPKTSZ,
- too many packets : num > MAXCHAINDPKTNUM.
- An entry of that length has already been entered.
- There are no more free slots in the array buffer queues.
- Allocation of buffers has failed for lack of RAM.

Returns

0 for success or -1 if an error occurred.

pk_copy

API Name

`pk_copy()`

Syntax

```
PACKET pk_copy(char * in, int len, int disp);
```

Parameters

in	Pointer to source buffer
len	Number of bytes to be copied
disp	Displacement. Number of bytes to be left in front of the data

Description

pk_copy () allocates a packet chain and copies the specifed amount of data from the source buffer to the newly allocated buffer chain. The nb_prot pointer is set "disp" bytes from nb_buff, and disp bytes are added to the length of the first buffer. The disp value may be negative. Implicitly, MaxLnh is adjusted according to disp.

Returns

Pointer to the newly allocated PACKET or NULL.

pk_gather

API Name

`pk_gather()`

Syntax

```
PACKET pk_gather(PACKET pkt, int headerlen);
```

Parameters

pkt	Pointer to first packet in chain to be coalesced
headerlen	Displacement from nb_buff to nb_prot in the output packet

Description

`pk_gather()` allocates a single packet with a buffer long enough to hold the entire input packet chain. It is typically used when it is necessary to gather a chain into single buffer, for user processing or to pass to a driver that does not do gather.

Note: `pk_gather` uses `npalloc()` to allocate the packet and the buffer, and it sets the flag `PKF_COALESCED`. When `PKF_COALESCED` is set, the interface code must free the packet using `npfree()`, rather than putting the packet back on the free queue.

Returns

Pointer to newly allocated PACKET or NULL.

7.2 ARP

make_arp_entry

make_arp_entry()

API Name

```
make_arp_entry()
```

Syntax

```
struct arptabent *make_arp_entry(ip_addr dest_ip, NET net);
```

Parameters

dest_ip	IP address to enter into table
net	Associated network interface

File

```
ip/et_arp.c
```

Description

Finds the first unused (or the oldest) ARP table entry and makes a new entry to prepare it for an ARP reply. If the IP address already has an ARP entry, the entry is returned with only the time stamp modified. The MAC address of the created entry is not resolved but left as zeros. The eventual ARP reply will fill in the MAC address.

Returns

Returns pointer to ARP table entry selected.

7.3 IP

add_route

add_route()

API Name

`add_route()`

Syntax

RTMIB add_route(ip_addr dest, ip_addr mask, ip_addr nexthop, int iface, int prot);

Parameters

```
ip_addr dest /* ultimate destination */

ip_addr mask /* net mask, 0xFFFFFFFF if dest is host address */

ip_addr nexthop /* where to forward to */

int iface /* interface (net) for nexthop */

int prot /* how we know it: icmp, table, etc */
```

File

`ip/ip.c`

Description

Make an entry in the route table directing `dest` to `nexthop`.

Returns

Returns a pointer to the table entry; so caller can process it further, i.e. add metrics.

ip_mymach

ip_mymach()

API Name

```
ip_mymach()
```

Syntax

```
ip_addr ip_mymach(ip_addr host);
```

Parameters

```
ip_addr host /* IP address of foreign host to find */
```

File

```
ip/ip.c
```

Description

Returns the address of our machine relative to a given foreign host IP address. On a single homed host this will always return the sole interface's IP address; on a router it will return the address of the interface to which packets for the host would be routed.

Returns

Our IP address on one of our networks interfaces.

iproute

iproute()

API Name

`iproute()`

Syntax

```
NET iproute(ip_addr host, ip_addr *hop1);
```

Parameters

```
ip_addr host /* IP address of final destination host */
```

```
ip_addr *hop1 /* IP address to use in resolving MAC address */
```

File

`ip/ip.c`

Description

Performs IP routing on an outgoing IP packet. Takes the Internet address to which we want to send a packet and returns the net interface through which to send it. An IP address is returned pointed to by the output parameter `hop1` which is the IP address for resolving the MAC destination address of the packets. If the target host is on our local segment, `hop1` will be the same as `host`, else it will be the IP address of the gateway or router through which we might be able to reach `host`.

Returns

Returns a pointer to a `net` structure which describes the interface of the MAC media we should send the packet on. Returns `NULL` when unable to route.

parse_ipad

API Name

`parse_ipad()`

Syntax

```
char * parse_ipad(ip_addr * ipout, unsigned * sbits, char * stringin);
```

Parameters

ipout	pointer to IP address to set
sbits	default subnet bit number
stringin	buffer with ascii to parse

File

`misclib/parseip.c`

Description

Looks for an IP address in `stringin` buffer, makes an IP address (in big-endian) in `ipout`.

Returns

Returns `NULL` upon success, else returns a pointer to a string describing the syntax problem in the input string.

print_ipad

API Name

`print_ipad()`

Syntax

```
char *print_ipad(unsigned long ipaddr);
```

Parameters

```
unsigned long ipaddr /* IP address to print, in Big-Endian (net order) */
```

File

`misclib/in_utils.c`

Description

Accepts a 32 bit IP address in big-endian format and returns a pointer to a volatile buffer with a printable version of the address. The buffer will be overwritten by each subsequent call to `print_ipad` , so the caller should copy it or use it immediately.

Note that the current implementation of `print_ipad()` is not re-entrant, and should not be used on a port to a pre-emptive RTOS.

Returns

Returns a pointer to the buffer with the printable IP address text.

7.4 ICMP

This section discusses IPv4 ICMP. The IPv6 equivalents are found elsewhere in this manual.

icmpEcho (Copy)

API Name

`icmpEcho()`

Syntax

```
int icmpEcho(ip_addr host, unsigned length, unshort pingseq);
```

Parameters

host	host to ping - 32 bit, local-endian
length	total desired length of packet on media
pingseq	ping sequence number

File

`net/ping.c`

Description

Send an ICMP echo request (the guts of "ping"). Callable from Applications. Sends a single "ping" (ICMP echo request) to the specified `host`. The application must provide an appropriate `pingDemux()` routine if ping replies are to be checked.

Returns

Returns `0` if ping sent OK, else negative error code.

7.5 UDP

These calls to the UDP layer are provided for systems which do not implement Sockets. They are much more lightweight, but do not offer the portability of Sockets.

udp_alloc

API Name

`udp_alloc()`

Syntax

```
PACKET udp_alloc(int datalen, int optlen, bool_t contig);
```

Parameters

datalen	length of UDP data (not including udp header)
optlen	length of IP options if any. Usually 0.
contig	Must use a single packet to hold of the headers and data

File

`ip/udp.c`

Description

This returns a `PACKET` big enough for the UDP data. It works by adding the space needed for UDP, IP, and MAC headers to the `datalen` passed and calling `pk_alloc()`. It also ensures that the FREEQ_RESID resource is locked around the call to `pk_alloc()`.

If the contig parameter is set, the call will fail if no free buffer queue contains a buffer large enough to hold the headers and all of the data. If contig is zero, the request can be satisfied by chaining packets together.

Returns

Returns a `PACKET` (pointer to struct `netbuf`) if OK, else `NULL` if a big enough packet was not available.

udp6_alloc

API Name

`udp6_alloc()`

Syntax

```
PACKET udp6_alloc(int datalen, int optlen, bool_t contig);
```

Parameters

datalen	length of UDP data (not including header)
optlen	length of IP options if any. Usually 0.
contig	Must use a single packet to hold of the headers and data

Description

This returns a `PACKET` big enough for the UDP data. It works by adding the space needed for UDP, IP, and MAC headers to the `datalen` passed and calling `pk_alloc()`. It also ensures that the `FREEQ_RESID` resource is locked around the call to `pk_alloc()`.

If the `contig` parameter is set, the call will fail if no free buffer queue contains a buffer large enough to hold the headers and all of the data. If `contig` is zero, the request can be satisfied by chaining packets together.

Returns

Returns a `PACKET` (pointer to struct `netbuf`) if OK, else `NULL` if a big enough packet was not available.

udp_free

API Name

`udp_free()`

Syntax

```
void udp_free(PACKET p);
```

Parameters

p	ptr to netbuf structure previously allocated by udp_alloc()

File

`ip/udp.c`

Description

`udp_free()` is used to return a previously allocated `PACKET` to the InterNiche stack's free pool. It works by calling `pk_free()`, but like `udp_alloc()` it ensures that the `FREEQ_RESID` resource is locked around the access to the free packet pool.

Returns

Void.

udp_open

API Name

`udp_open()`

Syntax

```
UDPCONN udp_open(ip_addr fhost,
                 unshort fsock,
                 unshort lsock,
                 int (*handler) (PACKET, void *, struct sockaddr *),
                 void * data);
```

Parameters

fhost	host to receive from, 0 if any is OK
fsock	foreign socket (port) number, 0 if any is OK
lsock	local socket (port) to receive on
handler	udp received callback function
data	returned on upcalls to aid de-muxing

File

`ip/udp_open.c`

Description

This routine creates a structure in the UDP layer to receive and upcall UDP packets which match the parameter passed. The foreign host and socket can use `0` as a wild card. This allows us to start "listens" for incoming SNMP Stations, TFTP applications, etc.

The handler routine is passed three parameters:

1. A pointer to the struct netbuf data structure for the received packet, with nb_prot and nb_tlen set to point to the starting address and total length of the application data.
2. A copy of the 'data' parameter that was passed into udp_open().
3. A pointer to the struct sockaddr_in data structure containing the IPv4 source address and UDP port number of the sender.

Returns

Pointer to UDP Connection structure, or `NULL` on failure.

udp6_open

API Name

`udp6_open()`

Syntax

```
UDPCONN udp6_open(ip6_addr f6host,
                  unshort fsock,
                  unshort lsock,
                  int (*handler) (PACKET, void *, struct sockaddr *),
                  void * data);
```

Parameters

f6host	host to receive from, 'ip6unspecified' if any is OK
fsock	foreign socket (port) number, 0 if any is OK
lsock	local socket (port) to receive on
handler	callback function to be invoked upon receipt of application data
data	returned on upcalls to aid de-muxing

Description

This routine creates a structure in the UDP layer to receive and upcall UDP packets which match the parameter passed. The foreign host and socket can use `0` as a wild card. This allows us to start "listens" for incoming SNMP Stations, TFTP applications, etc.

The handler routine is passed three parameters:

1. A pointer to the struct netbuf data structure for the received packet, with nb_prot and nb_tlen set to point to the starting address and total length of the application data.
2. A copy of the 'data' parameter that was passed into udp6_open().
3. A pointer to the `struct sockaddr_in6` data structure containing the IPv6 source address and UDP port number of the sender.

Returns

Pointer to UDP Connection structure, or `NULL` on failure.

udp_send

API Name

`udp_send()`

Syntax

```
int udp_send(unshort fport, unshort lport, PACKET p);
```

Parameters

fport	target UDP port
lport	local UDP port
p	packet to send, nb_prot ... nb_plen set to data, fhost set

File

`ip/udp.c`

Description

Send a UDP datagram to the foreign host in `p->fhost`. The '`local`' and '`remote`' ports in the UDP header are set from the values passed. Note: If `udp_send()` is called without having first called `udp_open()` on the associated port then any responses will be dropped.

Returns

`0` is OK, or a negative `ENP_` error code.

udp6_send

API Name

`udp6_send()`

Syntax

```
int udp6_send(ip6_addr * faddr. int scopeID, unshort fport, unshort lport,
PACKET p);
```

Parameters

faddr	Destination IPv6 address
scopeID	The scopeID for the destination address. This is only used on MULTI_HOMED systems when the destionation IP address is a link local address. This is a 1's based index of the egress interface.
fport	target UDP port
lport	local UDP port
p	packet to send, nb_prot ... nb_plen set to data, fhost set

Description

Send a UDP datagram to the foreign host in `p->fhost`. The '`local`' and '`remote`' ports in the UDP header are set from the values passed. Note: If `udp6_send()` is called without having first called `udp6_open()` on the associated port then any responses will be dropped.

Returns

`0` is OK, or a negative `ENP_` error code.

udp_close

API Name

`udp_close()`

Syntax

```
void udp_close(UDPCONN con);
```

Parameters

```
UDPCONN con /* an open UDP connection */
```

File

`net/udp_open.c`

Description

`udp_close()` closes a udp connection, by removing the connection from UDP's list of connections and deallocating its internal structures.

Returns

Nothing.

7.6 misclib

7.7 DNS Client

NicheStack provides the APIs listed below for obtaining one or more IP addresses for a specified domain name or for obtaining the domain name for a specified IP address. These APIs fall into three general categories (Early, Current and Proprietary) and are available when `DNS_CLIENT` is defined.

The so called, *Early standards* are now deprecated. Internally they arethread safe, but `gethostbyname()` and `gethostbyname2()` return a parameter that is not thread safe. These functions are provided only for existing applications that cannot be easily changed:

gethostbyname()	Early standard API.
gethostbyname2()	Standard deprecated by RFC 2553.
nslookupr()	InterNiche API for returning domain name for specified IPv4 address

The *Current Standard APIs* are implemented as specified by RFC 3493

getaddrinfo()	Replacement for `gethostbyname` APIs. Flexible but complex.
freeaddrinfo()	Frees memory for structures returned by `getaddrinfo()`.
getnameinfo()	Reverse lookup function.

In addition to the RFC specified API, InterNiche also supports a proprietary "mid-level" function which is simpler to use and often sufficient for use in embedded applications.

in_46reshost()	Returns IP address for a host name or a host name for an IP address

DNS Client API

gethostbyname

API Name

`gethostbyname()`

Syntax

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

Parameters

name	host name

Description

Get host information for named host. Implements a "standard" Unix version of gethostbyname(). Returns a pointer to a hostent structure if successful, NULL if not successful. The returned structure should NOT be freed by the caller.

Note

The returned hostent structure is not thread safe. It could be freed by internal DNS client routines if the entry ages out or if the table becomes full and space is needed for another entry.

Returns

Returns a pointer to host entry structure or NULL.

gethostbyname2

API Name

`gethostbyname2()`

Syntax

```
struct hostent *gethostbyname2(char *name, int af);
```

Parameters

name	host name
af	either `AF_INET` or `AF_INET6`

Description

Get host information for named host. Host information can be either in IPv4 or IPv6 format.

Note

Note: This API was deprecated by RFC 2553. The returned struct hostent has the same thread-safe problems described for gethostbyname().

Returns

Pointer to host entry structure or NULL

nslookupr

API Name

`nslookupr()`

Syntax

```
int nslookupr(char *name, char type, struct dns_querys **dns_entry);
```

Parameters

char *name	name to lookup
char type	lookup type (must be DNS_TYPE_PTR)
dns_querys **dns_entry	ptr to DNS query structure

Description

Performs a reverse lookup

Returns

Pointer to host entry structure or NULL

getaddrinfo

API Name

`getaddrinfo()`

Syntax

```
int getaddrinfo(CONST char *nodename, CONST char *servname,
                CONST struct addrinfo *hints, struct addrinfo **res);
```

Parameters

nodename	Domain name or an IP address
servname	Service name or port number
hints	Structure defined in RFC 3943
res	Ptr to array of 1 or more addrinfo structures.

Description

Translates a host name and/or service name and returns a set of socket addresses and associated info. to be used to create a socket to address the specified service with. This API is defined by RFC 3493 and intended to replace `gethostbyname()` and `gethostbyname2()`. It is thread safe. It is complex but provides many capabilities. It is available when DNSC_GETADDRINFO is defined.

The "hints" parameter is an `addrinfo` structure as defined in RFC 3943. On entry it contains a flags field, "`ai_flags`". The value in `ai_flags` is a hexidecimal `OR` of the desired "AI_" flags (`dns.h`). The flags direct the operation of the command and may limit the returned information.

The port number returned for a specified service name is based on `servtoportlist[ ]` in `dnsclnt.c`. The default array is limited in size. Add additional entries as needed for an implementation.

The function returns a pointer to an array of `addrinfo` structures with one structure for each address returned. On return, the calling application should use the info. in the structures as needed then call `freeaddrinfo()` to free the array.

Note

The `AI_V4MAPPED` flag is not currently supported, and the command does not currently support IP_V6 scope IDs other then 1

freeaddrinfo() must be called to free this array

Returns: 0 or one of the `EAI` error code defined in RFC_3493 and `dns.h`

freeaddrinfo

API Name

`freeaddrinfo()`

Syntax

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

Parameters

ai	Ptr to array of addrinfo structures returned by getaddrinfo()

Description

Frees the array of `addrinfo` structures returned by `getaddrinfo()`. It also frees the buffers within the structures that were used to hold names and addresses.

Returns

Nothing.

getnameinfo

API Name

`getnameinfo()`

Syntax

```
int getnameinfo(CONST struct sockaddr *sa, int salen, char *node,
                int nodelen, char *service, int servicelen, int flags)
```

Parameters

sa	Socket address. Either IPV4 or IP_V6
salen	Length of socket address
node	Buffer to contain the returned node name
nodelen	Length of buffer
service	IN: port number. OUT: Service name
servicelen	Length of service buffer
flags	Hexidecimal OR of desired NI_ flags (dns.h)

Description

Translates a socket address to a node name and/or a port number to a service name. The API behaves as defined in RFC 3493. The "`flags`" parameter can be used to change the default actons of the API. This API is available when `DNSC_GETADDRINFO` is defined.

Note

Note these NI_flags are NOT the same as the AI_flags for getaddrinfo().

Returns

0 or one of the `EAI` error code defined in RFC_3493 and `dns.h`

in46_reshost

API Name

`in46_reshost()`

Syntax

```
int in46_reshost(char *host, int type, struct dns_querys *ret_DNS_Entry,
int flags);
```

Parameters

host	Host name string or IP address string
type	DNS record type (see dns.h)
ret_DNS_Entry	In: allocated buffer. Out: copy of a DNS entry
flags	RH_VERBOSE, RH_BLOCK

Description

Mid-level thread-safe API used to resolve a host name to an IP address or to obtain a host name for an IP address. The DNS cache will be searched first for the requested information. If the information is not available there, it will make calls to the DNS servers. If `RH_BLOCK` is set the call will not return until the address is resolved or a timeout occurs. When called, `ret_DNS_Entry` must contain a buffer large enough to hold a `dns_querys` structure. When the function returns 0 (success), the "`ret_DNS_entry` " will contain a copy of a DNS entry. This buffer must be freed by the application.

Returns

0 if address was set, else one of the `ENP_` error codes

dns_update

API Name

```
dns_update()
```

Syntax

```
dns_update(char *soa_mname, char *hname, struct sockaddr *ipaddr,
           int r_type, unsigned long ttl, void *pio)
```

Parameters

soa_mname	domain name
hname	host name
ipaddress	IPv4 or IPv6 address using the appropriate struct sockaddr. IPv6 link local addresses on MULTI_HOMED systems should include a scopeID.
int r_type	type of "A" record: 4 for IPv4 "A" or 6 for IPv6 "AAAA" record
long ttl	Time to live value
pio	GIO handle for output (or NULL)

Description

Sends a DNS UPDATE packet to the authoritative server with the specified domain name. First sends `DNS_TYPE_SOA` to get IP address of authoratative server. It then sends the `DNS_UPDT` packet to the authoritative server.

Returns

- 0 if successful
- Negative `ENP` error if internal error occurs (eg timeout)
- One of the `DNSRC_` errors from network (all positive).

inet_ntop (Copy)

Name

`inet_ntop()`

Syntax

```
const char *inet_ntop(int af, const void *addr, char *str, size_t size);
```

Parameters

af	Address family (AF_INET or AF_INET6)
addr	Pointer to storage for IPv4 address ('ip_addr') or IPv6 address ('struct in6_addr') in network byte order
str	Pointer to storage for string that will contain IPv4 address in dotted decimal notation, or an IPv6 address in colon-separated notation
size	Length of output buffer ('str')

Description

This functions converts a binary representation of an IPv4 address or IPv6 address (in network byte order) into a string in dotted decimal notation. The output buffer must be at least 16 (or 40) bytes long for an IPv4 (or IPv6) address.

Returns

This function returns NULL if it encountered an error; otherwise, it returns the third argument ('str').

Copyright HCC Embedded 2017

inet_pton (Copy)

API Name

`inet_pton()`

Syntax

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

Parameters

af	Address family (AF_INET or AF_INET6)
src	Pointer to string containing IPv4 address in dotted decimal notation, or an IPv6 address in colon-separated notation
dst	Pointer to storage for IPv4 address ('ip_addr') or IPv6 address ('struct in6_addr') where the results of the conversion will be stored (in network byte order)

Description

This functions converts a string containing an IPv4 or IPv6 address in printable format into its equivalent binary representation (in network byte order).

Returns

This function returns 0 if the conversion was successful. A non-zero return value indicates a failure.

7.8 Syslog Client

InterNiche TCPIP stack ships with a syslog client. The syslog client can be used to send log messages (over UDP) to a syslog server. The API is based on the BSD syslog specification (man-pages). Hence part of the following documentation has been derived from BSD man pages. It inter-operates with all commercial syslog servers. InterNiche syslog client also has a special mechanism where-in different applications can send logs to different syslog servers.

Integration Notes

The syslog client is already integrated with the NicheStack. Hence it can be used out-of-the-box. It can be easily ported to other environments too. To do that, the following points should be addressed.

1. At system startup, call `syslog_init()` to initialize the syslog client. This is mainly needed to install the syslog sub-menu
2. At system shutdown, call `closelog()` to cleanup the syslog client

Interoperability Notes

1. The `openlog()`, `syslog()`, `closelog()`, `setlogmask()` functions work as per the BSD specs (man-pages).
2. In addition to the above, specific functions like `openlogaddr()` and `closelogfac()` are provided.
3. Support for other BSD syslog functions (like `vsyslog()`) is not provided.

Usability Notes

Applications can use syslog as follows.

1. Use syslog directly. Just call `syslog()` to send the log.
2. Use syslog when needed.
    - Call `openlog()` to open logging for a facility/application.
    - Call `syslog()` to send logs.
    - Call `closelog()` when done.
3. Use syslog with special features (for InterNiche syslog client). We have defined a special mechanism where-in different applications can send logs to different syslog servers. To use this feature, use the following sequence.
    - Call `openlog()` to open logging for a facility/application.
    - Call `openlogaddr()` to set the syslog server address.
    - Call `syslog()` to send logs to the specific syslog server.
    - Call `closelogfac()` when done. This will close the logging session for the particular application.

Additional information about the syslog client

- Call `setlogmask()` to mask the priorities of syslog messages.

- Calling closelog() closes all logging sessions, including the default. On a subsequent syslog() call, default session is recreated.
- syslog() does the following:
    - If LOG_CONS option was set, then log to console
    - If LOG_FILE option was set, then log to file.
    - Send message/log to syslog server
- If no facility/application is specified in syslog(), then the default facility LOG_USER is used.
- Here are some of the default values:
    - Default facility - LOG_USER
    - Default severity - LOG_NOTICE
    - Default options - LOG_FILE
- The test cases/functions provided at the end of the file syslog.c can be viewed for sample usage.

Syslog API

syslog, openlog, closelog, setlogmask

API Name

`syslog()`

`openlog()`

`closelog()`

`setlogmask()`

Syntax

```
void closelog (void);

void openlog (const char * ident, int logopt, int facility);

void syslog (int priority, const char * msg, ...);

int setlogmask (int maskpri);
```

Parameters

```
const char * ident; /* Identity of the application */

int logopt; /* Options for logging */

int facility; /* Application/facility doing the log */

int priority; /* Priority of the log */

int maskpri; /* Used to mask logs of lower priorities */
```

File

`misclib/syslog.c`

Description

The `syslog()` function writes message to the syslog server. The message is then written to the system console, log files, logged-in users, or forwarded to other machines as appropriate. The message is identical to a printf format string. ('%m' is supported by BSD, but not supported in this implementation). A trailing newline is added if none is present. The `vsyslog()` function of BSD is not supported. The message is tagged with priority. Priorities are encoded as a facility and a level. The facility describes the part of the system generating the message. The level is selected from the following ordered (high to low) list:

LOG_EMERG	A panic condition. This is normally broadcast to all users.

LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, e.g., hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but should possibly be handled specially.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

The `openlog()` function provides for more specialized processing of the messages sent by syslog(). The parameter ident is a string that will be prepended to every message. The logopt argument is a bit field specifying logging options, which is formed by OR'ing one or more of the following values:

LOG_CONS	If `syslog()` cannot pass the message to syslogd it will attempt to write the message to the console
LOG_NDELAY	Open the connection to `syslogd` immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.
LOG_PERROR	Write the message to standard error output as well to the system log.
LOG_PID	Log the process id with each message: useful for identifying instantiations of daemons.

The facility parameter encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_AUTH	The authorization system
LOG_AUTHPRIV	The same as LOG_AUTH, but logged to a file readable only by selected individuals.
LOG_CONSOLE	Messages written to console by the kernel console output driver.
LOG_CRON	The cron daemon
LOG_DAEMON	System daemons that are not provided for explicitly by other facilities.
LOG_FTP	The file transfer protocol daemons
LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_LPR	The line printer spooling system

LOG_MAIL	The mail system.
LOG_NEWS	The network news system.
LOG_SECURITY	Security subsystems
LOG_SYSLOG	Messages generated internally by `syslogd()`
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_UUCP	The uucp system.
LOG_LOCAL0	Reserved for local use. Similarly for `LOG_LOCAL1` through `LOG_LOCAL7`.

The `closelog()` function can be used to close the log file.

The `setlogmask()` function sets the log priority mask to maskpri and returns the previous mask. Calls to syslog() with a priority not set in maskpri are rejected. The mask for an individual priority pri is calculated by the macro LOG_MASK(pri); the mask for all priorities up to and including toppri is given by the macro LOG_UPTO(toppri);. The default allows all priorities to be logged.

Returns

The routines `closelog()`, `openlog()`, `syslog()` return no value.

`setlogmask()` always returns the previous log mask level.

openlogaddr, closelogfac

API Name

`openlogaddr()`

`closelogfac()`

Syntax

```
void openlogaddr(int facility,
                 char *iden,
                 int logopt,
                 struct sockaddr *sa,
                 int sa_len,
                 char *fname);
```

```
void closelogfac(int facility);
```

Parameters

facility	One of the facility codes defined in `h/syslog.h`
iden	Pointer to ID information
logopt	logging options (e.g., LOG_CONS, LOG_FILE)
sa	pointer to generic socket address structure
sa_len	length of socket address structure
fname	name of facility-specific logfile. Null if logopt = LOG_CONS

Description

The function `openlogadd()` is used to start logging to a particular syslog server. InterNiche syslog client allows separate logging for each facility. Hence different applications can use this feature to log to different syslog servers. The fname parameter gives the name of the file where messages are to be logged. When the application is done logging, it can call `closelogfac()` to close special logging for the particular facility.

Returns

Nothing

File

misclib/syslog.c

8 Semaphores

Semaphores are used as a method for waiting for an event to occur. For example, a semaphore is allocated when a socket is created. A task can then wait for a socket event to occur. When the event occurs, the semaphore is "signaled" and the task is run to handle the event. The semaphore control structure is defined as follows:

```
struct in_semaphore
{
 struct task *tk_owner;    /* Task that currently owns the semaphore (Not used by NicheTask) */
       struct task *tk_waitq;      /* Queue of tasks waiting for the semaphore */
       int16_t        tk_count;      /* Count of event occurances since semaphore was reset */
       int16_t        tk_maxcnt;  /* Limits the count of events that have occurred */
#ifdef DEBUG_TASK
       uint32_t      tk_tag;
#endif
};

typedef struct in_semaphore  IN_SEM
```

When the semaphore is signaled (the event has occurred) all tasks listed in `tk_waitq` will be marked as ready (runnable). NicheStack code uses binary semaphores. The code is only interested in whether or not an event occurred, not the number of times the event occurred. The `tk_maxcnt` will be set to 1, and therefore, the `tk_count` field can only be 0 (event has not occurred) or 1 (event has occurred one or more times).

It is possible to set `tk_maxcnt` to a larger number and have the semaphore keep track of the number of times the event has occurred. For example, it could be used to keep track of the number of messages received. While `tk_count` is > 0, the waiting task will be awakened and will handle one message. It would then yield so that other tasks could run, and then be reawakened, if `tk_count` is still greater than 0.

Within NicheStack code signals are not used to take ownership of a resource, so the the `tk_owner` field is not used.

8.1 tk_sem_pend

tk_sem_pend() - Wait for an event

API Name

```
tk_sem_pend() - Wait for an event
```

Syntax

```
void tk_sem_pend(TASK *task, IN_SEM *sem, int32_t timeout);
```

Parameters

task	Task waiting on the semaphore; a value of NULL means "the current task"
sem	Semaphore to wait for; a value of NULL means the task's semaphore
timeout	Number of CTICKs to wait before timing out

Description

A semaphore is used to signal the occurrence of an event, such as the arrival of a packet or a handshake from another task. If the event has already occurred, the calling task does not wait. If the event has not occurred, the calling task is suspended until the event occurs or the specified timeout expires.

The timeout parameters specifies the number of CTICKs to wait before giving up. A negative value or a value of zero means do wait at all. A value of `INFINITE_DELAY` will cause the task to never timeout.

Returns

`ESUCCESS`	if the event arrived before the timeout value
`TK_TIMEOUT`	the task timed out waiting for the event
`EFAILURE`	an error occurred in the call

8.2 tk_sem_post

tk_sem_post() - Signal an event

API Name

```
tk_sem_post() - Signal an event
```

Syntax

```
void tk_sem_post(TASK *task, IN_SEM *sem);
```

Parameters

task	Task waiting on the semaphore; a value of `NULL` means "the current task"
sem	Semaphore to wait for; a value of `NULL` means the task's semaphore

Description

Signals a semaphore that an event has occurred, such as the arrival of a packet or a handshake from another task. Any tasks that are waiting for the event will be marked ready to run. If the waiting task has a higher priority than the signalling task, the waiting task will immediately resume execution.

Returns

Nothing

9 Mutexes

Within the NicheStack, a mutex is used to claim ownership of a resource. No other task can use the resource until the current owner releases it.

```
struct in_mutex
{
   struct task *tk_owner;    /* task that owns the resource */
      struct task *tk_waitq;    /* Queue of tasks waiting for the resource */
      int16_t        tk_nesting; /* Count of times current task as locked this mutex */
#ifdef DEBUG_TASK
       uint32_t      tk_tag;
#endif
};


typedef struct in_mutex  IN_MUTEX;
```

The `tk_nesting` field is used when a task requests a mutex that it already owns. This is generally undesirable, but it is sometimes unavoidable. `tk_nesting` is set to 1 when a task claims ownership of the mutex. If the task requests the mutex again before it releases it, `tk_nesting` will be incremented. A task must be sure to release the mutex the same number of times that it successfully requested it.

9.1 tk_mutex_pend

tk_mutex_pend() - Request a mutex resource

API Name

```
tk_mutex_pend() - Request a mutex resource
```

Syntax

```
void tk_mutex_pend(IN_MUTEX *mutex, int32_t timeout);
```

Parameters

mutex	mutex resource to wait for
timeout	number of CTICKs to wait before timing out

Description

Ownership of a mutex allows multiple task to cooperatively access a shared resource, such as a queue of socket connections or a network table. A task requests ownership of a mutex by calling `tk_mutex_pend()`. If no other task owns the mutex, ownership is passed to the calling task and the function returns. If another task owns the mutex, the calling task is suspended until the current owner relinquishes the mutex.

The timeout parameters specifies the number of CTICKs to wait before giving up. A negative value or a value of zero means do wait at all. A value of `INFINITE_DELAY` will cause the calling task to never timeout.

To simplify coding, a task may request a mutex that it already owns. In this case, the mutex will continnue to be owned by the calling task, and the function will return immediately. Nested calls `to tk_mutex_pend()` must be matched with an equal number of nested calls to `tk_mutex_post()` before the mutex is available to other tasks.

Returns

ESUCCESS	if the calling task owns the mutex
TK_TIMEOUT	the task timed out waiting for the mutex
EFAILURE	an error occurred in the call

9.2 tk_mutex_post

tk_mutex_post() - Release a mutex resource

API Name

```
tk_mutex_post() - Release a mutex resource
```

Syntax

```
int tk_mutex_post(IN_MUTEX *mutex);
```

Parameters

mutex	mutex resource to wait for

Description

Relinquishes ownership of a mutex resource. If another task is waiting for the mutex, ownership of the mutex is passed to the waiting task. If the waiting task has a higher priority than the relinquishing task, the waiting task will immediately resume execution.

Returns

ESUCCESS or error code

10 Introduction to NicheStack IPv6

InterNiche IPv6 was designed to be added to existing InterNiche based systems with minimal impact on pre-existing applications and drivers. Applications using IPv4 sockets will run unchanged on the new "dual mode" (IPv4 and IPv6) stack, although these applications will need to be modified if they are to use IPv6 capabilities. Sockets modification for IPv6 are described in IPv6 Sockets.

To understand the design of the InterNiche IPv6 stack, it is useful to understand the ways in which IPv6 differs from IPv4. This discussion assumes the reader is familiar with IPv4 concepts, such as the IP address, subnets, broadcast and multicast; as well as related protocols such as ARP and ICMP.

10.1 Bigger IP address

The major factor driving the creating of a new IP standard is the lack of available addresses in IPv4. IPv4's 32 bit address theoretically yield about 4 billion unique addresses. While this may be enough to assign an address to every computer currently in existence, it will fall short when every person on earth owns several Internet-capable devices. The problem is actually more pressing than that. The entire IPv4 addresses space is already assigned, with some organizations holding large numbers of unused addresses. Most of the IPv4 address space is assigned to North American corporations, making the problem especially acute in Europe and Asia.

To solve this problem, IPv6 specified an IP header with 128 bit IP addresses, as compared to the 32 addresses used in IPv4. Like IPv4, the addresses are stored as the last two fields in the IPv6 header, source followed by destination.

In IPv4 stacks it was common practice to manipulate IP addresses by treating them as 32 bit unsigned integer values. This does not work with 128 bit IPv6 addresses. Throughout the InterNiche code, IPv6 addresses are compared and copied using the macros `IP6EQ()` and `IP6CPY()`, respectively.

At their simplest, these two macros may be left to the default definitions from `ip6.h` file. These definitions map the macros to `memcpy()` and `memcmp()` as follows:

```
#define IP6EQ(addr1, addr2)          (!MEMCMP(addr1, addr2, sizeof(ip6_addr)))
#define IP6CPY(addrptr1, addrptr2)  MEMCPY(addrptr1, addrptr2, sizeof(ip6_addr))
```

Both macros may be optimized by using inline assembly, and by taking advantage of the fact that the blocks to be copied/compared are always 16 bytes long. `IP6EQ()` may be further optimized by comparing the data at the end of the blocks first, since that is the data most likely to differ.

CPU architectures with alignment sensitivity issues (such as ARM) may assume that the IPv6 addresses are always properly aligned in memory and dispense with alignment tests.

10.2 Header Layers

IPv6 does not have the concept of IP header options, as IPv4 does. IPv6 headers also have fewer fields than the IPv4 headers. IPv6 headers have omitted the checksum, fragment and header length fields.

A new class of headers called IP "extension" headers provides a large portion of the functionality that was performed in IPv4 by the deprecated fields and option headers. IPv6 extension headers differ from IPv4 options primarily in that extension headers are technically not part of the IPv6 header (thus no IPv6 header length field is needed). Each extension header carries a 8-bit protocol type for the header which follows - e. g. `6` for TCP, `17` for UDP, etc. Each extension header has it's own assigned 8 bit type.

The new headers exist in the IPv6 packets between the IPv6 headers and the transport layer headers. Since any number of extension headers may be inserted be software layers between the transport layer and the MAC layer, the InterNiche code for IPv6 does not depend on preallocating a limited space for the IP and MAC headers at the front of data packets. This sort of preallocation worked on IPv4 since the size of the IP header was well known, and the maximum size of a MAC header could be determined at system startup time.

10.3 Hardware Limits

Datagram Size

IPv6 supports larger datagram sizes than IPv4. Assuming large IP packets will be fragmented, "normal" IPv6 packets are limited to 64K by the 16 bit offset field in the IP fragmentation header.

PMTU

IPv6 has different requirements from it's underlying network hardware than IPv4 does. Most importantly, the PMTU (Path Maximum Transmission Unit) on each network must be at least 1280 bytes. This is more than twice the size of IPv4's 580 byte requirement.

Routers Do Not Fragment

The benefits of the larger PMTU are offset by the requirement that all IP fragmentation MUST take place in the sending host, not on the IP routers. This means the sender must determine the PMTU for his connection to any peer prior to setting packet size parameters, such as TCP MSS options, or simply default to the 1280 PMTU mentioned in the previous section.

This makes routing simpler, but does so at the expense of forcing the end nodes to choose between PMTU discovery (which adds connection setup latency and overall complexity) or the default PMTU (which hurts performance).

10.4 Multicast Addressing

IPv6 does not use broadcast packets. Instead, a series of multicast addresses are defined for common target groups, such as the "all nodes" multicast address, or the "all routers" address. The "all nodes" multicast address is functionally equivalent to a broadcast for IPv6 hosts.

This means that the IP multicast option is required for IPv6.

Another side effect of this design is that MAC layers are also required to support multicast. An Ethernet device cannot support IPv6 by receiving only Unicast and Broadcast, since it will not receive packets multicast by adjacent nodes. For more details on dealing with the multicast requirement at the MAC layer, see Multicast is required.

10.5 Pseudo Checksum

The various "pseudo checksum" descriptions used by TCP, ICMP, and UDP over IPv4 are all replaced in IPv6 by a single standard pseudo checksum algorithm. Since the IPv6 layer lacks an IP layer header checksum, all protocols running IPv6 are required to use this checksum method. The upper layer checksum verifies the most important fields in the IP header - length, addresses, and protocol type.

The InterNiche code provides `ip6_pseudosum()`, a generic implementation of this checksum algorithm that should be suitable for all transport level protocols.

The IPv6 method of constructing the pseudo header is similar to IPv4's TCP/UDP pseudo header, however the longer IPv6 IP addresses are used. The underlying 16 bit ones complement summing algorithm is unchanged.

10.6 Interface Addresses

IPv6 usually has several IPv6 addresses per network Interface, as opposed to the single address per interface commonly used by IPv4. Addresses supported by the InterNiche code for each interface are given in the table below. The net structure has a field added for each of these.

Type	Scope	Description
Link local address	Local to MAC segment	Derived from the MAC address of the interface or assigned by some other means, such as PPP.
Global Local	Global	IPv6 equivalent of an IPv4 public IP address.
Unique Local	Global	A private address with global scope.
MAC-specific multicast	Local to MAC segment	A Solicited-Node multicast address derived from the interfaces MAC address

The leading bits of the IPv6 address indicate what type of address it is. Of interest are Link-local addresses, which begin with "`1111111010`" (usually `0xFE80`), and multicast addresses, which begin with "`11111111`" (`0xFF`).

For a complete list of all IPv6 addresses required per interface, see section 2.8 of RFC2373. There are several multicast addresses from this section which are not listed in the table above, however they are all contained in the interface's multicast list, and thus do not need additional fields added to the net structure.

10.7 Neighbor Discovery

IPv6 replaces ARP with the "Neighbor Discovery" (ND) protocol for resolving MAC addresses of Link-local IP addresses. ND "solicits" are roughly equivalent to ARP requests, and ND advertisements to ARP replies. ND may also be used to generically solicit local routers.

ND is actually a collection of ICMPv6 packet types, and thus has an IPv6 Ethernet type field; unlike ARP which had it's own Ethernet type field.

11 IPv6 Addresses

The meaning of the various bits in IPv6 addresses is considerably more complex than v4 - so much so that there is an entire RFC (RFC-2373) dedicated to it. The two major issues presented by IPv6 addresses are that they are large (usually too large to fit in a single CPU register) and there are many of them. The approach of assigning a single "`unsigned long`"-sized IP address field to each interface usually works quite well with IPv4, however IPv6 requires something more complex.

This section of the manual is merely an overview of these concepts and some insight on how they are implemented in the InterNiche code.

11.1 IPv6 Interfaces

When an InterNiche stack is built with the `IP_V6` compile-time `#ifdef` set in `ipport.h`, several fields are added to each `net` structure. Among these is an array of pointers to structures that contain IPv6 addressing information for that interface.

```
struct ip6_inaddr
{
   struct ip6_inaddr * next;     /* for application use */
   ip6_addr            addr;     /* address value (maybe unassigned) */
   int                 prefix;   /* number of bits in prefix */
   struct net *        ifp;      /* iface associated with it (maybe NULL) */
   int                 flags;    /* mask of the IA_ bits below */
   u_long              tmo1;     /* expiration ctick */
   u_long              tmo2;     /* expiration ctick of address */
   u_long              lasttm;   /* last action for dup checking, etc. */
   int                 dups;     /* Dup. check packets sent */
};
```

Programmers must use care when accessing these addresses, since the pointers in the `net` structure may be `NULL`. Further, the structure may exist but the pointer to the actual address data (`addr`) may be `NULL`.

When `#define MAC_LOOPBACK` is enabled a loopback interface is created. This interface is connected to a virtual link and is assigned the link local address of `FE80::01`. Because link local addresses are only required to be unique within the associated link this will not conflict with address assignments on other links. The address `FE80::1` is not a loopback address and may be assigned to other links.

11.2 Address Notation

The "dot notation" used to express IPv4 address in human-readable format would be rather tedious for IPv6, since IPv6 addresses have considerable length. A different notation has been set down by RFC-2373 to express IPv6 addresses. IPv6 notation conforms to the following rules:

- Values are expressed as hexadecimal, rather than the base-ten notation used for IPv4.
- 16 bits of address data exist between each delimiter, rather than the v4's 8 bits.
- Colons (":") are used as delimiter, rather than IPv4s dots(".").
- The longest sequence of sixteen `0` values may be shown as a single double colon - ("::")

Here is a typical example of a printed IPv6 address in this format:

```
FE80::0248:54FF:FE86:D329
```

where the `FE` value is located in byte 'zero'.

Ten bytes of non-zero data are displayed in five 16-bit values. Bytes three through eight of the address are assumed to be all zeros, as indicated by the double-colon after "`FE80`".

RFC-2553 "Basic Socket Interface Extensions for IPv6" specifies a C language API to convert these strings between 128-bit binary strings and ASCII. The InterNiche code provides both routines (`inet_pton()` and `inet_ntop()`) in the source file `parseip.c`.

11.3 Address passing and storage

In IPv4 implementations, IP addresses are almost always passed by value. Since most CPUs could store 32 bits on a CPU stack or in registers, there was not reason not to do it this way. The larger IPv6 addresses use more time and space when passed by value, so instead the InterNiche code makes every effort to pass IPv6 addresses by reference (pointers).

This helps performance, but makes the code somewhat harder to maintain, since the programmer must always be aware of the scope of the address passed by pointer. If the address is transient, for example a pointer into a `PACKET` buffer, then the pointer obviously should not be stored in a more permanent structure, such as a routing table entry. In cases like this, the address should be copied into the longer-lived structure via `IP6CPY()`. This means the structures also must be designed with some awareness of how they are to be used in the code - long lived structures which get info from short-lived storage should always provide for a local copy of the IP address.

The IP routing table entry mentioned above is a good example of a structure that gets a long-lived IP address from a potentially short-lived source. The "`nexthop`" field in the `PACKET` structure illustrates the inverse scenario. This IPv6 address is set by routing code when an IPv6 packet is about to be sent. It is set on every packet transmission, so setting it should be fast. The source is either the routing table or the ND cache (see Neighbor cache), both of which are very long lived when compared to the longevity of a `PACKET` . Additionally, the routing table and ND cache are protected from having entries deleted during the transmit process. Overall, it makes sense to have `nexthop` be a pointer to the IP address in the routing table or ND cache, rather than have an additional copy of the IP address.

11.4 Address classes

IPv6 supports a number of different types of IP address, with a wide variety of scopes and purposes. Excluding multicast and internal addresses, the two main types are:

Link local	Address valid only on the local network segment
Global Unicast	Globally unique, globally usable address
Unique Local	Private address, routable within an organization, globally unique.

These addresses are assigned and maintained on a per-interface basis. Additionally, the required predefined multicast addresses (see Predefined Addresses) are added to the interface's multicast list at startup time.

11.5 Link Local vs. Global addressing

The two types of IPv6 address that seem to have the most importance are the Link Local and the Global types. The global type is analogous to a public IPv4 address. Its assignment is controlled by the ICANN (via regional providers in each country), and once assigned these addresses can be used to reach any other global IPv6 host attached to the Internet.

The link local addresses are somewhat analogous to private IPv4 addresses (e.g. `10.0.0.1`). They can only be used to communicate between hosts on the same network segment. A router should never forward link local addresses.

The primary advantage of link-local addresses is that they can be automatically configured from the MAC addresses of Ethernet devices (and similar network adapters). The first 8 bytes are defined as:

```
FE80:0000:0000:0000
```

The remaining 8 bytes contain an encoding of the Ethernet MAC address. The ninth through eleventh bytes of the IPv6 address contain the first three bytes of the Ethernet MAC (the vendor ID). The next two bytes of the IPv6 address is predefined as "`0xFFFE`" and the last three bytes of the IPv6 address contain the last three bytes of the MAC address. An added complexity is that the second to last bit of the first byte is inverted. RFC-2373 explains the reasoning for MAC encoding.

11.6 Link Local Addresses and ScopeID

In a multi-homed system because any given link local address may exist on more than one link, the link local address is not sufficient to determine the interface on which to send the packet. The scopeID must be used to determine which interface should be used. The scopeID is the same as the 1's based index of the interface as shown by the "iface" command. The "iface" command without parameters will show the index value for all interfaces. An interface name cannot be used as the scopeID. If the scopeID is not given with an IPv6 address, then "`%1`" is assumed. On system with only one interface, the scopeID is not needed. The scopeID is not need with global addresses, and it will be ignored if it is present.

Note: The address "0xfe80::1" by itself is not a loopback address. An IPv6 address without a scopeID defaults to 1. However, interface 1 is not the loopback interface. Assuming the loopback interface is interface 2, then "0xfe80::1%2" would be a loopback address.

When writing applications that use `t_bind()`, `t_sendto()`, etc with IPv6 addresses the scopeID should be given as part of the `sockaddr_in6` structure.

The scopeID is only used internally to the host and never passed on the network; it is only meaningful at the host.

Global addresses are meant to be globally unique.

InterNiche's IPv6 may construct global addresses (Aggregatable Global Unicast Addresses (AGUA)), from the prefixes given in Router Advertisements. The prefix is concatenated with the last 8 bytes of the link-local address.

For example, the following global address uses the prefix from the 6bone IPv6 research network. Note that the 6bone is no longer in operation and so this is a safe example:

```
3FFE:501:FFFF::211:11FF:FEBE:85B9
```

The network administrator has the responsibility of loading the routers on a particular link to advertise specific prefixes. These globally unique prefixes have to be known by anyone forwarding traffic to a given link.

IPv6 routers will only forward unicast addresses if they are global. In other words, unicast traffic cannot leave a link unless it has a global address. For example, in order to even unicast ping across a router, a global address is required.

Correspondingly, link-local unicast addresses are never forwarded and cannot leave their link.

Thus the MAC address: `00 48 54 86 D3 29` produces the link-local IPv6 address: `FE80::0248:54FF:FE86:D329`.

The InterNiche IPv6 code configures each Ethernet interface with a link local address.

11.7 Unique Local Addresses

Unique Local Addresses, ULA, are private addresses that are global in scope; that is they are treated essentially like global addresses. Their assignment is not centrally managed but handled by the local organization. They can be routed but should not be routed to the public network. They are likely unique when assigned as specified by RFC 4193 but are not guaranteed to be unique; this is only one reason they should not be routed to the global internet.

11.8 Node Local

Intended for testing purposes, IPv6 specifications define a "node local address" (also called "interface local"). InterNiche IPv6 does not currently support this feature but instead relies on the IPv6 loopback address for testing.

11.9 Predefined Addresses

There is also a set of predefined IPv6 IP addresses which all systems are required to support. These addresses are summarized in this section, along with descriptions of how they are handled by InterNiche implementation.

Loopback - ::1

The IPv6 loopback address is all zeros except for a "1" in the final bit, a.k.a."::1". The InterNiche IPv6 code implements loopback by creating a dedicated loopback network interface. No attempt is made to allow for "looping back" packets without a loopback interface. Note: the address FE80::1 is not a loopback address.

All nodes multicast - FF02::1

The all-nodes multicast is a predefined multicast group that has as its members all IPv6 hosts. The address is FF02::1. This is essentially the same as the IPv4 broadcast address, 255.255.255.255. The InterNiche code supports this by adding the address to the multicast list via calls to in_addmulti(). One call is made for this address for every interface in the IPv6 system.

All routers multicast - FF02::2

The all-routers multicast is a predefined multicast group that has as its members all IPv6 routers. The address is FF02::2. On builds compiled with the IP_ROUTING flag set in ipport.h, the InterNiche code supports this by adding the address to the multicast list via calls to in_addmulti(). One call is made for this address for each interface in the IPv6 system.

Solicited node multicast

The solicited node multicast is an IPv6 address made for each unicast address on each interface by encoding part of the address into a predefined multicast prefix. The prefix value is FF02::01:FF00:0. The last three bytes of the address are copied from the last three bytes of the unicast address. The solicited node multicast address for MAC address 00 48 54 86 D3 29 is FF02::01:FF86:D329.

The InterNiche code supports solicited node addresses via two somewhat overlapping mechanisms. First, it builds the address for each MAC interface, and places the address in the interface's ip6_inaddr[ ] slot reserved for this purpose. Second, it adds the address to the multicast list via calls to in_addmulti(). One call is made for this address for every interface in the IPv6 system.

Unspecified address

Lastly, section 2.5.2 of RFC-2373 specifies an "unspecified address" which is designed to be an indicator that an IPv6 address field has not been set, or (in the case of some socket calls) is to be considered a wildcard address. The value of the unspecified address is all zeros (written as "::").

11.10 Neighbor cache

IPv6 on Ethernet LANs utilizes the "Neighbor Discovery" protocol to discover the MAC address of other nodes on the LAN and is the replacement for IPv4's ARP protocol. InterNiche Neighbor Discovery maintains a cache (the "ND cache") of link local addresses of known neighbors as described in RFC 2461. This cache is not intended to be statically configured, or to be manually modified.

The Neighbor cache is periodically purged of addresses that are in the `STALE` state. The lifetime of these entries can be set by the CLI `ip6cache` command.

11.11 More notes on IPv6 Addressing

IPv6 allows multiple unicast addresses per interface. It also allows different ways to configure the interface addresses including auto-configuration, manual or CLI configuration, and DHCPv6. The Iniche stack supports all three mechanisms. Because all three mechanisms can be used simultaneously there is the possibility that each will attempt to add or modify the same address. This is allowed with the exception that auto-configuration and DHCPv6 cannot override the lifetime of a manually set address. The lifetime of a manually set address is infinite; therefore, a manually set address can only be removed manually.

Address auto-configuration includes generating the link local address which is based on the MAC address (or equivalent). This address must exist, will never expire, and cannot be removed. Auto-configuration of global addresses can be initiated by receiving an appropriate router advertisement. The advertisement will include the lifetime of the address. Address lifetimes can be changed (extended or shortened) by subsequent advertisements. Auto-configuration can be "enabled" or "disabled" using the "`ip6cfg`" command. Because router advertisements happen periodically one could manually remove an ip address created automatically only to see it reappear a short time later unless auto-configuration is explicitly disabled.

Manual configuration with the "`setip`" command can add or remove addresses. These addresses get an infinite lifetime which means they will not expire and can only be removed using the "setip" command. Manual configuration can remove unicast addresses (except the link local address) regardless of how they were configured. Manual configuration takes precedence over auto-configuration and DHCPv6. That means that using "setip" to add an existing address will set the address lifetime to infinity; this lifetime will not be overridden by auto-configuration or DHCPv6. Manually removing an address leased by DHCP by using the "setip" command will not notify the DHCPv6 server to release the lease. To remove an address and notify the DHCPv6 server use the "`dhcpv6 lease -l release`" command (see below).

DHCPv6 (if enabled) can be used to acquire addresses and can be triggered either by an appropriate router advertisement or by a CLI command "`dhcpv6 lease -l new`". The command "`dhcpv6 lease -l new`" will look for a DHCPv6 server and request an address. DCHPv6 will then automatically attempt to renew the lease when appropriate. The command "dhcpv6 lease -d" will return all leased addresses and disable DHCPv6 so that router advertisements cannot trigger the acquisition of more addresses. To re-enable DHCPv6 use one of the DHCPv6 lease management commands such as "`dhcpv6 lease -l new`".

The number of IPv6 addresses allowed per interface by the InterNiche stack is controlled `MAX_V6_LOCALS` and `MAX_V6_GLOBALS`. Their values must be at least 1. The defaults are set in `net.h` and can be overridden in `ipport.h`. The defaults are:

```
#define  MAX_V6_LOCALS    1
#define  MAX_V6_GLOBALS   3
```

11.12 IPv6 Routing

InterNiche's static IPv6 router provides all the IPv6 router functionality in RFC2460 and RFC2461, with multiple IPv6 interfaces. It has only a very simple static router capability, however, and does not support RIPng. It is activated by defining IP6_ROUTING in ipport.h.

This note presumes that the reader is familiar with IPv6, the relevant RFC's, and the InterNiche documentation.

The router functionality is controlled via a command line interface (CLI). This allows the user to control router activity on specific interfaces, to load up the prefixes it will advertise and to manage the static router.

Here is a summary of the functionality:

1. (1 - n) IPv6 interfaces.
2. Full RFC2460 and RFC2461 router behavior.
3. Simple static routing. The router can use known prefixes for 1-hop routing or simple table lookup of {prefix, interface, prefix length} for multi-hop routing.
4. CLI to control the router, ping, and interface behavior, including "`%`" syntax for scope-IDs.

The interfaces

The maximum number of interfaces is set in the ipport.h file:

```
#define MAXNETS        8   /* max ifaces to support at one time */
#define STATIC_NETS    6   /* static nets to allow for... */
```

The low level initialization of the relevant tables is discussed in The nets[ ] Array and the netlist.

The interfaces can be referred to as "1,2,3,etc". These can be used as the IPv6 scope IDs, in a relatively standard manner, e.g.,

```
ping -a ff02::1%2
ping -a ff02::1%3
```

Typing the `net iface` command, will describe the interfacess being used:

```
if -i 1
if -i et1
```

All the available interfaces will be accessible after boot time, though there will be a short delay while Duplicate Address Detection (DAD) occurs, before the addresses are PREFERRED and available for use.

After receiving Router Advertisements (RA), NicheStack IPv6 may create a global address from the given prefix. The number of global addresses supported is set by the `MAX_V6_GLOBALS` compile-time macro.

Default routers will be accumulated on the link of each interface.

Controlling IPv6 router behavior

This section gives a conceptual overview of the router behavior, making reference to the CLI.

The router behaves according to RFC2460 and RFC2461, beginning "host-like" - doing duplicate address detection (DAD) and issuing Router Solicitations (RS). It will remain in host mode until the "rt6man" command is issued.

The command:

```
rt6man -f -r -d
```

will start sending RA's and set the forwarding as 'active'. It should be issued first. Until the "`rt6add`" command is given, the static routing table will be empty, but by using the prefix lists for each interface, one-hop routing can begin immediately.

The command:

```
rt6prfx -a 2005:501:ffff:1000::0%1/64 -p 1280 -i 10000 -l 10000
```

will cause the Router Advertisements (RAs) on the first interface to include the prefix `2005:501:ffff:1000` in the options, with length 64, on-link. The prefix will have invalid time `10000` and lifetime `10000`.

A similar command:

```
rt6prfx -a 2004:501:ffff:1000::0%2/64 -p 1280 -i 10000 -l 10000
```

will advertise the 2004 prefix on the second interface.

The command:

```
rt6add -a 2002:501:ffff:1000::0%1/64
```

places the prefix `2002:501:ffff:1000/64` in the static router table, to direct packets with matching destinations to use a default router on the first interface.

It is this command, which sets up the tables for allows multi-hop routing.

The router table entries have the simple format:

```
{prefix, IF, prefix length}
```

There is no consideration of metric, etc.

Almost all prefixes and addresses appear as complete addresses with scope ID and prefix length, e.g.,

```
2002:501:ffff:1000::0%1/64
```

An exception would be ping, where the prefix length is not necessary:

```
ping -a ff02::1%1
```

Where the scope ID has the syntax:

```
"%"<scope ID>
```

The <scope ID> may be the interface number, as discussed above.

ip6cfg

Display or configure IPv6 parameters in system

ip6cfg

```
Display or configure IPv6 parameters in system
```

Syntax

```
ip6cfg [{-i <interface id> {-a <global,local> | <global> | <local> } | {-m
<MTU>}} | -n <duration>]
```

Parameters

-a	Disable or enable auto-configuration of addresses from received Router Advertisement on a particular interface. The string "global" enables creation of Global Addresses. The string "local" enables the creation of Unique Local Addresses, ULA. Either global,local or both can be enabled. If the string is not present then that type of address will not be created. For example, 'ip6cfg -i 1 -a global' will enable the creation of Global Addresses and will disable the creation of ULAs. To disable both enter "ip6cfg -i 1 -a none". To enable both use "global,local" with no spaces between global and local.
-i	Specify interface index (ones-based)
-m	Specify IPv6 MTU for a particular interface in byes (must be >=1280)
-n	Specify ND cache entry lifetime in STALE state (seconds)

Description

This command is used to configure or display the value of IPv6 parameters.

Notes/Status

- When no arguments are specified, this command displays the values of IPv6 parameters.

ip6tbl

Display contents of IPv6 tables in system

ip6tbl

```
Display contents of IPv6 tables in system
```

Syntax

```
ip6tbl [-a | -n | -p | -r]
```

Parameters

`-a`	Display list of addresses associated with each interface
`-n`	Display contents of Neighbor Discovery cache
`-p`	Display list of prefixes associated with each interface
`-r`	Display contents of reassembly table

Description

This command displays the contents of the specified IPv6 tables.

Notes/Status

- When no arguments are specified, this command displays the contents of all IPv6 tables.

rt6add

Command Name

`rt6add - Create an IPv6 route table entry`

Syntax

```
rt6add -a <addr>%<scopeid>/<prefixlen> -b <addr>%<scopeid>
```

Parameters

`-a`	Argument of type `IPv6 address`, indicating the destination address.
`-b`	Argument of type `IPv6 address`, indicating the next hop address.

Description

This command rt6add will create an IPv6 route table entry.

<addr> is a global unicast IPv6 address.

<scopeid> is the scope ID (1, 2, etc) of the interface of the address.

<prefixlen> is the prefix length of the address.

Location

This command is provided by the `IPv6` module when `IP_V6` is defined.

rt6del

Command Name

`rt6del - Delete an IPv6 route table entry`

Syntax

```
rt6del -a <addr>%<scopeid>/<prefixlen>
```

Parameters

`-a`	Argument of type `IPv6 address`, indicating the address to be deleted.

Description

This command rt6add will delete an IPv6 route table entry.

<addr> is a global unicast IPv6 address.

<scopeid> is the scope ID (1, 2, etc) of the IF of the address.

<prefixlen> is the prefix length of the address.

Location

This command is provided by the `IPv6` module when `IP_V6` is defined.

rt6list

Command Name

`rt6list - List current IPv6 routes table`

Syntax

`rt6list`

Parameters

none

Description

This command rt6list will list current IPv6 routes table.

Location

This command is provided by the `IPv6` module when `IP_V6` is defined.

rt6man

Command Name

`rt6man - Manage IPv6 router`

Syntax

`rt6man` [-d] [-f] [-g] [-r] [-s] [-m number] [-n number] [-l number] [-t number] [-e number] [-c number] [-p (L | M | H)] [-v number]

Parameters

`-c`	Argument of type `UINT`, setting `the AdvCurHopLimit`
`-d`	Presence puts in all defaults, before any other args
`-e`	Argument of type `UINT`, setting `the AdvRetransTimer`
`-f`	Presence turns on forwarding
`-g`	Presence turns off forwarding
`-l`	Argument of type `UINT`, setting `the AdvLinkMTU`
`-m`	Argument of type `UINT`, setting `the MaxRtrAdvInterval`
`-n`	Argument of type `UINT`, setting `the MinRtrAdvInterval`
`-p`	Argument of type `STRING`, sets router preferences (RFC4191) `( L | M | H ) (low OR medium OR high)`
`-r`	Presence turns on sending of Router Advertisements
`-s`	Presence turns off sending of Router Advertisements
`-t`	Argument of type `UINT`, setting `the AdvReachableTime`
`-v`	Argument of type `UINT`, setting `the AdvDefaultLifetime`

Description

This command rt6man sets the basic behavior of the IPv6 static router. Each of the variables MaxRtrAdvInterval, MinRtrAdvInterval, AdvLinkMTU, AdvReachableTime, AdvRetransTimer, AdvCurHopLimit, AdvDefaultLifetime; are explained in detail in RFC4861.

Location

This command is provided by the `IPv6` module when `IP_V6` is defined.

rt6prfx

Command Name

`rt6prfx - Set router prefixes`

Syntax

`rt6prfx` -a <addr>%<scopeid>/<prefixlen> -p <pmtu> -i <valid_time> -l <life_time> [-o]

Parameters

`-a`	Argument of type `IPv6 address`. Adds a prefix to be advertised.
`-p`	Argument of type `UINT`. The Path MTU (PMTU) for this link to be sent in the router's advertisements.
`-i`	Argument of type `UINT`. Valid lifetime in seconds for the prefix
`-l`	Argument of type `UINT`. Preferred Lifetime in seconds for
`-o`	Presence of this sets the prefix as "off-link", on-link flag is reset. The default is "on-link"

Description

This command rt6prfx sets address, PMTU, and lifetimes of a prefix; these are explained in detail in RFC4861.

<addr> is a global unicast IPv6 address.

<scopeid> is the scope ID (1, 2, etc) of the IF of the address.

<prefixlen> is the prefix length of the address.

Notes/Status

- The value for the `-l` option must be less than or equal to the value for the `-r` option.
- Durations greater than INFINITE_DELAY (0x7FFFFFFF) will be set to INFINITE_DELAY.

Location

This command is provided by the `IPv6` module when `IP_V6` and `IP6_MENUS` are defined.

11.13 IPv6 over IPv4 Networks: 6TO4

NicheStack enables IPv6 applications to communicate with each other over an IPv4 network without explicitly setting up a tunnel. This feature is called “6TO4” and is defined in RFC 3056. Basically, the IPv6 packet will be encapsulated within an IPv4 packet; that is, the IPv6 header will be preceded by an IPv4 header. The IPv4 header will have its protocol field set to 41 (0x29), the assigned value for IPv6 packets that are tunneled inside of IPv4 frames. The IPv4 header contains the IPv4 source and destination addresses.

A 6TO4 address in the IPv6 headers contains a unique prefix. The first 16 bits are 0x2002 and the next 16 bits are the IPv4 address of the target. If the destination site supports the 6TO4 mechanism, then at least one of its DNS records should provide a 6TO4 address. You can then use this 6TO4 address as the destination address for the packet or connection. The alternative is to go through a gateway that acts as a "relay router" (described below).

The following are required to setup a NicheStack 6TO4 tunnel:

- It must be a dual stack with both IPv4 and IPv6 defined.
- IPV6_TUNNEL, INCLUDE_6TO4, and IP6_ROUTING must be defined.
- There must be an entry in the in_devices[] array (userdata.c) for a 6TO4 pseudo interface. The format of this entry should be exactly as follows:

```
{ prep_6to4, 0x00000000, 0x00000000, 0x00000000, NF_6TO4 },
```

- One interface in the device table must connect to an IPv4 network and its flag parameter must include the flag NF_FOR6TO4.
- When sending a message, the application must specify must specify the scopeID of the 6TO4 pseudo interface.
- If the IPv4 interface has a gateway that will act as a 6TO4 relay router, then the application will use the native IPv6 address of the target; otherwise, it must use the 6TO4 address of the target.

When the outbound message specifies the scopeID of the 6TO4 pseudo interface, NicheStack will automatically:

- Construct a 6TO4 source address with a prefix that contains the IPv4 address of the interface with the NF_FOR6TO4 flag.
- Encapsulate the IPv6 message within an IPv4 message.
- Send the message out the interface marked with the NF_FOR6TO4 flag

For inbound messages sent to the 6TO4 address of the local host, NicheStack will automatically strip off the IPv4 header before passing the message up the stack in the normal manner for an IPv6 message.

6TO4 Relay Router

A 6TO4 relay router offers native IPv6 connectivity to the external network. In this case, the application will use the normal IPv6 address of the target although it still must specify the scopeID of the pseudo-interface. NicheStack will format an IPv6 packet using its own 6TO4 address in the source field and the target's IPv6 address in the destination field. It will wrap the IPv6 packet in an IPv4 packet using the IPv4 address of the gateway.

As a relay router, the gateway understands 6TO4 addressing. When it sees that the IPv4 header has a type field of 41, it will strip off the IPv4 header and route the packet across the IPv6 network as if it were a native IPv6 packet. For incoming IPv6 packets containing NicheStack's 6TO4 address in the destination field, it will wrap the IPv6 packet within an IPv4 packet, using the NicheStack's IPv4 address as the destination. NicheStack will strip off the IPv4 packet and pass the packet up the stack as if it were a native IPv6 packet.

Notes:

- 6TO4 tunneling is only available with the dual version of NicheStack.
- The current version of NicheStack does not provide support for any other form of IPv6 tunneling.
- The current version allows only one 6TO4 tunnel.

11.14 DHCPv6

NOTE: DHCPv6 is an optional product and may not be present in your particular source code tree. If you are uncertain about whether it is, or should be contained in your product distribution, please contact InterNiche.

What the DHCPv6 Client Does

DHCP for IPv4 and DHCP for IPv6 provide some similar services, such as, address management; but they are distinctly different in implementation and the focus on a particular version of IP. RFC 3315 defines the Dynamic Host Configuration Protocol for IPv6 (DHCPv6).

IPv6 allows multiple unicast addresses per interface. Addresses can be set manually, can be created automatically, or can be received from a server.

The DHCPv6 client can request IPv6 addresses from a DHCPv6 server. All requests are initiated by the client. The DHCPv6 client depends upon already having a valid IPv6 link local address. This is commonly formed from the MAC address of the interface. The client can then request IPv6 global addresses from the server by sending requests (DHCPv6 Solicit message) to a permanently defined multicast address.

The client will also send "renew" or "rebind" messages to the server to extend the lifetime of the lease.

If the client is disconnected from the network and is then reconnected the "`lease -l confirm`" command should be used to check that the leases are still valid.

The client code implements address support. It does not implement support for other information such as DNS server addresses.

Porting Considerations

The DHCPv6 client requires a Client ID for the message exchange with the server. RFC 3315 defines the requirements for this identifier. It is intended to be globally unique and consistent over time and configuration changes. There are 3 options for the Client ID (also known as the client's DHCP Unique Identifier or DUID) and the porting engineer must insure the correct selection and implementation of the identifier for his or her product. An example for one type is provided in the `dhcpv6_port.c` file.

DHCPv6 also requires some randomness in the retransmission timers used in the protocol. The porting engineer may want to modify `dhcpv6_random()` and/or `dhcpv6_calc_rt()` found in the file `dhcpv6_utils.c` based on the capabilities and needs of their product. For example, finer granularity or more uniform distribution of values can be achieved in some environments compared to others.

DHCPv6 will request an address when an appropriate router advertisement (with the "M" bit set) is received. It will also request an address when the CLI command "`dhcpv6 lease -l new`" is invoked. An address can be programmatically requested by invoking the "`dhcpv6_solicit()`" function for a particular interface.

DHCPv6 requires that the client send a "confirm" message to the server when the client disconnects then reconnects to the network. This checks that the leased addresses are still valid for the link on which the client has been reconnected. The Iniche stack does not automatically recognize and act on the disconnect / reconnect events. The porting engineer can invoke the function "`dhcpv6_lease_confirm()`" to create and send the lease confirm message for a particular interface. The client implementation will then act on the server's response appropriately.

DHCPv6 Menu Commands

dhcpv6 lease

Command Name

`Lease - Request the DHCPv6 client to obtain/renew/confirm/release a lease`

Syntax

`lease [-i UINT ] [-d | -l STRING]`

Parameters

-i	Network interface number, 1 based. This is optional with interface 1 the default.
-d	Disable DHCPv6 client. Releases all DHCPv6 leased addresses back to the server removing them from the interface. Prevents DHCPv6 from requesting an address because of a router advertisement.
-l	Enable DHCPv6 client and perform a lease function.

Description

This command is used to enable or disable DHCPv6 client functions for the specified interface. This supported DHCPv6 functions are:

- New - request a new IPv6 address for the DHCPv6 server
- Renew - renew an IPv6 address lease
- Confirm - confirm all IPv6 leased address; for example, can be used after disconnecting and reconnecting to the network.
- Release - release all IPv6 leased addresses back to the DHCPv6 server

Location

This command is provided by the `DHCPv6` module when `DHCPv6_CLIENT` is defined.

dhcpv6 netstat

Command Name

`dhcpv6 netstat - display DHCPv6 client statistics`

Syntax

`dhcpv6 netstat`

Parameters

None

Description

This command displays statistics associated with the DHCPv6 client. It show the number of messages sent and received by type.

Location

This command is provided by the `DHCPv6` module when `DHCPv6_CLIENT` is defined.

12 Sockets API

12.1 Overview

This section is documentation for the InterNiche Sockets layer. Sockets is an API, primarily used today for TCP programming, which was developed during the early 1980s at U.C. Berkeley for UNIX. Dozens of books and tutorials are available for Sockets programming (one of the compelling arguments for their use), so this section is devoted to functional descriptions of the Sockets subset as supported by the InterNiche TCP/IP stack. It is not a tutorial.

Programmers new to Sockets may observe that there is a good deal of extra "baggage" here not required for a TCP API. There are historical reasons for this. When Sockets was developed, it was fashionable in academic computing to view all IO devices as a file-system-like, (or "streaming") device. This was, obviously, in the days before mice and GUIs. Sockets were initially developed to allow inter-process communication via stream devices - one process would write to a connection socket and another process would read what was written. Sockets was meant to be a one-size-fits-all solution for data IO. On many UNIX systems you can actually pass a socket to the `read()` and `write()` calls in place of a file descriptor. When the Berkeley researchers wanted to extend the endpoints of the socket outside the host system so that processes on two separate systems could talk, they implemented TCP (as well as other protocols) under the Sockets. This required extending Sockets to indicate what type of service was desired. The `AF_INET` (as opposed to `AF_UNIX`) parameter in the `t_socket()` call is a vestige of this; as is the multiplicity of overlapping routines such as `t_write()`, `t_send()`, and `t_sendto()` which all do similar things on different types of Sockets.

This use of TCP as a carrier for Sockets was TCP's first major popular application outside of the DARPA projects where it was developed. So in a very real sense, TCP owes its widespread popularity today to Berkeley UNIX and Sockets.

Over the years, many simpler, cleaner TCP APIs have been proposed, but, by the time TCP became popular on non-UNIX platforms, it was too late. As with the attempts in the 1970s to switch the United States to the metric system, people had become accustomed to the status quo and were unwilling to switch. Sockets had become entrenched.

So for better or worse, Sockets is the de-facto standard for TCP programming. For all its warts, it still compares favorably with many of the baroque API standards from big commercial software vendors.

The calls documented in this section are compatible with those on UNIX systems insofar as TCP use goes. Example networking code from other Sockets-based systems should work here, and most of what is in the books and tutorials apply as well. We've tried to update the man-pages herein to reflect any differences there are.

One general difference is that all the function names in the InterNiche package start with "`t_`", e.g. `socket()` is `t_socket()`. This is so that embedded systems which already use some of the socket names will not have a conflict at link time. By adding `#define`s in your `tcpport.h` file, the original Sockets function names can be used in source code.

Another is the UNIX `errno` mechanism has been replaced by an error holder attached to each socket structure and assigned a value whenever an error occurs. Thus when a socket call indicates failure, such as `t_socket()` returning `-1`, you can examine this member or call `t_errno(long s)` to find out what went wrong. Possible values for Sockets errors are listed below. These are a subset of the standard Berkeley errors.

12.2 Sockets Errors

```
/* BSD Sockets errors */
#define ENOBUFS         1
#define ETIMEDOUT       2
#define EISCONN         3
#define EOPNOTSUPP      4
#define ECONNABORTED    5
#define EWOULDBLOCK     6
#define ECONNREFUSED    7
#define ECONNRESET      8
#define ENOTCONN        9
#define EALREADY        10
#define EINVAL          11
#define EMSGSIZE        12
#define EPIPE           13
#define EDESTADDRREQ    14
#define ESHUTDOWN       15
#define ENOPROTOOPT     16
#define EHAVEOOB        17
#define ENOMEM          18
#define EADDRNOTAVAIL   19
#define EADDRINUSE      20
#define EAFNOSUPPORT    21
```

12.3 Quick List for Sockets Prototypes

```
extern long t_socket (int, int, int);
extern int t_bind (long, struct sockaddr *, int);
extern int t_listen (long, int);
extern long t_accept (long, struct sockaddr *, int *);
extern int t_connect (long, struct sockaddr *, int);
extern int t_getpeername (long, struct sockaddr *, int *);
extern int t_getsockname (long, struct sockaddr *, int *);
extern int t_setsockopt (long, int, int, void *, int);
extern int t_getsockopt (long, int, int, void *, int);
extern int t_recv (long, char *, int, int);
extern int t_send (long, char *, int, int);
extern int t_recvfrom (long s, char * buf, int len, int flags, struct sockaddr *, int *);
extern int t_sendto (long s, char * buf, int len, int flags, struct sockaddr *, int);
extern int t_shutdown (long, int);
extern int t_socketclose (long);
extern int t_errno (long s);
extern int t_select(fd_set * in, fd_set * out, fd_set * ev, long tmo_seconds);
```

12.4 Quick List for Socket Options

```
/* Generic/TCP socket options          */
#define SO_DEBUG           0x0001    /* turn on debugging info recording    */
#define SO_ACCEPTCONN      0x0002    /* socket has had listen()    */
#define SO_REUSEADDR       0x0004    /* allow local address reuse    */
#define SO_KEEPALIVE       0x0008    /* keep connections alive    */
#define SO_LINGER          0x0080    /* linger on close if data present    */
#define SO_OOBINLINE       0x0100    /* leave received OOB data in line    */
#define SO_TCPSACK         0x0200    /* Allow TCP SACK    */
#define SO_WINSCALE        0x0400    /* Set scaling window option    */
#define SO_TIMESTAMP       0x0800    /* Set TCP timestamp option    */
#define SO_BIGCWND         0x1000    /* Large initial Congenstion window    */
#define SO_HDRINCL         0x2000   /* user access to IP hdr for SOCK_RAW    */
#define SO_NOSLOWSTART     0x4000   /* suppress slowstart on this socket    */
#define SO_SNDBUF          0x1001    /* send buffer size    */
#define SO_RCVBUF          0x1002    /* receive buffer size    */
#define SO_SNDTIMEO        0x1005    /* send timeout    */
#define SO_RCVTIMEO        0x1006    /* receive timeout    */
#define SO_ERROR           0x1007    /* get error status and clear    */
#define SO_TYPE            0x1008    /* get socket type    */
#define SO_MAXMSG          0x1010    /* get/set TCP_MSS (max segment size)    */
#define SO_RXDATA          0x1011    /* get count of bytes in sb_rcv    */
#define SO_TXDATA          0x1012    /* get count of bytes in sb_snd    */
#define SO_MYADDR          0x1013    /* return my IP address    */
#define SO_NBIO            0x1014    /* set socket into NON-blocking mode    */
#define SO_BIO             0x1015    /* set socket into blocking mode    */
#define SO_NONBLOCK        0x1016    /* set/get blocking mode via param    */
#define SO_CALLBACK        0x1017    /* set/get zero_copy callback routine    */
/* Only the following options can be set on a running socket */
#define TCP_ACKDELAYTIME   0x2001    /* Set time for delayed acks    */
#define TCP_NOACKDELAY     0x2002    /* suppress delayed ACKs    */
#define TCP_MAXSEG         0x2003    /* set maximum segment size    */
#define TCP_NODELAY        0x2004    /* Disable Nagle Algorithm */


/* IP socket options           */
#define IP_OPTIONS          1   /* buf/ip_opts; set/get IP options    */
#define IP_HDRINCL          2   /* int; header is included with data    */
#define IP_TOS              3   /* int; IP type of service and preced.    */
#define IP_TTL_OPT          4   /* int; IP time to live    */

/* Multicast socket options          */
#define IP_MULTICAST_IF     9   /* u_char; set/get IP multicast i/f    */
#define IP_MULTICAST_TTL    10    /* u_char; set/get IP multicast ttl    */
#define IP_MULTICAST_LOOP   11    /* u_char; set/get IP multicast loopback    */
#define IP_ADD_MEMBERSHIP   12    /* ip_mreq; add an IP group membership    */
#define IP_DROP_MEMBERSHIP  13    /* ip_mreq; drop an IP group membership    */

#define  IP_SCOPEID                14  /* int; IPv6 IF scope  ID */
#define  IPV6_UNICAST_HOPS         15  /* int; set hopcount */
#define  IPV6_MULTICAST_IF         16  /* unisgned int; set IF for outgoing MC pkts */
#define  IPV6_MULTICAST_HOPS       17  /* int; set MC hopcount */
#define  IPV6_MULTICAST_LOOP       18  /* unisgned int; set to 1 to loop back */
#define  IPV6_JOIN_GROUP           19  /* ipv6_mreq; join MC group */
```

```
#define  IPV6_LEAVE_GROUP          20  /* ipv6_mreq; leave MC group */

#ifdef USE_IGMPV3
#define  IP_BLOCK_SOURCE           23  /* ip_mreq_source; block data from a src */
#define  IP_UNBLOCK_SOURCE         24  /* ip_mreq_source; undo block filter */
#define  IP_ADD_SOURCE_MEMBERSHIP  25  /* ip_mreq_source; add a single source */
#define  IP_DROP_SOURCE_MEMBERSHIP 26  /* ip_mreq_source; drop a single source */
#define  MCAST_JOIN_GROUP          70  /* group_req; */
#define  MCAST_BLOCK_SOURCE        71  /* group_source_req; */
#define  MCAST_UNBLOCK_SOURCE      72  /* group_source_req; */
#define  MCAST_LEAVE_GROUP         73  /* group_req; */
#define  MCAST_JOIN_SOURCE_GROUP   74  /* group_source_req; */
#define  MCAST_LEAVE_SOURCE_GROUP  75 / * group_source_req; */
#define  IP_MSFILTER               76 / * optname used by advanced apis to call setsockopt */
#endif /* USE_IGMPV3 */
```

12.5 Sockets API Calls Reference

t_socket

API Name

`t_socket()`

Syntax

```
long t_socket (int domain, int type, int protocol);
```

Parameters

domain	Communication domain (AF_INET or AF_INET6)
type	Socket type (SOCK_STREAM or SOCK_DGRAM)
protocol	0

Description

`t_socket()` creates an endpoint for communication and returns a descriptor. The `domain` parameter specifies a communications domain within which communication will take place; this selects the `protocol` family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file `socket.h`.

The socket has the indicated type, which specifies the semantics of communication. Currently defined types are:

```
SOCK_STREAM      /* TCP*/
SOCK_DGRAM       /* UDP */
SOCK_RAW         /* IP */
```

A `SOCK_STREAM` type provides sequenced, reliable, two-way connection based byte streams. A `SOCK_DGRAM` socket supports datagrams (connectionless, unreliable messages of a fixed, typically small, maximum length). A `SOCK_RAW` socket provides lower-layer protocol access.

Sockets of type `SOCK_STREAM` are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a `t_connect()` call. Once connected, data may be transferred using `t_send()` and `t_recv()` calls. When a session has been completed, a `t_socketclose()` may be performed. Out-of-band data may also be transmitted as described in the `t_send()` page and received as described in `t_recv()`.

The communications protocols used to implement a `SOCK_STREAM` ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully

transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with `-1` returns and with `ETIMEDOUT` as the specific code in the global variable `t_errno`. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (such as five minutes).

`SOCK_DGRAM` sockets allow sending of datagrams to correspondents named in `t_sendto()` calls. Datagrams are generally received with `t_recvfrom()`, which returns the next datagram with its return address.

`SOCK_RAW` sockets allow the application access to IP-layer protocols with a datagram-like interface; the application specifies the protocol of interest as the `protocol` argument to `t_socket()`.

The operation of sockets is controlled by socket level options. These options are defined in the file `socket.h`. `t_getsockopt()` and `t_setsockopt()` are used to get and set options, respectively.

The `SO_NOSLOWSTART` option suppresses the standard TCP "slow-start" feature. Normally when newly connected, the TCP socket which is passed a large block of data to send (for example an FTP data connection) will send about two full-sized data segments, and then wait for a response from the other side before sending more. If the speed of the response indicates the network can handle more traffic, the connection will send more segments in reply. The number of segments will keep increasing until they are limited by internal resources or the receiver's window size.

In situations where a machine on an ethernet is sending it's packets through a router to a slower media (i.e. DSL) this slow start behaviour prevents the socket from flooding the router. The `SO_NOSLOWSTART` option defeats this feature, allowing the first data burst on the net to be the maximum number of segments allowed.

`SO_FULLMSS` prevents the socket from sending any data until the socket has buffered enough data to send a full sized packet. The size is determined by the network hardware, usually about 1460 data bytes. This should be used judiciously, since it may prevent proper operation of typical network applications. The problem is that an application command will not be sent until enough commands are buffered to produce the full-sized packet. For applications like FTP and HTTP, the average command is much too small to trigger the send. This option will not be available unless the TCP layer has been compiled with the `#define SUPPORT_SO_FULLMSS` in `ipport.h`.

The `TCP_NODELAY` disables the Nagle algorithm, and prevents attempts to coalesce small packets less than the `TCP_MSS`, while awaiting acknowledgement for data already sent.

`TCP_ACKDELAYTIME` sets the delay time for TCP delayed ACKs. The number of milliseconds of delay is passed to `setsockopt()` as a parameter. The millisecond time specified will by rounded off to the nearest "`cticks`" value. This option will not be available unless the TCP layer has been compiled with the `#define DO_DELAY_ACKS` in `ipport.h`. Builds which are compiled with this define will do delayed acks on all sockets by default, with a value of 1 `ctick`.

The `TCP_NOACKDELAY` option defeats delayed acking on specific sockets and will not be available unless the TCP layer has been compiled with the `#define DO_DELAY_ACKS` in `ipport.h`. Builds which are compiled with this define will do delayed acks on all sockets by default, with a value of 1

`ctick`. Setting `TCP_NOACKDELAY` will cause sockets to ack immediately as decided by the TCP code, with no delay.

`TCP_MAXSEG` is used to set the TCP MSS (Maximum Segment Size) value of the socket. Normally this value default to the size of the largest datagram supported on the underlying media, minus room for TCP, IP, and media headers. On ethernet this number is 1460 octets. This option can be called anytime after the socket is created, and should be called before the socket is connected. Calling after connection will produce unpredictable results. The value passed should generally be smaller than the default value as larger values may result in IP fragmentation.

Returns

`t_socket()` returns a non-negative descriptor on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept, t_bind, t_connect, t_getsockname, t_getsockopt, t_listen, t_recv, t_select, t_send, t_shutdown

t_listen

API Name

`t_listen()`

Syntax

```
int t_listen(long s, int backlog);
```

Parameters

s	Socket identifier
backlog	Used to compute a limit on the maximum number of connections that can be pending in the completed (those for which the TCP three-way handshake has completed) and partially completed (those for which the TCP three-way handshake has started, but isn't complete) queues.

Description

To accept connections, a `socket` is first created with `t_socket()`, a backlog for incoming connections is specified with `t_listen()` and then the connections are accepted with `t_accept()`. The `t_listen()` call applies only to sockets of type `SOCK_STREAM`. The backlog parameter defines the maximum length for the queue of pending connections (not maximum open connections). If a connection request arrives with the queue full the client will receive an error with an indication of `ECONNREFUSED`.

Returns

Returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept(), t_connect()

t_connect

API Name

`t_connect()`

Syntax

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

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for remote end (peer). IPv6 link local addresses on MULTI_HOMED systems should include a scopeID.
namelen	Length of sockaddr_in structure (bytes)

Description

The parameter `s` is a socket. If it is of type `SOCK_DGRAM` or `SOCK_RAW`, then this call specifies the peer with which the socket is to be associated; the address to which datagrams are sent and the only address from which datagrams are received. If it is of type `SOCK_STREAM`, then this call attempts to make a connection to another socket. The other socket is specified by `name` which is an address in the communications space of the socket. Each communications space interprets the `name` parameter in its own way.

Generally, stream sockets may successfully call `t_connect()` only once, however in the Interniche Sockets implementation even for a streams socket, if `NB_CONNECT` is defined in `ipport.h` and the socket is a non-blocking socket, then a socket allows repeated calls to `t_connect()`. These calls will return `0` once the socket is connected, or a `SOCKET_ERROR` if it is in the process of connecting.

Datagram and raw sockets may use `t_connect()` multiple times to change their association. Datagram and raw sockets may also dissolve the association by connecting to an invalid address, such as a zero address.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept(), t_connect(), t_getsockname(), t_select(), t_socket()

t_socketclose

API Name

`t_socketclose()`

Syntax

```
int t_socketclose(long s);
```

Note: this is just `close()` on traditional Sockets systems.

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for remote end (peer)
namelen	Length of sockaddr_in structure (bytes)

Description

The `t_socketclose()` call causes all of a full-duplex connection on the socket associated with `s` to be shut down and the socket descriptor associated with `s` to be returned to the free socket descriptor pool. Once a socket is closed, no further socket calls should be made with it.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept(), t_socket()

t_select

Syntax

```
int t_select (fd_set * readfds, fd_set * writefds, fd_set * exceptfds, long
tv);

void FD_SET (long so, fd_set * set)

void FD_CLR (long so, fd_set * set)

void FD_ISSET (long so, fd_set * set)

void FD_ZERO (fd_set * set)
```

Parameters

s	Socket identifier
readfds	Set of descriptors that an application will wait to become ready for reading
writefds	Set of descriptors that an application will wait to become ready for writing
exceptfds	Set of descriptors that an application will wait for occurence of an exceptional condition on
tv	Timeout duration (ticks)

Description

`t_select()` examines the socket descriptor sets whose addresses are passed in `readfds`, `writefds`, and `exceptfds` to see if some of their descriptors are ready for reading, ready for writing or have an exception condition pending. On return, `t_select()` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned. Any of `readfds`, `writefds`, and `exceptfds` may be given as `NULL` pointers if no descriptors are of interest. Selecting true for reading on a socket descriptor upon which a `t_listen()` call has been performed indicates that a subsequent `t_accept()` call on that descriptor will not block.

In the standard Berkeley UNIX Sockets API, the descriptor sets are stored as bit fields in arrays of integers. This works in the UNIX environment because under UNIX socket descriptors are file system descriptors which are guaranteed to be small integers that can be used as indexes into the bit fields.

In the InterNiche stack, socket descriptor are pointers and thus a bit field representation of the descriptor sets is not feasible. Because of this, the InterNiche Sockets API differs from the Berkeley standard in that the descriptor sets are represented as instances of the following structure:

```
typedef struct fd_set {      /* the select socket array manager */
   unsigned fd_count;      /* how many are SET? */
   long fd_array[FD_SETSIZE];   /* an array of SOCKETs */
} fd_set;
```

Instead of a socket descriptor being represented in a descriptor set via an indexed bit, an InterNiche socket descriptor is represented in a descriptor set by its presence in the `fd_array` field of the associated `fd_set` structure. Despite this non-standard representation of the descriptor sets themselves, the following standard entry points are provided for manipulating such descriptor sets: `FD_ZERO(&fdset)` initializes a descriptor set `fdset` to the null set. `FD_SET(fd, &fdset)` includes a particular descriptor, `fd`, in `fdset`. `FD_CLR(fd, &fdset)` removes `fd` from `fdset`. `FD_ISSET (fd, &fdset)` is nonzero if `fd` is a member of `fdset`, zero otherwise. These entry points behave according to the standard Berkeley semantics.

The porting engineer should be aware that the value of `FD_SETSIZE` defines the maximum number of descriptors that can be represented in a single descriptor set. The default value of `FD_SETSIZE` of 12 is defined in `tcp/tcpport.h`. This value can be increased to accommodate a larger maximum number of descriptors at the cost of increased processor stack usage.

Another difference between Berkeley and InterNiche `t_select()` calls is the representation of the timeout. Under Berkeley, the timeout parameter is represented by a pointer to a structure. Under InterNiche Sockets, a timeout is specified by the `tv` parameter, which defines the maximum number of ticks that should elapse before the call to `t_select()` returns. A tv parameter equal to `0` implies that `t_select()` should return immediately (effectively a poll of the sockets in the descriptor sets). Note that there is no provision for no timeout, that is, there is no way to specify that `t_select()` block forever unless one of its descriptors becomes ready. The maximum value (longest time in ticks) that can be specified for the `tv` parameter can be calculated by dividing the largest value that can be represented in a variable of type `long` by the TPS constant (system ticks per second). On PC based systems where longs are typically 32 bits and TPS is 20, this works out to be over 3 years.

The final difference between the Berkeley and InterNiche versions of `t_select()` is the absence in the InterNiche version of the Berkeley `width` parameter. The `width` parameter is of use only when descriptor sets are represented as bit arrays and was thus deleted in the InterNiche implementation.

Returns

t_select() returns a non-negative value on success. A positive value indicates the number of ready descriptors in the descriptor sets. `0` indicates that the time limit specified by `tv` expired.

See Also: **t_accept(), t_connect(), t_listen(), t_recv(), t_send()**

Notes

Under rare circumstances, `t_select()` may indicate that a descriptor is ready for writing when in fact an attempt to write would block. This can happen if system resources necessary for a write are exhausted or otherwise unavailable. If an application deems it critical that writes to a file descriptor not block, it should set the descriptor for non-blocking I/O. See discussion of **t_setsockopt()**.

t_recv, t_recvfrom

API Name

`t_recv()`

`t_recvfrom()`

Syntax

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

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

Parameters

s	Socket identifier
buf	Start address of buffer where received data will be copied into
len	Length of data to be sent
flags	Flags for receiving process (e.g., MSG_PEEK)
from	Pointer to struct sockaddr_in structure containing addressing information for the remote end
fromlen	Pointer to storage for length of sockaddr_in structure (bytes)

Description

`s` is a socket created with `t_socket()`. `t_recv()` and `t_recvfrom()` are used to receive messages from another socket. `t_recv()` may be used only on a connected socket (see t_connect), while `t_recvfrom()` may be used to receive data on a socket whether it is in a connected state or not.

If `from` is not a `NULL` pointer, the source address of the message is filled in. `fromlen` is a value-result parameter, initialized to the size of the buffer associated with `from`, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see t_socket).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see t_setsockopt) in which case `-1` is returned with the external variable `t_errno` set to `EWOULDBLOCK`.

Note that `t_recv()` will return an `EPIPE` if an attempt is made to read from an unconnected socket.

The `t_select()` call may be used to determine when more data arrive.

The `flags` parameter is formed by OR-ing one or more of the following:

MSG_OOB	Read any "out-of-band" data present on the socket, rather than the regular "in-band" data.
MSG_PEEK	"Peek" at the data present on the socket; the data are returned, but not consumed, so that a subsequent receive operation will see the same data.

Returns

These calls return the number of bytes received, or `-1` if an error occurred. On failure, they set an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_connect(),t_getsockopt(),t_select(), t_send(), t_socket()

t_send, t_sendto

API Name

`t_send()`

`t_sendto()`

Syntax

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

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

Parameters

s	Socket identifier
buf	Start address of data to be sent
len	Length of data to be sent
flags	Flags for sending process (e.g., MSG_OOB)
to	Pointer to struct sockaddr_in structure containing addressing information for the remote end. IPv6 link local addresses on MULTI_HOMED systems should include a scopeID.
tolen	Length of sockaddr_in structure (bytes)

Description

`t_send()` and `t_sendto()` are used to transmit the message addressed by buf to another socket. `t_send()` may be used only when the socket is in a connected state, while `t_sendto()` may be used at any time, in which case the address of the target is given by the to parameter. The length of the message is given by `len`.

No indication of failure to deliver is implicit in a `t_send()`. Locally detected errors are indicated by a return value of `-1`.

If no messages space is available at the socket to hold the message to be transmitted, then `t_send()` normally blocks, unless the socket has been placed in non-blocking I/O mode. The `t_select()` call may be used to determine when it is possible to send more data.

The `flags` parameter may include one or more of the following:

```
#define MSG_OOB          0x1   /* process out-of-band data */
#define MSG_DONTROUTE    0x4   /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

Returns

The call returns the number of characters sent, or -1 if an error occurred. On failure, it sets an internal t_errno to one of the errors listed in Sockets Errors to indicate the error. The t_errno can be retrieved by a call to t_errno(s).

See Also

t_recv(), t_select(), t_getsockopt(), t_socket()

t_accept

Syntax

```
long t_accept(long s, struct sockaddr *addr, int *addrlen);
```

Parameters

s	Socket identifier
addr	Pointer to struct sockaddr_in structure containing addressing information for the remote end in newly accepted connection
addrlen	Pointer to storage for length of sockaddr_in structure (bytes)

Description

The argument `s` is a socket that has been created with `t_socket()`, bound to an address with `t_bind()` and is listening for connections after a `t_listen()`. `t_accept()` extracts the first connection on the queue of pending connections, creates a new socket with the same properties as `s` and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `t_accept()` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `t_accept()` returns an error as described below. The accepted socket is used to read and write data to and from the socket which connected to this one; it is not used to accept more connections. The original socket s remains open for accepting further connections.

The argument `addr` is a result parameter that is filled in with the address of the connecting entity as known to the communications layer, i.e. the exact format of the `addr` parameter is determined by the domain in which the communication is occurring. The `addrlen` is a value-result parameter. It should initially contain the amount of space pointed to by `addr`. On return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

It is possible to `t_select()` a socket for the purposes of doing an `t_accept()` by selecting it for read.

Returns

`t_accept()` returns a non-negative descriptor for the accepted socket on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also**: t_bind(), t_connect(), t_listen(), t_select(), t_socket()**

t_bind

API Name

`t_bind()`

Syntax

```
int t_bind(long , struct sockaddr *name, int namelen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for local end
namelen	Length of sockaddr_in structure (bytes)

Description

`t_bind()` assigns a name to an unnamed socket. When a socket is created with `t_socket()` it exists in a name space (address family) but has no name assigned. `t_bind()` requests that the name pointed to by name be assigned to the socket.

Returns

`t_bind()` returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno (s)`.

See Also

t_connect(), t_getsockname(), t_listen(), t_socket()

t_shutdown

API Name

`t_shutdown()`

Syntax

```
int t_shutdown(long s, int how);
```

Parameters

s	Socket identifier
how	Type of shutdown (SHUT_RD, SHUT_WR, or SHUT_RDWR)

Description

The `t_shutdown()` call causes all or part of a full-duplex connection on the socket associated with `s` to be shut down. If `how` is `0`, then further receives will be disallowed. If `how` is `1`, then further sends will be disallowed. If `how` is `2`, then further sends and receives will be disallowed.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_connect(), t_socket()

t_getpeername

API Name

`t_getpeername()`

Syntax

```
int t_getpeername(long s, struct sockaddr *name, int * addrlen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for local end
addrlen	Pointer to storage for length of sockaddr_in structure (bytes)

Description

Fills in the passed `struct sockaddr` with the IP addressing information of the connected host.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_bind(), t_socket()

t_getsockname

API Name

`t_getsockname()`

Syntax

```
int t_getsockname(long s, struct sockaddr *name, int * addrlen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for local end
addrlen	Pointer to storage for length of sockaddr_in structure (bytes)

Description

`t_getsockname()` returns the current name for the specified socket, in the passed `struct sockaddr`.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_bind(), t_getpeername(), t_socket()

t_getsockopt, t_setsockopt

API Name

`t_getsockopt()`

`t_setsockopt()`

Syntax

```
int t_getsockopt(long s, int level, int optname, char *optval, int optlen);
```

```
int t_setsockopt(long s, int level, int optname, char *optval, int optlen);
```

Parameters

s	Socket identifier
level	Level of socket option (IP_OPTIONS or SOL_SOCKET)
optname	Name of socket option (e.g., SO_ERROR)
optval	Pointer to storage for socket option (for reading (get) or writing (set))
optlen	Size of storage pointed to by 'optval'

Description

`t_getsockopt()` and `t_setsockopt()` manipulate options associated with a socket. The `optname` parameter identifies an option that is to be set with `t_setsockopt()` or retrieved with `t_getsockopt()`.

The parameter `optval` is used to specify option values for `t_setsockopt()`. On calls to `t_setsockopt()` it generally contains a pointer to a variable or structure, the contents of which will define the value of the option to be set. On calls to `t_getsockopt()` it generally points to a variable or structure into which the value for the requested option is to be returned.

The include file `socket.h` contains definitions for option names, described below. Most options take a pointer to an int variable for `optval`. For `t_setsockopt()`, the variable addressed by the parameter should be non-zero to enable a Boolean option or zero if the option is to be disabled.

`SO_LINGER` uses a `struct linger` parameter defined in `socket.h`. This parameter specifies the desired state of the option and the linger interval (see below).

In addition to those referenced in Quick List for Socket Options, the following options are recognized by the InterNiche stack. Except as noted, each may be examined with `t_getsockopt()` and set with `t_setsockopt()`.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)
SO_CALLBACK	set a callback function for the socket (set only)
IP_HDRINCL	set inclusion of IP header in data (`SOCK_RAW` only)

`SO_DEBUG` enables debugging in the underlying protocol modules.

`SO_REUSEADDR` indicates that the rules used in validating addresses supplied in a `t_bind()` call should allow reuse of local addresses.

`SO_KEEPALIVE` enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken. If the process is waiting in `t_select()` when the connection is broken, `t_select()` returns true for any read or write events selected for the socket.

`SO_DONTROUTE` indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

`SO_LINGER` controls the action taken when unsent messages are queued on socket and a `t_socketclose()` is performed. If the socket promises reliable delivery of data and `SO_LINGER` is set, the system will block the caller on the `t_socketclose()` attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the `linger interval`, is specified in the `t_setsockopt()` call when `SO_LINGER` is requested). If `SO_LINGER` is disabled and a `t_socketclose()` is issued, the system will process the close in a manner that allows the caller to continue as quickly as possible.

The option `SO_BROADCAST` requests permission to send broadcast datagrams on the socket. Note that the InterNiche stack supports the setting and getting of this option for compatibility but does not check its value when transmitting broadcast messages.

With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received. It will then be accessible with t_recv() calls without the MSG_OOB flag.

SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for the output and input buffers respectively. The buffer size may be increased for high-volume connections or may be decreased to limit possible backlog of incoming data. The system places an absolute limit on the values.

SO_TYPE and SO_ERROR are options used only with t_getsockopt(). SO_TYPE returns the type of the socket, for example SOCK_STREAM. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

IP_HDRINCL option is used only with SOCK_RAW sockets. The option value (pointed to by optval) is expected to be an integer; if it is non-zero it allows application access to the IP header, meaning that received datagrams include an IP header and sent datagrams are expected to be constructed with an IP header at the start of the buffer passed to the t_send() function. Its default setting is 0.

The options SO_NONBLOCK, SO_NBIO, and SO_BIO are unique to the InterNiche stack (these options do not appear in the Berkeley Sockets API) and are used to control whether a socket uses blocking or non-blocking IO.

SO_NONBLOCK allows the caller to specify blocking or non-blocking IO that works the same as the other Boolean socket options. That is, optval points to an integer value which will contain a non-zero value to set non-blocking IO or a 0 value to reset non-blocking IO. This means that we can get the current blocking or non-blocking status of a socket with t_getsockopt().

For compatibility, older InterNiche Sockets options SO_NBIO and SO_BIO are still supported. SO_NBIO is used to specify that a socket use non-blocking IO. SO_BIO is used to specify that a socket use blocking IO. The use of t_setsockopt() to set these options is different than that of the standard Boolean options in that the value in optval is not used. All that is necessary is to specify the appropriate option name in optname.

SO_NBIO	Set socket to use non-blocking IO.
SO_BIO	Set socket to use blocking IO.

The SO_CALLBACK option is also specific to the InterNiche stack and is only available if the stack has been built with the TCP_ZEROCOPY option enabled.

Returns

These return 0 on success. On failure, they return -1 and set an internal t_errno to one of the errors listed in Sockets Errors to indicate the error. The t_errno can be retrieved by a call to t_errno(s).

See Also

Quick List for Socket Options

t_socket()

tcp_sleep, tcp_wakeup

API Name

`tcp_sleep()`

`tcp_wakeup()`

Syntax

```
void tcp_sleep(void *address);

void tcp_wakeup(void *address);
```

Description

These functions provide a mechanism by which the InterNiche TCP code can yield control of the target processor while waiting for one or more events to occur. The functions' address parameters provide a mechanism by which the source of the events can be synchronized.

See the detailed description of this in the TCP Sleep section of this document.

Utility Functions

inet_ntop

inet_ntop()

Name

`inet_ntop()`

Syntax

```
const char *inet_ntop(int af, const void *addr, char *str, size_t size);
```

Parameters

af	Address family (AF_INET or AF_INET6)
addr	Pointer to storage for IPv4 address ('ip_addr') or IPv6 address ('struct in6_addr') in network byte order
str	Pointer to storage for string that will contain IPv4 address in dotted decimal notation, or an IPv6 address in colon-separated notation
size	Length of output buffer ('str')

Description

This functions converts a binary representation of an IPv4 address or IPv6 address (in network byte order) into a string in dotted decimal notation. The output buffer must be at least 16 (or 40) bytes long for an IPv4 (or IPv6) address.

Returns

This function returns NULL if it encountered an error; otherwise, it returns the third argument ('str').

inet_pton

inet_pton()

API Name

```
inet_pton()
```

Syntax

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

Parameters

af	Address family (AF_INET or AF_INET6)
src	Pointer to string containing IPv4 address in dotted decimal notation, or an IPv6 address in colon-separated notation
dst	Pointer to storage for IPv4 address ('ip_addr') or IPv6 address ('struct in6_addr') where the results of the conversion will be stored (in network byte order)

Description

This functions converts a string containing an IPv4 or IPv6 address in printable format into its equivalent binary representation (in network byte order).

Returns

This function returns 0 if the conversion was successful. A non-zero return value indicates a failure.

13 IPv6 Sockets

The IETF IPv6 working group has produced RFC-2133, which formally documents the use of "sockets" as a TCP/IP API. No API for IPv4 was ever formally documented by the IETF, which historically has taken the position that it should only specify protocols and not APIs. RFC-2133 is an "informational" RFC, which means it does not specify a standard - only a description of one way of doing things. Nevertheless, the InterNiche code provides extensions and changes to the InterNiche IPv4 sockets interface to bring the IPv6 product into compliance with this document.

IPv4 applications require changes to use IPv6 sockets. The scope of changes varies greatly, depending on the complexity of the application and the extent to which it encodes IP addressing information in data streams. This section reviews the potential changes throughout the InterNiche API. The reader should already be familiar with IPv4 sockets, as described in Sockets.

13.1 socket6.h

RFC-2133 recommends a set of new structures and definitions for using IPv6 sockets, and is implemented by the InterNiche code. All the additional definitions for IPv6 sockets are in the InterNiche header file "`socket6.h`". This header file should be the only additional header needed to port an IPv4 application to IPv6.

RFC-2133 also recommends a series of header files, including path names, for use with IPv6 sockets. These paths and files are not included are part of the InterNiche deliverables, however providing an environment which simulates them is very easy. The engineer should simply create the needed paths (if they don't already exist on the development system), and inside each path create the named header files (again, if they don't already exist). The contents of these files should simply be a line including the all-purpose InterNiche IPv6 socket header, socket6.h.

13.2 Socket Creation

In almost every IPv4 application, the code used to create and connect sockets will need to be changed to support IPv6. In some cases (such as the InterNiche telnet server), this is the only change required.

As you may suspect, IPv6 needs to associate the longer IPv6 addresses with socket structures in place of the IPv4 addresses. In addition, a new address family parameter, "`AF_INET6`", is defined in addition to the traditional `AF_INET`.

Throughout the IPv6 sockets extensions, structures are usually named by taking the name of the traditional IPv4 structure and appending a "`6`" to the name. `AF_INET` and `AF_INET6` are the first of many examples of this naming convention.

The type of socket (v4 or v6) is determined by the address family parameter passed to `socket()` when the socket is created. Once created, a socket may only be used for the family indicated. Below is an example IPv6 socket create call:

```
SOCKTYPE sock;

sock = t_socket(AF_INET6, SOCK_STREAM, 0);
if (sock == INVALID_SOCKET)
   return INVALID_SOCKET;
```

The IPv6 socket create call only differs from the IPv4 version by using `AF_INET6` instead of `AF_INET`.

13.3 Connecting

Once created, a socket is usually connected or put into a listen mode. Either step involves associating the socket with IP addressing information, and therein is the major difference between v4 and v6 sockets.

The IPv6 addresses are accommodated by a new type of "`sockaddr`" structure, `sockaddr_in6`. This replaces the "`sockaddr_in`" structure widely used in IPv4.

```
struct sockaddr_in6
{
   u_short        sin6_family;   /* AF_INET6 */
   u_short        sin6_port;     /* transport layer port # */
   uint32_t       sin6_flowinfo; /* IPv6 flow information */
   ip6_addr       sin6_addr;     /* IPv6 address */
   uint32_t       sin6_scope_id; /* set of interfaces for a scope */
};
```

Active TCP connections (those which call `connect()` rather than `listen()`) need to pass a `sockaddr_in6` structure to the `connect()` call, rather than the v4-ish `sockaddr_in` structure. As mentioned above, the socket must have been created as an IPv6 socket by passing `AF_INET6` to the `socket()` call.

The a `sockaddr_in6` structure is larger than the a `sockaddr_in` structure, so the structure size parameter passed to `connect()` must reflect this. The InterNiche code performs a check on the size of the sockaddr length field, but currently the only effect of an incorrect size is a `dtrap()` call.

The `sockaddr_in6` structure must have the `sin6_family` field set to `AF_INET6`, and the `sin6_flowinfo` field (which is new for IPv6) should be set to zero. [Note: The `sin6_flowinfo` field will be used for setting the IPv6 header's "flow label" field, pending specification by the IETF.]

The `sin6_port` field is identical to the IPv4 `sockaddr_in->sin_port` field. It contains the 16-bit TCP or UDP port number to which the socket is to be bound.

`sin6_addr` is the 128 bit IPv6 address for the socket. The `sockaddr_in6` structure contains a complete copy of the address, not a pointer. In InterNiche code this is best set with `IP6CPY()`.

Here is the example IPv6 active connect code from the FTP server:

```
IP6CPY(&ftpsin.sin6_addr, &ftp->ip6_host);
ftpsin.sin6_port = htons(ftp->dataport);
ftpsin.sin6_family = AF_INET6;
```

```
e = t_connect(sock, (struct sockaddr*)&ftpsin, sizeof(ftpsin));
if (e != 0)
{
   ... error handling
```

Passive TCP connections over IPv6 (those which form TCP connections via the `bind()` - `listen()` - `accept()` sequence) differ from IPv4 passive connections by passing the `sockaddr_in6` structure to `bind()`, rather than the IPv4 version `sockaddr_in`. As with `connect()`, the sockaddr size parameter must reflect the size of a `sockaddr_in6`. The members of the `sockaddr_in6` structure are set in the same way as they are for a `connect()` call (see above).

Here is the example IPv6 bind code from the FTP server:

```
IP6CPY(&ftpsin6.sin6_addr, &in6addr_any);
ftpsin6.sin6_port = htons(*lport);
ftpsin6.sin6_family = AF_INET6;
e = t_bind(sock, (struct sockaddr *)&ftpsin6, addrlen);
if (e != 0)
{
   ... error handling
```

Most other socket calls are the same for IPv6 and IPv4 sockets. Specifically, the calls to read write and close sockets are identical between the two IP versions.

13.4 FTP and IP addressing in the data stream

Of all the RFC-defined InterNiche TCP/IP applications, only FTP transmits IP addressing information (IP addresses and port numbers) in the data stream. RFC-2428 ("FTP Extensions for IPv6 and NATs") provides the specification for encoding IPv6 addresses in the FTP command socket's ASCII data stream. The RFC-2428 specification also provides for encoding IPv4 address in the FTP command socket's ASCII data stream.

The InterNiche FTP server code supports RFC-2428 IPv6 encoding, since the FTP standard (RFC-959) does not allow for IP addresses other than IPv4. The IPv4 encoding is still done per RFC-959, since it will probably be many years before FTP servers on IPv4 networks widely support RFC-2428 encoding.

13.5 Socket domain field

Throughout the sockets and transport layer InterNiche code, IPv4 sockets are distinguished from IPv6 sockets by the new socket structure field `so_domain`. This field is set to either `AF_INET` or `AF_INET6`, and is present no matter what compile-time definitions are used.

InterNiche IPv6 allows the scope ID to be set for outbound packets from a socket. This becomes very useful if the destination address of the socket is link local and is accomplished using the `IP_SCOPEID` option to `t_setsockopt()`.

```
t_setsockopt(data_sock, 1, IP_SCOPEID, &<scope ID>, sizeof(<scope ID>));
```

where <scope ID> is defined as an `int`.

More commonly the scopeID should simply be set in the `sockaddr_in6` structure passed to `t_bind()`, `t_sendto()`, etc.

13.6 ICMPv6 callbacks

ICMPv6 provides the potential for the receipt of a number of informational packets the may be useful to applications and higher layer protocol code (e.g. raw sockets). These received ICMPv6 packets may be accessed by installing optional callback routines with the following C definition:

```
int (*icmp6_callback)(PACKET);
```

Code that sets these pointers should first save the present value of the pointer as the "next callback". If the "next callback" pointer is non-null, it means that another code module has set the pointer and expects access to the received packet. In this case, the callback routine should "daisy-chain" to the non-null value, passing the unchanged received packet to "next callback".

The received packet is responsible for freeing the `PACKET` structure if the callback routine is the last (or only) routine in the daisy-chain as indicated by the "next callback" value being `NULL`. The `PACKET` structure should be freed (via a call to `pk_free(pkt)`) or have other arrangements made to ensure that it eventually returns to the free buffer queues.

13.7 IP6EQ and IP6CPY

`IP6EQ()` and `IP6CPY()` macros are used to manipulate IPv6 addresses. In addition to providing a mechanism for efficient per-port implementations, they are also designed to detect programming errors where IPv6 addresses and pointers to addresses are inadvertently confused. To this end, they should generally be implemented as C functions (possibly in-line), or in-line assembly language code. Default definitions of these routines are included in `h/ipv6.h`, however these defaults are inefficient and do not check the parameters. They should only be used during initial prototyping of a port.

The recommended implementation method is to provide these two routines with other names (e.g. `ip6cpy` and `ip6eq`), and `#define` them to `IP6EQ()` and `IP6CPY()`. This enforces global type checking of the passed IPv6 address parameters, and prevents the inefficient default macros in `ipv6.h` from being used.

Here are the reference definitions from the Windows port `ipport.h` file:

```
struct in6_addr;   /* predecl */
extern void ip6cpy(struct in6_addr * dest, struct in6_addr * src);
#define IP6CPY(a,b) ip6cpy(a,b)
extern  int ip6eq(struct in6_addr * dest, struct in6_addr * src);
#define IP6EQ(a,b) ip6eq(a,b)
```

Intel 386 Assembly versions of these are in `w32_in_vc/osport.c`.

`IP6CPY(destination, source)` copies an IPv6 address from the second pointer passed to the first pointer passed. It is meant to replace the simple assignment operator that was useful with IPv4 addresses.

Since the passed addresses are always 128 bits long, the best implementation is probably to do four get/put sequences of 32 bits each. CPUs that have a fast built-in string move operation (such as Intel x86) may opt to use that instead.

`IP6EQ(addr1, addr2 )` does a fast compare of the two IPv6 addresses. It returns `TRUE` if the IP addresses match exactly, and `FALSE` is they do not. Most often the addresses passed will not match, so the code should be optimized accordingly. Also, mismatches are more likely near the end of the IPv6 addresses than at the beginning. Optimize accordingly.

13.8 Address Identification Macros

The following macros can be used to determine the type of IPv6 address.

IN6_IS_ADDR_MULTICAST	identify IPv6 multicast addresses
IN6_IS_ADDR_MCSCOPE	identify IPv6 multicast scope
IN6_IS_ADDR_GLOBAL	identify IPv6 global address - highest 3 bits are 001
IN6_IS_ADDR_LINKLOCAL	identify IPv6 Link-local addresses
IN6_IS_ADDR_SITELOCAL	identify IPv6 Site-local addresses
IN6_IS_ADDR_UNSPECIFIED	identify IPv6 unspecified addresses
IN6_IS_ADDR_LOOPBACK	identify IPv6 loopback addresses

14 TCP Zero-Copy

14.1 Overview

This section documents an optional extension to the InterNiche Sockets layer, the TCP Zero-Copy API. This extension is only present if the stack has been built with the `TCP_ZEROCOPY` package option defined in `ipport.h`. See Package Options for information about how to enable this option.

The TCP Zero-Copy API is intended to assist the development of higher-performance embedded network applications by allowing the application direct access to the InterNiche TCP/IP stack's packet buffers. This feature can be used to avoid the overhead of having the stack copy data between application-owned buffers and stack-owned buffers in `t_send()` and `t_recv()`, but it comes at the cost that the application will have to fit its data into, and accept its data from, the stack's buffers.

The TCP Zero-Copy API comprises two functions for allocation and freeing of packet buffers, a third function for sending a packet buffer on an open socket, an application-supplied callback function for accepting received packets, and an extension to the Sockets `t_setsockopt()` function for registration of the callback function. The TCP Zero-Copy API can be this small because it is simply an extension to the existing Sockets API that provides an alternate mechanism for sending and receiving data on a socket, and the Sockets API is used for all other operations on the socket.

The two functions for allocation and freeing of packet buffers are straightforward requests to allocate a packet buffer from the stack's pool of packet buffers, `tcp_pktalloc()`, and free a packet buffer, `tcp_pktfree()`. Applications using the TCP Zero-Copy API are responsible for allocating packet buffers for use in sending data, as well as for freeing buffers that have been used to receive data and those that the application has allocated but decided not to use for sending data. As these packet buffers are a limited resource, it is important that applications free them promptly when they are no longer of use.

The function for sending data, `tcp_xout()`, sends a packet buffer of data via a socket. If successful, it is considered to have consumed the supplied buffer and so there is no need for the application to free the buffer via `tcp_pktfree()`.

Applications that use the TCP Zero-Copy API for receiving data must include a callback function for acceptance of received packets, and must register the callback function with the socket using the `t_setsockopt()` Sockets function with the `SO_CALLBACK` option name. The callback function, once registered, receives not only received data packets, but also connection events that result in socket errors.

`TCP_ZEROCOPY` may be used with the`ONEBUF` define. The intention of `ONEBUF` is to bypass NicheStack's `CHAINED_BUFFERS` feature and to place both the packet headers and packet data in a single contiguous buffer.

When `ONEBUF` is defined, `tcp_pktalloc()` will automatically add `HDRSLEN` or `HDRSLEN6` to the specified datasize, and it will set `nb_prot` to one byte beyond the headers. When it receives a packet `tcp_xout()`

will set the `SB_ONEBUF` buffer flag for the send buffer of that socket. The `SB_ONEBUF` tells the code to assume that there is room to add any required headers in front of the data within the packet and the chain buffer code will be disabled for that socket.

Note the `SB_ONEBUF` flag is socket specific. For example, when FTP transfers a file, it uses two sockets, one for control messages and one for data--the actual file transfer. FTP only uses `ZERO_COPY` for the data socket, so the `SB_ONEBUF` flag is only set for the data socket. TCP treats the two sockets differently. For the control socket it uses the chained buffer code, but for the data socket it puts the headers and data within the same packet.

14.2 Sending Data with the TCP Zero-Copy API

The first step in using the TCP Zero-Copy API is to ensure that, if `IP_V4` is defined, `HDRSLEN` is defined correctly in `tcpport.h`, and that if `IP_V6` is defined, `HDRSLEN6` is defined correctly in `ip6.h`. Normally no change will be required for these two defines.

When opening a socket, call `t_socket()`, and `t_connect()` as normal. Once a connection has been made, call `tcp_mss()` to determine the MSS (maximum segment size)—that is, how much data can the application put in one packet. Save this information for future writes.

The following is a brief listing of the steps needed to write a packet of data using the Zero-Copy API. The functions `tcp_pktalloc()`, `tcp_xout()`, and `tcp_pktfree()` are described in more detail in their separate sections.

1. Call `tcp_pktalloc(datasize, domain)` to allocate the packet.
2. Write up to MSS data in the packet starting at `pkt->nb_prot`.
3. Set `pkt->nb_plen` and `pkt->nb_tlen` to the size of the data written into the packet
4. Call `tcp_xout(socket, pkt)`
5. Handle the return from `tcp_xout()`. Note: If the return is negative, the system did not accept the packet. The application must either call `tcp_pktfree()` to free it or it can retry the packet later.

14.3 Receiving Data with the TCP Zero-Copy API

Writing a Callback Function

Using the TCP Zero-Copy API for receiving data requires the application developer to write a callback function that the stack can use to inform the application of received data packets and other socket events. This function is expected to conform to the following prototype:

```
int (*rx_upcall)(struct socket *so, void *pkt, int error);
```

The stack will call this function when it has a received data packet or other event to report for a socket. When invoked, the `pkt` will contain the address of a `struct netbuf`.

If the application is using the same callback function for several sockets, it can use `so` to identify the socket for which the callback has occurred. For example, the following code fragment walks a list of data structures to find one with a matching socket, and illustrates a way to compare the so argument with a socket returned by `t_socket()`.

```
for (ftps = ftplist; ftps; ftps = ftps->next)
   if((long)ftps->datasock == SO2LONG(so))
      break;
```

Once the callback function has identified the socket, it should examine the `pkt` and `code` parameters as these contain the information about the socket.

If `pkt` is not `NULL`, it is a pointer to a packet buffer containing received data for the socket. `pkt->nb_prot` points to the start of the received data, and `pkt->nb_plen` indicates the number of bytes of received data in the buffer. The amount of data in the buffer chain is available in `pkt->nb_tlen` field of the first buffer in the chain. If the callback function returns `0`, it indicates that it has accepted responsibility for the packet buffer and will return it to the stack (via the `tcp_pktfree()` function) when it no longer requires the buffer. If the callback function returns any non-zero value, it indicates to the stack that it has not accepted responsibility for the packet buffer. The stack will keep the packet buffer queued and will call the callback function again at a later time.

If `code` is not `0`, it is a socket error indicating that an error or other event has occurred on the socket. Typical non-zero values will be `ESHUTDOWN`, indicating that the connected peer has closed its end of the connection and will send no more data; and `ECONNRESET`, indicating that the connected peer has abruptly closed its end of the connection and will neither send nor receive more data.

Note that the callback function is called from the stack and is expected to return promptly. Some of the places where the stack calls the callback function require that the stack's data structures remain consistent through the callback, so the callback function should not call back into the stack except to call `tcp_pktfree()`.

Reading the Data

The following is a brief listing of the steps needed to read a packet of data using the Zero-Copy API.

- Start the listening task normally with `t_socket()`, `t_bind()`, `t_listen()` and `t_accept()`
- After accepting a socket connection, call `t_setsockopt()` to set the call back function, e.g.,

  ```
  t_setsockopt(sock, 0, SO_CALLBACK, (void *)rxupcall, 0);
  ```

- When packets are received, the callback function should find the application structure for the socket, save the value of the rxcode parameter, put the received packet on a queue for the application, and optionally wake the application.
- The input function for the application should follow the pseudo code below:

```
while (packets on queue)
{
   dequeue the packet;
   Handle the saved rxcode value;
   Process the packet;
   Call tcp_pktfree().
}
```

14.4 TCP Zero-Copy API Reference

tcp_pktalloc

API Name

`tcp_pktalloc()`

Syntax

```
PACKET tcp_pktalloc(int datasize, int domain);
```

Parameters

```
int datasize /* size of TCP data for packet */
```

```
int domain /* AF_INET for IPv4, AF_INET6 for IPv6 */
```

Description

`tcp_pktalloc()` allocates a packet buffer large enough to hold datasize bytes of TCP data, plus TCP, IP, and MAC headers. It is a small wrapper around the internal `pk_alloc()` function that provides the necessary synchronization and calculation of header length.

`tcp_pktalloc()` should be called to allocate a buffer for sending data via `tcp_xout()`. It will return the allocated packet buffer with its `pkt->nb_prot` field set to where the application should deposit the data to be sent.

Returns

Returns a `PACKET` (pointer to `struct netbuf`) if OK, else `NULL` if a big enough packet was not available.

Notes

The `domain` field is ignored unless `ONEBUF` is defined, in which case `tcp_pktalloc()` will automatically add the length of the headers for the domain to the size of the packet buffer and `pkt->nb_prot` will be moved to the next byte beyond the headers to point to where the application should write its data.

See Also

tcp_pktfree(), tcp_xout()

tcp_pktfree

API Name

`tcp_pktfree()`

Syntax

```
void tcp_pktfree(PACKET p);
```

Description

`tcp_pktfree()` frees a packet allocated by (presumably) `tcp_pktalloc()` or passed to the application by a callback. This is a simple wrapper around `pk_free()` to lock and unlock the free-queue resource.

Parameters

```
PACKET p /* the pointer to the packet to be returned to the Protocol stack
*/
```

Returns

No value is returned. If the passed packet is already in a free queue, has been corrupted, or does not appear to be a valid packet, a `dtrap()` may be generated by the debugging logic.

See Also

tcp_pktalloc()

tcp_xout

API Name

`tcp_xout()`

Syntax

```
int tcp_xout(long s, PACKET pkt);
```

Parameters

```
long s /* socket on which packet is to be sent */

PACKET pkt /* pointer to packet to be sent */
```

Description

The `tcp_xout()` call sends a packet buffer on a socket. The packet buffer must be initialized with `pkt->nb_prot` pointing to the start of the application data to be sent (this will have been set properly by `tcp_pktalloc()`), and with `pkt->nb_plen` set to the number of bytes of data to be sent.

Returns

An integer indicating the success or failure of the function. A returned value of zero indicates that the packet was sent successfully. Returned values less than zero indicate errors, and that the packet was not accepted by the stack (so the application must either re-send the packet via a later call to `tcp_xout()` or free the packet via `tcp_pktfree()`). Returned values greater than zero indicate that the packet has been accepted and queued on the socket but has not yet been transmitted.

See Also

tcp_pktalloc(), tcp_pktfree()

15 MAC Drivers

The InterNiche IPv6 stack uses essentially the same MAC layer API (defined in the NicheStack manual) as all other InterNiche products, with one additional requirement - the multicast feature, which is optional in IPv4, is required by IPv6. There is also a measurable performance penalty for MAC drivers that do not implement support for the scatter/gather method of chaining PACKET structures together.

Legacy MAC drivers (from products which predate the IPv6 release) should work with IPv6 as long as they implement the multicast option.

15.1 Scatter gather - performance

As mentioned in IPv6, the IPv6 stack may prepare a packet to be sent in multiple discontinuous buffers. These buffers are managed by a linked list of PACKET structures. The pk_prev and pk_next members of each PACKET point to (respectively) the previous and next members of the list. That packet's nb_plen field gives the number of bytes in each segment, and the nb_prot field points to the segment data.

MAC drivers may optionally provide support for handling the sending of these PACKET lists. The MAC driver indicates this support to the stack by setting the NF_GATHER bit in the net structure's nb_flags field. If this bit is set, then "scattered" packets will be passed to the interface's n_pkt_send() routine for sending, and the driver is responsible for collecting the separated data segments and sending them as a contiguous MAC packet.

Legacy drivers, or drivers which cannot support efficient sending of linked lists of PACKETs, should not set the NF_GATHER bit in the net structure's nb_flags field. This will cause the IPv6 code to assemble the linked list of PACKETs into a single large packet before passing the packet to n_pkt_send(). This will involve copying most of the packet data, so drivers that can support the linked lists should do so.

Receiving packets with NF_GATHER support is no different than receiving those without this support. The IP level code assumes the received packets are in a single contiguous buffer, with the total data length given by nb_plen. The only issue worth noting is that it is good form to set the nb_tlen field (see below) as well as the nb_plen field.

nb_tlen length - primarily on sends

IPv6 packets that are being prepared for sending are usually in linked lists as described above. The older data length field, nb_plen, is used is indicate how much data is in a single PACKET's buffer. When multiple PACKETs are linked into a list, it's convenient to have a total length for all PACKETs in the list. The nb_tlen field serves this function.

MAC drivers which support scatter/gather may use this field to determine buffer requirements without having to traverse the linked list of PACKETs. The nb_tlen field is only guaranteed to be accurate in the first PACKET of a linked list. This PACKET is identified by the tk_prev field being NULL.

15.2 Multicast is required

The one change that may be required for a MAC driver to support IPv6 is support for multicast packets. Multicast was an optional driver feature in previous releases, and the specification for driver support of multicast packets has not changed.

The key point is that the MAC driver must receive all multicast packets with a destination MAC address matching any of the addresses registered with the drivers via calls to the drivers `n_mcastlist()` routine.

Programming multicast addresses to Ethernet devices can be error prone, and some Ethernet hardware devices have limited space to store multicast address. For these reasons, many Ethernet devices offer a feature that allows the hardware to receive (and send) all multicast packets. This frees the programmer from having to keep the multicast address list in the hardware synchronized with the lists in the IPv6 layers, and considerably simplifies all aspects of programming the hardware.

Since multicast addresses make up a small portion of the traffic on most IPv6 networks, this "generic multicast" approach generally does not have problems with excessive interrupts from uninteresting multicast packets; however system engineers should keep the possibility in mind. Some forms of video and audio streaming rely on heavy amounts of multicast traffic. If these, or similar applications, become popular in the future it could create problems for embedded devices with under powered CPUs.

16 GIO - Generic IO

The Generic Input/Output (GIO) interface is an API for reading and writing data in a device-independent manner. This is particularly useful for applications, such as the Command Line Interpreter (CLI), which reads comands from a keyboard, file or socket, and sends command output to a display, file or socket. Separate data streams are maintained for input and output so that data can be read from one type of device and written to a different type of device.

GIO can be thought of as a wrapper around device-dependent read and write functions. A device is opened using device-specific functions. After a device is opened, a GIO context is created to manage the device, using the device's descriptor and device-specific read and write functions. When the GIO context is no longer needed, the device's close function is called, and then the GIO context can be freed.

There is not a one-to-one mapping between GIO devices and hardware devices. There can be several variants of a GIO device, each of which performs a different data transformation while transferring data between the application and the actual hardware device. For example, a GIO socket device may implement HTTP data chunking on data as it is being written to a socket, while another GIO socket device may implement Telnet options processing on a Telnet socket as the application is reading the data stream. This flexibility is accomplished by allowing the application to specify the device-specific read an write functions when the GIO context is created.

16.1 GIO Contexts

A GIO context implements two uni-directional data streams; one for reading and one for writing. Data transfers occur between an application and a device, such as a socket or file or console. A different device may be associated with each of the data streams. For example, an application could process data read from a file and send the results to a socket.

A GIO context structure can be created either statically or dynamically.

Each data stream in a GIO context is initialized with information about the associated device; device type code, device descriptor, and a device-specific I/O function. Once a GIO context has been created and initialized, an application can read and write data through the GIO data streams. When an application is finished, the device is closed and the GIO context destroyed.

GIO contexts support I/O redirection by allowing applications to "push" and "pop" a GIO context. A "push" operation saves the current GIO context and creates a new GIO context which is a copy of the previous GIO context. A "pop" operation destroys the current GIO context and restores the previous GIO context. The "push" and "pop" operations are performed in a manner that preserves the validity of any references to the current GIO context.

I/O redirection is accomplished by performing a "push" and then overwriting the data stream(s) with new device information. For example, a application creates a GIO context to read from and write to the console device. To read from a file, the application opens the file, "pushes" the GIO context, changes the input device to be the file descriptor, and changes the I/O function to a "read from file" function. Because the output stream was unchanged by the "push" operation, output data will continue to be written to the console

device. When the end of the file is reached, the application closes the file descriptor and "pops" the GIO contex. This action restores the previous console data streams.

A GIO context includes a set of flags which applications can use to configure the GIO context. These flags include the ability to select blocking or non-block I/O, to enable or disable echoing of console input, and to select binary or ASCII data processing modes. Not all device I/O functions support all of these flags. Additional flags can be defined as additional devices and I/O functions are implemented.

A GIO context also includes a pointer to a callback function. This function is intended to be used to signal asynchronous events to the GIO context and/or the application using the GIO context.

16.2 GIO Context Structure

A GIO context structure is defined as follows:

```
struct gio_stream
{
   void      *id;              /* device handle */
   GIO_FUNC io_func;           /* i/o function pointer */
   u_short  type;              /* device type */
   u_short  ref;               /* reference count */
};

struct gio {
   struct gio  *next;          /* previous gio struct */
   struct gio_stream in;       /* input stream */
   struct gio_stream out;      /* output stream */
                               /* i/o done callback function */
   int (*done)(struct gio *, void *, int32_t);
   void *param;                /* callback parameter */
   uint32_t flags;             /* i/o flags */
};
```

A GIO context contains two gio_stream structures, one for input and one for output. Each gio_stream structure includes a device id field, a reference to an I/O function, a device type code, and a reference count. The device id field is used to reference the actual device. For example, it can be a pointer to a socket structure, a file descriptor, a UART device index, etc. The I/O function should cast the device id value into a data type appropriate for the function.

The device type code should be one of the defines shown in the table below for the supported devices:

GIO_NONE_T	no device
GIO_NULL_T	“null” device
GIO_CONSOLE_T	Console
GIO_SOCKET_T	Socket
GIO_FILE_T	VFS file

GIO_TELNET_T	telnet connection
GIO_HTTP_T	HTTP connection
GIO_UART_T	UART
GIO_USER_T	user-defined device

Its presence is primarily for debugging purposes, but it can be used to select device-specific actions. One usage is in a command interpreter application, where some commands may not be valid for some devices. The "exit" command, for example, may only be valid for TELNET devices.

Note that a TELNET device is an example of a customized device. The device id is a pointer to the TCP socket used by the Telnet connection. The I/O functions are customized socket functions which perform Telnet options processing in addition to generic socket I/O processing. It is convenient to assign this GIO stream a unique device type.

The reference count field is initialized to one when the GIO context is created. It is incremented whenever the GIO context is "pushed" and decremented whenever the context is "popped". Applications can use this information to know when to close a device that is no longer in use.

A set of configuration flag bits is shared by the two GIO streams. These flags can be used to customize processing of the data without the need to implement additional I/O functions. The set of flags is expected to evolve as new devices or device variants are implemented. Some flags are specific to a GIO stream ("echo input" only applies to interactive character input streams) or to a device ("ASCII mode" only applies to TELNET devices).

The following is a list of the currently supported flags:

GIO_F_ASCII	ASCII mode
GIO_F_CR	<CR> is the line terminator
GIO_F_LF	<LF> is the line terminator
GIO_F_CRLF	<CR><LF> is the line terminator
GIO_F_BIN	Block on input
GIO_F_BOUT	Block on output
GIO_F_BIO	Block on both input and output
GIO_F_ECHO	Echo input

The callback function gives the application that owns the GIO context, a mechanism to handle asynchronous events, such as using a keystroke to abort an activity in progress. The application stores the callback function in the GIO context before starting the activity. When an event occurs, the event handler can call the GIO callback function to notify the application. A user-defined parameter is passed to the function, which can be used to identify the caller or select the action to be performed. How and when the callback mechanism is used is completely up to the application. Care must be taken to ensure that resource locks are respected during the callback in order to prevent deadlocks.

16.3 GIO API

The GIO API consists of the following functions:

gio_dev()	initialize a GIO context
gio_in()	read data from the GIO input stream
gio_out()	write data to the GIO output stream
gio_pop()	"pop" the current GIO context
gio_printf()	output formatted data
gio_push()	"push" the current GIO context

gio_dev

API Name

`gio_dev() - initialize a GIO context`

Syntax

```
GIO *gio_dev(GIO *gio, void *id, GIO_FUNC in, GIO_FUNC out, int rw, int
type)
```

Parameters

gio	pointer to the GIO context
id	device id
in	input function
out	output function
rw	update flags: GIO_R = update input stream fields GIO_W = update output stream fields GIO_RW = update input and output stream fields
type	device type code

Description

If `GIO_R` is set in the '`rw`' field, update the input stream with the '`id`', '`in`', and '`type`' values. If `GIO_W` is set in the '`rw`' field, update the output stream with the '`id`', '`out`', and '`type`' values.

Prototypes for generic device-specific input and output functions can be found in gio.h.

Returns

A pointer to the updated `GIO` context is returned.

gio_done

API Name

```
gio_done() - call the GIO callback function
```

Syntax

```
int gio_done(GIO *gio, int32_t code)
```

Parameters

gio	pointer to the GIO context
code	"done" code

Description

Call the GIO context's 'done' function. The syntax of the callback is:

```
ret = (gio->done)(gio, gio->param, code);
```

Returns

The return value from the '`done`' function is an integer completion code, where 0 means success and non-zero is a user-defined error code.

gio_in

API Name

```
gio_in() - read data from the GIO input stream
```

Syntax

```
int gio_in(GIO *gio, char *buf, uint32_t len)
```

Parameters

gio	pointer to the GIO context
buf	input buffer pointer
len	maximum input length

Description

Read a maxmimum of 'len' bytes of data from the GIO input stream. The data is stored in the buffer pointed to by 'buf'. If the `GIO_F_BIN` flag is set in the GIO context, the function blocks until 'len' bytes are read. If `GIO_F_BIN` is not set, the function returns immediately after copying any available data into the buffer.

Returns

If the return value is positive, the return value is the number of bytes read from the device. A return value of zero means there is no data available. A negative return value indicates an error occurred.

gio_out

API Name

```
gio_out() - write data to the GIO output stream
```

Syntax

```
int gio_out(GIO *gio, char *buf, uint32_t len)
```

Parameters

gio	pointer to the GIO context
buf	output buffer pointer
len	output buffer length

Description

Write 'len' bytes of data to the GIO output stream. The 'buf' parameter points to the first byte of data to be written. If the GIO_F_BOUT flag is set in the GIO context, the function blocks until 'len' bytes are written. If GIO_F_BOUT is not set, the function returns immediately after copying up to 'len' bytes into the buffer.

Returns

If the return value is positive, the return value is the number of bytes written to the device (possibly zero bytes). A negative return value indicates an error occurred.

If fewer than 'len' bytes were written to the output stream, the caller should update the buffer pointer and remaining byte count, and wait and retry the operation.

gio_pop

API Name

`gio_pop() - "pop" the current GIO context`

Syntax

```
int gio_pop(GIO **giop);
```

Parameters

giop	address of a pointer to the GIO context

Description

The current GIO context (pointed to by the GIO context variable pointed to by 'giop') is destroyed, and the GIO context pointer pointed to by the 'giop' parameter is updated to point to the previous GIO context.

Returns

The return code is a GIO error code defined in `gio.h` indicating the success or failure of the operation.

gio_printf

API Name

`gio_printf() - write formatted data to the GIO output stream`

Syntax

```
int gio_printf(GIO *gio, const char *format, ...);
```

Parameters

gio	pointer to the GIO context
format	printf()-compatible format specification string
...	output parameter list

Description

Creates a formatted output string using the format specification and the output parameter list. The formatted output string is then written to the GIO output stream. This function is equivalent to:

```
char buf[N];

sprintf(buf, format, ...);
ret = gio_out(gio, buf, strlen(buf));
return (ret);
```

The gio_out() operation is forced to be performed in blocking I/O mode (GIO_F_BOUT is set).

Returns

Function returns the result of the `gio_out()` call (see above).

gio_push

API Name

```
gio_push() - "push" the current GIO context
```

Syntax

```
int gio_push(GIO **giop, void *id, GIO_FUNC in, GIO_FUNC out, int rw, int
type)
```

Parameters

giop	pointer to the GIO context
id	device id
in	input function
out	output function
rw	update flags: GIO_R = update input stream fields GIO_W = update output stream fields GIO_RW = update input and output stream fields
type	device type code

Description

Similar to the gio_dev() function except that the current GIO context, pointed to by *giop, is saved and a new GIO context is created. The new GIO context is initialized with the values of the current GIO context, and then gio_dev() is called to update the new context with the function parameters.

gio_push() is used to change an existing context. If you were to use gio_dev() to change an existing context, then the previous context would be lost. For example, with gio_push(), if you change from reading from a socket to reading from a file, when you perform a gio_pop(), the application will again read input from the original socket.

Note: The "id" parameter is simply a pointer that is passed to the input and output routines. For example:

```
err = gio_push(&hp->ctx->gio,
               (void *)&hp->si,
               &wbs_io_in,
               &wbs_io_out,
               GIO_RW,
               GIO_SOCKET_T);
```

In this call, the "id" parameter, &hp->si, is the pointer to a structure that will be passed to both the input function wbs_io_in() and the outputfunction wbs_io_out().

Returns

The return code is a GIO error code defined in gio.h indicating the success or failure of the operation.

16.4 GIO Example

This code example illustrates how to use a GIO context to copy data from a file to the console. Allocation error checking is not included for reasons of program clarity.

```
#include "vfsfiles.h"
#include "gio.h"

static char filename[] = "myfile.txt";

void main(void)
{
   GIO *gio;

     gio = (GIO *)npalloc(sizeof(GIO));  /* allocate a GIO context */

   /* initialize the input and output streams to go to the console, using blocking output    */

   GIO_CONSOLE(gio, GIO_RW);
   gio->flags |= GIO_BOUT;

      display_file(gio, filename);   /* display a file on the console */
}

void display_file(GIO *gio, char *filename)
{
   char    inbuf[128];
   int     len;
   VFILE   fd;

   if ((fd = vfopen(filename, "r")) == NULL)
      panic("cannot open file");

   gio_printf(gio, "Display file: %s\n", filename);

   /* redirect the input stream to a VFS file */
   GIO_PUSH_FILE(&gio, fd, GIO_RD);

   /* read the file and copy it to the console */
   while (!vfeof(fd))
   {
      len = gio_in(gio, inbuf, 128);
      if (len > 0)
      {
         /* copy file data to the console */
         gio_out(gio, inbuf, len);
      }
      else if (len < 0)
      {
         break;    /* some kind of file error */
      }
   }

   /* close the file and destroy the GIO context */
```

```
    vfclose(fd);
    gio_pop(&gio);
}
```

17 NicheTool

17.1 The Menu System

NicheTool refers to both the menu system mechanism and the suite of commands available to an interactive user of InterNiche products. Broadly speaking, the NicheTool mechanism parses "command line input", and uses the initial tokens to identify the desired "C" function. Any parameters that follow the initial tokens are parsed and passed to the menu routine in a standardized format.

Since NicheTool functions use the GIO mechanism for their input and output processing, applications can often create command lines and pass them directly to the CLI system. If this is deemed inefficient or otherwise not desired, the commands provided can serve as examples for calling sequences or manipulation of structures.

NicheTool commands are parsed and processed by the CLI module. Inclusion of NicheTool commands and the CLI module is controlled by the `INCLUDE_CLI` precompiler directive. The command definitions are grouped within menus which are part of a NicheStack module. Inclusion of individual menus and the commands they contain and the code to be executed for each command is controlled by a module-specific symbol of the form `xxx_MENUS`.

A complete list of commands and the command reference document is provided with each product delivery.

The following sections describe the CLI Module's implementation of NicheTool. The level of detail is sufficient for the porting engineer to add their own commands to the NicheTool framework.

17.2 Menu Structures

The set of NicheTool commands are implemented in one or more menus. Each menu is a hierarchy of structures which define a set of commands and their parameters. The following code illustrates a simple menu:

```
static struct cli_parm user1_params[ ] = {
   { 'a', CLI_IPADDR },
   { 'p', CLI_UINT },
};

static struct cli_parm user2_params[ ] = {
   { 's', CLI_STRING },
   { 'a', CLI_NONE },
};

static struct cli_cmd user_cmds[ ] = {
   {
      "user1",
      "first user command",
      &user1_func,
      sizeof(user1_params)/sizeof(user1_params[0]),
      &user1_params
   },
   {
      "user2",
      "second user command",
      &user2_func,
      sizeof(user2_params)/sizeof(user2_params[0]),
      &user2_params
   },
};

struct cli_menu user_menu = { "user", 2, &user_cmds };
```

The `cli_menu` structure defines the menu. In this example, the name of the menu is "`user`" and it contains two commands. The `cli_cmd` structure defines the two commands. The first command is "`user1`" and it has two parameters; an IP address (either IPv4 or IPv6) and an unsigned 32-bit integer. The second command is "`user2`" and it has two parameters; a character string and a flag.

Before a menu of commands is available to the user, it must be added to the CLI module's array of menus. This is done by:

```
#include "cli.h"

int  err;

err = cli_install_menu(&user_menu);
if (err != SUCCESS)
{
   /* an error occurred. err is a non-zero error code */
}
```

If a menu is no longer needed, it can be removed from the CLI module's array of menus by:

```
cli_uninstall_menu(&user_menu);
```

17.3 Commands and Parameters

Within a menu, all of the command names must be unique. If two menus have a command with the same name, the command must be preceded by the name of the menu. For example, if the "`config`" command is part of both the "`his`" and "`hers`" menus, the commands can be entered as:

```
his config -y 1
hers config -x 10.0.0.100 -d
```

Since each "`config`" command is part of a separate menu definition, they do not have to have to same number or type of parameters.

A command name is followed by zero or more parameters. Each parameter consists of a letter preceded by a hyphen and optionally followed by a value. The parameter letter and the parameter value are separated by "whitespace". The parameter letter is not case-sensitive.

The type of each parameter is checked as the command is parsed. Supported parameter types are:

CLI_NONE	No parameter value. The command function can test for the presence or absence of the parameter.
CLI_INT	Signed 32-bit integer.
CLI_UINT	Unsigned 32-bit integer.
CLI_STRING	A character string delimited by "whitespace". Strings containing "whitespace" can be delimited by matching single quotes, double quotes, or parentheses: `-a 'hello world!'`, `-a "I don't know"`, or `-a (a, b, c)`.
CLI_IPADDR	IPv4 or IPv6 address. IPv6 addresses may include numeric scope id and prefix length modifiers, i.e. `-a FE80::%2/64`.

17.4 User-defined Menus and Commands

Developers can add commands to NicheTool to support their development needs. All of the NicheTool menus can be deleted and replaced by user-defined menus to create a custom command set. Commands can be added to existing menus by editing the menu structure in existing modules are created as part of implementing a new module. Menus are normally installed in a module's "`init`" function and uninstalled in a module's "`close`" function.

17.5 Command Line Parsing

The CLI module is responsible for parsing a command line and calling the command's execution function. This process is accomplished by calling the function:

```
int cli_command(CLI_CTX ctx, char *cmd);
```

The "`cmd`" parameter is the NUL-terminated command string to be parsed and executed. The "`ctx`" parameter is a structure that contains all of the information needed to parse and execute a menu command. Any task that intends to use the CLI module to execute menu commands on its behalf, must first allocate a CLI context structure and initialize its GIO component as illustrated by the following code snippet:

```
#include "gio.h"
#include "cli.h"

CLI_CTX my_ctx;
char cmdbuf[80];

my_ctx = cli_get_context();        /* get an empty context */
if (my_ctx == NULL)
   panic("get_context");                /* cannot allocate context */
GIO_PUSH_CONSOLE(&my_ctx->gio, GIO_RW);  /* input/output = Console */

strcpy(cmdbuf, "setip -i 1 -a 10.0.0.52");
err = cli_command(my_ctx, &cmdbuf[0]);
if (err != SUCCESS)
   gio_printf(ctx->gio, "error: %s\n", cli_get_errstr(err, &cmdbuf[0]);
```

This example creates a context and initializes the GIO input and output streams to the NicheTool console device. Refer to the GIO section of this manual for a discussion on redirecting the input and/or output device (s). The `cli_command()` function is called to parse and execute the NicheTool "`setip`" command. If the command is not successful, the function `cli_get_errstr()` can be called to convert the CLI error code into a error string. A list of CLI error codes can be found in `h/cli.h`.

The command line parser scans the command line to find the command name. The command name by be optionally preceded by a menu name. The command parser then searches the installed menus, looking for a match; names are not case-sensitive. Command names can be abbreviated to the first `N` characters of the name, as long as the name is still unique. If the symbol `NO_SHORT_NAMES` is defined, command names must be minimum of 3 characters.

If the command is found, the corresponding `cli_cmd` and `cli_parm` structures are used to parse and validate the command parameters. The parameter information is stored in the CLI Context. A parameter may appear at most once in a command line and must be of the correct type. If a parameter needs to support multiple types, for example a device name string or a device number, the `CLI_STRING` type can be used and parameter validation can be deferred to the command's execution function.

The command parser recognizes the special symbols, '`?`' and '`-?`' as "help" flags. Entering a "help" flag as part of a command line causes the command's description and syntax to be displayed on the output device. If the command was successfully parsed, the command's execution function will then be called. The function can test if "help" was entered and take appropriate action, such as displaying addition "help" text and/or not executing the command.

Note that **the command line is modified** during the command parsing process. Commands that are stored in read-only memory, they must be copied into a writable memory buffer before they can be parsed and executed.

17.6 Command Execution

The CLI command parser finds the command's "`cli_cmd`" structure in the installed menus. The structure includes a pointer to the function to be called to execute the command. The function prototype for any command's execution function is:

```
int (*cli_func)(CLI_CTX ctx);
```

The CLI Context structure provides the function with all of the command line parameter information. The following macros can be used to access the command's parameters:

`bool_t CLI_HELP(CLI_CTX ctx)`
Evaluates to TRUE if either '?' or '-?' was entered as part of the command line. Otherwise, it evaluates to FALSE.

`int CLI_COUNT(ctx)`
Returns the number of parameters that were present in the command line. The "help" symbols are not included in the count.

`bool_t CLI_DEFINED(CLI_CTX ctx, char c)`
Evaluates to TRUE if the command line included a '-<c>' parameter. Otherwise, it evaluates to FALSE.

`void *CLI_VALUE(CLI_CTX ctx, char c)`
Returns the value of the '-<c>' parameter. The value must be cast into the type of the parameter. For example:

```
if (CLI_DEFINED(ctx, 'p')
   port = (uint32_t)(CLI_VALUE(ctx, 'p'));
else
   port = MY_DEFAULT_PORT_NUMBER;
```

The type for each CLI parameter type is:

CLI_NONE	undefined
CLI_INT	32-bit signed integer
CLI_UINT	32-bit unsigned integer
CLI_STRING	pointer to a "C" string
CLI_IPADDR	pointer to a `cli_addr` structure

The `CLI_DEFINED()` macro should be used to test for the presence of a parameter before using `CLI_VALUE()` to get the its value.

When a parameter of type `CLI_IPADDR` is parsed, the information is stored in a `cli_addr` structure defined by:

```
struct cli_addr {
   int    type;              /* CLI_IPV4, CLI_IPV6, or CLI_MAC */
#ifndef IP_V6
   uint8_t  addr[8];         /* IPv4/MAC address (network-order) */
#else
   uint8_t  addr[128/8];     /* IPv6 address (network-order) */
   int      scopeID;         /* IPv6 scopeID of address */
   int      prefixLen;       /* IPv6 prefix length of the address */
#endif
```

The bytes of the IP address are stored in network byte-order. The "`type`" field of the structure can be tested to determine if a 32-bit IPv4 address or a 128-bit IPv6 address was entered in the command line:

```
#include "cli.h"

ip_addr  my_ipv4;
uint8_t  my_ipv6[16];
struct cli_addr  *my_ipaddr;

if (CLI_DEFINED(ctx, 'a'))    /* -a <ip address> */
{
   my_ipaddr = (struct cli_addr *)(CLI_VALUE(ctx, 'a'));
   if (my_ipaddr->type == CLI_IPV4)
   {
      ip_addr aval = *((ip_addr *)my_ipaddr->addr[0]);
      my_ipv4 = ntohl(aval);     /* convert to host byte-order */
   }
   else if (my_ipaddr->type == CLI_IPV6)
      MEMCPY(&my_ipv6[0], &my_ipaddr->addr[0], 16);
}
```

Any output processing that is performed within the command's execution function should use the `gio_printf()` function to format the output. If the variable "`ctx`" is of type `CLI_CTX`, then "`ctx->gio`" can be used as the GIO parameter for `gio_printf()`.

When the command's execution function has completed its processing, it returns an error code indicating the success or failure of the command. This error code is returned to the caller of `cli_command()`.

18 Virtual File System - VFS

18.1 VFS API Overview

The VFS exports an API that is approximately in conformance to a subset of the ISO 9899: 1990 ("ISO C") buffered file I/O API that is characterized by the functions fopen(), fclose(), fread(), fwrite(), etc. The functions which constitute the VFS API are listed below:

- vfopen
- vfclose
- vfread
- vfwrite
- vfseek
- vftell
- vgetc
- vferror
- vunlink
- vclearerr

With the exceptions of their function names (prefixed with a 'v') and the use of the `VFILE` (instead of a `FILE`) parameter, the calling syntax and semantics of a given VFS function is approximately the same as those of the correspondingly named standard C library function. There are small differences between the VFS API and the standard which are described later in this chapter.

18.2 VFS Implementation

The VFS is a flat (non-hierarchical) file system in which the set of files that exist in the file system is stored in target system memory and is implemented as a singly linked list of structures in which each structure has associated with it a buffer that is used to contain the associated file's contents. Part of this list can be contained as part of the target system executable. The InterNiche Web Server and VFS Compiler use this feature to link the files that contain Web server content with the target system executable. The set of files contained in the list and their contents can be modified at run time to allow this set of files to be updated dynamically.

The VFS supports a system dependent backing store that can be used to allow these dynamically created files to be stored to whatever non-volatile storage (typically solid state devices like FLASH EEPROM) that is provided by the target system. The VFS reads the backing store during system initialization in order to reconstruct the file system in memory. Applications can then open, read, write, and close files in the VFS using the VFS API. Data that is read from a `VFILE` is read from normal read/write system memory, like SRAM or DRAM. Data that is written to a `VFILE` is written to normal read/write memory.

Typically (although this behavior is configurable by the porting engineer) the entire subset of the VFS that has been marked as non-volatile is written from the volatile system memory (e. g. SRAM or DRAM) to the non-volatile backing store whenever any file to which modifications have been made is closed. This

behavior allows for simple implementations of backing store drivers for devices like FLASH which by their nature can make it complicated to implement random access writes.

18.3 Source Files that Constitute the VFS

vfsfiles.h	vfsfiles.h should be included by source files that intend to use the VFS API. It contains the definitions of data structures used by the VFS, prototypes of the functions that constitute the VFS API and various defined constants.
vfsfiles.c	vfsfiles.c contains the bulk of the implementation of the VFS API.
vfsutil.c	vfsutil.c contains functions which implement a user interface that allows access to and control of the VFS.
vfssync.c	vfssync.c contains the implementations of functions which write to and read from the VFS backing store .
vf_nt.c	vfs_nt.c contains the VFS menus used by the CLI and their underlying support functions.

18.4 VFS Configuration Options

The VFS provides several configuration options to allow the porting engineer to customize the VFS behavior for a particular target system. The options are described below.

VFS_FILES

The presence of the defined constant `VFS_FILES` enables the VFS described in this chapter. If `VFS_FILES` is not defined, the inclusion of vfsfiles.h causes the VFS API entry points to be defined to be equal to their standard C library equivalents, as in:

```
#define vfopen(n,m)   fopen(n,m)
#define vfclose(fd)   close(fd)
```

InterNiche applications which need access to a file system, like the Web and FTP Servers, perform file system access via the VFS API. By undefining `VFS_FILES`, the porting engineer can cause these applications to access the standard C library buffered I/O API that is provided by their target system's compiler package.

Example usage:

```
#define VFS_FILES 1
```

HT_RWVFS

HT_RWVFS defines whether the VFS is write enabled. The presence of this defined constant causes code that enables write access to the VFS to be included in the target system executable. If HT_RWVFS is not defined, files can be opened, read from, and closed via the VFS, but calls which would cause the VFS to be modified, such as vfwrite() will not be operational.

Example usage:

```
#define HT_RWVFS 1
```

HT_EXTDEV

The presence of the defined constant HT_EXTDEV causes the VFS to make calls to "external file systems". External file systems are described in more detail in the section "External File Systems".

HT_LOCALFS

The presence of the defined constant HT_LOCALFS causes the VFS API functions to make calls to their analogous standard C library buffered I/O function under certain circumstances. This is described in the section "Local File Systems".

FILENAMEMAX

The defined constant FILENAMEMAX defines the maximum length of a VFS file name. The default value of FILENAMEMAX is 16, but the porting engineer can modify this value if 16 characters is not an appropriate length for file names on the target system.

Example usage:

```
#define FILENAMEMAX 50
```

VFS_MAX_TOTAL_RW_SPACE

The defined constant VFS_MAX_TOTAL_RW_SPACE defines an upper limit on the amount of memory that the VFS will allocate for use in buffers for the containment of VFS file contents. This allows the porting engineer to limit the amount target system memory that the VFS will consume.

Example usage:

```
#define VFS_MAX_TOTAL_RW_SPACE   100000
```

With the above definition, attempts to write to a VFILE which requires more than 100 kilobytes of system memory to be allocated to contain the file contents will fail.

VFS_MAX_DYNA_FILES

The defined constant `VFS_MAX_DYNA_FILES` defines an upper limit on the number of files that the VFS will create dynamically. As with `VFS_MAX_TOTAL_RW_SPACE`, it is a tool that the porting engineer can use to limit the amount of memory that is consumed by the VFS.

Example usage:

```
#define VFS_MAX_DYNA_FILES 100
```

With the above definition, attempts to create more than 100 files on the target system will fail.

VFS_MAX_OPEN_FILES

The defined constant `VFS_MAX_OPEN_FILES` defines an upper limit on the number of files that the VFS will allow to be simultaneously open. It is another tool to limit VFS memory usage.

Example usage:

```
#define VFS_MAX_OPEN_FILES 5
```

With the above definition if five files have been opened and not closed, the next attempt to open a `VFILE` will fail.

18.5 Detailed Description of VFS API

In the following API description, the term "current file pointer" or `CFP` means the relative byte offset from the beginning of the file from which reads will be made and to which writes will be made.

vclearerr

API Name

`()`

Syntax

```
void vclearerr(VFILE *vfd);
```

Description

vclearerr() clears the error condition returned by vferror().

Returns

Nothing.

vfclose

API Name

`vfclose()`

Syntax

```
void vfclose(VFILE * vfd);
```

Description

Files that are opened with `vfopen()` should eventually be closed with vfclose(). Depending on how the VFS has been configured and whether any changes to the file have been made since it was opened, a call to vfclose() can cause the function `vfs_sync()` to be called which allows for the RAM resident VFS to be stored to the target system's backing store.

Returns

Nothing

vferror

API Name

`vferror()`

Syntax

```
int vferror(VFILE * vfd);
```

Returns

vferror() returns an error code describing what went wrong on the last attempt to write to the file.

vfopen

API Name

vfopen()

Syntax

```
VFILE* vfopen(char * name, char * mode);
```

Description

The calling semantics of vfopen() are similar to that of the standard C library fopen(). The name parameter points to a null terminated string that defines the name of the file to be opened. The first character of the string addressed by the mode parameter defines what actions are to be taken when opening the file, as shown below:

mode [0] == 'r'	If the named file does not exist, fail the open. If the file does exist, open the file and position the CFP to the beginning of the file.
mode [0] == 'w'	If the named file does not exist, create a file of 0 length with the given name and open it. If the named file does exist, truncate it to a length of 0 and open it. In both cases position the CFP to the beginning of the file.
mode [0] == 'a'	If the named file does not exist, create a file of 0 length with the given name and open it. If the named file does exist, open it without modifying its existing contents. In both cases position the CFP to the end of the file.

Returns

When vfopen() is successful, it returns a handle which is a pointer to the type VFILE. This handle should be passed to subsequent VFS functions which require a VFILE parameter to access the file's contents. When vfopen() is not successful it returns NULL and the reason for the error can be retrieved by calling the function get_vfopen_error()..

Notes

`vfopen()` differs from the Standard `fopen()` call in the following ways:

- Only the first character of the mode parameter is significant. The '`b`' and '`+`' suffixes that have special meaning in some `fopen()` implementations have no meaning to `vfopen()`. This means that the "open for read access only" semantic of the 'r' parameter that is present in `fopen()` does not apply. Writes to a file that is `vfopen()`'ed with mode '`r`' will not automatically fail like they do on some systems. In that sense '`r`' with `vfopen()` is more like '`r+`' on most system's `fopen()`. It also means that the 'ASCII' mode of file opening in which newline conversion is done in the API is not performed with the VFS. All reads and writes are strictly binary.
- The VFS supports only one current file pointer per `VFILE`. Some buffered I/O systems will do reads from the "current file pointer" which is settable with `fseek()` but will only allow writes to the end of the file (as weird a "standard" behavior as one can imagine). With the VFS, reads and writes are always initiated from the `CFP`.
- The VFS imposes no requirements on file names other than that they are not to exceed `FILENAMEMAX` characters in length. Embedded spaces and punctuation characters are legal, as are ASCII characters with the most significant bit set. A file name of 0 length is legal. Slash (forward slash), '/', and backslash, '\', have no special meaning. The one exception to this is that if a file name begins with a slash, '`/`', it will be removed from the file name before the file is created. Thus the file names /foo and foo refer to the same file.

Copyright HCC Embedded 2017

www.hcc-embedded.com

vfread

API Name

`vfread()`

Syntax

```
int vfread(char * buf, unsigned size, unsigned items, VFILE * vfd);
```

Description

The calling semantics of `vfread()` are similar to that of the standard `fread()`. An attempt to read the product of items times size bytes from the `CFP` of the `VFILE` addressed by the vfd parameter into the caller supplied buffer addressed by the buf parameter is made. If at least that many bytes are available in the file starting at the `CFP`, the call succeeds and returns items to the caller. If less than that many bytes are available, as much as is available is copied to the caller's buffer and the number of bytes copied divided by size is returned to the caller. This is an integer division, which implies that if it is important to know how many bytes were actually read, size should be 1. In all cases the `CFP` is incremented by the number of bytes successfully read.

Returns

The number of items successfully read into the caller's buffer.

vfseek

API Name

`vfseek()`

Syntax

```
long vfseek(VFILE * vfd, long offset, int mode);
```

Description

The calling syntax of vfseek() is similar to that of the standard C library fseek(), however the semantics are quite restricted. vfseek() allows the caller to change the `CFP` of a `VFILE`. The offset parameter must be 0. Two values are accepted for the mode parameter: SEEK_SET and SEEK_END. Thus vfseek() allows the caller to position the `CFP` to either the beginning (SEEK_SET) or the end (SEEK_END) of the file.

Returns

vfseek() returns the value of the modified `CFP` when successful. It returns -1 when unsuccessful. The reasons for failure usually have to do with invalid parameter values.

vftell

API Name

`vftell()`

Syntax

```
long vftell(VFILE * vfd);
```

Returns

For uncompressed files, `vftell()` functions much as the standard `ftell()`. It returns the `CFP` of the specified `VFILE`. For compressed files, `vftell()` returns the uncompressed size of the file if the `CFP` is at the end of the file, else is returns the byte offset into the compressed file image of the current `CFP`. File compression is described in the section, "Internal Data Structures".

vfwrite

API Name

`vfwrite()`

Syntax

```
int vfwrite(char * buf, unsigned size, unsigned items, VFILE * vfd);
```

Description

The calling semantics of `vfwrite()` are similar to that of the standard fwrite(). An attempt to write the product of items times size bytes from the caller's buffer addressed by the buf parameter to the `CFP` of the `VFILE` addressed by the vfd parameter is made. When successful, the `CFP` is incremented by the number of bytes written.

Returns

Because of its implementation, calls to `vfwrite()` either succeed completely and return items, or fail completely and return 0 to indicate that the file's contents were not modified. There is a possible exception to this when an external or local file system is used. The reason for the failure can be determined via a call to the vferror() function. The set of errors includes:

ENP_LOGIC	An attempt was made to do a write to a VFS in which write access is not enabled (`HT_RWVFS` is not defined).
ENP_FILEIO	An attempt was made to do a write to a VFS file that is write protected. Write protection of individual files is described later.
ENP_NOMEM	There was insufficient memory available to store the added file contents.

vgetc

API Name

`vgetc()`

Syntax

```
int vgetc(VFILE * vfd);
```

Returns

vgetc() returns the value of the byte at the current `CFP` and increments the `CFP`. It returns EOF (-1) when the end of the file is reached.

vunlink

API Name

`vunlink()`

Syntax

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

Description

`vunlink()` deletes the named file from the set of files maintained by the VFS. Depending on how the VFS has been configured, a call to vunlink() can cause the function `vfs_sync()` to be called which allows for the RAM resident VFS to be stored to the target system's backing store.

Returns

`0` if the file was successfully deleted, `-1` otherwise.

The reasons for failure are:

- The named file does not exist in the VFS.
- The named file exists but was not marked as writable.

`vunlink()` modifies the parameter in the same manner as does `vfopen()`.

18.6 Internal Data Structures

This section describes important data structures that are used by the VFS.

vfs_file Structure

```
struct vfs_file {
   struct vfs_file *next;
   char name[FILENAMEMAX + 1];   /* name of file under "path" */
   unsigned short flags;
   unsigned char * data;      /* pointer to file data, NULL if none */
   unsigned long real_size;   /* size in bytes of file before compression */
   unsigned long comp_size;   /* size in bytes of file compressed */
   unsigned long buf_size;    /* size in bytes of memory buffer used to store file */
#ifdef WEBSERVER
   /* routine to call if file is treated as CGI executable */
   int (*cgi_func)(struct httpd *, struct httpform *, char ** text);
#endif
#ifdef HT_EXTDEV
   void * method;          /* pointer depends on flags */
#endif
};
```

Each file in the VFS is represented by an instance of a `vfs_file` structure. These structures are linked together in a list using the next field. The head of the list is stored in the global:

```
struct vfs_file *vfsfiles;
```

The `name` field contains the name of the file. The `flags` field is a field of bits that describes various attributes of the file. The `flags` field is described in more detail in the section "Bits of the Flags ield".

The `data` field points to a buffer that contains the contents of the file. When a file is newly created, the `data` field contains NULL. A buffer is allocated and assigned to the `data` field when the first write is made to the file. As the file grows in size and exceeds the size of the allocated buffer, a new buffer is allocated to replace the old buffer, the file contents in the old buffer are copied to the new buffer and the old buffer is freed. This has an implication for the memory requirements of the VFS. When a large file is written to such that the write exceeds the size of the file, there is a short period between the time when the new buffer is allocated and old buffer is freed when there must be sufficient memory available to store effectively twice the size of the contents of the file. Porting engineers should keep this in mind when determining the memory requirements for a target system's VFS.

The `real_size` field contains the size of a compressed file before it was compressed. This information is used by the InterNiche Web Server. The `comp_size` field contains the size of actual file image. The `buf_size` field contains the size of the buffer addressed by the data field.

The `cgi_func` field is used by the InterNiche Web server. The `method` field is used in conjunction with external file systems, which are described later in this chapter.

Bits of the Flags Field

The following defined constants identify the bits of significance in the `flags` field of the `vfs_file` structure. For purposes of this document they are divided into two groups.

The following six bits are of significance only to the InterNiche Web Server.

```
VF_AUTHBASIC         0x02  /* check Basic user auth */
VF_AUTHMD5           0x04  /* check MD5 user auth */
VF_MAPFILE           0x08  /* data pointer is a (struct mapfile*) */
VF_CVAR              0x10  /* its C variable display */
VF_PUSH             0x400  /* its a web server "push" file */
VF_NOCACHE          0x800  /* File cannot be cached by any device in
                            the request/response chain  */
```

The remainder of the bits, shown below, are significant to the VFS itself.

```
VF_HTMLCOMPRESSED   0x001  /* If HTML, file is tag-compressed */
VF_WRITE            0x020  /* writable device */
VF_DYNAMICINFO      0x040  /* file info created dynamically */
VF_DYNAMICDATA      0x080  /* file data created dynamically */
VF_NONVOLATILE      0x100  /* copy to non-volatile storage on close */
VF_STALE            0x200  /* contents of file have been changed */
VF_NODATA          0x8000  /* file has a size, but no actual data */
```

The `VF_HTMLCOMPRESSED` bit indicates that the file image was compressed using the InterNiche VFS Compiler. When this bit is set, the functions which read the VFS, `vgetc()` and `vfread()`, apply a decompression algorithm to the image before returning the file's contents to the caller. When a new file is created with `vfopen()` and whenever a file is written to with `vfwrite()`, the `VF_HTMLCOMPRESSED` bit is reset. The bit can be set again after the file is closed using the user interface commands described later. The intent here is to allow a file to be compressed at one location, perhaps a central site or development center and uploaded to a target system using the InterNiche FTP server. Once the FTP transfer has been completed, the target system's user interface can be used to set the bit so that when the Web server reads the file, it will be decompressed. This allows target systems' memory resources to be minimized by taking advantage of the decompression while not incurring the code overhead on the target system that would be required if the compression algorithm was also located on the target system.

The `VF_WRITE` bit indicates that the file can be written to with `vfwrite()` and deleted with `vunlink()`. Files that are created dynamically with `vfopen()` get their `VF_WRITE` bits set. Files that are linked with the target system executable via the VFS Compiler may or may not have their `VF_WRITE` bits set depending on the requirements of the target system. The user interface allows the `VF_WRITE` bits to be set or reset.

The `VF_DYNAMICDATA` and `VF_DYNAMICINFO` bits are used internally by the VFS to track whether the data buffer associated with the `vfs_file` structure and the `vfs_file` structure itself were allocated dynamically.

The VF_NONVOLATILE bit indicates whether the file should be stored to the backing store or not. The VF_STALE bit is used to determine whether the file's contents have changed since it was opened. This enables the vfclose() function to determine whether vfs_sync() should be called.

The VF_NODATA bit indicates that the file has a size, but no actual data. This is useful for creating large test files that would otherwise exceed the memory available to the VFS.

vfs_open Structure

```
struct vfs_open {
   struct vfs_open * next;
   struct vfs_file * file;
   unsigned char * cmploc;    /* current position in data buf */
   unsigned char * tag;       /* current position in compressed tag, if any
   int error;                 /* last error, if any */
};
typedef struct vfs_open VFILE;
```

When a file is opened with vfopen(), an instance of a vfs_open structure is allocated. The address of the structure is what is returned as the file handle to the caller.

vfs_open structures are stored in a singly linked list using the structures' next field. The list is headed by the global:

```
VFILE *vfiles;
```

The file field of the structure points to the vfs_file structure that is associated with the opened file. The cmploc field points into the buffer addressed by the vfs_file structure's data field. It is the cmploc field that effectively implements the file's CFP. The tag field is used by the decompression algorithm to decompress compressed files. It is unused with regular, uncompressed files. The error field is used to store the error that is returned by vferror().

18.7 Porting Engineer Provided VFS Functions

The following constructs should be provided by the porting engineer when porting the VFS to a given target system. InterNiche provides direct support for some target systems. If the target system to which a port is to be made is similar to one of these supported target systems, the code located in the target system dependent directory can be used as a starting point.

VFS_VFS_FILE_ALLOC

`VFS_VFS_FILE_ALLOC()` should return a pointer to a zeroed block of memory into which the VFS will store the contents of a `vfs_file` structure, thus the block of memory returned by `VFS_VFS_FILE_ALLOC()` should be at least as large as a `vfs_file` structure. When memory is unavailable `VFS_VFS_FILE_ALLOC()` should return NULL.

For many target systems, the following defined constant implementation of `VFS_VFS_FILE_ALLOC()` which uses the InterNiche `npalloc()` function will work fine:

```
#define VFS_VFS_FILE_ALLOC() (struct vfs_file *) npalloc(sizeof(struct vfs_file)
```

VFS_VFS_FILE_FREE

The VFS will call `VFS_VFS_FILE_FREE()` when it no longer needs a buffer that had previously been allocated with `VFS_VFS_FILE_ALLOC()`. `VFS_VFS_FILE_FREE()` takes a single parameter which is the address of the buffer to be freed.

For many target systems, the following defined constant implementation of `VFS_VFS_FILE_FREE()` which uses the InterNiche `npfree()` function will work fine:

```
#define VFS_VFS_FILE_FREE(x) npfree(x)
```

VFS_VFS_OPEN_ALLOC

`VFS_VFS_OPEN_ALLOC()` should return a pointer to a zeroed block of memory into which the VFS will store the contents of a `vfs_open` structure, thus the block of memory returned by `VFS_VFS_OPEN_ALLOC()` should be at least as large as a `vfs_open` structure. When memory is unavailable `VFS_VFS_OPEN_ALLOC()` should return NULL.

For many target systems, the following defined constant implementation of `VFS_VFS_OPEN_ALLOC()` which uses the InterNiche `npalloc()` function will work fine:

```
#define VFS_VFS_OPEN_ALLOC() (struct vfs_open *) npalloc(sizeof(struct vfs_open)
```

VFS_VFS_OPEN_FREE

The VFS will call `VFS_VFS_OPEN_FREE()` when it no longer needs a buffer that had previously been allocated with `VFS_VFS_OPEN_ALLOC()`. `VFS_VFS_OPEN_FREE()` takes a single parameter which is the address of the buffer to be freed.

For many target systems, the following defined constant implementation of `VFS_VFS_OPEN_FREE()` which uses the InterNiche `npfree()` function will work fine:

```
#define VFS_VFS_OPEN_FREE(x) npfree(x)
```

vfs_lock() and vfs_unlock()

The VFS makes use of two singly linked lists to keep track of allocated data structures: the list of `vfs_file` structures headed by the global `vfsfiles` and the list of `vfs_open` structures headed by the global `vfiles`. Because some VFS API functions make additions to and deletions from these lists, while others simply traverse them, it is important to serialize access to these lists in order to prevent data corruption on systems in which it is possible for one process or task that is accessing the VFS to pre-empt another. `vfs_lock()` and `vfs_unlock()` are provided for this purpose.

Each VFS API function makes a call to `vfs_lock()` before it accesses the internal VFS data structures. Each of these functions call `vfs_unlock()` before returning to the caller. On target systems in which it is possible for task preemption to occur, the porting engineer should provide an implementation of `vfs_lock()` that blocks on the acquisition of an RTOS semaphore or mutex before returning to the caller and an implementation of `vfs_unlock()` that releases the semaphore or mutex. On superloop based systems (systems without an operating system in which only one task or process executes) or multitasking systems in which task preemption cannot occur, the implementations of these function can safely be no-ops. The functions take no parameters and return nothing to the caller.

One could implement `vfs_lock()` as a function that disabled interrupts, with `vfs_unlock()` re-enabling them, though the porting engineer should understand that there has been no attempt to make the VFS "real time" and the interrupt latency that would be introduced by such an implementation could be quite long. It has been assumed in its implementation that the VFS API will not be called from interrupt service routines. There is nothing to prevent it from functioning from ISR context, but again the interrupt latency involved would make such an approach unsuitable for most applications.

18.8 VFS NicheTool Commands

The VFS includes a user interface that presents several commands that are useful for viewing and manipulating the VFS. These commands are described below.

vfs attribute

Command Name

`attribute - set or clear VFS file attributes`

Syntax

`attribute -f F [-c C] [-s S]`

Parameters

`-f F`	file name
`-c C`	flags to clear: String of unseparated characters, e.g., "SWN"
`-s S`	flags to set: String of unseparated characters, e.g., "SWN"

Description

Set or clear VFS file attributes.

Notes/Status

The following characters represent flags that can be used to set or clear the attributes of a VFS file:

`H`	File has been HTML compressed
`B`	Access to file requires BASIC authentication
`S`	Access to file requires MD5 authentication
`M`	File is a MAPFILE
`W`	File is writable
`N`	File data should be copied to non-volatile storage on close

Location

This command is provided by the `VFS` module when `USE_VFS`, `VFS_MENUS`, and `VFS_RWFILES` are defined.

vfs delete

Command Name

`vfs delete - Delete a VFS file`

Syntax

`vfs delete -f F`

Parameters

`-f F`	String: File name

Description

This command is used to delete a VFS File.

Notes/Status

- The `-f` (filename) option is required

Location

This command is provided by the `VFS` module when `USE_VFS`, `VFS_MENUS`, and `VFS_RWFILES` are defined.

vfs directory

Command Name

`directory - display a directory listing`

Syntax

`directory [ -o ]`

Parameters

`-o`	Only list files that are open

Description

Lists the files in the current directory. If `'-o'` is present, only list the files that are currently open.

Notes/Status

- This output of this command is dependent upon the underlying file system implementation.

Location

This command is provided by the `VFS` module when `VFS_FILES` is defined.

vfs read

Command Name

`vfs read - Copy a VFS file to the output device`

Syntax

`read -f F`

Parameters

`-f F`	String: File name

Description

This command is used to copy a VFS file to the output device.

Notes/Status

- The `-f` (filename) option is required

Location

This command is provided by the `VFS` module when `USE_VFS` and `VFS_MENUS` are defined.

18.9 Local File Systems

When the constant HT_LOCALFS is defined, the functions which make up the VFS API will make their analogous standard C library buffered I/O function calls under certain circumstances. This behavior can be useful on some target systems to allow files to be accessed in the VFS and the local file system provided by the target system's C library.

The circumstances under which a VFS function will call its analogous standard C library function are described below.

vfopen() calls fopen()	When the name of the file passed in begins with a porting engineer provided prefix that is defined by the constant VFS_NATIVE_PREFIX. The default value of VFS_NATIVE_PREFIX is: `#define VFS_NATIVE_PREFIX "\\disk\\"`
vfopen() calls fopen()	When vfopen() is called with file name that does not exist in the VFS and HT_RWVFS is not defined. Therefore the VFS is not configured to allow files to be created dynamically.
vfopen() calls fopen()	When vfopen() is called with a file name that does not exist in the VFS but the mode parameter begins with 'r', indicating that the named file must exist for the call to succeed.
vunlink() calls unlink()	When vunlink() is called with a file name that does not exist in the VFS.
vfread() calls vfread()	
vfwrite() calls fwrite()	
vfseek() calls fseek()	
vftell() calls ftell()	
vgetc() calls getc()	
vferror() calls ferror()	
vclearerr() calls clearerr()	
vfclose() calls fclose()	When the passed in VFILE descriptor is not in the list of open files that is maintained by the VFS.

18.10 External File Systems

When the constant HT_EXTDEV is defined, the VFS will make calls to an "external file system" under some circumstances. The porting engineer can define an external file system using the following structure:

```
struct vfroutines {
   struct vfroutines * next;  /* keep these in a list */
   VFILE* (* fopen)(char * name, char * mode);
   void   (* fclose)(VFILE * vfd);
   int    (* fread)(char * buf, unsigned size, unsigned items, VFILE * vfd);
   int    (* fwrite)(char * buf, unsigned size, unsigned items, VFILE * vfd);
   long   (* fseek)(VFILE * vfd, long offset, int mode);
   long   (* ftell)(VFILE * vfd);
   int    (* fgetc)(VFILE * vfd);
   int    (* unlink)(char*);
};
extern struct vfroutines * vfsystems;
```

The vfroutines structure is used to define a set of entry points into an external file system. To define an external file system, the porting engineer should allocate an instance of a vfroutines structure, initialize its various fields with entry points into the code that implements the external file system and link the structure to the list of structures headed by the global vfsystems.

When vfopen() is called with a function name that does not exist in the list of vfs_file structures, vfopen() will traverse the list of vfroutines structures headed by vfsystems and call the function addressed by the fopen() field of each structure. This is the opportunity for the external file system to claim ownership of the file. If the file name "belongs" in the external file system, the fopen() function of the external file system should allocate a vfs_file structure for the file and set its method field to address the file system's vfroutines structure, link the vfs_file structure into the list of vfs_file structures headed by vfsfiles, allocate a vfs_open structure, set its file field to point to the allocated vfs_file structure, add the vfs_open structure to the list of such structures headed by vfiles and return a pointer to the allocated vfs_open structure. It should also do whatever initialization is necessary to create and maintain a file in the external file system. If the file name does not "belong" to the external file system, the external file system's fopen() function should return NULL. Note that for external file systems, the vfs_open_files variable must be incremented. It will be decremented by vfclose().

Subsequent to this, whenever a VFILE handle is passed to one of the VFS entry points, the method field of the associated VFILE file field is inspected and when this method field is found to be non-NULL, the appropriate vfroutine field function pointer is called to handle the API request. For example, assuming that a vfopen() of a given file caused the external file system to claim the file and allocated its own VFILE to represent the open file instance, a subsequent call to vfread() with that VFILE handle would result in vfread() calling the fread() entry point of the external file system because the method field of the vfs_file structure addressed by the VFILE file field would point to the file system's vfroutine structure.

18.11 VFSCOMP - External Utility for VFS Configuration

The VFScompiler is an external utility, designed to greatly facilitate the integration of the files required by your target's application into the VFS file system. Its output is a set of `.c` and `.h` files ready for direct use by your application. The utility can be used to:

- Add X.509 certificates and other data files to an InterNiche-based system;
- Translate binary data into "C" code (e.g. graphical files served by the HTTP Server);
- Import HTML and Web Script files for use by the HTTP Server;
- Compress HTML files for reduced ROM usage;
- Create direct references to functions and internal variables for access by the HTTP Server script files.

VFSComp - VFS Filesystem Compiler

Syntax

```
VFSComp [-options]
```

Options

These options apply to all files and so must be set on the command line

-arr <arrayname>	File system array name. Default = "<basename> files"
-base <basename>	Base name for the following fields. Default = "vfs"
-i <infile>	Input filename. Default = "<basename>.txt"
-html	Normal Webserver mode. The equivalent to setting the following options on the command line: `-c -arr webfiles -i webfiles.txt -o htmldata`
-nvfs	Do not generate vfiles structure
-ss	Sort tags in cmprreport.txt by bytes saved. Default is to sort alphabetically
-v	Toggle verbose mode
-z const	Generate code using `Const` keyword
-D	Emit dependency information for make

These options may appear either on the command line or within `<infile>.txt`

-b	Require Basic Authentication (user name/password) for access to this file
-c	Do file compression
-cs	File is case sensitive
-d	Require Digest Authentication (MD5) for access to this file
-o <outfile>	Put output for this file into the specified file

These per-file options are only valid within `<infile>.txt`

-cgi <funcname>	Generate a CGI function stub for this symbol

`-cvar type nat TOKEN`	Generate full mapping of C variable in `htmldata.c`. See Displaying C Variables in Web Pages section of the HTTP and Web Servers Technical Reference
`-uhdr`	Call `ht_adduserheaders()` to add additional user-specified HTTP headers. The user must add code to this stub in `democgi.c`

Description

The VFS compiler is a developer tool used to prepare and insert a list of files into the VFS filesystem. These files may be web pages, images, private key files, configuration, or any other file that will not grow in size over the product lifetime. To use the resulting VFS files, the output `.c` file should be added to the build, and a call to the `XXX_setup()` routine should be called before first use.

The `-cgi` and `-cvar` options are primarily used in relation to web pages for the HTTP server. They provide code generation features to facilitate the calling of CGI functions and the display of certain variable values within the code.

Notes/Status

- The VFS Compiler replaces the HTML Compiler

VFS Compiler Use for Web Pages

Use of the VFS Compiler for HTTP Server applications is discussed in detail in the *HTTP and Web Servers Technical Reference Manual*.