QNX® Neutrino® RTOS

Core Networking with io-pkt

User's Guide

For QNX® Neutrino® 6.3.2 or later

Published under license by:

QNX Software Systems International Corporation
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: `info@qnx.com`
Web: `http://www.qnx.com/`

Electronic edition published 2009

Contents

List of Figures

About This Guide

What you'll find in this guide

This guide introduces you to the QNX Neutrino Core Networking stack and its manager, **`io-pkt`**.

The following table may help you find information quickly in this guide:

For information on:	Go to:
`io-pkt` and its architecture	Overview
Examining and modifying packets, and creating a firewall	Packet Filtering
Setting up secure connections	IP Security and Hardware Encryption
802.11 a/b/g (WiFi)	WiFi Configuration Using WPA and WEP
`io-pkt` and Qnet	Transparent Distributed Processing
Support for different types of network drivers	Network Drivers
Related utilities, etc.	Utilities, Managers, and Configuration Files
`io-pkt` and **`io-net`**	Migrating from **`io-net`**
Terms used in this document	Glossary

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL )`
Command options	**`-lR`**
Commands	**`make`**
Environment variables	**PATH**
File and pathnames	**`/dev/null`**
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete

continued...

Reference	**Example**
Keyboard input	`something you type`
Keyboard keys	Enter
Program output	`login:`
Programming constants	NULL
Programming data types	`unsigned short`
Programming literals	`0xFF`, `"message string"`
Variable names	*stdin*
User-interface components	**Cancel**

We use an arrow (→) in directions for accessing menu items, like this:

> You'll find the **Other...** menu item under **Perspective→Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

WARNING: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support

To obtain technical support for any QNX product, visit the **Support + Services** area on our website (**www.qnx.com**). You'll find a wide range of support options, including community forums.

Chapter 1

Overview

In this chapter. . .

What's new in the networking stack?

The QNX Neutrino networking stack is called **io-pkt**. It replaces the previous generation of the stack, **io-net**, and provides the following benefits:

- performance improvements. Since **io-pkt** removes the **npkt**-to-**mbuf** translation and mandatory queuing, and also reduces context switching on the packet receive path, the IP receive performance is greatly improved.
- simplified locking of shared resources, resulting in simpler SMP support
- it closely follows the NetBSD code base and architecture, meaning:
  - easier maintenance / upgrade capability of IP stack source
  - existing applications that use BSD standard APIs will port more easily (e.g. **tcpdump**)
  - enhanced features included with NetBSD stack are also included with **io-pkt**
  - NetBSD drivers will port in a straightforward manner
- far richer stack feature set, drawing on the latest in improvements from the NetBSD code base
- 802.11 Wi-Fi client and access point capability

The **io-pkt** manager is intended to be a drop-in replacement for **io-net** for those people who are dealing with the stack from an outside application point of view. It includes stack variants, associated utilities, protocols, libraries and drivers.

The stack variants are:

io-pkt-v4
: IPv4 version of the stack with no encryption or Wi-Fi capability built in. This is a "reduced footprint" version of the stack that doesn't support the following:
  - IPv6
  - Crypto / IPSec
  - 802.11 a/b/g WiFi
  - Bridging
  - GRE / GRF
  - Multicast routing
  - Multipoint PPP

io-pkt-v4-hc
: IPv4 version of the stack that has full encryption and Wi-Fi capability built in and includes hardware-accelerated cryptography capability (Fast IPsec).

io-pkt-v6-hc
: IPv6 version of the stack (includes IPv4 as part of v6) that has full encryption and Wi-Fi capability, also with hardware-accelerated cryptography.

In this guide, we use "**io-pkt**" to refer to *all* the stack variants. When you start the stack, use the appropriate variant (**io-pkt** isn't a symbolic link to any of them).

We've designed **io-pkt** to follow as closely as possible the NetBSD networking stack code base and architecture. This provides an optimal path between the IP protocol and drivers, tightly integrating the IP layer with the rest of the stack.

The **io-pkt** stack isn't backward-compatible with **io-net**. However, both can exist on the same system. For more information, see the Migrating from **io-net** appendix in this guide.

The **io-pkt** implementation makes significant changes to the QNX Neutrino stack architecture, including the following:

- **io-net** is replaced by the stack's link layer
- **mbuf**s are used throughout, including in the drivers
- all buffer management is handled by the stack
- **mount** and **umount** capabilities:
  - Only **io-net** drivers may be both mounted and unmounted. Other drivers may allow you to detach the driver from the stack, by using the **ifconfig** *iface* **destroy** command (if the driver supports it).
  - The IP stack is an integral part of **io-pkt**; you can't start **io-pkt** without it. This means that you don't need to specify the **-ptcpip** option to the stack unless there are additional parameters (e.g. **prefix=**) that you need to pass to it. If you specify the **-ptcpip** option without additional parameters, **io-pkt** accepts it with no effect.
  - Protocols and enhanced stack functionality (e.g. TDP, NAT / IP Filtering) can be mounted, but not unmounted.
- The concepts of producers and consumers no longer exist within **io-pkt**. Filters still exist, but they use a different API than with **io-net**. The **io-pkt** stack does provide other hooks into the stack that you can use to provide a similar level of functionality. These include:

  Berkeley Packet Filter interface

  : Lets you read and write, but not modify or discard, both IP and Ethernet packets from your application.

  Packet Filter interface

  : **pfil** hooks, enabled when the PF filter module is loaded, let you read, write, and modify IP and Ethernet packets within the context of the stack process.
- Driver changes:

- The driver model has changed to provide better integration with the protocol stack. (For example, in **io-net**, **npkt**s had to be converted into **mbuf**s for use with the stack. In **io-pkt**, **mbuf**s are used throughout.)
- The driver API and behavior have been changed to closely match those of the NetBSD stack, allowing NetBSD drivers to be ported to **io-pkt**.
- A shim layer, **devnp-shim.so**, is provided that lets you use an **io-net** driver as-is.
- By default, driver interfaces are no longer sequentially numbered with **en***x* designations; they're named according to driver type (e.g. **fxp0** is the interface for an Intel 10/100 driver). You can use the **name=** driver option (processed by **io-pkt**) to specify the interface name.
- Drivers no longer present entries in the name space to be directly opened and accessed with a *devctl()* command (e.g. **open(/dev/io-net/en0)**). Instead, a socket file descriptor is opened and queried for interface information. The *ioctl()* command is then sent to the stack using this device information (see the source to **nicinfo** for an example of how this works).

- IP Filtering and NAT are now handled through the PF interface with **pfctl**. This replaces the **io-net ipf** interface, which is no longer supported.
- SCTP isn't supported in the initial release of **io-pkt**.
- In **io-net**, loopback-checksumming was done with **ifconfig**. In **io-pkt** this is controlled via **sysctl**:

```
# sysctl -a | grep do_loopback_cksum
net.inet.ip.do_loopback_cksum = 0
net.inet.tcp.do_loopback_cksum = 0
net.inet.udp.do_loopback_cksum = 0
```

- The **nicinfo** utility operates slightly differently from the way it did under **io-net**:
  - The default operation with no arguments is to list information on all interfaces. The behavior with **io-net** was to show stats for **/dev/io-net/en0**.
  - Under **io-net**, all Ethernet interface names were in the form **en***X*. Under **io-pkt**, this name will vary, but you can use the **name=** driver option (processed by **io-pkt**) to override this.
  - Ported NetBSD drivers might not support the **nicinfo** *ioctl()* command.

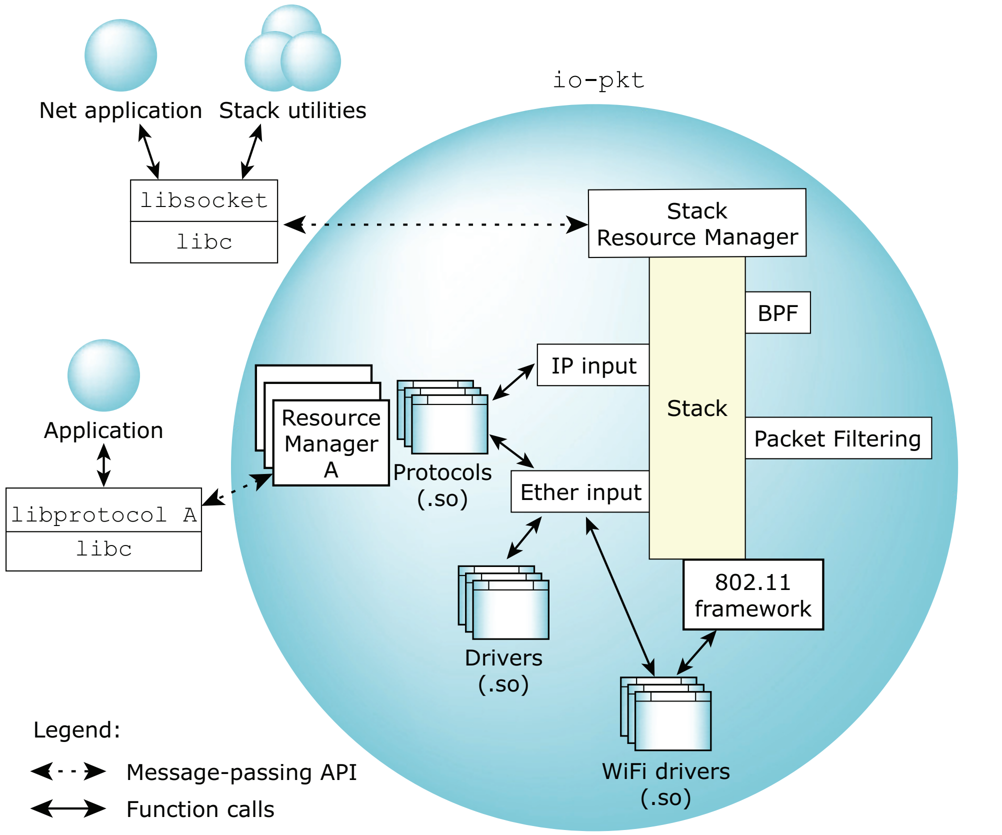
Architecture of io-pkt

The **io-pkt** stack is very similar in architecture to other component subsystems inside of the Neutrino operating system. At the bottom layer are drivers that provide the mechanism for passing data to, and receiving data from, the hardware. The drivers hook into a multi-threaded layer-2 component (that also provides fast forwarding and bridging capability) that ties them together and provides a unified interface into the layer-3 component, which then handles the individual IP and upper-layer protocol-processing components (TCP and UDP).

In Neutrino, a resource manager forms a layer on top of the stack. The resource manager acts as the message-passing intermediary between the stack and user applications. It provides a standardized type of interface involving *open()*, *read()*, *write()*, and *ioctl()* that uses a message stream to communicate with networking applications. Networking applications written by the user link with the socket library. The socket library converts the message-passing interface exposed by the stack into a standard BSD-style socket layer API, which is the standard for most networking code today.

One of the big differences that you'll see with this stack as compared to **io-net** is that it isn't currently possible to decouple the layer 2 component from the IP stack. This was a trade-off that we made to allow increased performance at the expense of some reduced versatility. We might look at enabling this at some point in the future if there's enough demand.

In addition to the socket-level API, there are also other, programmatic interfaces into the stack that are provided for other protocols or filtering to occur. These interfaces — used directly by Transparent Distributed Processing (TDP, also known as Qnet) — are very different from those provided by **io-net**, so anyone using similar interfaces to these in the past will have to rewrite them for **io-pkt**.



A detailed view of the **io-pkt** *architecture.*

At the driver layer, there are interfaces for Ethernet traffic (used by all Ethernet drivers), and an interface into the stack for 802.11 management frames from wireless drivers. The **hc** variants of the stack also include a separate hardware crypto API that allows the stack to use a crypto offload engine when it's encrypting or decrypting data for secure links. You can load drivers (built as DLLs for dynamic linking and prefixed with **devnp-**) into the stack using the **-d** option to **io-pkt**.

APIs providing connection into the data flow at either the Ethernet or IP layer allow protocols to coexist within the stack process. Protocols (such as Qnet) are also built as DLLs. A protocol links directly into either the IP or Ethernet layer and runs within the stack context. They're prefixed with **lsm** (loadable shared module) and you load them into the stack using the **-p** option. The **tcpip** protocol (**-ptcpip**) is a special option that the stack recognizes, but doesn't link a protocol module for (since the IP stack is already present). You still use the **-ptcpip** option to pass additional parameters to the stack that apply to the IP protocol layer (e.g. **-ptcpip prefix=/alt** to get the IP stack to register **/alt/dev/socket** as the name of its resource manager).

A protocol requiring interaction from an application sitting outside of the stack process may include its own resource manager infrastructure (this is what Qnet does) to allow communication and configuration to occur.

In addition to drivers and protocols, the stack also includes hooks for packet filtering. The main interfaces supported for filtering are:

Berkeley Packet Filter (BPF) interface

: A socket-level interface that lets you read and write, but not modify or block, packets, and that you access by using a socket interface at the application layer (see **`http://en.wikipedia.org/wiki/Berkeley_Packet_Filter`**). This is the interface of choice for basic, raw packet interception and transmission and gives applications outside of the stack process domain access to raw data streams.

Packet Filter (PF) interface

: A read/write/modify/block interface that gives complete control over which packets are received by or transmitted from the upper layers and is more closely related to the **`io-net`** filter API.

For more information, see the Packet Filtering chapter.

Threading model

The default mode of operation is for **`io-pkt`** to create one thread per CPU. The **`io-pkt`** stack is fully multi-threaded at layer 2. However, only one thread may acquire the "stack context" for upper-layer packet processing. If multiple interrupt sources require servicing at the same time, these may be serviced by multiple threads. Only one thread will service a particular interrupt source at any point in time. Typically an interrupt on a network device indicates that there are packets to be received. The same thread that handles the receive processing may later transmit the received packets out another interface. Examples of this are layer-2 bridging and the "ipflow" fastforwarding of IP packets.

The stack uses a thread pool to service events that are generated from other parts of the system. These events include:

- time-outs
- ISR events
- other things generated by the stack or protocol modules

You can use a command-line option to the driver to control the priority of threads that receive packets. Client connection requests are handled in a floating priority mode (i.e. the thread priority matches that of the client application thread accessing the stack resource manager).

Once a thread receives an event, it examines the event type to see if it's a hardware event, stack event, or "other" event:

- If the event is a hardware event, the hardware is serviced and, for a receive packet, the thread determines whether bridging or fast-forwarding is required. If so, the thread performs the appropriate lookup to determine which interface the packet should be queued for, and then takes care of transmitting it, after which it goes back to check and see if the hardware needs to be serviced again.

- If the packet is meant for the local stack, the thread queues the packet on the stack queue. The thread then goes back and continues checking and servicing hardware events until there are no more events.
- Once a thread has completed servicing the hardware, it checks to see if there's currently a stack thread running to service stack events that may have been generated as a result of its actions. If there's no stack thread running, the thread *becomes* the stack thread and loops, processing stack events until there are none remaining. It then returns to the "wait for event" state in the thread pool.

This capability of having a thread change directly from being a hardware-servicing thread to being the stack thread eliminates context switching and greatly improves the receive performance for locally terminated IP flows.

If **io-pkt** runs out of threads, it sends a message to **slogger**, and anything that requires a thread blocks until one becomes available. You can use command-line options to specify the maximum and minimum number of threads for **io-pkt**.

Threading priorities

There are a couple of ways that you can change the priority of the threads responsible for receiving packets from the hardware. You can pass the **rx_prio_pulse** option to the stack to set the default thread priority. For example:

```
io-pkt-v4 -ptcpip rx_pulse_prio=50
```

This makes all the receive threads run at priority 50. The current default for these threads is priority 21.

The second mechanism lets you change the priority on a *per-interface* basis. This is an option passed to the driver and, as such, is supported only if the driver supports it. When the driver registers for its receive interrupt, it can specify a priority for the pulse that is returned from the ISR. This pulse priority is what the thread will use when running. Here's some sample code from the **devn-mpc85xx.so** Ethernet driver:

```
if ((rc = interrupt_entry_init(&mpc85xx->inter_rx, 0, NULL,
    cfg->priority)) != EOK) {
        log(LOG_ERR, "%s(): interrupt_entry_init(rx) failed: %d",
           __FUNCTION__, rc);
        mpc85xx_destroy(mpc85xx, 9);
               return rc;
}
```

Driver-specific thread priorities are assigned on a *per-interface* basis. The stack normally creates one thread per CPU to allow the stack to scale appropriately in terms of performance on an SMP system. Once you use an interface-specific parameter with multiple interfaces, you must get the stack to create one thread per interface in order to have that option picked up and used properly by the stack. This is handled with the **-t** option to the stack.

For example, to have the stack start up and receive packets on one interface at priority 20 and on a second interface at priority 50 on a single-processor system, you would use the following command-line options:

```
io-pkt-v4 -t2 -dmpc85xx syspage=1,priority=20,pci=0 \
-dmpc85xx syspage=1,priority=50,pci=1
```

If you've specified a per-interface priority, and there are more interfaces than threads, the stack sends a warning to **slogger**. If there are insufficient threads present, the per-interface priority is ignored (but the **rx_pulse_prio** option is still honored).

The actual options for setting the priority and selecting an individual card depend on the device driver; see the driver documentation for specific option information.

Legacy **io-net** drivers create their own receive thread, and therefore don't require the **-t** option to be used if they support the **priority** option. These drivers use the **devnp-shim.so** shim driver to allow interoperability with the **io-pkt** stack.

Components of core networking

The **io-pkt** manager is the main component; other core components include:

pfctl, **lsm-pf-v6.so**, **lsm-pf-v4.so**
: IP Filtering and NAT configuration and support.

ifconfig, **netstat**, **sockstat** (see the NetBSD documentation), **sysctl**
: Stack configuration and parameter / information display.

pfctl
: Priority packet queuing on Tx (QoS).

lsm-autoip.so
: Auto-IP interface configuration protocol.

brconfig
: Bridging and STP configuration along with other layer-2 capabilities.

pppd, **pppoed**, **pppoectl**
: PPP support for **io-pkt**, including PPP, PPPOE (client), and Multilink PPP.

devnp-shim.so
: **io-net** binary-compatibility shim layer.

nicinfo
: Driver information display tool (for native and **io-net** drivers).

libsocket.so BSD socket application API into the network stack.

libpcap.so, **tcpdump**
: Low-level packet-capture capability that provides an abstraction layer into the Berkeley Packet Filter interface.

lsm-qnet.so Transparent Distributed Processing protocol for **io-pkt**.

hostapd, **hostapd_cli** (see the NetBSD documentation), **wpa_supplicant**, **wpa_cli**
: Authentication daemons and configuration utilities for wireless access points and clients.

QNX Neutrino Core Networking also includes applications, services, and libraries that interface to the stack through the socket library and are therefore not directly dependent on the Core components. This means that they use the standard BSD socket interfaces (BSD socket API, Routing Socket, **PF_KEY**, raw socket):

libssl.so, **libssl.a**
: SSL suite ported from the source at **http://www.openssl.org**.

libnbdrvr.so BSD porting library. An abstraction layer provided to allow the porting of NetBSD drivers.

libipsec(S).a, **setkey**
: NetBSD IPsec tools.

inetd Updated Internet daemon.

route Updated route-configuration utility.

ping, **ping6** Updated **ping** utilities.

ftp, **ftpd** Enhanced FTP.

Getting the source code

If you want to get the source code for **io-pkt** and other components, go to Foundry27, the community portal for QNX developers (**http://community.qnx.com/sf/sfmain/do/home**).

Chapter 2

Packet Filtering

In this chapter. . .

Packet Filters

In principle, the pseudo-devices involved with packet filtering are as follows:

- **pf** is involved in filtering network traffic
- **bpf** is an interface that captures and accesses raw network traffic.

The **pf** pseudo-device is implemented using **pfil** hooks; **bpf** is implemented as a tap in all the network drivers. We'll discuss them briefly from the point of view of their attachment to the rest of the stack.

Although the NetBSD documenation talks about *ioctl()*, you should use *ioctl_socket()* instead in your packet-filtering code. With the microkernel message-passing architecture, *ioctl()* calls that have pointers embedded in them need to be handled specially. The *ioctl_socket()* function will default to *ioctl()* for functionality that doesn't require special handling.

Packet Filter interface

The **pfil** interface is purely in the stack and supports packet-filtering hooks. Packet filters can register hooks that are called when packet processing is taking place; in essence, **pfil** is a list of callbacks for certain events. In addition to being able to register a filter for incoming and outgoing packets, **pfil** provides support for interface attach/detach and address change notifications.

The **pfil** interface is one of a number of different layers that a user-supplied application can register for, to operate within the stack process context. These modules, when compiled, are called Loadable Shared Modules (**lsm**) in QNX Neutrino nomenclature, or Loadable Kernel Modules (**lkm**) in BSD nomenclature.

There are two levels of registration required with **io-pkt**:

- The first allows the user-supplied module to connect into the **io-pkt** framework and access the stack infrastructure.
- The second is the standard NetBSD mechanism for registering functions with the appropriate layer that sends and receives packets.

In Neutrino, shared modules are dynamically loaded into the stack. You can do this by specifying them on the command line when you start **io-pkt**, using the **-p** option, or you can add them subsequently to an existing **io-pkt** process by using the **mount** command.

The application module must include an initial module entry point defined as follows:

```
#include "sys/io-pkt.h"
#include "nw_datastruct.h"

int mod_entry( void *dll_hdl, struct _iopkt_self *iopkt,
               char *options)
{
```

```
}
```

The calling parameters to the entry function are:

void * *dll_hdl*
: An opaque pointer that identifies the shared module within **io-pkt**.

struct _iopkt_self * *iopkt*
: A structure used by the stack to reference its own internals.

char * *options*
: The options string passed by the user to be parsed by this module.

The header files aren't installed as OS header files, and you must include them from the relevant place in the networking source tree (available from Foundry27).

This is followed by the registration structure that the stack will look for after calling *dlopen()* to load the module to retrieve the entry point:

```
struct _iopkt_lsm_entry IOPKT_LSM_ENTRY_SYM(mod) =
  IOPKT_LSM_ENTRY_SYM_INIT(mod_entry);
```

This entry point registration is used by all shared modules, regardless of which layer the remainder of the code is going to hook into. Use the following functions to register with the **pfil** layer:

```
#include <sys/param.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/pfil.h>


struct pfil_head *
pfil_head_get(int af, u_long dlt);

struct packet_filter_hook *
pfil_hook_get(int dir, struct pfil_head *ph);

int
pfil_add_hook(int (*func)(), void *arg, int flags,
              struct pfil_head *ph);

int
pfil_remove_hook(int (*func)(), void *arg, int flags,
                 struct pfil_head *ph);

int
(*func)(void *arg, struct mbuf **mp, struct ifnet *, int dir);
```

The *head_get()* function returns the start of the appropriate **pfil** hook list used for the hook functions. The *af* argument can be either PFIL_TYPE_AF (for an address family hook) or PFIL_TYPE_IFNET (for an interface hook) for the "standard" interfaces.

If you specify PFIL_TYPE_AF, the Data Link Type (*dlt*) argument is a protocol family. The current implementation has filtering points for only AF_INET (IPv4) or AF_INET6 (IPv6).

When you use the interface hook (PFIL_TYPE_IFNET), *dlt* is a pointer to a network interface structure. All hooks attached in this case will be in reference to the specified network interface.

Once you've selected the appropriate list head, you can use *pfil_add_hook()* to add a hook to the filter list. This function takes as arguments a filter hook function, an opaque pointer (which is passed into the user-supplied filter *arg* function), a *flags* value (described below), and the associated list head returned by *pfil_head_get()*.

The *flags* value indicates when the hook function should be called and may be one of:

- PFIL_IN — call me on incoming packets.
- PFIL_OUT — call me on outgoing packets.
- PFIL_ALL — call me on all of the above.

When a filter is invoked, the packet appears just as if it came off the wire. That is, all protocol fields are in network-byte order. The filter returns a nonzero value if the packet processing is to stop, or zero if the processing is to continue.

For interface hooks, the *flags* argument can be one of:

- PFIL_IFADDR — call me when the interface is reconfigured (**mbuf **** is an *ioctl_socket()* number).
- PFIL_IFNET — call me when the interface is attached or detached (**mbuf **** is either PFIL_IFNET_ATTACH or PFIL_IFNET_DETACH)

Here's an example of what a simple **pfil** hook would look like. It shows when an interface is attached or detached. Upon a detach (**ifconfig** *iface* **destroy**), the filter is unloaded.

```
#include <sys/types.h>
#include <errno.h>
#include <sys/param.h>
#include <sys/conf.h>
#include <sys/socket.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/pfil.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include "sys/io-pkt.h"
#include "nw_datastruct.h"


static int in_bytes = 0;
static int out_bytes = 0;


static int input_hook(void *arg, struct mbuf **m,
                      struct ifnet *ifp, int dir)
{
    in_bytes += (*m)->m_len;
    return 0;
}
```

```
static int output_hook(void *arg, struct mbuf **m,
                       struct ifnet *ifp, int dir)
{
    out_bytes += (*m)->m_len;
    return 0;
}


static int deinit_module(void);
static int iface_hook(void *arg, struct mbuf **m,
                      struct ifnet *ifp, int dir)
{
    printf("Iface hook called ... ");
    if ( (int)m == PFIL_IFNET_ATTACH) {
        printf("Interface attached\n");
        printf("%d bytes in, %d bytes out\n", in_bytes,
               out_bytes);
    } else if ((int)m == PFIL_IFNET_DETACH) {
        printf("Interface detached\n");
        printf("%d bytes in, %d bytes out\n", in_bytes,
               out_bytes);
        deinit_module();
    }
    return 0;
}

static int ifacecfg_hook(void *arg, struct mbuf **m,
                         struct ifnet *ifp, int dir)
{

    printf("Iface cfg hook called with 0x%08X\n", (int)(m));

    return 0;
}

static int deinit_module(void)
{
    struct pfil_head *pfh_inet;

    pfh_inet = pfil_head_get(PFIL_TYPE_AF, AF_INET);
    if (pfh_inet == NULL) {
        return ESRCH;
    }
    pfil_remove_hook(input_hook, NULL, PFIL_IN | PFIL_WAITOK,
                     pfh_inet);
    pfil_remove_hook(output_hook, NULL, PFIL_OUT | PFIL_WAITOK,
                     pfh_inet);

    pfh_inet = pfil_head_get(PFIL_TYPE_IFNET, 0);
    if (pfh_inet == NULL) {
        return ESRCH;
    }

    pfil_remove_hook(ifacecfg_hook, NULL, PFIL_IFNET, pfh_inet);

    pfil_remove_hook(iface_hook, NULL, PFIL_IFNET | PFIL_WAITOK,
                     pfh_inet);
    printf("Unloaded pfil hook\n" );

    return 0;
}
```

```
int pfil_entry(void *dll_hdl, struct _iopkt_self *iopkt,
               char *options)
{
    struct pfil_head *pfh_inet;

    pfh_inet = pfil_head_get(PFIL_TYPE_AF, AF_INET);
    if (pfh_inet == NULL) {
        return ESRCH;
    }
    pfil_add_hook(input_hook, NULL, PFIL_IN | PFIL_WAITOK,
                  pfh_inet);
    pfil_add_hook(output_hook, NULL, PFIL_OUT | PFIL_WAITOK,
                  pfh_inet);

    pfh_inet = pfil_head_get(PFIL_TYPE_IFNET,0);
    if (pfh_inet == NULL) {
        return ESRCH;
    }

    pfil_add_hook(iface_hook, NULL, PFIL_IFNET, pfh_inet);
    pfil_add_hook(ifacecfg_hook, NULL, PFIL_IFADDR, pfh_inet);
    printf("Loaded pfil hook\n" );

    return 0;
}

struct _iopkt_lsm_entry IOPKT_LSM_ENTRY_SYM(pfil) =
  IOPKT_LSM_ENTRY_SYM_INIT(pfil_entry);
```

You can use **pfil** hooks to implement an **io-net** filter; for more information, see the Migrating from **io-net** appendix in this guide.

Packet Filter (pf) module: firewalls and NAT

The **pfil** interface is used by the Packet Filter (**pf**) to hook into the packet stream for implementing firewalls and NAT. This is a loadable module specific to either the v4 or v6 version of the stack (**lsm-pf-v4.so** or **lsm-pf-v6.so**). When loaded (e.g. **mount -Tio-pkt /lib/dll/lsm-pf-v4.so**), the module creates a **pf** pseudo-device.

The **pf** pseudo-device provides roughly the same functionality as **ipfilter**, another filtering and NAT suite that also uses the **pfil** hooks.

If you've downloaded the source code from Foundry27, you can find the portion of **pf** that loads into **io-pkt** in **sys/dist/pf/net**. The source for the accompanying utilities is located under **dist/pf**. For more information, see the following in the *Utilities Reference*:

pf
: Packet Filter pseudo-device

pf.conf
: Configuration file for **pf**

pfctl
: Control the packet filter and network address translation (NAT) device

To start **pf**, use the **pfctl** utility, which issues a DIOCSTART *ioctl_socket()* command. This causes **pf** to call *pf_pfil_attach()*, which runs the necessary **pfil** attachment routines. The key routines after this are *pf_test()* and *pf_test6()*, which are called for IPv4 and IPv6 packets respectively. These functions test which packets should be sent, received, or dropped. The packet filter hooks, and therefore the whole of **pf**, are disabled with the DIOCSTOP *ioctl_socket()* command, usually issued with **pfctl -d**.

For more information about using PF, see **ftp://ftp3.usa.openbsd.org/pub/OpenBSD/doc/pf-faq.pdf** in the OpenBSD documentation. Certain portions of the document (related to packet queueing, CARP and others) don't apply to our stack, but the general configuration information is relevant. This document covers both firewalling and NAT configurations that you can apply using PF.

Berkeley Packet Filter

The Berkeley Packet Filter (BPF) (in **sys/net/bpf.c** in the downloaded source) provides link-layer access to data available on the network through interfaces attached to the system. To use BPF, open a device node, **/dev/bpf**, and then issue *ioctl_socket()* commands to control the operation of the device. A popular example of a tool using BPF is **tcpdump** (see the *Utilities Reference*).

The device **/dev/bpf** is a cloning device, meaning you can open it multiple times. It is in principle similar to a cloning interface, except BPF provides no network interface, only a method to open the same device multiple times.

To capture network traffic, you must attach a BPF device to an interface. The traffic on this interface is then passed to BPF for evaluation. To attach an interface to an open BPF device, use the BIOCSETIF *ioctl_socket()* command. The interface is identified by passing a **struct ifreq**, which contains the interface name in ASCII encoding. This is used to find the interface from the kernel tables. BPF registers itself to the interface's **struct ifnet** field, *if_bpf*, to inform the system that it's interested in traffic on this particular interface. The listener can also pass a set of filtering rules to capture only certain packets, for example ones matching a given combination of host and port.

BPF captures packets by supplying a *bpf_tap()* tapping interface to link layer drivers, and by relying on the drivers to always pass packets to it. Drivers honor this request and commonly have code which, along both the input and output paths, does:

```
#if NBPFILTER > 0
    if (ifp->if_bpf)
        bpf_mtap(ifp->if_bpf, m0);
#endif
```

This passes the **mbuf** to the BPF for inspection. BPF inspects the data and decides if anyone listening to this particular interface is interested in it. The filter inspecting the data is highly optimized to minimize the time spent inspecting each packet. If the filter matches, the packet is copied to await being read from the device.

The BPF tapping feature and the interfaces provided by **pfil** provide similar services, but their functionality is disjoint. The BPF mtap wants to access packets right off the wire without any alteration and possibly copy them for further use. Callers linking into **pfil** want to modify and possibly drop packets. The **pfil** interface is more analogous to **io-net**'s filter interface.

BPF has quite a rich and complex syntax (e.g. **http://www.rawether.net/support/bpfhelp.htm**) and is a standard interface that is used by a lot of networking software. It should be your interface of first choice when packet interception / transmission is required. It will also be a slightly lower performance interface given that it does operate across process boundaries with filtered packets being copied before being passed outside of the stack domain and into the application domain. The **tcpdump** and **libpcap** library operate using the BPF interface to intercept and display packet activity. For those of you currently using something like the **nfm-nraw** interface in **io-net**, BPF provides the equivalent functionality, with some extra complexity involved in setting things up, but with much more versatility in configuration.

Chapter 3

IP Security and Hardware Encryption

In this chapter. . .

The **io-pkt-v4-hc** and **io-pkt-v6-hc** stack variants include full, built-in support for IPsec.

You need to specify the **ipsec** parameter option to the stack in order to enable IPsec when the stack starts.

There's a good reference page in the NetBSD man pages covering IPsec in general. There are some aspects that don't apply (you obviously don't have to worry about rebuilding the kernel), but the general usage is the same.

Setting up an IPsec connection: examples

The following examples illustrate how to set up IPsec:

- between two boxes manually
- with authentication using the preshared-key method

Between two boxes manually

Suppose we have two boxes, A and B, and we want to establish IPsec between them. Here's how:

1. On each box, create a script file (let's say its name is **my_script**) having the following content:

```
#!/bin/ksh
# args: This script takes two arguments:
#     - The first one is the IP address of the box that is to
#       run it on.
#     - The second one is the IP address of the box that this
#       box is to establish IPsec connection to.
Myself=$1
Remote=$2

# The following two lines are to clean the database.
# They're here simply to demonstrate the "hello world" level
# connection.
#
setkey -FP
setkey -F

# Use setkey to input all of the SA content.
setkey -c << EOF

spdadd $Myself $Remote any -P out ipsec esp/transport/$Myself-$Remote/require;
spdadd $Remote $Myself any -P in ipsec esp/transport/$Remote-$Myself/require;

add  $Myself $Remote esp 1234 -m any -E 3des-cbc "KeyIsTwentyFourBytesLong";
add  $Remote $Myself esp 1234 -m any -E 3des-cbc "KeyIsTwentyFourBytesLong";
EOF
```

2. On BoxA, run **./my_script BoxA BoxB**, or give the IP address of each box if the name can't be resolved.

3. Similarly, on BoxB, run **./my_script BoxB BoxA**.

Now you can check the connection by pinging each box from the other. You can get the IPsec status by using **setkey -PD**.

With authentication using the preshared-key method

Consider the simplest case where there are two boxes, BoxA and BoxB. User A is on BoxA, User B is on Box B, and the two users have a shared secret, which is a string of **hello_world**.

1 On Box A, create a file, **psk.txt**, that has these related lines:

```
usera@qnx.com   "Hello_world"
userb@qnx.com   "Hello_world"
```

The IPsec IKE daemon, **racoon**, will use this file to do the authentication and IPsec connection job.

2 The **root** user must own **psk.txt** and the file's permissions must be read/write only by **root**. To ensure this is the case, run:

```
chmod 0600 psk.txt
```

3 The **racoon** daemon needs a configuration file (e.g. **racoon.conf**) that defines the way that **racoon** is to operate. In the remote session, specify that we're going to use the preshared key method as authentication and let **racoon** know where to find the secret. For example:

```
...
# Let racoon know where your preshared keys are:

path pre_shared_key "your_full_path_to_psk.txt" ;
remote anonymous
{
    exchange_mode aggressive,main;
    doi ipsec_doi;
    situation identity_only;

    #my_identifier address;
    my_identifier user_fqdn "usera@qnx.com";
    peers_identifier user_fqdn "userb@qnx.com";

    nonce_size 16;
    lifetime time 1 hour;   # sec,min,hour
    initial_contact on;
    proposal_check obey;    # obey, strict or claim

    proposal {
        encryption_algorithm 3des;
        hash_algorithm sha1;
        authentication_method pre_shared_key ;
        dh_group 2 ;
    }
}
...
```

4 Set up the policy using **setkey**. You can use the following script (called **my_script**) to tell the stack that the IPsec between BoxA and BoxB requires key negotiation:

```
#!/bin/sh
# This is a simple configuration for testing racoon negotiation.
#

Myself=$1
Remote=$2

setkey -FP
setkey -F
setkey -c << EOF
#
spdadd $Remote $Myself any -P in  ipsec esp/transport/$Remote-$Myself/require;
spdadd $Myself $Remote any -P out ipsec esp/transport/$Myself-$Remote/require;
#
EOF
```

Run this on BoxA as **./my_script BoxA BoxB**.

5 Repeat the above steps on BoxB. Needless to say, on BoxB you need to run as **./my_script BoxB BoxA** (and so on).

6 On both boxes, run **racoon -c full_path_to_racoon.conf**. When you initiate traffic, say by trying to **ping** the peer box, **racoon** will do its job and establish the IPsec connection by creating Security Associations (SAs) for both directions, and then you can see the traffic passing back and forth, which indicates that the IPsec connection is established.

IPsec tools

The Neutrino Core Networking uses the IPsec tools from the NetBSD source base and incorporates it into its source base. The tools include:

libipsec PF_KEY library routines.

setkey Security Policy Database and Security Association Database management tool.

racoon IKE key-management daemon. This utility is available only in binary form on request. Under encryption export law, we must track to whom we send this technology and report the information to the US government.

racoonctl A command-line tool that controls **racoon**.

OpenSSL support

We've ported the OpenSSL crypto and SSL libraries (from **http://www.openssl.org**) for your applications to use.

Hardware-accelerated crypto

The **io-pkt-v4-hc** and **io-pkt-v6-hc** managers have the (hardware-independent) infrastructure to load a (hardware-dependent) driver to take advantage of dedicated hardware that can perform cryptographic operations at high speed. This not only speeds up the crypto operations (such as those used by IPsec), but also reduces the CPU load.

This interface is carefully crafted so that the stack doesn't block on the crypto operation; rather, it continues, and later on, using a callback, the driver returns the processed data to the stack. This is ideal for DMA-driven crypto hardware.

Supported hardware crypto engines

The MPCSEC crypto hardware core (present on the E-series PowerQUICC III and PowerQUICC-II PRO series of processors from Freescale) is supported by the **devnp-mpcsec.so** driver. This driver loads just like a standard Ethernet driver from the command line:

```
# io-pkt-v6-hc -d mpcsec -d mpc85xx
```

where **devnp-mpc85xx.so** is the Ethernet driver for the Freescale TSEC (Triple Speed Ethernet Controller) hardware block. For information, see **sys/dev_qnx/mpcsec/README** in the source code, which you can download from Foundry 27.

Chapter 4

WiFi Configuration Using WPA and WEP

In this chapter. . .

802.11 a/b/g Wi-Fi Support

Wi-Fi capability is built into the two **hc** variants of the stack (**io-pkt-v4-hc** and **io-pkt-v6-hc**). The NetBSD stack includes its own separate 802.11 MAC layer that's independent of the driver. Many other implementations pull the 802.11 MAC inside the driver; as a result, every driver needs separate interfaces and configuration utilities. If you write a driver that conforms to the stack's 802.11 layer, you can use the same set of configuration and control utilities for all wireless drivers.

The networking Wi-Fi solution lets you join or host WLAN (Wireless LAN) networks based on IEEE 802.11 specifications. Using **io-pkt**, you can:

- connect using a peer-to-peer mode called *ad hoc mode*, also referred to as *Independent Basic Service Set (IBSS)* configuration
- either act as a client for a Wireless Access Point (WAP, also known as a *base station*) or configure Neutrino to act as a WAP. This second mode is referred to as *infrastructure mode* or *BSS (Basic Service Set)*.

Ad hoc mode lets you create a wireless network quickly by allowing wireless nodes within range (for example, the wireless devices in a room) to communicate directly with each other without the need for a wireless access point. While being easy to construct, it may not be appropriate for a large number of nodes because of performance degradation, limited range, non-central administration, and weak encryption.

Infrastructure mode is the more common network configuration where all wireless hosts (clients) connect to the wireless network via a WAP (Wireless Access Point). The WAP centrally controls access and authentication to the wireless network and provides access to rest of your network. More than one WAP can exist on a wireless network to service large numbers of wireless clients.

The **io-pkt** manager supports WEP, WPA, WPA2, or no security for authentication and encryption when acting as the WAP or client. WPA/WPA2 is the recommended encryption protocol for use with your wireless network. WEP isn't as secure as WPA/WPA2 and is known to be breakable. It's available for backward compatibility with already deployed wireless networks.

For information on connecting your client, see "Using **wpa_supplicant** to manage your wireless network connections" later in this chapter.

NetBSD 802.11 layer

The **net80211** layer provides functionality required by wireless cards. If you've downloaded the Core Networking source from Foundry27 (**http://community.qnx.com/sf/sfmain/do/home**), you'll find the **net80211** layer under **sys/net80211**. The code is meant to be shared between FreeBSD and NetBSD, and you should try to keep NetBSD-specific bits in the source file **ieee80211_netbsd.c** (likewise, there's **ieee80211_freebsd.c** in FreeBSD).

For more information about the **ieee80211** interfaces, see Chapter 9 (Kernel Internals) of the NetBSD manual pages at **http://www.netbsd.org/documentation/**.

The responsibilities of the **net80211** layer are as follows:

- MAC-address-based access control
- crypto
- input and output frame handling
- node management
- radiotap framework for **bpf** and **tcpdump**
- rate adaption
- supplementary routines, such as conversion functions and resource management

The **ieee80211** layer positions itself logically between the device driver and the ethernet module, although for transmission it's called indirectly by the device driver instead of control passing straight through it. For input, the **ieee80211** layer receives packets from the device driver, strips any information useful only to wireless devices, and in case of data payload proceeds to hand the Ethernet frame up to **ether_input**.

Device management

The way to describe an **ieee80211** device to the **ieee80211** layer is by using a **struct ieee80211com**, declared in **<sys/net80211/ieee80211_var.h>**. You use it to register a device to the **ieee80211** from the device driver by calling *ieee80211_ifattach()*. Fill in the underlying **struct ifnet** pointer, function callbacks, and device-capability flags. If a device is detached, the **ieee80211** layer can be notified with *ieee80211_ifdetach()*.

Nodes

A node represents another entity in the wireless network. It's usually a base station when operating in BSS mode, but can also represent entities in an ad hoc network. A node is described by a **struct ieee80211_node**, declared in **<sys/net80211/ieee80211_node.h>**. This structure includes the node unicast encryption key, current transmit power, the negotiated rate set, and various statistics.

A list of all the nodes seen by a certain device is kept in the **struct ieee80211com** instance in the field *ic_sta* and can be manipulated with the helper functions provided in **sys/net80211/ieee80211_node.c**. The functions include, for example, methods to scan for nodes, iterate through the node list, and functionality for maintaining the network structure.

Crypto support

Crypto support enables the encryption and decryption of the network frames. It provides a framework for multiple encryption methods, such as WEP and null crypto. Crypto keys are mostly managed through the *ioctl()* interface and inside the **ieee80211** layer, and the only time that drivers need to worry about them is in the send routine when they must test for an encapsulation requirement and call *ieee80211_crypto_encap()* if necessary.

Using Wi-Fi with io-pkt

When you're connecting to a Wireless Network in Neutrino, the first step that you need to do is to start the stack process with the appropriate driver for the installed hardware. For information on the available drivers, see the **devnp-*** entries in the *Utilities Reference*. For this example, we'll use the driver for network adapters using the RAL chipset, **devnp-ral.so**. After a default installation, all driver binaries are installed under the staging directory **/***cpu***/lib/dll**.

The **io-pkt-v4** stack variant doesn't have the 802.11 layer built in, and therefore you can't use it with Wi-Fi drivers. If you attempt to load a Wi-Fi driver into **io-pkt-v4**, you'll see a number of unresolved symbol errors, and the driver won't work.

In this example, start the stack using one of these commands:

- **io-pkt-v4-hc -d /lib/dll/devnp-ral.so**

  or:

- **io-pkt-v6-hc -d ral**

If the network driver is installed to a location other than **/lib/dll**, you'll need to specify the full path and filename of the driver on the command line.

Once you've started the stack and appropriate driver, you need to determine what wireless networks are available. If you already have the name (SSID or Service Set Identifier) of the network you want to join, you can skip these steps. You can also use these steps to determine if the network you wish to join is within range and active:

1. To determine which wireless networks are available to join, you must first set the interface status to up:

   ```
   ifconfig ral0 up
   ```

2. Check to see which wireless networks have advertised themselves:

   ```
   wlanctl ral0
   ```

   This command lists the available networks and their configurations. You can use this information to determine the network name (SSID), its mode of operation (ad hoc or infrastructure mode), and radio channel, for example.

3. You can also force a manual scan of the network with this command:

```
ifconfig ral0 scan
```

This will cause the wireless adapter to scan for WAP stations or ad hoc nodes within range of the wireless adapter, and list the available networks, along with their configurations. You can also get scan information from the **wpa_supplicant** utility (described later in this document).

Once you've started the appropriate driver and located the wireless network, you'll need to choose the network mode to use (ad hoc or infrastructure mode), the authentication method to attach to the wireless network, and the encryption protocol (if any) to use.

We recommend that you implement encryption on your wireless network if you aren't using any physical security solutions.

By default, most network drivers will infrastructure mode (BSS), because most wireless networks are configured to allow network access via a WAP. If you wish to implement an ad hoc network, you can change the network mode by using the **ifconfig** command:

- To create or join ad hoc networks, use:

  ```
  ifconfig ral0 mediaopt adhoc
  ```

- If you wish to switch back to infrastructure mode, you can use this command to connect to WAP on Infrastructure networks (note the minus sign in front of the **mediaopt** command):

  ```
  ifconfig ral0 -mediaopt adhoc
  ```

For information about your driver's media options, see its entry in the *Utilities Reference*. When you're in ad hoc mode, you advertise your presence to other peers that are within physical range. This means that other 802.11 devices can discover you and connect to your network.

Whether you're a client in infrastructure mode, or you're using ad hoc mode, the steps to implement encryption are the same. You need to make sure that you're using the authentication method and encryption key that have been chosen for the network. If you wish to connect with your peers using an ad hoc wireless network, all peers must be using the same authentication method and encryption key. If you're a client connecting to a WAP, you must use the same authentication method and encryption key as have been configured on the WAP.

Connecting to a wireless network

For the general case of connecting to a Wi-Fi network, we recommend that you use the **wpa_supplicant** daemon. It handles unsecure, WEP, WPA, and WPA2 networks and provides a mechanism for saving network information in a configuration file that's scanned on startup, thereby removing the need for you to constantly reenter network parameters after rebooting or moving from one network domain to another. The information following covers the more specific cases if you don't want to run the supplicant.

You can connect to a wireless network by using one of the following:

- no encryption
- WEP (Wired Equivalent Privacy) for authentication and encryption
- WPA/WPA2 for authentication and encryption
- **wpa_supplicant** to manage your wireless network connections

We will also be rolling out a new Wi-Fi configuration GUI as part of Photon that interfaces into **wpa_supplicant** directly.

Once connected, you need to configure the interface in the standard way:

- TCP/IP configuration in a wireless network (client in Infrastructure Mode, or ad hoc Mode)

Using no encryption

CAUTION:

If you're creating a wireless network with no encryption, anyone who's within range of the wireless network (e.g. someone driving by your building) can easily view all network traffic. It's possible to create a network without using encryption, but we don't recommend it unless the network has been secured by some other mechanism.

Many consumer devices (wireless routers to connect your internal LAN to the Internet for example) are shipped with security features such as encryption turned off. We recommend that you enable encryption in these devices rather than turn off encryption when creating a wireless network.

To connect using no encryption or authentication, type:

```
ifconfig ral0 ssid "network name" -nwkey
```

The **-nwkey** option disables WEP encryption and also deletes the temporary WEP key.

The **io-pkt** manager doesn't support a combination of Shared Key Authentication (SKA) and WEP encryption disabled.

Once you've entered the network name, the 802.11 network should be active. You can verify this with **ifconfig**. In the case of ad hoc networks, the status will be shown as active only if there's at least one other peer on the (SSID) network:

```
ifconfig ral0

ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500%%
    ssid "network name" %%
```

```
    powersave off %%
    bssid 00:11:22:33:44:55 chan 11%%
    address: 11:44:88:44:88:44%%
    media: IEEE802.11 autoselect (OFDM36 mode 11g) %%
    status: active%%
```

Once the network status is active, you can send and receive packets on the wireless link.

You can also use **wpa_supplicant** to associate with a security-disabled Wi-Fi network. For example, if your **/etc/wpa_supplicant.conf** file can contain a network block as follows:

```
network = {
    ssid = "network name"
    key_mgmt = NONE
}
```

you can then run:

```
wpa_supplicant -i ral0 -c/etc/wpa_supplicant.conf
```

You may also use **wpa_cli** to tell **wpa_supplicant** what you want to do. You can use either **ifconfig** or **wpa_cli** to check the status of the network. To complete your network configuration, see "Client in infrastructure or ad hoc mode" in the section on TCP/IP interface configuration.

Using WEP (Wired Equivalent Privacy) for authentication and encryption

WEP can be used for both authentication and privacy with your wireless network. Authentication is a required precursor to allowing a station to associate with an access point. The IEEE 802.11 standard defines the following types of WEP authentication:

Open system authentication

The client is *always* authenticated with the WAP (i.e. allowed to form an association). Keys that are passed into the client aren't checked to see if they're valid. This can have the peculiar effect of having the client interface go "active" (become associated), *but* data won't be passed between the AP and station if the station key used to encrypt the data doesn't match that of the station.

If your WEP station is active, but no traffic seems to be going through (e.g. **dhcp.client** doesn't work), check the key used for bringing up the connection.

Shared key authentication

This method involves a challenge-response handshake in which a challenge message is encrypted by the stations keys and returned to the access point for verification. If the encrypted challenge doesn't match that expected by the access point, then the station is prevented from forming an association.

Unfortunately, this mechanism (in which the challenge and subsequent encrypted response are available over the air) exposes information that could leave the system more open to attacks, so we don't recommended you use it. While the stack does support this mode of operation, the code hasn't been added to **ifconfig** to allow it to be set.

Note that many access points offer the capability of entering a passphrase that can be used to generate the associated WEP keys. The key-generation algorithm may vary from vendor to vendor. In these cases, the generated hexadecimal keys *must* be used for the network key (prefaced by **`0x`** when used with **`ifconfig`**) and not the passphrase. This is in contrast to access points, which let you enter keys in ASCII. The conversion to the hexadecimal key in that case is a simple conversion of the text into its corresponding ASCII hexadecimal representation. The stack supports this form of conversion.

Given the problems with WEP in general, we recommend you use WPA / WPA2 for authentication and encryption where possible.

The network name can be up to 32 characters long. The WEP key must be either 40 bits long or 104 bits long. This means you have to give either 5 or 13 characters for the WEP key, or a 10- or 26-digit hexadecimal value.

You can use either **`ifconfig`** or **`wpa_supplicant`** to configure a WEP network.

If you use **`ifconfig`**, the command is in the form:

`ifconfig` *if_name* **`ssid`** *the_ssid* **`nwkey`** *the_key*

For example, if your interface is **`ral0`**, and you're using 128-bit WEP encryption, you can run:

```
ifconfig ral0 ssid "corporate lan" nwkey corpseckey456 up
```

Once you've entered the network name and encryption method, the 802.11 network should be active (you can verify this with **`ifconfig`**). In the case of ad hoc networks, the status will be shown as active only if there's at least one other peer on the (SSID) network:

```
ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    ssid "corporate lan" nwkey corpseckey456
    powersave off
    bssid 00:11:22:33:44:55 chan 11
    address: 11:44:88:44:88:44
    media: IEEE802.11 autoselect (OFDM36 mode 11g)
    status: active
```

Once the network status is active, you can send and receive packets on the wireless link.

If you use **`wpa_supplicant`**, you need to edit a configuration file to tell it what you want to do. For example:

```
network = {
  ssid = "corporate lan"      # The Wi-Fi network you want to associate to.
  key_mgmt= NONE              # NONE is for WEP or no security.
  wep_key0 = "corpseckey456"  # Most of the time, you may specify a list
                              # from wep_key0 to wep_key3 and use
                              # key index to specify which one to use.
}
```

Then you may run:

`wpa_supplicant -i ral0 -c` *your_config_file*

By default, the configuration file is **/etc/wpa_supplicant.conf**. Alternatively you may use **wpa_cli** to tell the **wpa_supplicant** daemon what you want to do. To complete your network configuration, see "Client in Infrastructure or ad hoc mode" in the section on TCP/IP interface configuration.

Using WPA/WPA2 for authentication and encryption

Background on WPA

The original security mechanism of the IEEE 802.11 standard wasn't designed to be strong and has proven to be insufficient for most networks that require some kind of security. Task group I (Security) of the IEEE 802.11 working group (**http://www.ieee802.org/11/**) has worked to address the flaws of the base standard and has in practice completed its work in May 2004. The IEEE 802.11i amendment to the IEEE 802.11 standard was approved in June 2004 and published in July 2004.

The Wi-Fi Alliance used a draft version of the IEEE 802.11i work (draft 3.0) to define a subset of the security enhancements, called Wi-Fi Protected Access (WPA), that can be implemented with existing WLAN hardware. This has now become a mandatory component of interoperability testing and certification done by Wi-Fi Alliance. Wi-Fi provides information about WPA at its website, **http://www.wi-fi.org/**.

The IEEE 802.11 standard defined a Wired Equivalent Privacy (WEP) algorithm for protecting wireless networks. WEP uses RC4 with 40-bit keys, a 24-bit initialization vector (IV), and CRC32 to protect against packet forgery. All these choices have proven to be insufficient:

- The key space is too small to guard against current attacks.
- RC4 key scheduling is insufficient (the beginning of the pseudo-random stream should be skipped).
- The IV space is too small, and IV reuse makes attacks easier.
- There's no replay protection.
- Non-keyed authentication doesn't protect against bit-flipping packet data.

WPA is an intermediate solution for these security issues. It uses the Temporal Key Integrity Protocol (TKIP) to replace WEP. TKIP is a compromise on strong security, and it's possible to use existing hardware. It still uses RC4 for the encryption as WEP does, but with per-packet RC4 keys. In addition, it implements replay protection and a keyed packet-authentication mechanism.

Keys can be managed using two different mechanisms; WPA can use either of the following:

WPA-Enterprise
: An external authentication server (e.g. RADIUS) and EAP, just as IEEE 802.1X is using.

WPA-Personal
: Pre-shared keys without the need for additional servers.

Both mechanisms generate a master session key for the Authenticator (AP) and Supplicant (client station).

WPA implements a new key handshake (4-Way Handshake and Group Key Handshake) for generating and exchanging data encryption keys between the Authenticator and Supplicant. This handshake is also used to verify that both Authenticator and Supplicant know the master session key. These handshakes are identical regardless of the selected key management mechanism (only the method for generating master session key changes).

WPA utilities

The **wlconfig** library is a generic configuration library that interfaces to the supplicant and provides a programmatic interface for configuring your wireless connection. If you've downloaded the source from Foundry27 (**http://community.qnx.com/sf/sfmain/do/home**), you can find it under **trunk/lib/wlconfig**.

The main utilities required for Wi-Fi usage are:

wpa_supplicant

Wi-Fi Protected Access client and IEEE 802.1X supplicant. This daemon provides client-side authentication, key management, and network persistence.

The **wpa_supplicant** requires the following libraries and binaries be present:

- **libcrypto.so** — crypto library
- **libssl.so** — Secure Socket Library (created from OpenSSL)
- **random** — executable that creates **/dev/urandom** for random-number generation
- **libm.so** — math library required by **random**
- **libz.so** — compression library required by **random**

The **wpa_supplicant** also needs a read/write filesystem for creation of a **ctrl_interface** directory (see the sample **wpa_supplicant.conf** configuration file).

You can't use **/dev/shmem** because it isn't possible to create a directory there.

---

wpa_cli WPA command-line client for interacting with **wpa_supplicant**.

wpa_passphrase

Set the WPA passphrase for a SSID.

hostapd Server side (Access Point) authentication and key-management daemon.

There are also some subsidiary utilities that you likely won't need to use:

wiconfig Configuration utility for some wireless drivers. The **ifconfig** utility can handle the device configuration required without needing this utility.

wlanctl Examine the IEEE 802.11 wireless LAN client/peer table.

Connecting with WPA or WPA2

Core Networking supports connecting to a wireless network using the more secure option of WPA (Wi-Fi Protected Access) or WPA2 (802.11i) protocols.

The **wpa_supplicant** application can manage your connection to a single access point, or it can manage a configuration that includes settings for connections to multiple wireless networks (SSIDs) either implementing WPA, or WEP to support roaming from network to network. The **wpa_supplicant** application supports IEEE802.1X EAP Authentication (referred to as WPA), WPA-PSK, and WPA-NONE (for ad hoc networks) key-management protocols along with encryption support for TKIP and AES (CCMP). A WAP for a simple home or small office wireless network would likely use WPA-PSK for the key-management protocol, while a large office network would use WAP along with a central authentication server such as RADIUS.

To enable a wireless client (or supplicant) to connect to a WAP configured to use WPA, you must first determine the network name (as described above) and get the authentication and encryption methods used from your network administrator. The **wpa_supplicant** application uses a configuration file (**/etc/wpa_supplicant.conf** by default) to configure its settings, and then runs as a daemon in the background. You can also use the **wpa_cli** utility to change the configuration of **wpa_supplicant** while it's running. Changes done by the **wpa_cli** utility are saved in the **/etc/wpa_supplicant.conf** file.

The **/etc/wpa_supplicant.conf** file has a rich set of options that you can configure, but **wpa_supplicant** also uses various default settings that help simplify your wireless configuration. For more information, see **http://netbsd.gw.com/cgi-bin/man-cgi?wpa_supplicant.conf++NetBSD-4.0**.

If you're connecting to a WAP, and your WPA configuration consists of a network name (SSID) and a pre-shared key, your **/etc/wpa_supplicant.conf** would look like this:

```
network={
    ssid="my_network_name"  #The name of the network you wish to join
    psk="1234567890"        #The preshared key applied by the access point
}
```

Make sure that only **root** can read and write this file, because it contains the key information in clear text.

Start **wpa_supplicant** as:

```
wpa_supplicant -B -i ral0 -c /etc/wpa_supplicant.conf
```

The **-i** option specifies the network interface, and **-B** causes the application to run in the background.

The **wpa_supplicant** application by default negotiates the use of the WPA protocol, WPA-PSK for key-management, and TKIP or AES for encryption. It uses infrastructure mode by default.

Once the interface status is active (use **ifconfig ral0**, where **ral0** is the interface name, to check), you can apply the appropriate TCP/IP configuration. For more information, see "TCP/IP configuration in a wireless network," later in this chapter.

If you were to create an ad hoc network using WPA, your **/etc/wpa_supplicant.conf** file would look like this:

```
network={
    mode=1                    # This sets the mode to be ad hoc.
                              # 0 represents Infrastructure mode
    ssid="my_network_name"    # The name of the ad hoc network
    key_mgmt=NONE             # Sets WPA-NONE
    group=CCMP                # Use AES encryption
    psk="1234567890"          # The preshared key applied by the access point
 }
```

Again, make sure that this file is readable and writable only by **root**, because it contains the key information in clear text.

Start **wpa_supplicant** with:

```
wpa_supplicant -B -i ral0 -c /etc/wpa_supplicant.conf
```

where **-i** specifies the network interface, and **-B** causes the application to run in the background.

Personal-level authentication and Enterprise-level authentication

WPA is designed to have the following authentication methods:

- WPA-Personal / WPA2-Personal, which uses a preshared key that's the same passphrase shared by all network users
- WPA-Enterprise / WPA2-Enterprise, which uses an 802.1X authentication RADIUS-based server to authenticate each user

This section is about the Enterprise-level authentication.

The Enterprise-level authentication methods that have been selected for use within the Wi-Fi certification body are:

- EAP-TLS, which is the initially certified method. Both the server's certificates and the user's certificates are needed.
- EAP-TTLS/MSCHAPv2: TTLS is short for "Tunnelled TLS." It works by first authenticating the server to the user via its CA certificate. The server and the user then establish a secure connection (the tunnel), and through the secure tunnel, the user gets authenticated. There are many ways of authenticating the user through the tunnel. The EAP-TTLS/MSCHAPv2 uses MSCHAPv2 for this authentication.
- PEAP/MSCHAPv2: PEAP is the secondmost widely supported EAP after EAP-TLS. It's similar to EAP-TTLS, however, it requires only a server-side CA certificate to create a secure tunnel to protect the user authentication. Again, there are many ways of authenticating the user through the tunnel. The PEAP/MSCHAPV2 again uses MSCHAPV2 for authentication.
- PEAP/GTC: This uses GTC as the authentication method through the PEAP tunnel.
- EAP-SIM: This is for the GSM mobile telecom industry.

The **io-pkt** manager supports all the above, except for EAP-SIM. Certificates are placed in **/etc/cert/user.pem**, and CA certificates in **/etc/cert/root.pem**. The following example is the network definition for **wpa_supplicant** for each of the above Enterprise-level authentication methods:

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
update_config=1

# 3.1.2 linksys -- WEP
network={
    ssid="linksys"
    key_mgmt=NONE
    wep_key0="LINKSYSWEPKEY"
}


# 3.1.3 linksys -- WPA
network={
    ssid="linksys"
    key_mgmt=WPA-PSK
    psk="LINKSYSWPAKEY"
}

# 3.1.4 linksys -- WPA2
network={
    ssid="linksys"
    proto=RSN
    key_mgmt=WPA-PSK
    psk="LINKSYS_RSN_KEY"
}

# 3.1.5.1 linksys -- EAP-TLS
network={
   ssid="linksys"
   key_mgmt=WPA-EAP
   eap=TLS
   identity="client1"
```

```
   ca_cert="/etc/cert/root.pem"
   client_cert="/etc/cert/client1.pem"
   private_key="/etc/cert/client1.pem"
   private_key_passwd="wzhang"
}


# 3.1.5.2 linksys -- PEAPv1/EAP-GTC
network={
   ssid="linksys"
   key_mgmt=WPA-EAP
   eap=PEAP
   identity="client1"
   password="wzhang"
   ca_cert="/etc/cert/root.pem"
   phase1="peaplabel=0"
   phase2="autheap=GTC"
}

# 3.1.5.3 linksys -- EAP-TTLS/MSCHAPv2
network={
   ssid="linksys"
   key_mgmt=WPA-EAP
   eap=TTLS
   identity="client1"
   password="wzhang"
   ca_cert="/etc/cert/root.pem"
   phase2="autheap=MSCHAPV2"
}

# 3.1.5.4 linksys -- PEAPv1/EAP-MSCHAPV2
network={
   ssid="linksys"
   key_mgmt=WPA-EAP
   eap=PEAP
   identity="client1"
   password="wzhang"
   ca_cert="/etc/cert/root.pem"
   phase1="peaplabel=0"
   phase2="auth=MSCHAPV2"
}
```

Run **wpa_supplicant** as follows:

```
wpa_supplicant -i if_name -c full_path_to_your_config_file
```

to pick up the configuration file and get the supplicant to perform the required authentication to get access to the Wi-Fi network.

Using wpa_supplicant to manage your wireless network connections

The **wpa_supplicant** daemon is the “standard” mechanism used to provide persistence of wireless networking information as well as manage automated connections into networks without user intervention.

The supplicant is based on the open-source supplicant (albeit an earlier revision that matches that used by the NetBSD distribution) located at **http://hostap.epitest.fi/wpa_supplicant/**.

In order to support wireless connectivity, the supplicant:

- provides a consistent interface for configuring all authentication and encryption mechanisms (unsecure, wep, WPA, WPA2)
- supports configuration of ad hoc and infrastructure modes of operation
- maintains the network configuration information in a configuration file (by default **/etc/wpa_supplicant.conf**)
- provides auto-connectivity capability allowing a client to connect into a WAP without user intervention

A sample **wpa_supplicant.conf** file is installed in **/etc** for you. It contains a detailed description of the basic supplicant configuration parameters and network parameter descriptions (and there are lots of them) and sample network configuration blocks.

In conjunction with the supplicant is a command-line configuration tool called **wpa_cli**. This tool lets you query the stack for information on wireless networks, as well as update the configuration file on the fly.

We're also in the process of developing a library of routines that will be pulled into a GUI (or that you can use yourself to create a Wi-Fi configuration tool). This library can be found under the source tree in **lib/wlconfig** and creates a **libwlconfig** library for applications to use.

If you want **wpa_cli** to be capable of updating the **wpa_supplicant.conf** file, edit the file and uncomment the **update_config=1** option. (Note that when **wpa_cli** rewrites the configuration file, it strips all of the comments.) Copy the file into **/etc** and make sure that **root** owns it and is the only user who can read or write it, because it contains clear-text keys and password information.

Given a system with a USB-Wi-Fi dongle based on the RAL chips, here's a sample session showing how to get things working with a WEP based WAP:

```
# cp $HOME/stage/etc/wpa_supplicant.conf /etc
# chown root:root /etc/wpa_supplicant.conf
# chmod 600 /etc/wpa_supplicant.conf
# io-pkt-v4-hc -dural
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
    inet 127.0.0.1 netmask 0xff000000
ural0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
    ssid ""
    powersave off
    address: 00:ab:cd:ef:d7:ac
    media: IEEE802.11 autoselect
    status: no network
# wpa_supplicant -B -iural0
# wpa_cli
wpa_cli v0.4.9
Copyright (c) 2004-2005, Jouni Malinen <jkmaline@cc.hut.fi> and contributors

This program is free software. You can distribute it and/or modify it
under the terms of the GNU General Public License version 2.

Alternatively, this software may be distributed under the terms of the
```

```
BSD license. See README and COPYING for more details.


Selected interface 'ural0'

Interactive mode

> scan
OK
> scan_results
bssid / frequency / signal level / flags / ssid
00:02:34:45:65:76   2437    10  [WPA-EAP-CCMP]  A_NET
00:23:44:44:55:66   2412    10  [WPA-PSK-CCMP]  AN_OTHERNET
00:12:4c:56:a7:8c   2412    10  [WEP]   MY_NET
> list_networks
network id / ssid / bssid / flags
0   simple  any
1   second ssid any
2   example any
> remove_network 0
OK
> remove_network 1
OK
> remove_network 2
OK
> add_network
0
> set_network 0 ssid "MY_NET"
OK
> set_network 0 key_mgmt NONE
OK
> set_network 0 wep_key0 "My_Net_Key234"
OK
> enable_network 0
OK
> save
OK
> list_network
network id / ssid / bssid / flags
0   QWA_NET any
> status
<2>Trying to associate with 00:12:4c:56:a7:8c (SSID='MY_NET' freq=2412 MHz)
<2>Trying to associate with 00:12:4c:56:a7:8c (SSID='MY_NET' freq=2412 MHz)
wpa_state=ASSOCIATING
> status
<2>Trying to associate with 00:12:4c:56:a7:8c (SSID='MY_NET' freq=2462 MHz)
<2>Associated with 00:12:4c:56:a7:8c
<2>CTRL-EVENT-CONNECTED - Connection to 00:12:4c:56:a7:8c completed (auth)
bssid=00:12:4c:56:a7:8c
ssid=MY_NET
pairwise_cipher=WEP-104
group_cipher=WEP-104
key_mgmt=NONE
wpa_state=COMPLETED
> quit
# dhcp.client -i ural0
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
    inet 127.0.0.1 netmask 0xff000000
ural0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    ssid MY_NET nwkey My_Net_Key234
    powersave off
    bssid 00:12:4c:56:a7:8c chan 11
```

```
    address: 00:ab:cd:ef:d7:ac
    media: IEEE802.11 autoselect (OFDM54 mode 11g)
    status: active
    inet 10.42.161.233 netmask 0xfffffc00 broadcast 10.42.160.252
#
```

Using a Wireless Access Point (WAP)

A Wireless Access Point (WAP) is a system that allows wireless clients to access the rest of the network or the Internet. Your WAP will operate in BSS mode. A WAP will have at least one wireless network interface to provide a connection point for your wireless clients, and one wired network interface that connects to the rest of your network. Your WAP will act as a bridge or gateway between the wireless clients, and the wired intranet or Internet.

Creating A WAP

To set up your wireless access point, you first need to start the appropriate driver for your network adapters.

Not all network adapter hardware will support operating as an access point. Refer to the documentation for your specific hardware for further information.

For the wireless access point samples, we'll use the **devnp-ral.so** driver for the RAL wireless chipsets, and the **devnp-i82544.so** driver for the wired interface. After a default installation, all driver binaries are installed under the directory **$QNX_TARGET/***cpu***/lib/dll** (or in the same location in your staging directory if you've built the source yourself).

Use one of the following commands:

- **io-pkt-v4-hc -d ral -d i82544**

  or:

- **io-pkt-v4-hc -d /lib/dll/devnp-ral.so -d /lib/dll/devnp-i82544.so**

  or:

- **io-pkt-v6-hc -d ral -d i82544**

If the driver is installed in a location other than **/lib/dll**, you need to specify the full path and filename of the driver on the command line.

The next step to configure your WAP is to determine whether it will be acting as a gateway or a bridge to the rest of the network, as described below.

Acting as a gateway

When your WAP acts as a gateway, it forwards traffic between two subnets (your wireless network and the wired network). For TCP/IP, this means that the wireless TCP/IP clients can't directly reach the wired TCP/IP clients without first sending their packets to the gateway (your WAP). Your WAP network interfaces will also each be assigned an IP address.

This type of configuration is common for SOHO (small office, home office) or home use, where the WAP is directly connected to your Internet service provider. Using this type of configuration lets you:

- keep all of your network hosts behind a firewall/NAT
- define and administer your own TCP/IP network

The TCP/IP configuration of a gateway and firewall is the same whether your network interfaces are wired or wireless. For details of how to configure a NAT, visit **`http://www.netbsd.org/documentation/`**.

Once your network is active, you will assign each interface of your WAP an IP address, enable forwarding of IP packets between interfaces, and apply the appropriate firewall and NAT configuration. For more information, see "DHCP server on WAP acting as a gateway" in the section on TCP/IP interface configuration.

Acting as a bridge

When your WAP acts as a bridge, it's connecting your wireless and wired networks as if they were one physically connected network (broadcast domain, layer 2). In this case, all the wired and wireless hosts are on the same TCP/IP subnet and can directly exchange TCP/IP packets without the need for the WAP to act as a gateway.

In this case, you don't need to assign your WAP network interfaces an IP address to be able to exchange packets between the wireless and wired networks. A bridged WAP could be used to allow wireless clients onto your corporate or home network and have them configured in the same manner as the wireless hosts. You don't need to add more services (such as DHCP) or manipulate routing tables. The wireless clients use the same network resources that the wired network hosts use.

While it isn't necessary to assign your WAP network interfaces an IP address for TCP/IP connectivity between the wireless clients and wired hosts, you probably will want to assign at least one of your WAP interfaces an IP address so that you can address the device in order to manage it or gather statistics.

To enable your WAP to act as a bridge, you first need to create a bridge interface:

```
ifconfig bridge0 create
```

In this case, **`bridge`** is the specific interface type, while 0 is a unique instance of the interface type. There can be no space between **`bridge`** and 0; **`bridge0`** becomes the new interface name.

Use the **brconfig** command to create a logical link between the interfaces added to the bridge (in this case **bridge0**). This command adds the interfaces **ral0** (our wireless interface) and **wm0** (our wired interface). The **up** option is required to activate the bridge:

```
brconfig bridge0 add ral0 add wm0 up
```

Remember to mark your bridge as up, or else it won't be activated.

To see the status of your defined bridge interface, you can use this command:

```
brconfig bridge0

bridge0: flags=41<UP,RUNNING>
    Configuration:
        priority 32768 hellotime 2 fwddelay 15 maxage 20
    Interfaces:
        en0 flags=3<LEARNING, DISCOVER>
            port 3 priority 128
        ral0 flags=3<LEARNING,DISCOVER>
            port 2 priority 128
    Address cache (max cache: 100, timeout: 1200):
```

WEP access point

If you're creating a new wireless network, we recommend you use WPA or WPA2 (RSN) rather than WEP, because WPA and WPA2 provide more better security. You should use WEP only if there are devices on your network that don't support WPA or WPA2.

Enabling WEP network authentication and data encryption is similar to configuring a wireless client, because both the WAP and client require the same configuration parameters.

To use your network adapter as a wireless access point, you must first put the network adapter in host access point mode:

```
ifconfig ral0 mediaopt hostap
```

You will also likely need to adjust the media type (link speed) for your wireless adapter as the auto-selected default may not be suitable. You can view all the available media types with the **ifconfig -m** command. They will be listed in the supported combinations of media type and media options. For example, if the combination of:

```
media OFDM54 mode 11g mediaopt hostap
```

is listed, you could use the command:

```
ifconfig ral0 media OFDM54 mediaopt hostap
```

to set the wireless adapter to use 54 Mbit/s.

The next parameter to specify is the network name or SSID. This can be up to 32 characters long:

```
ifconfig ral0 ssid "my lan"
```

The final configuration parameter is the WEP key. The WEP key must be either 40 bits or 104 bits long. You can either enter 5 or 13 characters for the key, or a 10- to 26-digit hexadecimal value. For example:

```
ifconfig ral0 nwkey corpseckey456
```

You must also mark your network interface as “up” to activate it:

```
ifconfig ral0 up
```

You can also combine all of these commands:

```
ifconfig ral0 ssid "my lan" nwkey corpseckey456 mediaopt hostap up
```

Your network should now be marked as up:

```
ifconfig ral0

ral0: flags=8943<UP,BROADCAST, RUNNING, PROMISC, SIMPLEX, MULTICAST> mtu 1500
    ssid "my lan" apbridge nwkey corpseckey456
    powersave off
    bssid 11:22:33:44:55:66 chan 2
    address: 11:22:33:44:55:66
    media: IEEE802.11 autoselect hostap (autoselect mode 11b hostap)
    status: active
```

WPA access point

WPA/WPA2 support in Neutrino is provided by the **hostapd** daemon. This daemon is the access point counterpart to the client side **wpa_supplicant** daemon. This daemon manages your wireless network adapter when in access point mode. The **hostapd** configuration is defined in the **/etc/hostapd.conf** configuration file.

Before you start the **hostapd** process, you must put the network adapter into host access point mode:

```
ifconfig ral0 mediaopt hostap
```

You will also likely need to adjust the media type (link speed) for your wireless adapter, as the auto-selected default may not be suitable. You can view all the available media types with the **ifconfig -m** command. They will be listed in the supported combinations of media type and media options. For example, if the combination of:

```
media OFDM54 mode 11g mediaopt hostap
```

is listed, you could use the command:

```
ifconfig ral0 media OFDM54 mediaopt hostap
```

to set the wireless adapter to use 54 Mbit/s.

The remainder of the configuration is handled with the **hostapd** daemon. It automatically sets your network interface as up, so you don’t need to do this step with the **ifconfig** utility. Here’s a simple **hostapd** configuration file (**/etc/hostapd.conf**):

```
interface=ral0
ssid=my home lan
macaddr_acl=0
auth_algs=1
wpa=1
wpa_passphrase=myhomelanpass23456
wpa_key_mgmt=WPA-PSK
wpa_pairwise=CCMP
```

This configuration uses WPA-PSK for authentication, and AES for data encryption.

The **auth_algs** and **wpa** are bit fields, not numeric values.

You can now start the **hostapd** utility, specifying the configuration file:

```
hostapd -B /etc/hostapd.conf
```

The **ifconfig** command should show that the network interface is active:

```
ifconfig ral0

ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2290
        ssid "my home lan" apbridge nwkey 2:"",0x49e2a9908872e76b3e5e0c32d09b0b52,0x00000000dc710408c04b32b07c9735b0,""
        powersave off
        bssid 00:15:e9:31:f2:5e chan 4
        address: 00:15:e9:31:f2:5e
        media: IEEE802.11 OFDM54 hostap (OFDM54 mode 11g hostap)
        status: active
```

Your WAP should now be available to your clients.

TCP/IP configuration in a wireless network

Client in infrastructure or ad hoc mode

Assigning an IP address to your wireless interface is independent of the 802.11 network configuration and uses the same utilities or daemons as a wired network. The main issue is whether your TCP/IP configuration is dynamically assigned, or statically configured. A static TCP/IP configuration can be applied regardless of the state of your wireless network connection. The wireless network could be active, or it could be unavailable until later. A dynamically assigned TCP/IP configuration (via the DHCP protocol) requires that the wireless network configuration be active, so that it can reach the DHCP server somewhere on the network. This is typically applied in a network that is centrally administered (using infrastructure mode with a WAP).

The most common usage case is that you're a client using a Wireless Access Point to connect to the network. In this kind of network, there should be a DHCP server available. After the 802.11 network status is active, you just need to start **dhcp.client** to complete your TCP/IP configuration. For example:

```
dhcp.client -iral0
```

As an alternative, you could use **lsm-autoip.so**. Auto IP is a special case in that it negotiates with its peers on the network as they become available; you don't need to wait until the network link becomes active to launch it. Auto IP will assign your network interface an IP address and resolve any IP address conflicts with your network peers as they're discovered when either your host or the peer changes its current IP address. You will be able to use this IP address once the wireless network is active. For more information, see the documentation for Auto IP.

The last configuration option is a static configuration, which doesn't change without intervention from the user. Here's an example of a static configuration that uses **10.0.0.5** for the wireless interface IP address, and **10.0.0.1** for the network gateway:

```
ifconfig ral0 10.0.0.5
route add default 10.0.0.1

cat /etc/resolv.conf

    domain company.com
    nameserver 10.0.0.2
    nameserver 10.0.0.3
```

The other usage case is an ad hoc network. This network mode is typically made up of a number of standalone peers with no central services. Since there's no central server, it's likely that DHCP services won't be available.

If there are Windows or Apple systems on your ad hoc network, they'll enable the Auto IP protocol to assign an IP address. By using Auto IP, you avoid IP address conflicts (two or more hosts using the same IP address), and you avoid having to configure a new IP address manually. Your IP address will be automatically configured, and you'll be able to exchange TCP/IP packets with your peers.

If you're using a static configuration in an ad hoc network, you'll have the added task of deciding what IP address to use on each system, making sure that there are no conflicts, and that all the IP addresses assigned are on the same subnet, so that the systems can communicate.

DHCP server on WAP acting as a gateway

If you've configured your WAP to act as a gateway, you will have your wireless network on a separate subnet from your wired network. In this case, you could be using infrastructure mode or ad hoc mode. The instructions below could work in either mode. You'll likely be using infrastructure mode, so that your network is centrally administered. You can implement DHCP services by running **dhcpd** directly on your gateway, or by using **dhcprelay** to contact another DHCP server elsewhere in the ISP or corporate network that manages DHCP services for subnets.

If you're running **dhcpd** on your gateway, it could be that your gateway is for a SOHO. In this case, your gateway is directly connected to the Internet, or to an IP network for which you don't have control or administrative privileges. You may also be using NAT in this case, as you've been given only one IP address by your Internet Service Provider. Alternatively, you may have administrative privileges for your network subnet which you manage.

If you're running **dhcprelay** on your gateway, your network subnet is managed elsewhere. You're simply relaying the DHCP client requests on your subnet to the

DHCP server that exists elsewhere on the network. Your relay agent forwards the client requests to the server, and then passes the reply packets back to the client.

These configuration examples assume that you have an interface other than the wireless network adapter that's completely configured to exchange TCP/IP traffic and reach any servers noted in these configurations that exist outside of the wireless network. Your gateway will be forwarding IP traffic between this interface and the wireless interface.

Launching the DHCP server on your gateway

This section describes how to launch the DHCP server on the gateway.

DHCP server configuration file

This is a simple **dhcpd** configuration file, **dhcpd.conf**. This file includes a subnet range that's dynamically assigned to clients, but also contains two static entries for known servers that are expected to be present at certain IP addresses. One is a printer server, and the other is a network-enabled toaster. The DHCP server configuration isn't specific to wireless networks, and you can apply it to wired networks as well.

```
ddns-update-style none;

#option subnet-mask 255.255.255.224;
default-lease-time 86400;
#max-lease-time 7200;

subnet 192.168.20.0 netmask 255.255.255.0 {
        range 192.168.20.41 192.168.20.254;
        option broadcast-address 192.168.20.255;
        option routers 192.168.20.1;
        option domain-name-servers 192.168.20.1;
        option domain-name "soho.com";

        host printerserver {
               hardware ethernet 00:50:BA:85:EA:30;
               fixed-address 192.168.20.2;
        }

        host networkenabledtoaster {
               hardware ethernet 00:A0:D2:11:AE:81;
               fixed-address 192.168.20.40;
        }
}
```

The nameserver, router IP, and IP address will be supplied to your wireless network clients. The router IP address is the IP address of the gateway's wireless network interface that's connected to your wireless network. The nameserver is set to the gateway's wireless network adapter, since the gateway is also handling name serving services. The gateway nameserver will redirect requests for unknown hostnames to the ISP nameserver. The internal wireless network has been defined to be `192.168.20.0`. Note that we've reserved IP address range `192.168.20.1` through `192.168.20.40` for static IP address assignment; the dynamic range starts at `192.168.20.41`.

Now that we have the configuration file, we need to start **dhcpd**.

We need to make sure that the directory **/var/run** exists, as well as **/var/state/dhcp**. The file **/var/state/dhcp/dhcpd.leases** must exist. You can create an empty file for the initial start of the **dhcpd** binary.

When you start **dhcpd**, you must tell it where to find the configuration file if it isn't in the default location. You also need to pass an interface name, as you want only **dhcpd** to service your internal wireless network interface. If we used the adapter from the wireless discussion, this would be **ral0**:

```
dhcpd -cf /etc/dhcpd.conf ral0
```

Your DHCP server should now be running. If there are any issues, you can start **dhcpd** in a debugging mode using the **-d** option. The **dhcpd** daemon also logs messages to the system log, **slogger**.

Launching the DHCP relay agent on your gateway

The **dhcprelay** agent doesn't require a configuration file as the DHCP server does; you just need to launch a binary on the command line. What you must know is the IP address of the DHCP server that's located elsewhere on the network that your gateway is connected to. Once you've launched **dhcprelay**, it forwards requests and responses between the client on your wireless network and the DHCP server located elsewhere on the ISP or corporate network:

```
dhcprelay -i ral0 10.42.42.42
```

In this case, it relays requests from wireless interface (**ral0**), and forward these requests to the DHCP server **10.42.42.42**.

Configuring an access point as a router

To configure an access point as a router:

1. Make sure the outside network interface on your access point is active. That is, make sure your access point is active on the wired network that it's connected to.
2. Configure the access point interface. The simplest mechanism to use for this is WEP.

   Say we want our wireless network to advertise **MY_WIRELESS_NET**, and our WEP secret is **MYWIRELESSWEP**. We have to do the following:

   2a Allow packets coming in from one interface to be forwarded (routed) out another:

   ```
   #sysctl -w net.inet.ip.forwarding=1
   ```

   2b Place the wireless interface into access point mode:

   ```
   #ifconfig in_nic mediaopt hostap
   ```

   2c Configure the wireless interface to be a WEP network with an associated key:

```
#ifconfig in_nic ssid MY_WIRELESS_NET nwkey MYWIRELESSWEP
```

2d Bring up the interface:

```
#ifconfig in_nic 10.42.0.1 up
```

3 See above for how you set up DHCP to distribute IP addresses to the wireless client. Briefly, you provide a **dhcpd.conf** with a configuration section as follows, which defines the internal network:

```
subnet 10.42.42.0 netmask 255.255.255.0 {
    range 10.42.0.2 10.42.0.120;
    ...;
}
```

Then you run **dhcpd**:

```
#dhcpd -cf full_path_to_your_dhcp_config_file -lf \
full_path_to_your_release_file ni_nic
```

You don't need to specify where your **dhcpd.conf** and release file are if you put them in the default place under **/etc**. For more information, see the entry for **dhcpd** in the *Utilities Reference*.

To use WPA or WPA2, you need to set up and run **hostapd** (the server-side application associated with the client's **wpa_supplicant**) to do the authentication and key exchange for your network.

You can also configure your access point as a NAT network router as follows:

```
#mount -Ttcpip lsm-pfv4.so
```

so that the PF module is loaded, and then use **pfctl** to do the configuration.

For details of how to configure a NAT, visit **http://www.netbsd.org/documentation/**.

Chapter 5

Transparent Distributed Processing

In this chapter. . .

TDP and io-pkt

Transparent Distributed Processing (also known as Qnet) functions the same under the old **io-net** and new **io-pkt** infrastructures, and the packet format and protocol remain the same. For both **io-net** and **io-pkt**, Qnet is just another protocol (like TCP/IP) that transmits and receives packets.

The Qnet module in Core Networking is now a loadable shared module, **lsm-qnet.so**. We support only the **l4_lw_lite** variant; we no longer support the **qnet-compat** variant that was compatible with Neutrino 6.2.1.

To start the stack with Qnet, type this command:

```
io-pkt-v4 -ddriver -pqnet
```

(assuming you have your **PATH** and **LD_LIBRARY_PATH** environment variables set up properly). You can also mount the protocol after the stack has started, like this:

```
mount -Tio-pkt full_path_to_dll/lsm-qnet.so
```

Note that **mount** still supports the **io-net** option, to provide backward compatibility with existing scripts.

The command-line options and general configuration information are the same as they were with **io-net**. For more information, see **lsm-qnet.so** in the *Utilities Reference*.

Using TDP over IP

TDP supports two modes of communications: one directly over Ethernet, and one over IP. The “straight to Ethernet” L4 layer is faster and more dynamic than the IP layer, but it isn’t possible to route TDP packets out of a single layer-2 domain. By using TDP over IP, you can connect to any remote machine over the Internet as follows:

1. TDP must use the DNS resolver to get an IP address from a hostname (i.e. use the **resolve=dns** option). Configure the local host name and domain, and then make sure that *gethostbyname()* can resolve all the host names that you want to talk to (including the local machine):
   - Use **hostname** to set the host name.
   - Use **setconf** to set the _CS_DOMAIN configuration string to indicate your domain.
   - If the hosts aren’t in a DNS database, create an appropriate name to host resolution file in **/etc/hosts** which includes the fully qualified node name (including domain) and change the resolver to use the host file instead of using the DNS server.
     (e.g. **setconf _CS_RESOLVE lookup_file_bind**)
     For more information on name resolution, see the **"**Name servers**"** section in TCP/IP Networking.
2. Start (or mount) Qnet with the **bind=ip,resolve=dns** options. For example:

```
io-pkt-v4-hc -di82544 -pqnet bind=ip,resolve=dns
```

or:

mount -Tio-pkt -o bind-ip,resolve=dns *full_path_to_dll*/**lsm-qnet.so**

With raw Ethernet transport, names automatically appear in the **/net** directory. This doesn't happen with TDP over IP; as you perform TDP operations (e.g. **ls /net/host1**), the entries are created as required.

Chapter 6

Network Drivers

In this chapter...

Types of network drivers

The networking stack supports the following types of drivers:

- *Native drivers* that are written specifically for the **io-pkt** stack and as such are fully featured, provide high performance, and can run with multiple threads.
- **io-net** drivers that were written for the legacy networking stack **io-net**.
- Ported NetBSD drivers that were taken from the NetBSD source tree and ported to **io-pkt**.

You can tell a native driver from an **io-net** driver by the name:

- **io-net** drivers are named **devn-***xxxxxx***.so**
- **io-pkt** native drivers are named **devnp-***xxxxxx***.so**

NetBSD drivers aren't as tightly integrated into the overall stack. In the NetBSD operating system, these drivers operate with interrupts disabled and, as such, generally have fewer mutexing issues to deal with on the transmit and receive path. With a straight port of a NetBSD driver, the stack defaults to a single-threaded model, in order to prevent possible transmit and receive synchronization issues with simultaneous execution. If the driver has been carefully analyzed and proper synchronization techniques applied, then a flag can be flipped during the driver attachment, saying that the multi-threaded operation is allowed.

If one driver operates in single-threaded mode, all drivers operate in single-threaded mode.

The native and NetBSD drivers all hook directly into the stack in a similar manner. The **io-net** drivers interface through a "shim" layer that converts the **io-net** binary interface into the compatible **io-pkt** interface. We have a special driver, **devnp-shim.so**, that's automatically loaded when you start an **io-net** driver.

The shim layer provides binary compatibility with existing **io-net** drivers. As such, these drivers are also not as tightly integrated into the stack. Features such as dynamically setting media options or jumbo packets for example aren't supported for these drivers. Given that the driver operates within the **io-net** design context, the drivers won't perform as well as a native one. In addition to the packet receive / transmit device drivers, device drivers are also available that integrate hardware crypto acceleration functionality directly into the stack.

For information about specific drivers, see the *Utilities Reference*:

- **devnp-*** for native **io-pkt** and ported NetBSD drivers. The entry for each driver indicates which type it is.
- **devn-*** for legacy **io-net** drivers

- Source code and/or binaries for some native drivers (especially those for WiFi chipsets) might not be generally available due to issues concerning licensing or non-disclosure agreements.
- We might not be able to support ported drivers for which the source is publicly available if the vendor doesn't provide documentation to us. While we'll make every effort to help you, we can't guarantee that we'll be able to rectify problems that may occur with these drivers.

For information about converting drivers, see the "Porting an **io-net** driver to **io-pkt**" technote.

Differences between ported NetBSD drivers and native drivers

There's a fine line between native and ported drivers. If you do more than the initial "make it run" port, the feature sets of a ported driver and a native driver aren't really any different.

If you look deeper, there are some differences:

- From a source point of view, a ported driver has a very different layout from a native **io-pkt** driver. The native driver source looks quite similar in terms of content and files to what an **io-net** driver looks like and has all of the source for a particular driver under one directory. The NetBSD driver source is quite different in layout, with source for a particular driver spread out under a specific driver directory, as well as **ic**, **pci**, **usb**, and other directories, depending on the driver type and bus that it's on.
- Ported NetBSD drivers don't allow the stack to run in multi-threaded mode. NetBSD drivers don't have to worry about Rx / Tx threads running simultaneously when run inside of the NetBSD operating system, so there's no need to pay close attention to appropriate locking issues between Rx and Tx.

  For this reason, a configuration flag is, by default, set to indicate that the driver doesn't support multi-threaded access. As a result, the entire stack runs in a single-threaded mode of operation (if one driver can't run in multithreaded mode, no drivers will run with multiple threads). You can change this flag once you've carefully examined the driver to ensure that there are no locking issues.
- NetBSD drivers don't include support for Neutrino-specific utilities, such as **nicinfo**.
- Unless otherwise indicated, we provide source and allow you to build NetBSD drivers that we've ported, but, unless we have full documentation from the silicon vendors, we can't classify the device as supported.
- The NetBSD drivers have two different delay functions, both of which take an argument in microseconds. From the NetBSD documentation, *DELAY()* is reentrant

(i.e it doesn't modify any global kernel or machine state) and is safe to use in interrupt or process context.

However, Neutrino's version of *delay()* takes a time in *milliseconds*, so this could result in very long timeouts if used directly as-is in the drivers. We've defined *DELAY()* to do the appropriate conversion of the delay from microseconds to milliseconds, so all NetBSD ported drivers should define *delay()* to be *DELAY()*.

Differences between `io-net` drivers and other drivers

The differences between legacy **io-net** drivers and other drivers include the following:

- The **io-net** drivers export a name space entry, **/dev/io-net/en***x*. Native drivers don't.
- You can unmount an **io-net** driver (**umount /dev/io-net/en***x*). With a native driver, you have to destroy it (**ifconfig tsec0 destroy**).
- The **io-net** drivers are all prefixed with **en**. Native drivers have different prefixes for different hardware (e.g. **tsec** for Freescale TSEC devices), although you can override this with the **name=** driver option (processed by **io-pkt**).
- The **io-net** drivers support the **io-net** *devctl()* commands. Native drivers don't.
- The **io-net** drivers are slower than native drivers, since they use the same threading model as that used in **io-net**.
- The **io-net** driver DLLs are prefixed by **devn-**. Core Networking drivers are prefixed by **devnp-**.
- The **io-net** drivers used the **speed** and **duplex** command-line options to override the auto-negotiated link defaults once. Often the use of these options caused more problems than they fixed. Native (and most ported NetBSD drivers) allow their speed and duplex setting to be determined at runtime via a device *ioctl()*, which **ifconfig** uses. See **ifconfig -m** and **ifconfig mediaopt**.

Loading and unloading a driver

You can load drivers into the stack from the command line just as with **io-net**. For example:

```
io-pkt-v4-hc -di82544
```

This command-line invocation works whether or not the driver is a native driver or an **io-net**-style driver. The stack automatically detects the driver type and loads the **devnp-shim.so** binary if the driver is an **io-net** driver.

Make sure that all drivers are located in a directory that can be resolved by the **LD_LIBRARY_PATH** environment variable if you don't want to have to specify the fully qualified name of the device in the command line.

You can also **mount** a driver in the standard way:

```
mount -Tio-pkt /lib/dll/devnp-i82544.so
```

The **mount** command still supports the **io-net** option, to provide backward compatibility with existing scripts:

```
mount -Tio-net /lib/dll/devnp-i82544.so
```

The standard way to remove a driver from the stack is with the **ifconfig** *iface* **destroy** command. For example:

```
ifconfig wm0 destroy
```

Troubleshooting a driver

For native drivers and **io-net** drivers, the **nicinfo** utility is usually the first debug tool that you'll use (aside from **ifconfig**) when problems with networking occur. This will let you know whether or not the driver has properly negotiated at the link layer and whether or not it's sending and receiving packets.

Ensure that the **slogger** daemon is running, and then after the problem occurs, run the **sloginfo** utility to see if the driver has logged any diagnostic information. You can increase the amount of diagnostic information that a driver logs by specifying the **verbose** command-line option to the driver. Many drivers support various levels of verbosity; you might even try specifying **verbose=10**.

For ported NetBSD drivers that don't include **nicinfo** capabilities, you can use **netstat -I** *iface* to get very basic packet input / output information. Use **ifconfig** to get the basic device information. Use **ifconfig -v** to get more detailed information.

Problems with shared interrupts

Having different devices sharing a hardware interrupt is kind of a neat idea, but unless you really need to do it — because you've run out of hardware interrupt lines — it generally doesn't help you much. In fact, it can cause you trouble. For example, if your driver doesn't work (e.g. no received packets), check to see if it's sharing an interrupt with another device, and if so, reconfigure your board so it doesn't.

Most of the time, when shared interrupts are configured, there's no good reason for it (i.e. you haven't really run out of interrupts) and this can decrease your performance, because when the interrupt fires, *all* of the devices sharing the interrupt need to run and check to see if it's for them. If you check the source code, you can see that some drivers do the "right thing," which is to read registers in their interrupt handlers to see if the interrupt is really for them, and then ignore it if not. But many drivers don't; they schedule their thread-level event handlers to check their hardware, which is inefficient and reduces performance.

If you're using the PCI bus, use the **pci -v** utility to check the interrupt allocation.

Sharing interrupts can vastly increase interrupt latency, depending upon exactly what each of the drivers does. After an interrupt fires, the kernel doesn't reenable it until *all* driver handlers tell the kernel that they've finished handling it. So, if one driver takes a long time servicing a shared interrupt that's masked, then if another device on the same interrupt causes an interrupt during that time period, processing of that interrupt can be delayed for an unknown duration of time.

Interrupt sharing can cause problems, and reduce performance, increase CPU consumption, and seriously increase latency. Unless you really need to do it, don't. If you must share interrupts, make sure your drivers are doing the "right thing."

Writing a new driver

If you've downloaded the source from Foundry27 (**http://community.qnx.com/sf/sfmain/do/home**), you'll find a technote in the source tree under **/trunk/sys/dev_qnx/doc** that describes how to write a native driver. Sample driver code is also available under the **/trunk/sys/dev_qnx/sample** directory.

Debugging a driver using gdb

If you want to use **gdb** to debug a driver, youfirst have to make sure that your source is compiled with debugging information included. With your driver code in the correct place in the **sys** tree (**dev_qnx** or **dev**), you can do the following:

```
# cd sys
# make CPULIST=x86 clean
# make CPULIST=x86 CCOPTS=-O0 DEBUG=-g install
```

Now that you have a debug version, you can start **gdb** and set a breakpoint at *main()* in the **io-pkt** binary.

Don't forget to specify your driver in the arguments, and ensure that the **PATH** and **LD_LIBRARY_PATH** environment variables are properly set up.

After hitting the breakpoint in *main()*, do a **sharedlibrary** command in **gdb**. You should see **libc** loaded in. Set a breakpoint in *dlsym()*. When that's hit, your driver should be loaded in, but **io-pkt** hasn't done the first callout into it. Do a **set solib-search-path** and add the path to your driver, and then do a **sharedlibrary** again. The debugger should load the symbols for your driver, and then you can set a breakpoint where you want your debugging to start.

Dumping 802.11 debugging information

The stack's 802.11 layer can dump debugging information. You can enable and disable the dumping by using **sysctl** settings. If you do:

```
sysctl -a | grep 80211
```

with a Wi-Fi driver, you'll see **net.link.ieee80211.debug** and **net.link.ieee80211.vap0.debug**. To turn on the debug output, type the following:

```
sysctl -w net.link.ieee80211.debug = 1
sysctl -w net.link.ieee80211.vap0.debug=0xffffffff
```

You can then use **sloginfo** to display the debug log.

Jumbo packets and hardware checksumming

Jumbo packets are packets that carry more payload than the normal 1500 bytes. Even the definition of a jumbo packet is unclear; different people use different lengths. For jumbo packets to work, the protocol stack, the drivers, and the network switches must all support jumbo packets:

- The **io-pkt** (hardware-independent) stack supports jumbo packets.
- Not all network hardware supports jumbo packets (generally, newer GiGE NICs do).
- Native drivers for **io-pkt** support jumbo packets. For example, **devnp-i82544.so** is a native **io-pkt** driver for PCI, and it supports jumbo packets. So does the **devnp-mpc85xx.so** for MPC 83xx/85xx.

If you can use jumbo packets with **io-pkt**, you can see substantial performance gains because more data can be moved per packet header processing overhead.

To configure a driver to operate with jumbo packets, do this (for example):

```
# ifconfig wm0 ip4csum tcp4csum udp4csum
# ifconfig wm0 mtu 8100
# ifconfig wm0 10.42.110.237
```

For maximum performance, we also turned on hardware packet checksumming (for both transmit and receive) and we've arbitrarily chosen a jumbo packet MTU of 8100 bytes. A little detail: **io-pkt** by default allocates 2 KB clusters for packet buffers. This works well for 1500 byte packets, but for example when an 8 KB jumbo packet is received, we end up with 4 linked clusters. We can improve performance by telling **io-pkt** (when we start it) that we're going to use jumbo packets, like this:

```
# io-pkt-v6-hc -d i82544 -p tcpip pagesize=8192,mclbytes=8192
```

If we pass the **pagesize** and **mclbytes** command-line options to the stack, we tell it to allocate contiguous 8 KB buffers (which may end up being two adjacent 4 KB pages, which works fine) for each 8 KB cluster to use for packet buffers. This reduces packet processing overhead, which improves throughput and reduces CPU utilization.

Padding Ethernet packets

If an Ethernet packet is shorter than ETHERMIN bytes, padding can be added to the packet to reach the required minimum length. In the interests of performance, the driver software doesn't automatically pad the packets, but leaves it to the hardware to do so if supported. If hardware pads the packets, the contents of the padding depend on the hardware implementation.

Transmit Segmentation Offload (TSO)

Transmit Segmentation Offload (TSO) is a capability provided by some modern NIC cards (see, for example, **http://en.wikipedia.org/wiki/Large_segment_offload**). Essentially, instead of the stack being responsible for breaking a large IP packet into MTU-sized packets, the driver does it. This greatly offloads the amount of CPU required to transmit large amounts of data.

You can tell if a driver supports TSO by typing **ifconfig** and looking at the capabilities section of the interface output. It will have **tso** marked as one of its capabilities. To configure the driver to use TSO, type (for example):

```
ifconfig wm0 tso4
ifconfig wm0 10.42.110.237
```

Chapter 7

Utilities, Managers, and Configuration Files

The utilities, drivers, configuration files, and so on listed below are associated with **io-pkt**. For more information, see the *Utilities Reference*.

brconfig	Configure network bridge parameters
hostapd	Authenticator for IEEE 802.11 networks
ifconfig	Configure network interface parameters
ifwatchd	Watch for addresses added to or deleted from interfaces and call up/down-scripts for them
io-pkt	Network I/O support
lsm-autoip.so	AutoIP negotiation module for link-local addresses
lsm-qnet.so	Transparent Distributed Processing (native QNX network) module
nicinfo	Display information about a network interface controller
pf	Packet Filter pseudo-device
pf.conf	Configuration file for **pf**
pfctl	Control the packet filter (PF) and network address translation (NAT) device
ping	Send ICMP ECHO_REQUEST packets to network hosts (UNIX)
pppoectl	Display or set parameters for a **pppoe** interface
setkey	Manually manipulate the IPsec SA/SP database
sysctl	Get or set the state of the socket manager
tcpdump	Dump traffic on a network
wpa_cli	WPA command-line client
wpa_passphrase	Set WPA passphrase for a SSID
wpa_supplicant	Wi-Fi Protected Access client and IEEE 802.1X supplicant

For information about drivers, see the **devnp-*** entries in the *Utilities Reference*.

Appendix A

Migrating from `io-net`

In this appendix...

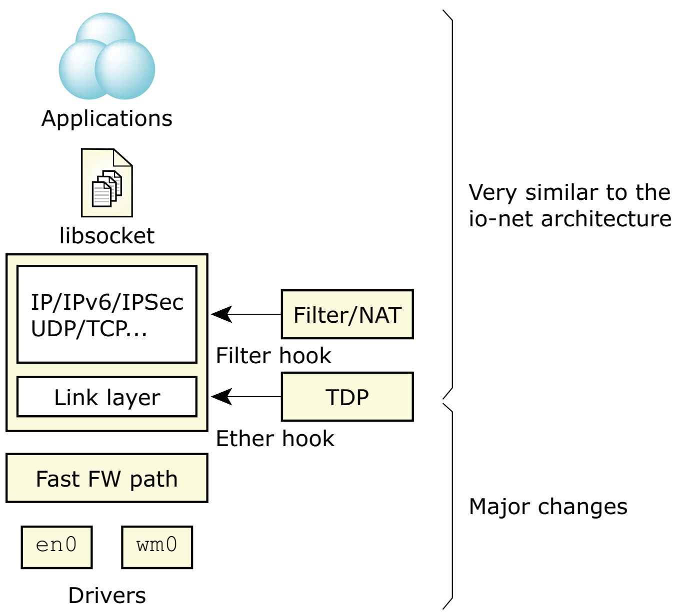
This appendix describes the compatibility between **io-net** and **io-pkt**.

For information about converting drivers, see the "Porting an **io-net** driver to **io-pkt**" technote.

Overview

The previous generation of the QNX Neutrino networking stack (**io-net**) was designed based on a modular approach. It served its purpose in the past by allowing users to separate protocols and drivers, but this came at the expense of incurring a significant amount of overhead when converting to a particular protocol's domain from **io-net** and vice versa.

Note that **io-pkt** and the new utilities and daemons aren't backward-compatible with **io-net**.



Changes to the networking stack.

Compatibility between io-net and io-pkt

Both **io-net** and **io-pkt** can co-exist on the same system. The updated socket library provided with **io-pkt** is compatible with **io-net**. This lets you run both **io-net** and **io-pkt** simultaneously.

The reverse isn't true; if you use the **io-net** version of the socket library with **io-pkt**, unresolved symbols will occur when you attempt to use the **io-pkt** configuration utilities (e.g. **ifconfig**).

We've updated the following binaries for **io-pkt**:

- **netmanager**
- **ifconfig**
- **route**
- **sysctl**
- **arp**
- **netstat**
- **ping**
- **ping6**
- **sockstat** (see the NetBSD documentation)
- **ftp**
- **ftpd**
- **inetd**
- **nicinfo**
- **pppd**
- **pppoed** — this is now simply a shim layer that **phdialer** uses to dial up PPPOE

Compatibility issues

Binaries

The following replaced binaries are known to have compatibility issues with **io-net**. Essentially, new utilities are likely to contain enhanced features that aren't supported by the old stack:

- **pppoed**
- **inetd**
- **ftp**
- **sysctl**
- **ifconfig**

The following **io-net** binaries are known to have compatibility issues with **io-pkt**:

- **snmpd**

Sockets

The socket library is fully backward-compatible with all BSD socket API applications. This also extends to the routing socket.

Protocols

A *bind()* on an AF_INET socket now requires that the second argument have its **(struct sockaddr *)->af_family** member be initialized to AF_INET. Previously a value of 0 was accepted and assumed to be this value.

Drivers — A special "shim" layer makes drivers that were written for **io-net** compatible with **io-pkt**. For more information, see the Network Drivers chapter in this guide.

Behavioral differences

- If **io-pkt** has problems loading a device, it doesn't print failure messages to the console by default; they're automatically sent to **slogger**. You can use the **-v** option to **io-pkt** to force the output to the console for debugging purposes.

- The way in which the SIOCGIFCONF *ioctl()* command was used in our **io-net** code was incorrect but it worked. We've changed the implementation, but applications that use the old method will no longer work. Here's some code that illustrates the old and new methods:

```
/*
 * Example demonstrating a common pitfall with SIOCGIFCONF handling.
 */


#include <sys/sockio.h>
#include <net/if.h>
#include <malloc.h>
#include <stdlib.h>
#include <err.h>
#include <ifaddrs.h>

void gifconf(int);
void gifaddrs(int);


int
main(void)
{
int             s;

if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
err(EXIT_FAILURE, "socket");

gifconf(s);     /* Old code often used SIOCGIFCONF ioctl        */
gifaddrs(s);    /* New code should use getifaddrs()             */

close(s);

return 0;
}

void
gifconf(int s)
{
struct ifconf   ifc;
struct ifreq    *inext, *iend, *icur;
size_t size;


size = 4096;
ifc.ifc_len = size;
if ((ifc.ifc_buf = malloc(ifc.ifc_len)) == NULL)
err(EXIT_FAILURE, "malloc");
```

```
if (ioctl(s, SIOCGIFCONF, &ifc) == -1)
err(EXIT_FAILURE, "SIOCGIFCONF");

if (ifc.ifc_len >= size) {
/* realloc and try again */
errx(EXIT_FAILURE, "SIOCGIFCONF: buf too small");
}

inext = ifc.ifc_req;
iend = (struct ifreq *)(ifc.ifc_buf + ifc.ifc_len);
for (;;) {
icur = inext;
#if 0
/*
 * Broken code.  This would happen to work for most cases
 * because previously:
 *
 * sizeof(struct sockaddr) + IFNAMSIZ == sizeof(struct ifreq)
 *
 * Under this scenario  the two 'if' cases in the working
 * case below work out to the same thing.
 */
inext = (struct ifreq *)
    ((char *)inext + inext->ifr_addr.sa_len + IFNAMSIZ);
#else
/* This will work against old / new libsocket */
if (inext->ifr_addr.sa_len + IFNAMSIZ > sizeof(struct ifreq))
inext = (struct ifreq *)
    ((char *)inext + inext->ifr_addr.sa_len + IFNAMSIZ);
else
inext++;
#endif
if (inext > iend)
break;

/* process icur */
}

free(ifc.ifc_buf);
}

void
gifaddrs(int s)
{
struct ifaddrs *ifaddrs, *ifap;

if (getifaddrs(&ifaddrs) == -1)
err(EXIT_FAILURE, "getifaddrs");

for (ifap = ifaddrs; ifap != NULL; ifap = ifap->ifa_next) {
continue;
}

freeifaddrs(ifaddrs);
}
```

If you compile the test case against the old headers, the SIOCGIFCONF *ioctl()* will produce the expected results in all environments: old / new stack, old / new **libc**. If you compile it against the **io-pkt** headers, the SIOCGIFCONF command will work only with the **io-pkt** stack and the 6.4 **libc**. In either case, there's no dependency on any version of **libsocket**.

The *getifaddrs()* call will work everywhere and is recommended for new code.

Simultaneous support

You can run both **io-net** and **io-pkt** simultaneously on the same target if the relevant utilities and daemons for each stack are present. Here are some specific issues you should be aware of:

- The socket library is backward-compatible with all BSD socket API applications. This also extends to the routing socket.
- Applications with a tight coupling to the TCP/IP stack, including **ifconfig**, **netstat**, **arp**, **route**, **sysctl**, **inetd**, **pppd**, and **pppoed**, aren't backward-compatible. Stack-specific versions need to be maintained and executed.
- Socket library and headers aren't saved. The new versions are backward-compatible.
- The following components may be compatible with **io-pkt**:
  - **phdialer** is compatible.
  - The network usage widget in **pwm** won't work with non-**io-net**-style drivers since they aren't compatible with the *devctl()* interface used by the widget.
- We've updated **nicinfo** to work with native **io-pkt** drivers (e.g. **nicinfo wm0**). NetBSD drivers don't operate with **nicinfo**.
- You can run both **io-net** and **io-pkt** simultaneously, but you have to provide different instance numbers and prefixes to the stack. For example:

  ```
  io-pkt -d pcnet pci=0
  io-net -i1 -dpcnet pci=1 -ptcpip prefix=/alt
  ```

  Note that the **io-net** versions of the utilities must be present on the target (assumed, for this example, to have been placed in a separate directory) and run with the **io-net** stack. For example:

  ```
  SOCK=/alt /io-net/ifconfig en0 192.168.1.2
  SOCK=/alt /io-net/inetd
  ```

Discontinued features

The **io-pkt** networking stack doesn't support the following features:

- **npm-qnet-compat.so** (Neutrino 6.2.1 Qnet compatibility protocol)
- **npm-ttcpip.so** (tiny TCP/IP) stack

Using `pfil` hooks to implement an `io-net` filter

We recommend that you use **pfil** hooks to rewrite your **io-net** filter to work in **io-pkt**. Here are the basic steps:

- Change your entry point function to remove the second argument (**dispatch_t**) and change the third option to be **struct _iopkt_self**.

- Remove the **io_net_dll_entry_t** and replace it with the appropriate version of:

  ```
  struct _iopkt_lsm_entry IOPKT_LSM_ENTRY_SYM(mod) =
  IOPKT_LSM_ENTRY_SYM_INIT(mod_entry);
  ```

- The *rx_up* and *rx_down* functions are essentially **pfil_hook** entries with a flag of PFIL_IN and PFIL_OUT, respectively.

  Information about the interface that the packet was received on is contained in the **ifp** pointer (defined in **usr/include/net/if.h**). For example, the external interface name is in **ifp->if_xname**; you can use this to determine where the packet came from.

  The *cell*, *endpoint*, and *iface* parameters are essentially wrapped up in the **ifp** pointer.

- There are no advertisement packets used within **io-pkt**. For information about interfaces being added, removed, or reconfigured, you can add an interface hook as shown in the sample code in "Packet Filters" in the Packet Filtering chapter in this guide.

- Buffering in **io-pkt** is handled using a different structure than in **io-net**. The **io-net** stack uses **npkt_t** buffers, whereas **io-pkt** uses the standard **mbuf** buffers. You have to modify your **io-net** code to deal with the different buffer format.

  You can find information about using **mbuf** buffers in many places on the web. The header file that covers the **io-pkt mbuf** support is in **$QNX_TARGET/usr/include/sys/mbuf.h**.

- The *shutdown1* and *shutdown2* routines are somewhat similar to the "remove_hook" options, in which the filtering functions are removed from the processing stream. Typically, a filter requiring interaction with the user (e.g. read, write, umount) would export a resource manager interface to provide a mechanism for indicating that the filter has to be removed. You can then use *pfil_remove_hook()* calls after cleaning up to remove the functions from the data path.

- There isn't an equivalent of *tx_done* for **pfil**. In **io-net**, buffers are allocated by endpoints (e.g. a driver or a protocol) and therefore an endpoint-specific function for freeing buffers needs to be called to release the buffer back to the endpoint's buffer pool.

  In **io-pkt**, all buffer allocation is handled explicitly by the stack middleware. This means that any element requiring a buffer goes directly to the middleware to get it, and anything freeing a buffer (e.g. the driver) puts it directly back into the middleware buffer pools. This makes things easier to deal with, because you now no longer have to track and manage your own buffer pools as an endpoint. As soon as the code is finished with the buffer, it simply performs an *m_freem()* of the buffer which places it back in the general pool.

There is one downside to the global buffer implementation. You *can't* create a thread using *pthread_create()* that allocates or frees a buffer from the stack, because the thread has to modify internal stack structures. The locking implemented within **io-pkt** is optimized to reduce thread context-switch times, and this means that non-stack threads can't lock-protect these structures. Instead, the stack provides its own thread-creation function (defined in **trunk/sys/nw_thread.h**):

```
int nw_pthread_create(pthread_t *tidp,
                      pthread_attr_t *attrp,
                      void *(*funcp)(void *),
                      void *arg,
                      int flags,
                      int (*init_func)(void *),
                      void *init_arg);
```

The first four arguments are the standard arguments to *pthread_create()*; the last three are specific to **io-pkt**. You can find a fairly simplistic example of how to use this function in **net/ppp_tty.c**.

- In terms of transmitting packets, the **ifp** interface structure contains the *if_output* function that you can use to transmit a user-built Ethernet packet. The *if_output* maps to *ether_output* for an Ethernet interface. This function queues the **mbuf** to the driver queue for sending, and subsequently calls *if_start* to transmit the packet on the queue. The preliminary queuing is implemented to allow traffic shaping to take place on the interface.

Glossary

AES

An abbreviation for **Advanced Encryption Standard)**.

BPF

An abbreviation for **Berkley Packet Filter**.

BSS

An abbreviation for **Basic Service Set**. Also known as **Infrastructure Mode**.

CA

An abbreviation for **Certification Authority**.

EAP-TLS

An abbreviation for **Extensible Authentication Protocol - Transport Layer Security**.

IBSS

An abbreviation for **Independent Basic Service Set**.

`mbuf`

An abbreviation for **memory buffer**, the internal representation of a packet used by NetBSD and `io-pkt`.

NAT

An abbreviation for **Network Address Translation**.

`npkt`

The name of the internal representation of a packet used by `io-net`.

SA

An abbreviation for **Security Association**.

SOHO

An abbreviation for **Small Office, Home Office**.

SPD

An abbreviation for **security policy database**.

spoofing

The faking of IP addresses, typically for malicious purposes.

SSID

An abbreviation for **Service Set Identifier**.

STP

An abbreviation for **Spanning Tree Protocol**.

TDP

An abbreviation for **Transparent Distributed Processing**. Neutrino's native networking (Qnet) that lets you access resources on other Neutrino systems as if they were on your own machine.

TKIP

An abbreviation for **Temporal Key Integrity Protocol**.

TLS

An abbreviation for **Transport Layer Security**.

TTLS

An abbreviation for **Tunneled Transport Layer Security**.

UDP

An abbreviation for **User Datagram Protocol**.

WAP

An abbreviation for **Wireless Access Point**. Also known as a **base station**.

WEP

An abbreviation for **Wired Equivalent Privacy**.

WLAN

An abbreviation for **Wireless Local Area Network**.

WPA

An abbreviation for **Wi-Fi Protected Access**.

Index

!

A

B

C

J

K

L

M

N

R

S

T

U

W