QNX® Momentics® PE DDK

Network Devices

For targets running QNX® Neutrino® 6.3.0

Printed under license by:

QNX Software Systems Co.
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/**

Publishing history

Electronic edition published 2005.

Technical support options

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website (**www.qnx.com**). You'll find a wide range of support options, including our free web-based **Developer Support Center**.

Contents

List of Figures

About the Network DDK

What you'll find in this guide

The following table may help you find information quickly:

For information about:	See:
`io-net`, drivers, and protocols	Introduction to the Network Subsystem
Network drivers	Writing Your Own Driver
Data structures, code, etc.	Network DDK API

You must use this DDK with QNX Neutrino 6.3.0 or later.

Building DDKs

You can compile the DDK from the IDE or the command line.

- To compile the DDK from the IDE:

  Please refer to the Managing Source Code chapter, and "QNX Source Package" in the Common Wizards Reference chapter of the *IDE User's Guide*.

- To compile the DDK from the command line:

  Please refer to the release notes or the installation notes for information on the location of the DDK archives.

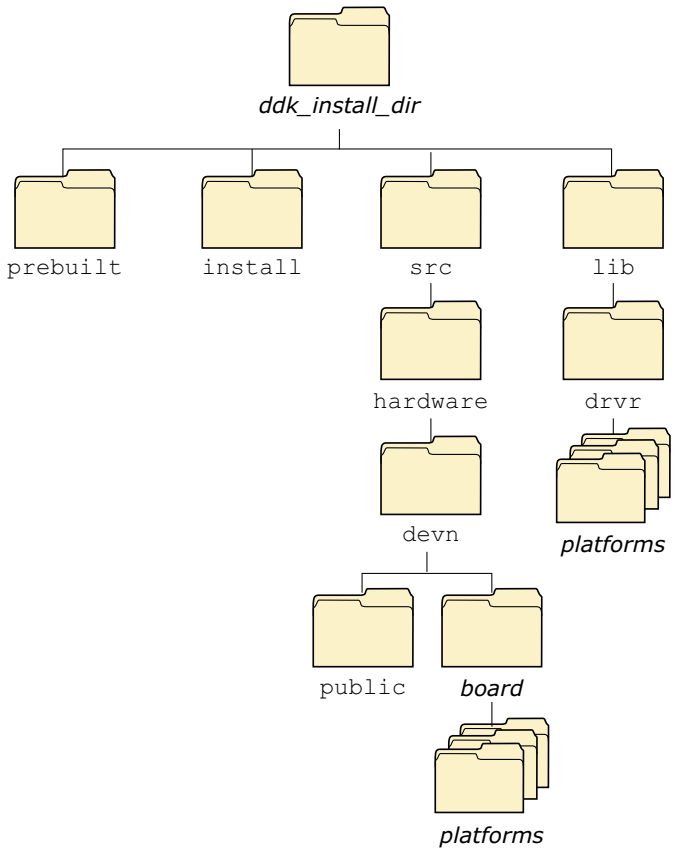
  DDKs are simple zipped archives, with no special requirements. You must manually expand their directory structure from the archive. You can install them into whichever directory you choose, assuming you have write permissions for the chosen directory.

  Historically, DDKs were placed in **`/usr/src/ddk_VERSION`** directory, e.g. **`/usr/src/ddk-6.2.1.`** This method is no longer required, as each DDK archive is completely self-contained.

  The following example indicates how you create a directory and unzip the archive file:

```
# cd ~
# mkdir my_DDK
# cd my_DDK
# unzip /path_to_ddks/ddk-device_type.zip
```

The top-level directory structure for the DDK looks like this:



Directory structure for this DDK.

You must run:

```
. ./setenv.sh
```

before running **make**, or **make install**.

Additionally, on Windows hosts you'll need to run the **Bash** shell (**bash.exe**) before you run the **. ./setenv.sh** command.

If you fail to run the **. ./setenv.sh** shell script prior to building the DDK, you can overwrite existing binaries or libs that are installed in **$QNX_TARGET**.

Each time you start a new shell, run the **. ./setenv.sh** command. The shell needs to be initialized before you can compile the archive.

The script will be located in the same directory where you unzipped the archive file. It must be run in such a way that it modifies the current shell's environment, not a sub-shell environment.

In **ksh** and **bash** shells, All shell scripts are executed in a sub-shell by default. Therefore, it's important that you use the syntax

```
. <script>
```

which will prevent a sub-shell from being used.

Each DDK is rooted in whatever directory you copy it to. If you type **make** within this directory, you'll generate all of the buildable entities within that DDK no matter where you move the directory.

all binaries are placed in a scratch area within the DDK directory that mimics the layout of a target system.

When you build a DDK, everything it needs, aside from standard system headers, is pulled in from within its own directory. Nothing that's built is installed outside of the DDK's directory. The makefiles shipped with the DDKs copy the contents of the **prebuilt** directory into the **install** directory. The binaries are built from the source using include files and link libraries in the **install** directory.

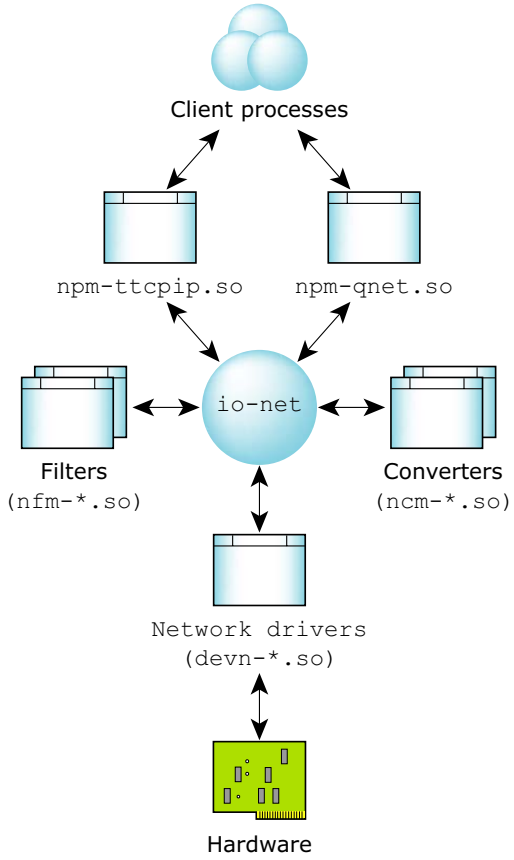
Chapter 1

Introduction to the Network Subsystem

In this chapter. . .

Overview of io-net and the networking subsystem

The QNX Neutrino network subsystem consists of a process called **`io-net`** that loads a number of shared objects. These shared objects typically consist of a protocol stack (such as **`npm-tcpip.so`**), a network driver, and other optional components such as filters and converters. These shared objects are arranged in a hierarchy, with the end user on the top, and hardware on the bottom.



The **io-net** *component can load one or more protocol interfaces and drivers.*

This document focuses on writing new network drivers, although most of the information applies to writing any module for **io-net**.

As indicated in the diagram, the shared objects that **io-net** loads don't communicate directly. Instead, each shared object registers a number of functions that **io-net** calls, and **io-net** provides functions that the shared object calls.

Each shared object provides one or more of the following types of service:

Up producer	Produces data for a higher level (e.g. an Ethernet driver provides data from the network card to a TCP/IP stack).
Down producer	Produces data for a lower level (e.g. the TCP/IP stack produces data for an Ethernet driver).
Up filter	A filter that sits between an up producer and the bottom end of a converter (e.g. a protocol sniffer).
Down filter	A filter that sits between a down producer and the top end of a converter (e.g. Network Address Translation, or NAT).
Converter	Converts data from one format to another (e.g. between IP and Ethernet)

Note that these terms are relative to **`io-net`** and don't encompass any non-**`io-net`** interactions.

For example, a network card driver (while forming an integral part of the communications flow) is viewed only as an up producer as far as **`io-net`** is concerned — it doesn't *produce* anything that **`io-net`** interacts with in the downward direction, even though it actually transmits the data originated by an upper module to the hardware.

A producer can be an up producer, a down producer, or both. For example, the TCP/IP module produces both types (up and down) of packets.

When a module is an up producer, it may pass packets on to modules above it. Whether a packet originated at an up producer, or that producer received the packet from another up producer below it, from the next recipient's point of view, the packet came from the up producer directly below it.

Connecting modules

Only an up or down producer can connect with converters. A converter can't connect directly to another converter.

For example, in a PPP (Point-to-Point Protocol) over Ethernet implementation, we already have an IP producer (the stack) and an IP-PPP converter (the **npm-pppmgr.so** module). A PPP-EN converter is needed to convert a PPP frame into an Ethernet frame. However, since two converters can't be directly connected to each other, a PPP producer is needed to bridge converters.

The **npm-pppoe.so** module registers with **io-net** twice: once as a PPP producer, and a second time as a PPP-EN converter. This PPP producer serves as a "dummy" module and serves only to pass packets. This way the packets can go from IP producer (**npm-tcpip.so**) to EN producer (**devn-xxx.so**)

This complete sequence of events is as follows: the packets run a chain from an IP producer (**npm-tcpip.so**) to an IP-PPP converter (the **npm-pppmgr.so** module) to the dummy PPP producer (**npm-pppoe.so**) to the PPP-EN converter (**npm-pppoe.so**) to an EN producer (**devn-xxx.so**), to the Internet.

Threading

The **io-net** module exists in a multi-thread environment which does not create any threads when loading a module unless a module using *pthread_create()* creates threads on its own.

If a module is using *pthread_create()*, the thread is an execution entity when the second thread in **io-net** tries to send an IP packet through the Point-to-Point over Ethernet (PPPoE) interface. The thread will call **io-net**'s *tx_down()* function, which calls into **npm-pppmgr.so**'s *rx_down()* function. After this function converts an IP packet to a PPP frame, it calls **io-net**'s *tx_down()*, then that function calls into *rx_down()* of the dummy PPP producer in **npm-pppoes.so**.

From the perspective of the functionality of a module, the module is exposed to all **io-net** function calls, which could allow two

functions to access the same data structures. It's important that the module is protected using a mutex or another synchronous object.

When calling the initialize function on a module, **io-net** passed in a dispatch handler (dpp). This dispatcher handles all the path names **io-net** created (**/dev/io-net/***). There is a thread pool associated with this function. The maximum number of threads in the thread pool is controlled by the **-t** option of **io-net**. As the thread pool is created and destroyed dynamically, it's common for **pidin -p io-net** to have noncontinuous thread IDs.

A module can create its own path name with the dpp, and have its private resource manager. For example, the **npm-pppmgr.so** module can attach a **/dev/socket/pppmgr** call which gives out statistics while being queried.

Starting io-net

When you start **io-net** from the command line, you tell it which drivers and protocols to load:

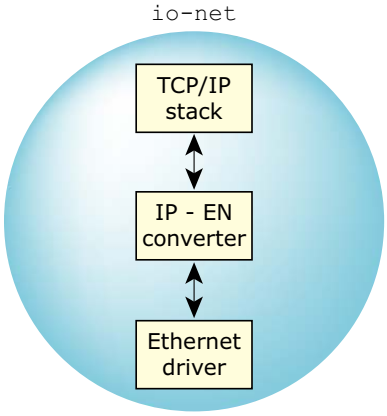
```
$ io-net -del900 verbose -pttcpip if=en0:11.2 &
```

This causes **io-net** to load the **devn-el900.so** Ethernet driver and the tiny TCP/IP protocol stack. The **verbose** and **if=en0:11.2** options are suboptions that are passed to the individual components.

Alternatively, you can use the **mount** and **umount** commands to start and stop modules dynamically. The previous example could be rewritten as:

```
$ io-net &
$ mount -Tio-net -overbose devn-el900.so
$ mount -Tio-net -oif=en0:11.2 npm-ttcpip.so
```

Regardless of the way that you've started it, here's the "big picture" that results:



Big picture of **io-net**.

In the diagram above, we've shown **io-net** as the "largest" entity. This was done simply to indicate that **io-net** is responsible for loading all the other modules (as shared objects), and that it's the one that "controls" the operation of the entire protocol stack.

Let's look at the hierarchy, from top to bottom:

TCP/IP stack — This is at the top of the hierarchy, as it presents a user-accessible interface. A user typically uses the socket library function calls to access the exposed functionality. (The mechanism used by the TCP/IP stack to present its interface isn't defined by **io-net** — it's a private interface that **io-net** has no knowledge of or control over.)

IP-EN converter

In order to use the Ethernet interface, the TCP/IP stack needs the services of a converter module to add/remove the Ethernet header. As we'll see, this isolation of hardware specifics from the down producer allows for easy addition of future hardware types. It also allows for the insertion of

filter modules between the down producer and the converter, or between the converter and the up producer. In this case, the IP-EN converter basically provides ARP (Address Resolution Protocol) services.

Ethernet driver — At the lowest level, there's an Ethernet driver that accepts Ethernet packets (generated by the IP module), and sends them out the hardware (and the reverse: it receives Ethernet packets from the hardware and gives them to the IP module).

As far as Neutrino's namespace is concerned, the following entries exist:

/dev/io-net

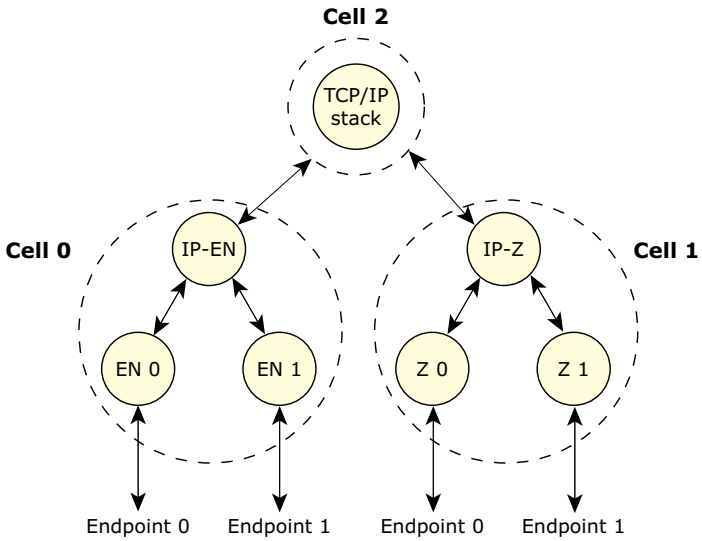
The main device created by **io-net** itself.

/dev/io-net/en*N*

The Ethernet device corresponding to LAN *N* (where *N* is 0 in our example).

At this point, you could *open()* **/dev/io-net/en0**, for example, and perform *devctl()* operations on it — this is how the **nicinfo** command gets the Ethernet statistics from the driver.

Here's another view of **io-net**, this time with two different protocols at the bottom:



Cells and endpoints.

As you can see, there are three levels in this hierarchy. At the topmost level, we have the TCP/IP stack. As described earlier, it's considered to be a down producer (it doesn't produce or pass on anything for modules above it.)

In reality, the stack probably registers as both an up *and* down producer. This is permitted by **io-net** to facilitate the stacking of protocols.

When the TCP/IP stack registered, it told **io-net** that it produces packets in the downward direction of type IP — there's no other binding between the stack and its drivers. When a module registers, **io-net** assigns it a cell number, 2 in this case.

Joining the stack (down producer) to the drivers (up producers), we have two converter modules. Take the converter module labeled IP-EN as an example. When this module registered as type _REG_CONVERTOR, it told **io-net** that it takes packets of type IP on top and packets of type EN on the bottom.

Again, this is the only binding between the IP stack and its lower level drivers. The IP-EN portion, along with its Ethernet drivers, is called cell 0 and the IP-Z portion, along with its Z-protocol drivers is called cell 1 as far as **io-net** is concerned.

The purpose of the intermediate converters is twofold:

1. It allows for increased flexibility when adding future protocols or drivers (simply write a new converter module to connect the two).

2. It allows for filter modules to be inserted either above or below the converter.

   Finally, on the bottom level of the hierarchy, we have two different Ethernet drivers and two different Z-protocol drivers. These are up producers from **io-net**'s perspective, because they generate data only in the upward direction. These drivers are responsible for the low-level hardware details. As with the other components mentioned above, these components advertise themselves to **io-net** indicating the name of the service that they're providing, and that's what's used by **io-net** to "hook" all the pieces together.

   Since all seven pieces are independent shared objects that are loaded by **io-net** when it starts up (or later, via the **mount** command), it's important to realize that the keys to the interconnection of all the pieces are:

   - the module type
   - the type of packet they produce or accept on the way up and down.

The life cycle of a packet

The next thing we need to look at is the life cycle of a packet — how data gets from the hardware to the end user, and back to the hardware.

The main data structure that holds all packet data is the **npkt_t** data type. (For more information about the data structures described in this

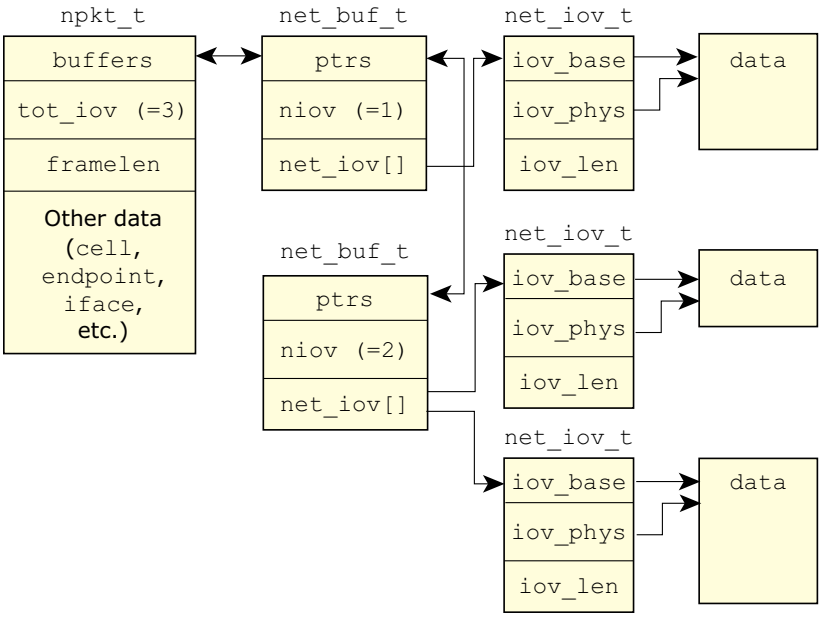
section, see the Network DDK API chapter.) The **npkt_t** structure maintains a *tail queue* of buffers that contain the packet's data.

A tail queue uses a pair of pointers, one to the head of the queue and the other to the tail. The elements are doubly linked; an arbitrary element can be removed without traversing the queue. New elements can be added before or after an existing element, or at the head or tail of the queue. The queue may be traversed only in the forward direction.

The buffers form a doubly-linked list, and are managed via the *TAILQ* macros from **<sys/queue.h>**:

- *TAILQ_EMPTY()*
- *TAILQ_FIRST()*
- *TAILQ_INSERT_AFTER()*
- *TAILQ_INSERT_BEFORE()*
- *TAILQ_INSERT_HEAD()*
- *TAILQ_INSERT_TAIL()*
- *TAILQ_LAST()*
- *TAILQ_NEXT()*
- *TAILQ_PREV()*
- *TAILQ_REMOVE()*

Buffer data is stored in a **net_buf_t** data type. This data type consists of a list of **net_iov_t** structures, each containing a virtual (or base) address, physical address, and length, that are used to indicate one or more buffers:



Data structures associated with a packet.

The *TAILQ* macros let you step through the list of elements. The following code snippet illustrates:

```
net_buf_t *buf;
net_iov_t *iov;
int       i;

// walk all buffers
for (buf = TAILQ_FIRST (&npkt -> buffers); buf;
                       buf = TAILQ_NEXT (buf, ptrs)) {
    for (i = 0, iov = buf -> net_iov; i < buf ->
         niov; i++, iov++) {
        // buffer is        :  iov -> iov_base
        // length is        :  iov -> iov_len
        // physical addr is :  iov -> iov_phys
    }
}
```

Going down

We'll start with the downward direction (from the end user to the hardware). A message is sent from the end user (via the socket library), and arrives at the TCP/IP stack. The TCP/IP stack does whatever error checking and formatting it needs to do on the data. At some point, the TCP/IP stack sends a fully formed IP packet down **io-net**'s hierarchy. No provision is made for any link-level headers, as this is the job of the converter module.

Since the TCP/IP stack and the other modules aren't bound to each other, it's up to **io-net** to do the work of accepting the packet from the TCP/IP stack and giving it to the converter module. The TCP/IP stack informs **io-net** that it has a packet that should be sent to a lower level by calling the *tx_down()* function within **io-net**. The **io-net** manager looks at the various fields in the packet and the arguments passed to the function, and calls the *rx_down()* function in the IP-EN converter module.

The *contents* of the packet aren't copied — since all these modules (e.g. the TCP/IP stack and the IP module) are loaded as shared objects into **io-net**'s address space, all that needs to be transfered between modules is pointers to the data (and not the data itself).

Once the packet arrives in the IP-EN converter module, a similar set of events occurs as described above: the IP-EN converter module converts the packet to an Ethernet packet, and sends it to the Ethernet module to be sent out to the hardware. Note that the IP-EN converter module needs to add data in front of the packet in order to encapsulate the IP packet within an Ethernet packet.

Again, to avoid copying the packet data in order to insert the Ethernet encapsulation header in front of it, only the data pointers are moved. By inserting a **net_buf_t** at the start of the packet's queue, the Ethernet header can be prepended to the data buffer without actually copying the IP portion of the packet that originated at the TCP/IP stack.

Going up

In the upward direction, a similar chain of events occurs:

- The Ethernet driver receives data from its hardware, and allocates one or more packets into which it places the data. (For efficiency, it may use memory-mapping tricks to cause the hardware to directly place the packet into a preallocated area.)
- The Ethernet driver then calls **io-net**'s *tx_up()* function, telling it that it has a packet that's ready to be given to a higher level.
- The **io-net** manager figures out which module should get the packet and calls that module's *rx_up()* function. In our example, this is the IP-EN converter module, as it now needs to look at the packet and get at just the IP portion (the packet arrived from the hardware with Ethernet encapsulation).

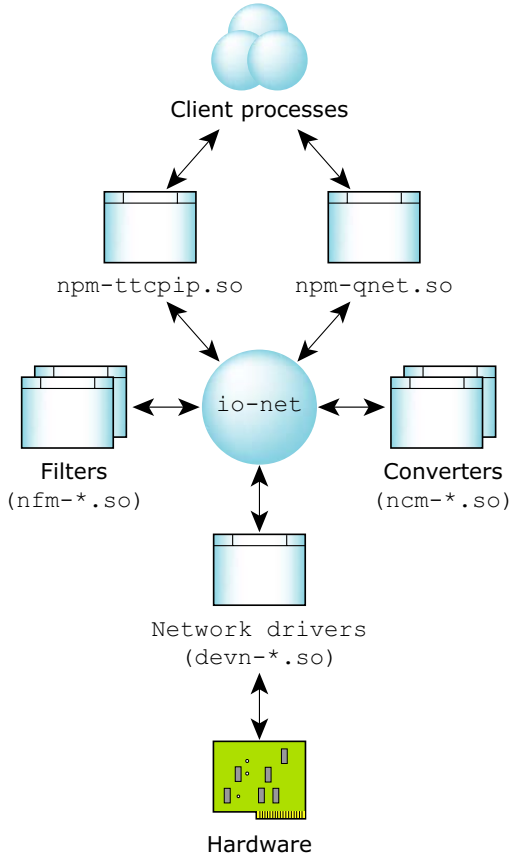
Note that in an upward-headed packet, data is *never* added to the packet as it travels up to the various modules, so the list of **net_buf_t** structures isn't manipulated. For efficiency, the arguments to **io-net**'s *tx_up()* function (and correspondingly to a registered module's *rx_up()* function) include *off* and *framelen_sub*. These are used to indicate how much of the data within the buffer is of interest to the level to which it's being delivered.

For example, when an IP packet arrives over the Ethernet, there are 14 bytes of Ethernet header at the beginning of the buffer. This Ethernet header is of no interest to the IP module — it's relevant only to the Ethernet and IP-EN converter modules. Therefore, the *off* argument is set to 14 to indicate to the next higher layer that it should ignore the first 14 bytes of the buffer. This saves the various levels in **io-net** from continually having to copy buffer data from one format to another.

The *framelen_sub* argument operates in a similar manner, except that it refers to the tail end of the buffer — it specifies how many bytes should be ignored at the end of the buffer, and is used with protocols that place a tail-end encapsulation on the data.

See **io-net** in the *Utilities Reference* for more details.

The purpose of the network driver is to detect and initialize one or more NIC (Network Interface Controller) devices, and allow for transmission and reception of data via the NIC. Additional tasks typically performed by a network driver include link monitoring and statistics gathering.



The networking subsystem and **`io-net`**.

Driver initialization

The network driver is loaded by **io-net**. This happens either when **io-net** starts, or later in response to a "mount" request.

A network driver must contain a global structure called **io_net_dll_entry**, of type **io_net_dll_entry_t**. The **io-net** process finds this structure by calling *dlsym()*. This structure must contain a function pointer to the driver's main initialization routine, which **io-net** calls when the driver has been loaded. This function is responsible for parsing the option string that was passed to the driver (if any), and detecting any network interface hardware, in accordance with the supplied options. For each NIC device that the driver detects, it creates a software instance of the interface, by allocating the necessary structures, and registering the interface with **io-net**. After it registers with **io-net**, the driver advertises its capabilities to the other components within the networking subsystem.

By default, the driver should attempt to detect and instantiate every NIC in the system that the driver supports. However, the driver may be requested to instantiate a specific NIC interface, via one or more driver options.

Sometimes, certain options are mandatory. For example, in the case of a non-PCI device, the driver may not be able to automatically determine the interrupt number and base address of the device. Also, on many embedded systems, their driver may not be able to determine the station (MAC) address of the device. In this case, the MAC address needs to be passed to the driver via an option.

Device detection

After parsing the options, the driver knows how and where to look for NIC devices.

For a PCI device, the driver typically searches using *pci_attach_device()*. The driver searches based on the values specified via the **vid**, **did**, and **pci** options. If no options were specified, the driver will typically search based on a well-known internal list of PCI

Vendor and Device IDs that correspond to the devices for which the driver was developed.

For non-PCI devices, the driver usually relies on a memory or I/O base address being specified in order to locate the NIC device. On certain systems, the driver may be able to find the device at well-known locations, without the need for the location to be specified via an option. The driver will then typically do some sanity checks to verify that an operational device indeed exists at the expected location.

Device instantiation

Once it's been determined that a NIC device is present, the driver initializes and configures the interface. It's always a good idea for the driver to reset the device before proceeding, since the device could be in an unknown state (e.g. in the case of a previous incarnation of **`io-net`** terminating prematurely without getting a chance to shut off the device properly).

Next, the driver typically allocates some structures in order to store information about the device state. The normal practice is to allocate a driver-specific structure, whose layout is known only to the network driver. The driver may pass a pointer to this structure to other networking subsystem functions. When calls are made into the driver's entry points, the subsystem passes this pointer, so the driver always has access to any data associated with the device. It's a good idea for the driver to store any configuration-related data in the **`nic_config_t`** structure so that the driver can take advantage of more of the functionality in **`libdrvr`**. The **`nic_config_t`** structure can be included in the driver-specific structure.

Also, if higher-level software queries the driver for configuration information, it will expect the information to be in the format defined by the **`nic_config_t`** structure, so it's convenient to keep a copy of this structure around.

Avoid the use of global variables in your driver if possible. Referencing global variables is much slower than accessing data that resides in a structure that the system allocated (e.g. *calloc()*).

Network drivers generally use an interrupt handler to receive notification of events such as packet reception. We *strongly* recommended that network drivers use the *InterruptAttachEvent()* call instead of *InterruptAttach()* to handle interrupts. In an RTOS, we must keep the amount of time spent in an interrupt handler to an absolute minimum so as not to negatively impact the overall realtime determinism of the system. Therefore, the type of operations performed by network drivers, such as copying packet data or traversing linked lists or iterating ring buffers, should be performed at process time.

The driver normally creates a thread during initialization, to handle events (such as interrupt events, timer events etc.). Note that you need to be extra careful, since multiple threads could simultaneously call into your driver. In addition, the driver's own event thread could be running, so it's very important that all data and device entities are protected (e.g. using mutexes). Make sure you are familiar with multi-threaded programming concepts before attempting to write a network driver.

Once the device is initialized and is ready to be made operational, the driver registers the interface with **`io-net`**. See the *reg* field of the **`io_net_self_t`** structure for details on how to register with **`io-net`**. When the driver registers, it provides various information to **`io-net`**, to allow the networking subsystem to call into the driver's various entry points.

After registering with **`io-net`**, the driver must advertise its capabilities to the networking subsystem. See the *dl_advert* field of the **`io_net_registrant_funcs_t`** structure for more details.

At this point, the device is ready to begin operation. The driver's entry points may now be called at any time to perform various tasks such as packet transmission. The driver may also call back into the

networking subsystem, for example, to deliver packets that have been received from the medium.

Chapter 2

Writing a Network Driver

In this chapter. . .

In this chapter, we look at the work that you must do to write a driver for your own network interface controller.

The network driver interface

This section describes the interface between a network driver and the rest of the networking subsystem.

Driver initialization

Once the driver is loaded, **io-net** looks for a global structure, which must be present in every network driver. The structure must be named **io_net_dll_entry**, and it must be of type **io_net_dll_entry_t**. The **io_net_dll_entry_t** structure is declared in **<sys/io-net.h>** and its members are as follows:

```
int nfuncs;
int (*init)(void *dll_hdl,
    dispatch_t *dpp, io_net_self_t *ion, char *options);
int (*shutdown) (void *dll_hdl);
```

The *nfuncs* variable should be set to the number of function pointers in this structure, that the driver knows about. Set its initial value to 2.

The *init* function pointer should point to the primary network driver entry point. The **io-net** command will call this entry point once for every *-d* argument to **io-net**, and once for every subsequent attempt to load a network driver via the "mount" interface. Its arguments are as follows:

- *dll_hdl*

  This parameter will be needed when the driver wants to register an interface with **io-net**.

- *dpp*

  This parameter should be ignored.

- *ion*

  This points to an **io_net_self_t** structure. This structure contains function pointers that allow the driver to interact with the

networking subsystem. The driver should always keep a copy of this pointer.

- *options*

  This points to an ASCII string, which, if non-NULL, should be parsed by the driver. See the *Driver option definitions* section for more details on driver options.

The *shutdown* function will be called before the driver is unloaded from memory. Note that before this happens, each active interface that was instantiated by the driver will have been individually shut down, so typically this function has nothing to do. However, it may be necessary to use this entry point to do additional cleanup, to ensure that any resources that we allocated during the lifetime of the driver, have been de-allocated. If the driver doesn't need to use this entry point, it should set the *shutdown* field to NULL.

Option parsing

One of the parameters to the driver's main initialization entry point is a pointer to a character string. This pointer may be NULL, or it may point to an ASCII string of driver options. The option string is in a form that is parseable by the *getsubopt()* function. Some of the options, by convention, have a standard meaning that is consistent across all network drivers. These options can be parsed by the *nic_parse_options()* function. A driver may also support other options which do not have a standardized meaning.

If the driver uses the *nic_parse_options()* function to do option parsing, the **`nic_config_t`** structure is used to store the results.

See the *Driver option definitions* section for definitions of the options that have a standardized meaning.

Calling back into the networking subsystem

The **`io_net_self_t`** structure, declared in **`<sys/io-net.h>`**, contains pointers to functions that allow the driver to interact with the networking framework. Each of the supported functions is described below in detail:

```
int (*reg)(void *dll_hdl, io_net_registrant_t *registrant,
        int *reg_hdlp, uint16_t *cell, uint16_t *endpoint);
```

This function registers an interface with **io-net**. It should be called once for each NIC interface that the driver wishes to instantiate. This function must be called before any of the other functions in the **io_net_self_t** structure. Its arguments are as follows:

- *dll_hdl*

  Specifies the value that was passed into the primary driver entry point.

- *registrant*

  Points to a structure that contains information as to how the device should be instantiated, as well as a pointer to a table of additional driver entry points. The **io_net_registrant_t** structure is described here .

- *reg_hdlp*

  The driver should specify a pointer to an integer. A handle will be stored at this location, which will be used in subsequent calls to the networking subsystem.

- *cell*

  The driver should specify a pointer to a 16-bit variable. A value will be stored at this location, which the driver will later use when delivering received packets to the upper layers.

- *endpoint*

  This is actually the interface number (LAN number). The driver should specify a pointer to a 16-bit variable. A value will be stored at this location, which the driver will later use when delivering received packets to the upper layers. Typically, io-net will decide how the LAN number is chosen, but it is possible for the driver to influence how the LAN number is chosen, as described in the **io_net_registrant_t** section.

Data packets

There are two types of packets, data packets and message packets. A network driver typically deals only with data packets, with one exception. After a driver registers an interface with **io-net**, the driver will construct a message packet that encapsulates a structure of type **io_net_msg_dl_advert_t**, and send it upstream in order to advertise the interfaces capabilities to the other components within the networking subsystem.

A packet consists of an **npkt_t** structure, which has one or more data buffers associated with it. The **npkt_t** structure is defined in **<sys/io-net.h>**.

If you're using the new lightweight Qnet, a network driver developed with the QNX Neutrino 6.2 release could malfunction because the assignment of the bits in the *flags* field of the **npkt_t** structure has changed. See _NPKT_ORG_MASK and _NPKT_SCRATCH_MASK in **<sys/io-net.h>**.

The driver can use the eight most significant bits while it's processing a packet. The driver shouldn't make assumptions about the state of these bits when it receives a packet from the upper layers.

The next four most significant bits are for the use of the originator of a packet. The driver can use these flags for packets being sent upstream. If a packet didn't originate with the driver, the driver must not alter these flags.

If the driver wants to create a packet to send upstream, it should call *alloc_up_npkt()*.

A data buffer is described by a structure of type **net_buf_t**, as defined in **<sys/io-net.h>**. The data in a buffer is comprised of one or more contiguous fragments. Each fragment is described by a **net_iov_t** structure (also defined in **<sys/io-net.h>**), which contains a pointer to the fragment's data, the size of the fragment, and the physical address of the fragment.

The following fields of the **npkt_t** structure are of importance to the network driver:

- *buffers*

  Points to a queue of data buffers. The buffer queues can be manipulated and traversed by a set of macros defined in **<sys/queue.h>**. See below for examples of the kind of operations a driver would need to perform on buffer queues.

- *next*

  Used for chaining packets into a linked list. The last item in the list is set to NULL.

- *org_data*

  For the sole use of the originator of the packet. The driver should only modify or interpret this field if the driver was the originator of the packet.

- *flags*

  The logical OR of zero or more of the following:

  _NPKT_NOT_TXED
  : If the driver couldn't transmit a packet, for whatever reason, it should set this flag before calling *tx_done* to indicate that the packet is known to have been dropped.

  _NPKT_UP
  : Should be set for packets originating from the driver.

  _NPKT_MSG
  : Indicates that the packet does not contain data, but rather contains a message. A driver will set this flag when it is sending a capabilites advertisement message upstream. The upper 12 bits of the flags field are reserved for the driver's own internal purposes.

- *framelen*

  The total size of the packet data, in bytes, including the Ethernet header.

- *tot_iov*

  The total number of fragments which comprise the packet data.

- *csum_flags*

  Used for hardware checksum offloading. See the *Hardware checksum offloading* section for a full description of the hardware checksum offloading interface.

- *ref_cnt*

  For packets originating from the driver, this should be set to 1.

- *req_complete*

  For packets originating from the driver, this should be set to 0.

A queue of structures of type **`net_buf_t`** is used to describe the data fragments that are associated with the packet. The members of this structure are as follows:

- *ptrs*

  Use by the queue manipulation macros to create queues of buffers.

- *niov*

  The number of data fragments associated with the buffer.

- *net_iov*

  Points to an array of data fragment descriptors.

The members of the **`net_iov_t`** structure are as follows:

- *iov_base*

  Points to the data fragment.

- *iov_phys*

  The physical address of the data fragment.

- *iov_len*

  The size of the data fragment, in bytes.

In order to traverse all of the data fragments associated with a packet (e.g. when transmitting a packet), the driver should use the TAILQ_FIRST and TAILQ_NEXT macros. The following example shows how a driver could traverse an entire packet in order to copy the data into a contiguous buffer:

```
#include <sys/io-net.h >

void
defrag(npkt_t *npkt, uint8_t *dst)
{
    net_iov_t   *iov;
    net_buf_t   *buf;
    int     i;

    for (buf = TAILQ_FIRST(& npkt->buffers); buf != NULL;
        buf = TAILQ_NEXT(buf, ptrs)) {
        for (i = 0, iov = buf->net_iov; i < buf->niov; i++, iov++) {
            memcpy(dst, iov->iov_base, iov->iov_len);
            dst += iov->iov_len;
        }
    }
```

When constructing a packet to be sent upstream, the driver will need to associate one or more fragments of data with a packet. The following example shows how a driver could use the TAILQ_INSERT_HEAD macro to create a packet and associate a piece of contiguous data with the packet:

```
#include <sys/io-net.h >

npkt_t *
make_packet(io_net_self_t *ion, uint8_t *data_ptr, int data_len)
{
    npkt_t      *npkt;
    net_buf_t   *nb;
    net_iov_t   *iov;

    /*
     * Allocate the npkt_t structure, along with extra memory
     * to store the net_buf_t and the net_iov_t
     */
    if ((npkt = ion->alloc_up_npkt(sizeof(net_buf_t) +
        sizeof(net_iov_t), (void **)& nb)) == NULL)
        return NULL;
```

```
    /* Get a pointer to the net_iov_t, which follows the net_buf_t */
    iov = (net_iov_t *)(nb + 1);

    /* Associate a buffer with the packet */
    TAILQ_INSERT_HEAD(& npkt->buffers, nb, ptrs);

    nb->niov = 1;
    nb->net_iov = iov;

    iov->iov_base = data_ptr;
    iov->iov_len = data_len;
    iov->iov_phys = ion->mphys(iov->iov_base);

    npkt->flags = _NPKT_UP
    npkt->org_data = data_ptr;
    npkt->next = NULL;
    npkt->tot_iov = 1;
    npkt->ref_cnt = 1;
    npkt->req_complete = 0;

    return npkt;
}
```

Note that we used the *alloc_up_npkt()* function to allocate the memory for the **npkt_t** structure, and the associated **net_buf_t** and **net_iov_t** structures, all at once. When the packet is no longer needed, this memory must all be freed all at once.

Advertising device capabilities

Once a driver has registered an interface with **io-net**, it must then advertise the device's capabilities by sending a special type of message upstream. This message should contain a single buffer and a single fragment of data. The data portion should contain a message in the format defined by the **io_net_msg_dl_advert_t** structure, as defined in **<sys/io-net.h>**.

The driver should zero-out the entire structure, then initialize the members as described in **io_net_msg_dl_advert_t**.

Here is an example of a function that creates a capabilities advertisement structure and sends it upstream. Note that it uses the *make_packet()* function from a previous code example:

```
#include <stdlib.h >
```

```
#include <net/if.h >
#include <net/if_types.h >
#include <sys/io-net.h >

int
advertise(void *dev_hdl, io_net_self_t *ion, int reg_hdl, int cell, int lan,
    const char *uptype, int iftype, uint8_t *macaddr)
{
    io_net_msg_dl_advert_t  *ap;
    npkt_t          *npkt;

    if ((ap = calloc (1, sizeof (*ap))) == NULL)
        return -1;

    ap->type = _IO_NET_MSG_DL_ADVERT;

    ap->iflags = IFF_SIMPLEX | IFF_BROADCAST | IFF_MULTICAST | IFF_RUNNING;
    ap->mtu_min = 0;
    ap->mtu_preferred = ap->mtu_max = 1514;
    strcpy(ap->up_type, uptype);
    itoa(lan, ap->up_type + strlen(ap->up_type), 10);

    strcpy(ap->dl.sdl_data, ap->up_type);

    ap->dl.sdl_len = sizeof (struct sockaddr_dl);
    ap->dl.sdl_family = AF_LINK;
    ap->dl.sdl_index = lan;
    ap->dl.sdl_type = iftype;
    ap->dl.sdl_nlen = strlen(ap->dl.sdl_data);
    ap->dl.sdl_alen = 6;

    memcpy(ap->dl.sdl_data + ap->dl.sdl_len, macaddr, 6);
    npkt = make_packet(ion, (void *)ap, sizeof (*ap));

    if (npkt == NULL)
        return -1;

    npkt->flags |= _NPKT_MSG;

    /*
     * At some point, the packet will be returned to the driver.  We
     * set this driver-owned flag, so that we will be able to tell
     * later on that this is an advertise message, and that we
     * will not be able to use it to store an ethernet packet, since
     * it's not big enough to store a full-sized ethernet packet.
     */
    npkt->flags |= (1<<20);

    npkt = ion->tx_up_start(reg_hdl, npkt, 0, 0, cell, lan, 0, dev_hdl);
```

```
    if (npkt != NULL) {
        /* Nobody took the packet, discard it */
        free(npkt->org_data);
        ion->free(npkt);
        return -1;
    }

    return 0;
}
```

The `io_net_registrant_t` structure

This structure is passed to the *reg()* function when the driver registers an interface with **`io-net`**. This structure is defined in **`<sys/io-net.h>`**. More information about this structure can be found in the Network DDK API chapter.

Driver entry points

The **`io_net_registrant_funcs_t`** structure, which is referenced from the **`io_net_registrant_t`** structure, contains function pointers to all of the driver entry points. After registration, the networking framework may call into the driver through these function pointers.

Interface statistics

Drivers are expected to keep track of statistical information. Some statistics are mandatory, some are optional. Some statistics apply to certain types of devices only. For example, the statistics tracked for an 802.3 device are different from those tracked for an 802.11 wireless device.

The driver should initialize all counters to zero when the interface is instantiated.

Higher-level software may query the driver's statistical counters by issuing the DCMD_IO_NET_GET_STATS *devctl()* function. Upon receipt of this *devctl()*, the driver will store the statistical infomation into a structure of type **`nic_stats_t`**. This structure is defined in **`<hw/nicinfo.h>`**.

Packet reception filtering

There are various *devctl()* functions that the driver can support in order to provide control over how packets are to be filtered upon reception. Packets are filtered based on the destination address in the Ethernet header. Most Ethernet devices have hardware that can be programmed to automatically accept or reject packets, based on this destination address. Destination addresses can be broken down into three categories:

- broadcast addresses — addresses that consists of the byte sequence ff:ff:ff:ff:ff:ff (all ones) as defined by the 802.3 specification.
- multicast addresses — addresses that have the least-significant bit of the first byte set to one.
- unicast addresses — all other addresses.

Network drivers should always receive broadcast packets and pass them upstream. Unicast packets that have the interface's current MAC address as their destination address, should also be passed upstream.

When the device is in promiscuous mode, the driver should attempt to receive all packets seen on the medium, irrespective of their destination address, and pass them upstream. The interface can be put into promiscuous mode:

- if the "promiscuous" driver option was specified, during initialization.
- via a *devctl()* function. The DCMD_IO_NET_PROMISCUOUS *devctl()* function can also be used to take the interface out of promiscuous mode.

When not in promiscuous mode, the driver may be required to receive certain multicast packets. The driver is instructed as to how it should filter multicast packet reception via the DCMD_IO_NET_CHANGE_MCAST *devctl()* function.

It's not considered an error if the driver passes packets upstream that it was not required to receive. The upper-layer software will filter out any unwanted packets.

You should avoid passing unrequired packets if possible, since it puts an additional load on the CPU.

This means that imperfect filters, which are usually implemented in hardware using a hashing algorithm, may be employed to perform multicast packet filtering. Note, however, that it's considered an error if a packet is rejected due to address filtering when the driver was expected to receive it.

Where an Ethernet device can't filter multicast addresses in hardware, the *driver* could put the device into promiscuous mode. This would mean that any packet transmitted on the medium by any device would be received and potentially need to be filtered-out by software. This potentially could place a high burden on the CPU, but at least software that depended on the multicast functionality would be able to operate.

Some devices can be placed into a "promiscuous multicast" mode. This means that they receive all multicast packets, but receive unicast packets destined only for the station's MAC address. You could use this method instead of full promiscuous mode to avoid receiving unicast packets unecessarily.

Some types of embedded systems may not have any software running on the device that needs to receive multicast packets. However, the TCP/IP stack always enables a small number of multicast addresses by default. This would allow the scenario described in the previous paragraph to occur if the device didn't have selective multicast filtering capabilities. The CPU would be burdened with unnecessary packet reception and software address filtering, even though no software on the system actually required packets with the enabled multicast addresses to be received.

To avoid this scenario, the "nomulticast" driver option tells the driver via the DCMD_IO_NET_CHANGE_MCAST *devctl()* function, that it can turn off reception of multicast packets, and to ignore any requests to enable multicast packet reception.

Multicast address filtering is controlled by the DCMD_IO_NET_CHANGE_MCAST *devctl()* function. A structure of type **`struct _io_net_msg_mcast,`** which is defined in **`<sys/io-net.h>`**, is passed to the driver, which contains information describing the required change in multicast address filtering. The fields are defined in the **`io_net_msg_mcast`** structure.

In certain cases a device may lose track of which multicast address ranges are enabled for reception. For example, if a device maintains its list of enabled addresses in the form of a list of individual addresses, the list could potentially overflow if too many addresses are enabled. At this point, the driver will need to put the device into promiscuous multicast mode (or, if that's not possible, into full promiscuous mode).

If the list subsequently shrinks to the point where the device is once again able to hold the entire list, the device can be taken out of promiscuous mode. The driver will then need to reprogram the device with the most up-to-date list.

A driver can reference the entire list of enabled multicast address ranges at any time by issuing the _IO_NET_CHANGE_MCAST *devctl()* function through the *devctl()* callback. This wil cause the driver's *devctl()* entry point to be called, at which point it can follow the "next" field of the **`_io_net_msg_mcast`** structure to traverse the entire list of enabled ranges.

CAUTION: Be careful, because when the driver calls the **`devctl`** function, it could result in its *devctl()* entry point being re-entered, *before* the *devctl()* callback returns!

Hardware checksum offloading

Some devices support offloading of the computation of IP header, TCP, and UDP checksums from the CPU onto the hardware. Devices that support computation of these checksums in hardware are becoming increasingly more common.

Driver support for checksum offloading involves:

- advertising the device's checksum offloading capabilities, if any
- enabling/disabling checksumming
- generating the checksum during transmission
- verifying the checksum upon packet reception, and reporting the result.

Advertising checksum capabilities

Advertising of the device's checksum offloading capabilities is performed by setting flags in the *capabilities_rx* and *capabilities_tx* fields of the **`io_net_msg_dl_advert_t`** structure. Valid flags are defined in **`<net/if.h>`**.

Enabling/disabling checksums

Checksum offloading is enabled or disabled via the SIOCSIFCAP *devctl()*, defined in **`<sys/socket.h>`**. The driver's *devctl()* handler is passed a pointer to a **`struct ifcapreq`**.

Receive flags for checksum verification

If the following flags are set for ifcr_capenable_rx, then the checksums can be verified for:

- IFCAP_CSUM_IPv4 — IP version 4 header checksums.
- IFCAP_CSUM_TCPv4 — TCP version 4 payload checksums.
- IFCAP_CSUM_UDPv4 — UDP version 4 payload checksums.
- IFCAP_CSUM_TCPv6 — TCP version 6 payload checksums.
- IFCAP_CSUM_UDPv6 — UDP version 6 payload checksums.

If a flag other than one of the above is set, the driver's *devctl()* handler should return ENOTSUP to reject the request.

Transmit flags for checksum generation

If the following flags are set for ifcr_capenable_tx, then the checksums can be generated for:

- IFCAP_CSUM_IPv4 — IP version 4 header checksums.
- IFCAP_CSUM_TCPv4 — TCP version 4 payload checksums.
- IFCAP_CSUM_UDPv4 — UDP version 4 payload checksums.
- IFCAP_CSUM_TCPv6 — TCP version 6 payload checksums.
- IFCAP_CSUM_UDPv6 — UDP version 6 payload checksums.

If a flag other than one of the above is set, the driver's *devctl()* handler should return ENOTSUP to reject the request.

Verifying checksums for received data

If offloading of checksum verification for received packets is enabled, the driver should set the *csum_flags* field of the **npkt_t** structure as appropriate before sending the packet upstream.

For received packets, the flags for the *csum_flags* field, described in **<sys/mbuf.h>**, are defined as:

- M_CSUM_IPv4 — an IPv4 header is present, and the hardware has computed the header checksum.
- M_CSUM_IPv4_BAD — if set, the computed checksum for the IP header didn't match the checksum in the IP header.
- M_CSUM_TCPv4 — a packet contains a version 4 TCP payload, and the hardware has computed the payload checksum.
- M_CSUM_UDPv4 — a packet contains a version 4 UDP payload, and the hardware has computed the payload checksum.
- M_CSUM_TCPv6 — a packet contains a version 6 TCP payload, and the hardware has computed the payload checksum.

- M_CSUM_UDPv6 — a packet contains a version 6 UDP payload, and the hardware has computed the payload checksum.
- M_CSUM_TCP_UDP_BAD — if set, the computed checksum for the payload didn't match the checksum in the TCP or UDP header.

Generating checksums during transmission

If offloading of checksum generation upon packet transmission is enabled, the driver should ensure that a checksum is generated in accordance with the information supplied in the *csum_flags* field of the **`npkt_t`** structure. Upon packet transmission, flags for the *csum_flags* field, described in **`<sys/mbuf.h>`**, are defined as follows:

- M_CSUM_IPv4 — the hardware must compute an IP version 4 header checksum, and insert the computed value into the checksum field of the packet's IP header.
- M_CSUM_TCPv4 — the hardware must compute a TCP version 4 payload checksum, and insert the computed value into the checksum field of the packet's TCP header.
- M_CSUM_UDPv4 — the hardware must compute a UDP version 4 payload checksum, and insert the computed value into the checksum field of the packet's UDP header.
- M_CSUM_TCPv6 — the hardware must compute a TCP version 6 payload checksum, and insert the computed value into the checksum field of the packet's TCP header.
- M_CSUM_UDPv6 — the hardware must compute a UDP version 6 payload checksum, and insert the computed value into the checksum field of the packet's UDP header.

The `nic_config_t` structure

There are two main purposes for the **`nic_config_t`** structure:

- to provide device configuration information to higher-level software, via the DCMD_IO_NET_GET_CONFIG *devctl()*

- to store information that's obtained by parsing the driver option string. Since the driver generally needs to be able to readily access its configuration information, a driver typically includes a **nic_config_t** structure as part of its internal state structure.

The **nic_config_t** structure is defined in **<hw/nicinfo.h>**.

The nic_wifi_dcmd_t structure

When the driver receives a DCMD_IO_NET_WIFI *devctl()*, it's passed a pointer to a structure of **nic_wifi_dcmd_t**. This **devctl** either gets or sets various WiFi-specific parameters.

Driver option definitions

Options are passed to the driver as an ASCII string that is parseable using the *getsubopt()* function. The standardized options are defined here (note that unless otherwise specified, each option takes a parameter):

ioport — Specifies the base address of a range of registers in I/O space. A device may have more than one range of I/O mapped registers. In this case, multiple ranges may be specified, but the order in which the ranges must be specified is defined on a per-driver basis. For certain types of devices (e.g. PCI devices), the driver may be able to automatically determine the I/O base(s). If this is the case, I/O bases specified via this option take precedence.

irq — Specifies the number of the interrupt that the driver attaches to in order to receive interrupt events from the device. A driver may need to attach to more than one interrupt. If this is the case, multiple interrupt numbers may be specified, but the order in which the interrupts must be specified is defined on a per-driver basis. For certain types of devices (e.g. PCI devices), the driver may be able to automatically

determine the interrupt numbers. When this occurs, interrupts specified via this option take precedence.

dma Specifies the channel that the device uses for DMA transfers. A device may need to use more than one channel. If this is the case, multiple DMA channels may be specified, but the order in which they must be specified is defined on a per-driver basis.

vid For PCI devices, this option limits the devices automatically detected to those having the specified PCI *vendor* ID.

did For PCI devices, this option limits the devices automatically detected to those having the specified PCI *device* ID.

pci For PCI devices, this option limits the devices automatically detected to those having the specified PCI index.

A PCI device is uniquely identified by its vendor ID, device ID, and PCI index. See *pci_attach_device()* for more details.

mac Specifies the physical station address (MAC address) of the interface. If no MAC address is specified, the driver should attempt to read the station address from the hardware in a device-specific manner (if possible). If this isn't possible, the driver should attempt to obtain the MAC address by calling *nic_get_syspage_mac()*. If this fails, the interface can't be instantiated unless a MAC address is supplied via driver option.

It's an error to specify a multicast address for a MAC address. That is, the first byte of the MAC address must not have the least-significant bit set. If an attempt is made to use a multicast address for the MAC address, TCP/IP will not work.

lan Specifies the instance number to assign to the interface. By default, the interface instance numbers are assigned by **`io-net`**, starting at zero, in the order that the interfaces are registered. This option allows the default numbers to be overridden.

mtu Specifies the maximum transmittable unit of the device. This limits the size of the packets that are sent to the driver for transmission. This value includes the 14-byte Ethernet header.

mru Specifies the maximum receivable unit of the device. This indicates to the driver that it should attempt to receive from the media packets that are no bigger than this value. This value includes the 14-byte Ethernet header. For devices with DMA capability, the driver may need to pre-allocate buffers of at least this size in order to store the packets as they're transferred to memory.

speed

Although the *speed* and *duplex* options are presented separately here, they are interrelated.

The speed option specifies the rate at which the device should operate, in megabits per second.

If the device supports link auto-negotiation, as per the IEEE 802.3 spec, the device may use auto-negotiation to determine the speed and duplex.

If neither the speed nor the duplex option is specified, the driver should use auto-negotiation to determine the speed and duplex, if possible.

When only the speed option is specified, it's recommended that the driver use auto-negotiation to determine the duplex setting. The link speed can be forced to a specific value by limiting the capabilities identified during the auto-negotiation process.

If the speed option isn't specified, the driver should default the link speed to a reasonable value.

duplex The duplex option specifies whether the device should operate in full-duplex or half-duplex mode. For duplex, a value of 0 specifies half-duplex operation; a value of 1 specifies full-duplex operation.

If the device supports link auto-negotiation, as per the IEEE 802.3 spec, the device may use auto-negotiation to determine the speed and duplex.

If neither the speed nor the duplex option is specified, the driver should use auto-negotiation to determine the speed and duplex, if possible.

If the duplex option is specified, the driver should disable auto-negotiation in all cases, and force the speed and duplex to specific values.

media Specifies the media that the NIC should operate with. This is a numeric value and should be one of the **`nic_media_types`** enumerated types.

promiscuous Specifies that when the interface is activated, it should be put into "promiscuous" mode. This means the device should receive all packets possible from the media, regardless of their destination address. This option doesn't take a parameter.

nomulticast Tells the driver that it can disable reception of all multicast packets, and ignore any requests to enable reception of multicast packets. This option doesn't take a parameter.

connector Specifies the connector type that the driver should activate. This is useful for devices that have multiple connectors, such as "Combo" Ethernet cards that have both BNC and RJ-45 connectors. This is a numeric value and should be one of the **`nic_connector_types`** enumerated types, defined in **`<hw/nicinfo.h>`**.

deviceindex This option applies to non-PCI devices. For PCI devices, the *vid*, *did*, and *pci* options are used instead. When a system has multiple network interfaces that the driver knows how to control, this option specifies which interface the driver should instantiate. If this option isn't specified, the driver should instantiate all interfaces that are known to be present in the system.

phy Specifies the address of the PHY device. An 802.3-compliant physical layer device (PHY) has a unique address that can be used to access its internal registers. A driver can detect the PHY by probing at all possible PHY addresses, but in some cases it's necessary to tell the driver what the PHY address is (e.g. when there are multiple PHYs connected).

memrange Specifies the base (physical) address, and optionally the size, of a range of memory that the device uses. This memory typically contains memory-mapped device registers, or is used as a buffer to store packet data. A device may have more than one range of memory. If this is the case, multiple ranges may be specified, but the order in which the ranges must be specified is defined on a per-driver basis. For certain types of devices, e.g. PCI devices, the driver may be able to automatically determine the location and size of the memory ranges. Any memory ranges specified via this option will take precedence. To specify a size as well as a base address, the

parameter is specified as a pair of numeric values separated by a colon.

iorange Specifies the I/O base address, and optionally the size, of a range of I/O space that the device uses. This memory typically contains I/O-mapped device registers. A device may have more than one range of I/O space, and if so, multiple ranges may be specified, but the order in which the devices must be specified is defined on a per-driver basis. For certain types of devices, e.g. PCI devices, the driver may be able to automatically determine the location and size of the I/O ranges. Any I/O ranges specified via this option will take precedence. To specify a size as well as a base address, the parameter is specified as a pair of numeric values separated by a colon.

verbose Used for debugging. Specifies the verbosity level of the driver's debug output. This option can be specified without a parameter, in which case the verbosity level is set to 1.

iftype Tells the driver what type of interface it should declare itself as when it advertises its capabilities to the networking subsystem. Ethernet drivers normally advertise themselves as being of type IFT_ETHER. See **<net/if_types.h>** for a list of possible interface types.

uptype Tells the driver what kind of interface it should register with **io-net**. It's specified as a string value, and tells **io-net** what kind of filter to use to handle packets going to and from the driver. An Ethernet driver normally defaults to **en**.

priority Specifies the priority of the driver's event-handling thread. The recommended default is 21.

The driver utility library

A library of utility functions for network drivers is available. It's provided as a static library, which is compiled so that it may be linked with shared objects. A driver may be linked with this library via the **`-ldrvrS`** option to the linker. If a driver is built within the Network DDK framework, it will automatically be linked with this library. A large portion of this library deals with handling MII management for 802.3-compliant Physical Layer (PHY) devices. This portion of the library is described separately from the rest of the library. Drivers that use the utility library should include the header file **`<drvr/nicsupport.h>`** to provide the necessary structures and function prototypes.

The MII management library

A utility library is provided for network device drivers which control 802.3-compliant Physical Layer (PHY) devices via the MII (Media Independent Interface) management interface.

Typically, a PHY device is located on a separate chip from the MAC device, although it's getting increasingly common to have the PHY integrated into the same ASIC as the MAC device. Traffic data is transferred between the MAC and the PHY via the MII. The network device driver uses the MII management interface, which is a serial bus between the MAC and the PHY, to control the PHY. The MII management interface consists of a data and a clock line, and the MAC device acts as the master device during data transfers to and from the PHY.

Each PHY is assigned a unique address. The address is a 5-bit value that makes it possible to have up to 32 PHY devices on a particular MII management bus. Internally, each PHY has a register set. The driver uses control registers on the MAC device, in order to read from and write to these registers.

These registers make it possible to obtain status information from the PHY (e.g.link integrity, link speed, etc.) and to configure the PHY

(e.g. to set the link speed, or to control the link auto-negotiation process with the link partner).

A variety of PHY devices from many different vendors exists on the market. When you write a device driver for a particular MAC device, you may need to support multiple PHY devices that could potentially operate with that MAC device. Since there's a standard definition for the register layout of a PHY device, it's possible to provide a generic library that should be able to control any fully compliant PHY.

In addition to containing code for controlling a compliant PHY device via the standard register set, the MII management library contains some code which is necessary to work around problems in certain PHYs.

Whenever you write a new network driver, you'll need to worry only about the specifics of programming the MAC device; you can use the MII management library to take care of controlling the PHY.

Overview of library usage

In order to properly use the library, first the driver must call *MDI_Register_Extended()*, optionally specifying whether it wishes to receive link-monitor pulses. The driver supplies pointers to callback functions that the library uses to access the PHY registers. Typically, the driver calls:

1. *MDI_FindPhy()* — the *MDI_FindPhy()* function either searches for a PHY by iterating all the possible PHY addresses, or verifies that a PHY exists at the address where the driver expects to find one.
2. *MDI_InitPhy()* — the *MDI_InitPhy()* function is called for each PHY it wishes to control, so the driver can use the library to configure the PHY, and optionally initiate the link auto-negotiation process. If the driver enables the link monitor, it will receive pulses on a periodic basis.
3. *MDI_MonitorPhy()* — the *MDI_MonitorPhy()* function is called when the driver receives a link-monitor pulse. It uses the driver's PHY access callbacks to determine the link state. If the

driver detects a change to the link state, the library issues the notification callback that handles link-state changes. In this callback, the driver may need to reconfigure the MAC to deal with the link's state change (for example, if the link went from full-duplex to half-duplex, the MAC would need to be set to operate at half-duplex). Also, the driver will need to record the link state, so that it can report the correct state information to the upper layers.

The MII management library interface contains the following functions:

- *MDI_AutoNegotiate()*
- *MDI_DeIsolatePhy()*
- *MDI_DeRegister_Extended()*
- *MDI_DisableMonitor()*
- *MDI_EnableMonitor()*
- *MDI_FindPhy()*
- *MDI_GetActiveMedia()*
- *MDI_GetAdvert()*
- *MDI_GetLinkStatus()*
- *MDI_GetPartnerAdvert()*
- *MDI_InitPhy()*
- *MDI_IsolatePhy()*
- *MDI_MonitorPhy()*
- *MDI_Register_Extended()*
- *MDI_ResetPhy()*
- *MDI_SetAdvert_Extended()*

- *MDI_SetSpeedDuplex()*
- *MDI_SyncPhy()*

See the Network DDK API chapter for a detailed description of the functions.

Other `libdrvr` functionality

The following routines are also supported in the driver utility library:

- *drvr_mphys()*
- *nic_calc_crc_le()*
- *nic_calc_crc_be()*
- *nic_dump_config()*
- *nic_get_syspage_mac()*
- *nic_parse_options()*
- *nic_slogf()*
- *nic_strtomac()*

See the Network DDK API chapter for a detailed description of the functions.

Guidelines for designing a driver

This section discusses various aspects of driver design. The intent is to provide various guidelines to help you create portable, robust, high-performance drivers that don't have a negative impact on other parts of the system. We'll look at the issues of:

- cache coherency
- portability considerations
  - accessing I/O ports
  - endian issues

- performance tips
- interrupt handling.

Cache coherency

The concept of cache coherency is to make sure that the host CPU(s) and the network device have the same view of memory structures (i.e. data buffers and buffer descriptors) that both components can access. This is an issue only for devices that directly access system memory via bus-mastering or through a DMA channel. If the driver copies the data from a memory-mapped or I/O mapped register area into system RAM buffers, there is no coherency issue; since the CPU transferred the data, it knows what the contents of the data buffers should be.

You need to be aware of coherency only when *all* of the following conditions are true:

- The network device can directly access system memory.
- The CPU may cache some of the data that the device is able to directly access.
- The system doesn't have a "smart cache" snooping mechanism.

A cache-snooping mechanism always exists on x86 systems that support caching. This means that when an external device modifies memory, the processor(s) "snoop" the memory cycle and perform the necessary operation with respect to the caches. For example, the processor can invalidate information in the cache when the device modifies data, or can flush data from the cache out to system memory when the device attempts to read it.

On an x86 system, the third condition (the system doesn't have a "smart cache" snooping mechanism) is always false, and you don't need to worry about cache coherency. Additionally, many higher-end non-x86 systems also have a smart cache. But, if the driver is targeted at non-x86 platforms, and potentially needs to work on any system that doesn't have a smart cache (true for all supported ARM and SH4-based systems, most MIPS-based systems, and many

PowerPC-based systems), then you need to be aware of cache coherency.

A simple, effective way to enforce cache coherency is to disable caching for all data structures that the device may directly access. However, this carries a severe performance penalty, as operations performed on the non-cacheable data (such as checksum calculation and header parsing) can't benefit from caching. Typically, allocating packet data buffers as uncacheable doubles the CPU-usage required to transfer data across the network. This can halve throughput on low-end systems.

The solution for supporting systems that don't have a smart cache, while still using cacheable buffers, is to explicitly perform operations on the cache, within the driver. For example, if a data buffer is submitted to the device, to be filled with packet data from the network, any cached data associated with this buffer needs to be invalidated. Then, after the device has copied data into the buffer, the CPU can read the correct data from the buffer. Since any cached data for this buffer was previously invalidated, CPU accesses to the memory won't retrieve stale data from the cache. The correct data is fetched from system memory instead.

When transmitting data, before the buffer is submitted to the device for transmisson, the driver should make sure any data associated with the buffer is flushed out to system memory, so that when the device fetches the data, it gets the most current copy.

The way data in the cache is flushed or invalidated is CPU-dependent, and involves issuing processor-dependent assembly instructions. If a driver will run on a single type of processor family, the driver could just use inline assembly language macros to perform the necessary cache synchronization.

We've provided a platform-independent library to help with the task of maintaining portable drivers that need to deal with cache coherency. This library should be used when writing a portable driver. The library takes the correct action for the CPU it's running on.

On x86 systems, these functions do nothing, whereas on an SH4 system, for example, they issue assembly instructions to manipulate the cache.

Portability considerations

Two factors that affect portability are:

- accessing I/O ports
- endian issues.

Accessing I/O ports

When you access I/O ports always use *mmap_device_io()* to map the I/O address, and use the mapped version of the address with the *in8()*/*out8()* etc. functions. If you attempt to use the I/O base without mapping it, your code will work only on x86 systems.

Endian issues

If you want your device to run on both little-endian and big-endian systems, you may find the macros in **<gulliver.h>** useful. For example, if you have a little-endian device that must work on both a little-endian and a big-endian system, you could use the *ENDIAN_LE32()* macro to access a 4-byte variable that the device stored in memory. On a little-endian system, this macro won't modify the value, since both the device and the host are little-endian.

On a big-endian system, the individual bytes within the variable are swapped, reversing their order. The value stored by the hardware is converted to big-endian before it's used by the big-endian host CPU. Also, some hardware can automatically swap the bytes without the need for software to do it. In this case, the macro could swap the value back to little-endian again! Sometimes it may be necessary for you to create your own macros to handle endian conversions, and create independant binaries to support systems with different swapping behaviour, using conditional compilation.

With a little care, you can offset the performance penalty that endian-swapping imposes. You can use the *inle32()*/*outle32()* calls to

read values from I/O ports. If you need to perform endian-swapping, these functions are the most efficient way to do this for the target processor. Also, whenever possible, write your code so that the swapping occurs at compile time instead of at runtime. For example, suppose “foo” is a pointer to a value that was stored in memory by the device, and you want to check bit 7 of this value. The following code would perform a data swap at runtime:

```
if (ENDIAN_LE32(*foo) & (1<<7)) {
    /* The bit is set */
}
```

This code lets you achieve the same effect, but the swap occurs at compile time, since the swapping is being performed on a constant:

```
if (*foo & ENDIAN_LE32(1<<7)) {
    /* The bit is set */
}
```

Performance tips for designing a driver

The following issues can be addressed so that you can design a driver that performs better:

- decoupling the packet transmission and reception
- transmit completion interrupts
- strategies for organizing data structures
- avoiding data copying.

Decoupling the packet transmission and reception

For most newer network devices, the packet transmission logic and packet reception logic in the device operate independently. This means that the driver can effectively treat the device as two separate pieces of hardware. When you determine how to protect access to the hardware (e.g. using mutexes) it’s worth taking this into consideration.

Some devices don't have decoupling of transmit and receive logic. For example, on some devices, the registers are accessible in banks, or windows: the driver must switch to the correct bank/window before it can access a particular register. In this case, it's unsafe for more than one thread to program the device at any given time, since one thread could switch windows, and the other thread could switch the window to something else before the first thread completes the register access. For a device like this, the driver would typically employ a per-interface mutex, to ensure exclusive access to the hardware. Any thread running in the driver would need to make sure it has ownership of this mutex before touching the device's registers.

If the receive and transmit logic is separated in the hardware, you can implement a more fine-grained locking policy. The objective is to reduce the number of threads that contend for a given locking primitive. This yields major performance gains on an SMP system.

Even a non-SMP system can be helped a great deal, since receiving and transmitting threads don't need to preempt each other due to lock contention. The usual approach is to have the driver's event-handling thread only ever access receive-related hardware, and the driver's packet-transmission entry point only access the transmit-related hardware. Note that multiple threads can be executing in the driver's transmit entry point concurrently!

The driver would create a mutex to protect transmit-related hardware and data structures. Since the driver has only one receive thread, it would need to acquire a mutex only if one of the driver's entry points could potentially access a resource that relates to packet reception. For example, the driver's *tx_done* entry point, which can be used to recycle packet buffers for packet reception, might access a linked list or similar structure that the driver's packet reception handler might also access. In this case, another mutex would be used to protect the linked list.

It's possible that the driver's event-handling thread might want to perform some disruptive operation on the device, such as resetting the hardware to recover from an error. To prevent this from interfering

with the operation of threads trying to transmit data, the event thread would simply lock the transmit mutex before resetting the device.

Transmit-completion interrupts

One fairly simple performance optimization is to implement the driver's transmit logic so that it doesn't generate transmit interrupts. On devices that use DMA to transmit chains of packets, e.g. by using a descriptor ring, it's generally possible to operate without using transmit completion interrupts at all. This helps reduce the interrupt load and leads to better throughput.

Transmit-completion interrupts are designed to inform the driver that buffers that contained data for a pending transmit are no longer needed, and can be freed or re-used (the driver would typically call the *tx_done* callout, to return the packet to the originator). The driver doesn't need to do buffer reclaim in an interrupt event handler; it can simply turn off transmit-completion interrupts, and reclaim the buffers the next time its transmit entry point is called. The only slight problem with this is that a burst of packets could be queued for transmission, after which nothing is sent to be transmitted for a long period of time. A bunch of packet buffers could be left outstanding, without getting reclaimed. The driver could use a timer that fires periodically (every couple of seconds). When the timer fires, the driver's event-handling thread receives a pulse and checks for outstanding transmits, then reclaims any outstanding buffers.

Strategies for organizing data structures

For optimal performance, make sure variables are naturally aligned, that is, 32-bit values should start on a 4-byte boundary, and 64-bit values should start on an 8-byte boundary. Use padding when necessary.

A driver typically creates a structure internally, one per interface, to keep track of various state information pertaining to the interface. If you organize the members in this structure carefully, you can achieve dramatic improvements in performance on SMP systems by minimizing data-access contention on a given cache line.

All you need to do is separate variables that are accessed during packet transmission from those that are accessed during packet reception. Also, some padding should be placed between the two sets of variables (about a cache line's worth, typically 32 or 64 bytes) to ensure that the variable sets are stored on separate cache lines.

Avoiding data copying

For devices with DMA capability, the driver should avoid copying the data with the CPU, if at all possible, and instead use DMA to copy the data directly to/from the packet data buffers.

On receive, the driver typically sets up a list/ring of packet buffers, then the device or DMA engine fills the buffers as the data arrives from the network. Each time a full packet is received, the driver encapsulates the buffer with an **`npkt_t`** structure, and sends the packet upstream via the *tx_up_start* callback. The driver shouldn't modify the buffer or the associated **`npkt_t`** structure until the packet is returned to the driver via the *tx_done* driver entrypoint.

The driver should allocate the receive data buffers with padding if necessary. This way, the buffers can be aligned to enable the hardware to use DMA to write to the buffers. When a buffer is sent upstream, the driver typically allocates another buffer to replace the one that was sent upstream. To avoid having to allocate buffers and their associated data structures in the receive event-handler, the driver could create a pre-allocated linked list of spare **`npkt_t`** structures. When packets are returned to the driver, the driver could put the packets onto the linked list, instead of freeing them, for re-use later. The driver would, however, want to put checks in place to prevent this list from growing too large, and using up too much memory.

Upon transmission, the driver must handle a packet that consists of multiple fragments. Multiple hardware-buffer descriptors are typically needed to submit a single packet to the hardware. Since these fragments could be of arbitrary alignment, the hardware must be able to address the data fragments with byte-aligned granularity for this to work. If, due to hardware restrictions, a packet can't be directly DMA-ed to the device, the data fragments of the packet could be

concatenated into a single, aligned, contiguous buffer, before being sent to the hardware (obviously this incurs a performance penalty).

When data fragments are enqueued to the hardware to be transmitted, the driver must not modify or release the data buffers. Instead, it must use some device-dependent method to learn that the DMA transfer has completed (e.g. by calling the *tx_done* callback), before it releases the buffer.

Handling interrupts

We recommend that network drivers not use "real" interrupt handlers (i.e. by calling *InterruptAttach()*), but use the *InterruptAttachEvent()* function instead. This way, all interrupt processing is done at process time, by a normal, preemptable thread. This means that the way the network interface operates won't have a negative impact on the system's realtime responsiveness.

Drivers that run on systems where interrupt lines may be shared with other devices (a common scenario on x86 systems) need to be handled a little more carefully. When the driver receives an event that indicates an interrupt was generated, it should consider that the interrupt event could have been triggered as a result of a different device generating an interrupt.

When the driver receives an interrupt event as a result of the *InterruptAttachEvent()* mechanism, the kernel masks the interrupt, so that no more events are generated until the driver receives the event and handles the interrupt condition. In order to receive subsequent interrupts, the driver must unmask the interrupt by calling *InterruptUnmask()*.

For interrupt-sharing to work well, the driver should call *InterruptUnmask()* as soon as possible after receiving the event so that the other device(s) sharing the interrupt won't experience delayed interrupt-event delivery (which, in the case of devices such as audio devices could cause undesirable results). If the driver unmasks the interrupt before clearing the source of the interrupt at the device level, spurious interrupt events will be delivered. To prevent this, the driver should first mask the interrupt at the device level, then call

InterruptUnmask(), as soon as it receives the interrupt-notification event. Then other devices can receive interrupts on the same interrupt line, while the driver is processing events such as sending received packets upstream. Once the driver finishes processing events that can cause an interrupt, it can unmask the interrupt at the device level, before exiting its event-handling loop.

CAUTION: When an interface is being shut down, the driver should ensure that all interrupts coming from the device are masked at the device level. This is typically done in the driver's *shutdown1()* entry point, since after *shutdown1()* has been called, the driver should no longer be sending received packets upstream, and therefore shouldn't need to receive interrupt events.

If a device generates an interrupt after the driver has gone away, and the interrupt line is being shared by another device, this could lock up the entire system, since the interrupt being asserted by the device for which no driver is running will never get cleared! (The interrupt will be unmasked at the CPU level, since the second device has attached to the interrupt.)

Chapter 3

Network DDK API

This chapter includes reference pages for the following functions and data structures used in writing network drivers:

Function/Structure:	Description:
drvr_mphys()	Get physical address of mapped memory
io_net_dll_entry_t	Global symbol exported by shared objects to be loaded by **io-net**
io_net_msg_dl_advert_t	Structure used to advertise a driver's capabilities
io_net_msg_mcast	Structure used to control multicast address filtering
io_net_registrant_funcs_t	Functions in your driver that **io-net** can call
io_net_registrant_t	Information used when registering with **io-net**
io_net_self_t	Functions in **io-net** that your driver can call
MDI_AutoNegotiate()	Initiate the auto-negotiation process
MDI_DeIsolatePhy()	De-isolate the PHY from the MII interface
MDI_DeRegister_Extended()	De-register with the MII management library and free allocated resources
MDI_DisableMonitor()	Disable the link monitor
MDI_EnableMonitor()	Enable the link monitor

continued…

Function/Structure:	Description:
MDI_FindPhy()	Determine if a PHY exists at a given address
MDI_GetActiveMedia()	Query the active media type
MDI_GetAdvert()	Query the advertised media types
MDI_GetLinkStatus()	Determine the status of the PHY link
MDI_GetPartnerAdvert()	Store currently advertised media types
MDI_InitPhy()	Initialize the PHY
MDI_IsolatePhy()	Isolate the PHY from the MII interface
MDI_MonitorPhy()	Check all PHY status
MDI_Register_Extended()	Register with the MII management library
MDI_ResetPhy()	Reset the PHY
MDI_SetAdvert_Extended()	Select media types to advertise
MDI_SetSpeedDuplex()	Force the link-state setting
MDI_SyncPhy()	Synchronize the PHY
nic_calc_crc_le()	Generate CRC32 checksums for little-endian mode
nic_calc_crc_be()	Generate CRC32 checksums for big-endian mode
`nic_config_t`	Structure used to store device configuration
`nic_dump_config`	Output configuration information

continued...

Function/Structure:	Description:
nic_ethernet_stats_t	Ethernet statistics for a **nicinfo** command
nic_get_syspage_mac()	Retrieve a stored MAC address
nic_parse_options()	Parse an option string for a network driver
nic_slogf()	Output messages and debug information
nic_stats_t	Statistical information for devices
nic_strtomac()	Convert a MAC address
nic_wifi_dcmd_t	Get WiFi-specific parameters
nic_wifi_stats_t	Get WiFi-specific statistics
npkt_t	Data structure for describing a packet

drvr_mphys()

Get the physical address of mapped memory

Synopsis:

```
uint64_t  drvr_mphys  ( void  *vaddr)
```

Arguments:

vaddr A pointer to the virtual address of some memory that's mapped into the process's virtual address space.

Description:

The *drvr_mphys()* function returns the physical address of the data pointed to by the *vaddr* argument.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

io_net_dll_entry_t

Global symbol exported by shared objects to be loaded by **io-net**

Synopsis:

```
typedef struct _io_net_dll_entry {
    int nfuncs;
    int (*init) (void *dll_hdl,
                 dispatch_t *dpp,
                 io_net_self_t *ion,
                 char *options );
    int (*shutdown) (void *dll_hdl );
} io_net_dll_entry_t;
```

Description:

The **io_net_dll_entry_t** data structure defines a network driver's primary entry points. Your driver must contain a public symbol of type **io_net_dll_entry_t** called *io_net_dll_entry*.

The *nfuncs* member specifies the number of functions in the **io_net_dll_entry_t** structure. Your driver should set this to 2, because two functions are currently defined: *init()* and *shutdown()*.

init()

A pointer to your driver's initialization function, which is mandatory. This is the first of your driver's functions that **io-net** calls. The prototype is:

```
int (*init) (void *dll_hdl,
             dispatch_t *dpp,
             io_net_self_t *ion,
             char *options );
```

The arguments are:

dll_hdl	An internal handle used by **io-net** — you'll need this handle for future calls into the **io-net** framework.
dpp	This pointer should be ignored.
ion	A pointer to a data structure of the **io-net** functions that your driver can call to interact with the networking

subsystem. The driver should always keep a copy of this pointer. For more information, see **io_net_self_t**.

options A pointer to an ASCII string, which, if non-NULL, should be parsed by the driver.

Your *init()* function should return 0 on success. If an error occurs, this function should set *errno* and return -1.

shutdown()

The **shutdown** function will be called before the driver is unloaded from memory.

The prototype is:

```
int (*shutdown) (void *dll_hdl );
```

The *dll_hdl* is the handle that was passed to the driver's initialization function.

When a particular registration instance (a "registrant") is shut down, its *shutdown1()* and *shutdown2()* functions (from the **io_net_registrant_t** structure's **io_net_registrant_funcs_t** function pointer array) are called.

When *all* of the shared object's registrants are closed, this *shutdown()* function is called. It may be necessary to use this entry point to do additional cleanup to ensure that any resources that we allocated during the lifetime of the driver have been deallocated. If you don't wish to supply this function, place a NULL in this member.

Your driver's *shutdown()* function should return 0.

Classification:

QNX Neutrino

See also:

io_net_registrant_t, **io_net_registrant_funcs_t**

io_net_msg_mcast

Structure used for multicast address filtering

Synopsis:

```
typedef struct _io_net_msg_mcast;

struct _io_net_msg_mcast {
      uint16_t      type;
      uint32_t      flags;
      uint16_t      next;
      uint32_t      mc_min;
      uint32_t      mc_max;
};
```

Description:

The **io_net_msg_mcast** structure contains information concerning the changes required to multicast addresses that have been identified by the DCMD_IO_NET_CHANGE_MCAST **devctl**.

The members include:

type The type of request. If it's set to _IO_NET_JOIN_MCAST, the request specifies a range of multicast addresses for which reception should be enabled. If it's set to IO_NET_REMOVE_MCAST, the request specifies a range of addresses for which multicast packet reception should no longer be enabled. (In this case, the address range was previously enabled via an _IO_NET_JOIN_MCAST request.)

If the field doesn't contain either of these two values, it means that the networking subsystem is responding to a DCMD_IO_NET_CHANGE_MCAST **devctl** that the driver issued.

flags Currently only one flag is defined:

- _IO_NET_MCAST_ALL — if set, specifies that the device should be put into, or taken out of, promiscuous multicast mode.

  If the *type* field was set to _IO_NET_JOIN_MCAST, the device should be put into promiscuous multicast mode.

If the *type* field was set to _IO_NET_REMOVE_MCAST, the device should be taken out of promiscuous multicast mode, in which case the device should revert to filtering based on the enabled range(s) of multicast addresses. When in promiscuous mode, the device should receive *all* multicast packets.

next Used to chain multiple **_io_net_msg_mcast** structures into a linked-list when the driver issues a DCMD_IO_NET_CHANGE devctl. When the driver traverses the list, it accesses the entire database of currently enabled multicast address ranges.

mc_min and *mc_max*

Specifies the minimum and maximum addresses within a range of multicast addresses. The LLADDR macro, defined in **<net/if_dl.h>**, can be used to obtain a pointer to the actual, physical MAC address, e.g. if *mcast* points to struct **io_net_msg_mcast**, the start and end of the address range can be obtained as follows:

```
LLADDR(mcast->mc_min.addr_dl)
LLADDR(mcast->mc_max.addr_dl)
```

If *type* was _IO_NET_JOIN_MCAST, the driver should enable reception of packets whose destination addresses are within these inclusive address ranges.

If *type* was _IO_NET_REMOVE_MCAST, the driver should disable reception of packets whose destination addresses are within these inclusive address ranges (in this case, the address range will have previously been enabled via an _IO_NET_JOIN_MCAST message).

If the _IO_NET_MCAST_ALL flag was set, *mc_min* and *mc_max* are irrelevant, and shouldn't be referenced.

Classification:

QNX Neutrino

io_net_msg_dl_advert_t

Structure used to advertise a driver's capabilities

Synopsis:

```
typedef struct _io_net_msg_dl_advert
                  io_net_msg_dl_advert_t;

struct _io_net_msg_dl_advert {
    uint16_t            type;
    uint32_t            iflags;
    uint32_t            mtu_min;
    uint32_t            mtu_max;
    uint32_t            mtu_preferred;
    char                up_type[20];
    uint32_t            capabilities_rx;
    uint32_t            capabilities_tx;
    struct sockaddr_dl  dl;
};
```

Description:

The **io_net_msg_dl_advert_t** structure is used when a module wants to advertise its capabilities to **io-net** and its other modules. The advertising module fills in this structure and places it in an upgoing packet.

The members include:

type — Set this to _IO_NET_MSG_DL_ADVERT.

iflags — Flags that describe the module's capabilities. This member is a combination of the following bits, which are defined in **<net/if.h>**:

- IFF_SIMPLEX — the module can't hear its own transmissions.
- IFF_BROADCAST — the broadcast address is valid.
- IFF_RUNNING — the module has allocated resources.
- IFF_MULTICAST — supports multicasting.

mtu_min The minimum preferred MTU (Maximum Transmission Unit). Set the *mtu_min* value to zero.

mtu_max The maximum MTU. The values for *mtu_max* and *mtu_preferred* should be set the same.

mtu_preferred The preferred MTU. For an Ethernet device, this value is **1514**. If an MTU size was specified via the *mtu* driver option, specify the value associated with that driver option instead.

up_type The type of upgoing packet produced by the module. For an Ethernet device, set to *en* followed by an ASCII representation of the LAN number, e.g. *en0*. If a different string was specified by the *uptype* driver option, use that value instead of *en*. The LAN number should be the number obtained from **io-net** at registration.

dl A link-layer **sockaddr_dl** structure, as described below. Its members are specific to the link layer.

The **sockaddr_dl** structure is defined as follows:

```
struct sockaddr_dl {
    char       sdl_data;
    u_char     sdl_len;
    u_char     sdl_family;
    u_int16_t  sdl_index;
    u_char     sdl_type;
    u_char     sdl_nlen;
    u_char     sdl_alen;
};
```

The members are:

sdl_data The value found in the *up_type* field, followed by the numeric form of the interface's current MAC address. Don't include NULL-string terminators.

sdl_len The total length of the **sockaddr_dl** structure.

sdl_family Set this to AF_LINK (defined in **<sys/socket.h>**).

sdl_index Set this to the LAN number that was obtained when the driver registered the interface with **io-net**.

sdl_type Set this to IFT_ETHER (defined in **<net/if_types.h>**). If a value was specified via the "iftype" command-line option, use this value instead.

sdl_nlen Set this to specify the number of characters in the alphabetic portion of the *dl.sdl_data* field.

sdl_alen Set this to the length of the device's MAC address. The length for an Ethernet address is six.

capabilities_rx and *capabilities_tx*

Advertise the device's hardware checksum-offloading capabilities. The following flags are defined for *capabilities_rx* and *capabilities_tx*:

- IFCAP_CSUM_IPv4 — the device can verify an IP version 4 header checksum.
- IFCAP_CSUM_TCPv4 — the device can verify a TCP version 4 payload checksum.
- IFCAP_CSUM_UDPv4 — the device can verify a UDP version 4 payload checksum.
- IFCAP_CSUM_TCPv6 — the device can verify a TCP version 6 payload checksum.
- IFCAP_CSUM_UDPv6 — the device can verify a UDP version 6 payload checksum.

Classification:

QNX Neutrino

io_net_registrant_funcs_t

Functions in your driver that **io-net** *can call*

Synopsis:

```
typedef struct _io_net_registrant_funcs {
    int nfuncs;
    int (*rx_down) (...);
    int (*tx_done) (...);
    int (*shutdown1) (...);
    int (*shutdown2) (...);
    int (*dl_advert) (...);
    int (*devctl) (...);
    int (*flush) (...);
} io_net_registrant_funcs_t;
```

Description:

The **io_net_registrant_funcs_t** structure is a table of functions that your driver wants to register with **io-net**. The *funcs* member of the **io_net_registrant_t** structure is a pointer to an instance of this structure.

The *nfuncs* member specifies the number of function pointers in the structure. For the structure as given above, this is 8. The functions are described below.

rx_down()

This function is called when your module receives a down-headed packet from a module above you. The driver must traverse the buffers and their associated data fragments, and send the data to the hardware to be transmitted. Depending on the nature of the NIC device, the driver may do one of the following:

- concatenate the data fragments into a buffer, from which the data will be transmitted
- pass pointers to the data fragments to the hardware, if the hardware has DMA capability.

If the driver passes pointers, then the hardware will typically be programmed with the physical addresses of the data fragments, which are stored in the **net_iov_t** structure.

If the driver copies the data, it can then call the *tx_done()* function to return the buffer to the sender, since it's finished with the buffer. However, if the hardware has DMA capability, there may be a long time delay from when the packet is submitted to the NIC device until the NIC copies the data from the buffers. For performance reasons, the driver wouldn't want to wait around until the packet has been copied. Instead, it would store a pointer to the packet, perhaps by maintaining a linked-list of packets that are pending transmission, and return from the entry point. At a later time (e.g. the next time the transmit entry point is called, or perhaps when a hardware interrupt or timer triggers an event) the packet can be returned to the sender by calling *tx_done()*. The prototype is:

```
int (*rx_down) (npkt_t *npkt,
                void *func_hdl);
```

The arguments are:

npkt This is the driver's packet transmission entry point. The *npkt* argument points to a structure which describes the packet to be transmitted. The *func_hdl* variable is the pointer that the driver supplied in the **io_net_registrant_t** structure when it registered the interface with **io-net**.

func_hdl The pointer that the driver supplied in the **io_net_registrant_t** structure when it registered the interface with **io-net**.

tx_done()

This function is called when the upper layers have finished processing packets that originated from the driver, effectively indicating that they have been consumed and may now be "recycled" (or disposed of). A previous call to *tx_up_start()* results in this function being called. The prototype is:

```
int (*tx_done) (npkt_t *npkt,
                void *done_hdl,
                void *func_hdl);
```

This entry point should not be confused with the **tx_done** callback in the **io_net_self_t** structure. This function will be called when the upper layers have finished processing packets that originated from the driver. This function will be called as a result of a previous call to *tx_up_start()*. In this entry point, the driver can either free the packets, and their associated buffer(s), or it can reuse them. The arguments are:

npkt A pointer to the packets returned to the driver.

done_hdl A pointer to the handle the driver passed to *tx_up_start()* when the packet was sent upstream. It's specified when your driver calls **io-net**'s *tx_up_start()* function (see the description of **io_net_self_t**).

func_hdl The pointer that the driver supplied in the **io_net_registrant_t** structure when it registered the interface with **io-net**.

shutdown1()

The shutdown entry points are called when the interface is no longer needed. They are called when either when the user unmounts the interface, or when the **io-net** process is terminating. Since the process of removing an interface from the networking subsystem is a delicate operation, the shutdown of an interface is done in two stages. After *shutdown1()* is called, the driver should no longer send packets upstream. However, it should still be prepared to have outstanding packets that were previously sent upstream be returned. It should also allow any pending transmits to complete, and return the buffers to the sender when they do (by calling the **tx_done** callback). The prototype is:

```
int (*shutdown1) (int registrant_hdl,
                  void *func_hdl);
```

The arguments are:

registrant_hdl The registrant handle that was filled in when your driver registered by calling **io-net**'s *reg()* function.

func_hdl The handle you specified for your driver in **io_net_registrant_t**.

This function can return:

- EOK to let the shutdown occur.

  or:

- Some other indication (for example, EBUSY to indicate that there are active transmissions occurring) to prevent the driver from being shut down. It's up to the higher level to retry later. The implication here is that one *can't* force a shutdown of a driver that returns an error indication.

Your driver is still connected to the other modules when *shutdown1()* is called. It's your last chance to transmit data either up or down, which must be done using the thread that called *shutdown1()* because of **io-net**'s locking mechanism.

shutdown2()

The prototype is:

```
int (*shutdown2) (int registrant_hdl,
                  void *func_hdl);
```

When *shutdown2()* is called, the driver should throw away any pending transmits, and return them by calling the *tx_done* callback. The driver should then de-activate the NIC device, and free up any resources that were allocated by the driver, during and since the instantiation of the interface.

The arguments to *shutdown2()* are:

registrant_hdl The handle that was obtained when the driver registered the interface with **io-net**.

func_hdl The pointer that the driver supplied in the **io_net_registrant_t** structure when it registered the interface with **io-net**.

dl_advert()

The prototype is:

```
int (*dl_advert) (int registrant_hdl,
                  void *func_hdl);
```

After a driver registers an interface with **io-net** it sends a message packet upstream in order to advertise its capabilities. However, sometimes the networking subsystem requires that the driver resend the advertisement message upstream. In this case, this entry point will be called. The driver should create a message packet as per "Advertising device capabilities" in the *Writing a Network Driver* chapter and send it upstream.

The arguments are:

registrant_hdl The handle that was obtained when the driver registered the interface with **io-net**.

func_hdl The pointer that the driver supplied in the **io_net_registrant_t** structure when it registered the interface with **io-net**.

devctl()

The prototype is:

```
int (*devctl) (void *func_hdl,
               int dcmd,
               void *data,
               size_t size,
               union _io_net_dcmd_ret_cred *ret);
```

This entry point is called when a *devctl()* (device control) message is sent to be processed by the driver.

The arguments are:

func_hdl The pointer that the driver supplied in the **io_net_registrant_t** structure when it registered the interface with **io-net**.

dcmd The type of *devctl()* that the driver is being asked to process. It may be one of the following:

- DCMD_IO_NET_PROMISCIOUS — this command selects whether or not the interface should be in promiscuous mode. For this **devctl**, *data* points to an integer value. If this value is 0, the interface should be taken out of promiscuous mode. If the value is nonzero, the device should be put into promiscuous mode.
- DCMD_IO_NET_CHANGE_MCAST — this command configures how the interface filters the multicast packets it receives. For this **devctl**, *data* is a pointer to the structure declared as struct **_io_net_msg_mcast**, which is defined in **<sys/io-net.h>**.
- DCMD_IO_NET_GET_CONFIG — this command queries the configuration of the NIC device. For this **devctl**, *data* points to a structure of type **nic_config_t**, defined in **<hw/nicinfo.h>**. The driver fills this structure with information describing the device's configuration.
- DCMD_IO_NET_GET_STATS — this command queries the NIC device's configuration. For this **devctl**, *data* points to a structure of type **nic_stats_t**, defined in **<hw/nicinfo.h>**. The driver fills this structure with statistical information and keeps track of it.
- DCMD_IO_NET_WIFI — this command is intended for 802.11 wireless devices. For this **devctl**, *data* points to a structure of type **nic_wifi_dcmd_t**, defined in **<hw/nicinfo.h>**. The driver should get or set various 802.11 related parameters, based on the contents of this structure.
- SIOCSIFCAP — this command enables or disables offloading of IP header TCP and UDP checksum

computation. For this **devctl**, *data* points to a **struct ifcapreq**, defined in **<net/if.h>**.

This function should return ENOTSUP if the driver receives a *devctl()* it doesn't recognise. It should return EOK if the *devctl()* was processed correctly; otherwise, an appropriate error code should be returned. See **<errno.h>** for details.

data A pointer to data to be passed to the driver, filled in by the driver, or both, depending on the command.

size The maximum amount of data to be sent to the driver or filled in by the driver. If *size* is 0, an unspecified amount of data is transferred.

ret A pointer to additional device data to be returned.

flush()

This function is called to flush out any packets that are pending for transmission on the medium. This is needed for drivers that return from their packet transmission entry point without immediately returning the packet to the originator without having called the *tx_done* callback. When this function returns, the driver should call the *tx_done* callback to make sure all outstanding packets have been returned. The prototype is:

```
int (*flush) (int registrant_hdl,
               void *func_hdl);
```

The arguments are:

registrant_hdl The registrant handle that was filled in when your driver registered by calling **io-net**'s *reg()* function.

func_hdl The handle you specified for your driver in **io_net_registrant_t**.

This function should call **io-net**'s *tx_done()* function for each queued packet, and should return 0.

Classification:

QNX Neutrino

See also:

io_net_registrant_t, **io_net_self_t**, **npkt_t**

io_net_registrant_t

Information used when registering with **io-net**

Synopsis:

```
typedef struct _io_net_registrant {
    uint32_t  flags;
    char      *name;
    char      *top_type;
    char      *bot_type;
    void      *func_hdl;
    io_net_registrant_funcs_t *funcs;
    uint16_t   endpoint;
    int       ndependencies;
} io_net_registrant_t;
```

Description:

The **io_net_registrant_t** structure contains information that's used when registering your driver with **io-net**. It's a member of the **io_net_self_t** structure.

The members are defined as follows:

flags The type and characteristics of the driver being registered. The driver should set this to _REG_PRODUCER_UP. The driver can also optionally set the _REG_ENDPOINT flag.

- _REG_PRODUCER_UP — a producer in the up direction.
- _REG_ENDPOINT — the LAN number was specified by the *lan* driver options; see the description of the *endpoint* field for more details.

name Points to the name of the driver module. For example, the **devn-smc9000.so** driver DLL would set this to point to the string **devn-smc9000**.

top_type An Ethernet driver should make this member point to the string *en*. However, if a different string was specified via the *uptype* driver option, the driver should use this instead.

bot_type Set this field to NULL.

func_hdl This is a function that will be passed to all of the driver's entry points. The driver should specify a pointer that points to its internal data structures associated with the interface.

funcs Specifies a pointer to the driver's entry point table, which is of type **io_net_registrant_funcs_t**, also defined in **<sys/io-net.h>**.

endpoint If the driver wants to let **io-net** automatically assign an interface (LAN) number to the interface, the contents of this field are ignored. If the driver wants to select a LAN number (for example, if a LAN number was specified via the *lan* driver option), the driver should set the _REG_ENDPOINT flag in the *flags* field. In this case, the contents of this field will specify the LAN number, and **io-net** will attempt to assign this LAN number to the interface.

However, the specified LAN number may already be assigned to another interface. If this is the case, **io-net** will assign a different, available interface number instead. In any case, after a successful attempt at registration, the *endpoint* parameter that was passed to the *reg()* function will point to the value of the actual LAN number that was assigned.

Classification:

QNX Neutrino

See also:

io_net_registrant_funcs_t, **io_net_self_t**

io_net_self_t

Functions in **io-net** *that your driver can call*

Synopsis:

```
typedef struct _io_net_self {
    int    *(reg);
    int    *(dereg) (...);
    void    (*alloc) (...);
    npkt_t *(alloc_up_npkt) (...);
    int     (*free) (...);
    paddr_t (*mphys) (...);
    npkt_t  (*tx_up_start) (...);
    int     (*tx_done) (...);
    int     (*devctl) (...);
} io_net_self_t;
```

Description:

The driver should always use the function table to call these functions. In addition, the driver should use a pointer to the function table that was passed to its primary entry point rather than copying the table or caching individual function pointers. The function pointers can change during the interface's lifetime.

The **io_net_self_t** structure points to functions that allow your driver to call back into the networking framework.

The functions are described below.

reg()

This function registers an interface with **io-net**. It should be called once for each NIC interface that the driver wishes to instantiate.

This function *must* be called before any of the other functions in the **io_net_self_t** structure.

The prototype is:

```
int (*reg) (void *dll_hdl,
            io_net_registrant_t *registrant,
            int *reg_hdlp,
            uint16_t *cell,
```

```
uint16_t *endpoint)
```

The arguments are:

dll_hdl The driver should specify the value that was passed into the primary driver entry point.

registrant A pointer to an **io_net_registrant_t** structure that describes how the interface should be instantiated. It also contains a pointer to a table of additional driver entry points.

reg_hdlp On success, a value is stored at the location that this points to, which should be used as the *registrant_hdl* parameter to subsequent calls into **io_net**.

cell The driver should specify a pointer to a 16-bit variable. A value is stored at this location for later use when the driver delivers received packets to the upper layers.

endpoint This is the interface number (LAN number) The driver should specify a pointer to a 16-bit variable. A value is stored at this location for later use when the driver delivers received packets to the upper layers. Typically **io-net** decides how the LAN number is chosen, but it's possible for the driver to influence how the LAN number is chosen. For details, see the *io_net_registrant_t()* entry.

This function returns:

0 Success.

-1 An error occurred; *errno* is set.

dereg()

The prototype is:

```
int (*dereg) (int registrant_hdl)
```

Deregister an interface from **io-net**. Typically, this function is called only if the driver encounters an error after registering with **io-net** and wishes to undo the registration.

The *registrant_hdl* argument is the registrant handle that was filled in via the *reg_hdlp* parameter when your driver registered by calling the *reg()* callback.

This function returns EOK on success, or an error code.

alloc()

The prototype is:

```
void *(*alloc) (size_t size,
                int flags)
```

This function allocates a buffer of the given size that's safe to pass to any other module. It's typically used to allocate buffers to store data that's received from the medium.

The *size* parameter specifies the amount of memory, in bytes, that's to be allocated.

There are currently no flags defined.

This function returns a pointer to the allocated memory, or NULL if an error occurred.

alloc_up_npkt()

The prototype is:

```
npkt_t *(*alloc_up_npkt) (size_t size,
                          void **data)
```

Allocates an **npkt_t** structure suitable to deliver packet data upstream. This function allocates only the structures that describe the packet data, and not the packet data itself.

The arguments are:

size Specifies additional data that should be allocated in addition to the *npkt_t()* structure.

data Points to an address where a pointer to the additional data that was allocated, will be stored.

The additional allocated data can be used to hold buffer descriptor (*net_buf_t()*) structures, and *iov_t()* structures, that point to the packet data. Note that the memory for the *npkt_t()* structure and the additional data will be allocated as a contiguous block of memory, and should be freed with a single call (as opposed to being freed piecemeal.)

This function returns a pointer to the allocated structure, or NULL if an error occurred.

free()

The prototype is:

```
int (*free) (void *ptr)
```

This function frees a buffer, pointed to by *ptr*, that was allocated by the *alloc()* or *alloc_up_npkt()* callbacks.

This function returns:

0 Success.

-1 An error occurred; *errno* is set.

mphys()

The prototype is:

```
paddr_t (*mphys) (void *ptr)
```

This function does a quick lookup of the physical address of the memory, pointed to by *ptr*, that was allocated by either *alloc()*, or *alloc_up_npkt()*.

This function returns the physical address of the buffer on success, or -1 if an error occurred (*errno* is set).

tx_up_start()

The prototype is:

```
npkt_t *(*tx_up_start) (int registrant_hdl,
                        nptk_t *npkt,
                        int off,
                        int framelen_sub,
                        uint16_t cell,
                        uint16_t endpoint,
                        uint16_t iface,
                        void *tx_done_hdl)
```

A function used to send data packet and advertisement messages upstream. This function can be called from the driver's receive event-handler. Note that the thread that calls this function could re-enter the driver through one of its entry points. Be careful not to hold locks when calling this function; one of the driver's entry points could attempt to reacquire them. Note that for all packets sent upstream, the data must be contained in a single fragment.

The arguments are:

registrant_hdl	The registrant handle that was filled in when your driver registered by calling **io-net**'s *reg()* callback.
npkt	A pointer to a linked list of packets to be sent upstream.

off	Zero should be specified.
framelen_sub	Zero should be specified.
cell	Specify the *cell* value supplied to you by **io-net** when you registered.
endpoint	Specify the *endpoint* value supplied to you by **io-net** when you registered.
iface	Zero should be specified.
tx_done_hdl	When the packets that were sent upstream have been processed by the upper layers, a driver entry point is called to return the packet to the driver. The driver can then either release or reuse the *npkt_t()* structures and their associated buffers. The driver entry point is passed an additional argument, so that the driver can access its internal state structures. This parameter specifies the value that's passed to the driver entry point.

This function returns NULL upon success. If this is non-NULL, *errno* is set, and a linked list of *npkt* structures for which a *tx_done()* callback couldn't be registered (i.e. for which *reg_tx_done()* failed) is returned. The driver should immediately release (or reuse) any packets that are returned in this way.

tx_done()

The prototype is:

```
int (*tx_done) (int registrant_hdl,
                 npkt_t *npkt)
```

Notifies the packet's originator that the packet is ready for release or reuse. When the packet data has been copied or transmitted, the driver's transmit routine calls this function. This function should also be called if your driver decides to discard the packet rather than attempt to transmit it.

The *registrant_hdl* argument is the registrant handle that was filled in when your driver registered by calling **io-net**'s *reg()* callback. The *npkt* argument points to a linked list of packets that the driver has finished processing.

This function returns:

0 Success.

-1 An error occurred; *errno* is set.

devctl()

The prototype is:

```
int (*devctl) (int registrant_hdl,
                int dcmd,
                void *data,
                size_t size,
                int *ret)
```

Send a *devctl()* (device control) command to **io-net**.

The arguments are:

registrant_hdl The registrant handle that was filled in when your driver registered by calling **io-net**'s *reg()* callback.

dcmd The command being sent to your driver. Only one value for *dcmd* is currently supported:

- DCMD_IO_NET_CHANGE_MCAST — for multicast support. If the driver loses track of which multicast addresses to accept packets from, it can send this *devctl* so that **io-net** can resend the list of enabled multicast addresses to the driver.

data A pointer to data to be passed to the driver, filled in by the driver, or both, depending on the command.

size	The maximum amount of data to be sent to the driver or filled in by the driver. If *size* is 0, an unspecified amount of data is transferred.
ret	A pointer to additional device data to be returned. (currently unused)

This function returns EOK on success, or an error code.

Classification:

QNX Neutrino

See also:

io_net_dll_entry_t, **npkt_t**

MDI_AutoNegotiate()

Initiate the autonegotiation process

Synopsis:

```
int MDI_AutoNegotiate  ( mdi_t *mdi,
                         int PhyAddr,
                         int Timeout)
```

Arguments:

mdi	A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY).
Timeout	The maximum time in seconds, for the autonegotiation process to complete. Make sure the value specified for *timeout* is greater than one. The recommended value for *timeout* is seven seconds.

If *MDI_NoWait* is specified as the *timeout*, the function returns immediately after initiating autonegotiation.

Description:

The *MDI_AutoNegotiate()* function initiates the autonegotiation process between the PHY and its link partner.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_DeIsolatePhy()

De-isolate the Phy from the MII interface

Synopsis:

```
int MDI_DeIsolatePhy  ( mdi_t * mdi,
                        int PhyAddr)
```

Arguments:

mdi	A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY).

Description:

The *MDI_DeIsolatePhy()* function electrically de-isolates the *PhyAddr* belonging to PHY from the MII interface.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_DeRegister_Extended()

Frees allocated resources

Synopsis:

```
int MDI_DeRegister_Extended ( mdi_t **mdi)
```

Arguments:

mdi A pointer to the **mdi_t** pointer to invalidate.

Description:

This function frees any resources that were allocated in the *MDI_Register_Extended()* function call, and invalidates the **mdi_t** pointer.

Returns:

MDI_SUCCESS if deregistration succeeds.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_DisableMonitor()

Prevent a change callback or a new link

Synopsis:

```
int MDI_DisableMonitor  ( mdi_t *mdi)
```

Arguments:

mdi A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.

Description:

The *MDI_DisableMonitor()* function prevents *MDI_MonitorPhy()* from calling the callback for the driver's link-down status change, or from attempting to establish a new link when no link is detected.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_EnableMonitor()

Allow the link monitor and PHY to communicate

Synopsis:

```
int MDI_EnableMonitor  ( mdi_t *mdi,
                         int LDownTest)
```

Arguments:

mdi	A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.
LDownTest	A test for the link down state.

When the value of *LDownTest* is one, *MDI_MonitorPhy()* attempts to establish a new link by writing to various PHY registers.

Description:

The *MDI_EnableMonitor()* function allows the link monitor to communicate with the PHY, and call the driver's link state change when appropriate. This function doesn't affect the delivery of link monitor pulses to the driver.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_FindPhy()

Determine if a PHY exists

Synopsis:

```
int MDI_FindPhy  ( mdi_t * mdi,
                   int PhyAddr )
```

Arguments:

mdi	A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY). The address range *must* be between 0 and 31 inclusively.

Description:

The *MDI_FindPhy()* function determines if a PHY with an address of PhyAddr exists.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_GetActiveMedia()

Store the active media type for PhyAddr

Synopsis:

```
int MDI_GetActiveMedia ( mdi_t * mdi,
                         int PhyAddr,
                         int *Media)
```

Arguments:

mdi	A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.
phyAddr	The physical address of the physical layer device (PHY).
Media	A pointer to the media-type specified. Possible media types are: • MDI_10bT — 10 Base T, half-duplex • MDI_10bTFD — 10 Base T, full-duplex • MDI_100bT — 100 Base T, half-duplex • MDI_100bTFD — 100 Base T, full-duplex • MDI_1000bT — 1000 Base T, half-duplex • MDI_1000bTFD — 1000 Base T, full-duplex.

Description:

The *MDI_GetActiveMedia()* function stores the curently active media-type for the PHY that the *media* address specifies.

Returns:

- MDI_BADPARAM — *PhyAddr* is out of range
- MDI_LINK_DOWN — no valid link was detected
- MDI_LINK_UP — a valid link was detected, and the link-media type was stored at the address pointed to by *media*.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_GetAdvert()

Store media types currently advertised by the PHY

Synopsis:

```
int MDI_GetAdvert  ( mdi_t * mdi,
                     int PhyAddr,
                     int Advert)
```

Arguments:

mdi — A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.

PhyAddr — The physical address of the physical layer device (PHY).

Advert — A pointer to the memory location where the media types are stored. Valid media types are:

- MDI_10bT — 10 Base T, half-duplex
- MDI_10bTFD — 10 Base T, full-duplex
- MDI_100bT — 100 Base T, half-duplex
- MDI_100bTFD — 100 Base T, full-duplex
- MDI_1000bT — 1000 Base T, half-duplex
- MDI_1000bTFD — 1000 Base T, full-duplex.

The above-mentioned media types are flags. They may be OR'd together.

Description:

The *MDI_GetAdvert()* function stores the media types that are currently advertised by the PHY.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_GetLinkStatus()

Determine the status of the PHY link

Synopsis:

```
int MDI_GetStatusLink  ( mdi_t  *mdi,
                         int  PhyAddr)
```

Arguments:

mdi	A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY).

Description:

This function call gets the link status of the PHY specified by *PhyAddr*.

Returns:

Possible return values for *PhyAddr*are:

- MDI_BADPARAM — *PhyAddr* is out of range
- MDI_LINK_UP — a valid link was detected
- MDI_LINK_DOWN — no link was detected
- MDI_LINK_UNKOWN — the link state is not known.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_GetPartnerAdvert()

Store media types currently advertised by the link partner

Synopsis:

```
int MDI_GetPartnerAdvert ( mdi_t *mdi,
                           int PhyAddr,
                           uint8_t *Advert)
```

Arguments:

mdi — A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.

PhyAddr — The physical address of the physical layer device (PHY).

Advert — A pointer to the memory location where the media types are stored. Valid media-type values are:

- MDI_10bT — 10 Base T, half-duplex
- MDI_10bTFD — 10 Base T, full-duplex
- MDI_100bT — 100 Base T, half-duplex
- MDI_100bTFD — 100 Base T, full-duplex
- MDI_1000bT — 1000 Base T, half-duplex
- MDI_1000bTFD — 1000 Base T, full-duplex.

The above-mentioned media values are flags that may be OR'd together.

Description:

The *MDI_GetPartnerAdvert_Extended()* function stores the media types that are currently advertised by the link partner.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_InitPhy()

Initialize the PHY

Synopsis:

```
int MDI_InitPhy ( mdi_t * mdi,
                  int PhyAddr)
```

Arguments:

mdi	A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY).

Description:

This function initializes the PHY whose address is *PhyAddr*.

You must call this function before you can configure or query the PHY further.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_IsolatePhy()

Isolate the PHY from the MII interface

Synopsis:

```
int MDI_IsolatePhy  ( mdi_t * mdi,
                      int PhyAddr)
```

Arguments:

mdi	A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY).

Description:

The *MDI_IsolatePhy()* function electrically isolates the *PhyAddr* belonging to PHY from the MII interface.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_MonitorPhy()

Check all PHY status

Synopsis:

```
int MDI_MonitorPhy  ( mdi_t *mdi)
```

Arguments:

mdi A pointer to the **mdi_t** structure obtained from the *MDI_Register_Extended()* function call.

Description:

The driver can call this function when it receives a link monitor pulse or a link event interrupt. The *MDI_MonitorPhy()* function checks the status of all PHYs that were initialized with *MDI_InitPHY()*. The function calls the link state change callback if it detects a change to the link state since the last callback, or if this is the first time that *MDI_MonitorPhy()* was called since the PHY was reset.

If the *MDI_EnableMonitor()* function passes a value of one as the *LDownTest* argument, and *MDI_MonitorPhy()* doesn't detect a link, it attempts to establish a new link by writing to various PHY registers.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_Register_Extended()

Register with the MII management library

Synopsis:

```
int MDI_Register_Extended ( void * handle,
                            MDIWriteFunc write,
                            MDIReadFunc read,
                            MDICallBack callback,
                            mdi_t **mdi,
                            struct sigevent *event,
                            int priority,
                            int callback_interval)
```

Arguments:

handle A handle that the library passes to each of the driver's callbacks.

write A pointer to a function which writes to a PHY register through the MAC device. An **MDIWriteFunc** structure is declared as:

```
typedef void    (*MDIWriteFunc)(void *handle, uint8_t phy_id,
                 uint8_t location, uint16_t val);
```

where

handle is the handle that was passed to *MDI_Register_Extended()*,

phy_id is the address of the PHY on the MII management bus,

location is the index of the PHY register to write to, and

val is the value to write to the register.

read A pointer to a function which reads a PHY register through the MAC device. An MDIReadFunc is declared as:

```
typedef uint16_t    (*MDIReadFunc)(void *handle,
                     uint8_t phy_id, uint8_t location);
```

where

handle is the handle that was passed to MDI_Register_Extended(),

phy_id is the address of the PHY on the MII management bus, and

location is the index of the PHY register to read from.

callback A pointer to a function which the library calls if the link state changes. An MDICallback is declared as:

```
typedef void        (*MDICallback) (void *handle,
                    uint8_t phy_id, uint8_t state)
```

where

handle is the handle that was passed to *MDI_Register_extended()*,

phy_id is the address of the PHY on the MII management bus,

state is the link state.

The *state* can be one of:

- MDI_LINK_UP
- MDI_LINK_DOWN
- MDI_LINK_UNKNOWN

If the link state is MDI_LINK_UP, the driver calls *MDI_GetActiveMedia()* to get further information about the link state.

mdi A pointer to an **mdi_t**, structure that the library initializes. The driver passes the pointer to the **mdi_t** structure upon all subsequent calls to the library associated with this registration.

event If the driver wishes to receive link monitor pulses, it should pass a pointer to a **struct sigevent** as the

event argument. The structure's *sigev_coid* field should contain the connection ID through which the driver receives the pulse messages. If the driver doesn't wish to receive the pulses, it should pass NULL.

priority The priority of the link monitor pulses that are delivered. The recommended value is 10.

callback_interval

Specifies the frequency, in seconds, of link monitor pulses. The recommended value is three.

Some device drivers may be able to receive an interrupt upon a link state change event. It's more efficient to use this interrupt, if possible, instead of using link monitor pulses.

Description:

This function registers with the MII management library, and *must* be called before calling any other MII management library function.

Returns:

MDI_SUCCESS if registration succeeds. If any other value is returned, the pointer to the **`mdi_t`** that was returned is invalid, and can't be used used to call other MII management library functions.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_ResetPhy()

Reset the Phy

Synopsis:

```
int MDI_ResetPhy  ( mdi_t * mdi,
                    int PhyAddr,
                    MDI_WaitType Wait)
```

Arguments:

mdi	A pointer to the mdi_t structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY).
Wait	The type of wait to be performed. If you want to complete the reset by the time the function returns, specify *MDI_WaitBusy*. If you want to try to have the driver receive an interrupt after the PHY reset completes, you can specify the *MDI_NoWait* argument. If you use the *MDI_NoWait* argument, this call returns immediately after the reset is initiated. When the reset completes, the driver must call *MDI_SyncPhy()* so the library can perform post-reset servicing.

Description:

The *MDI_ResetPhy()* function resets the *PhyAddr* that belongs to the PHY.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes

continued...

Safety	
Thread	Yes

MDI_SetAdvert_Extended()

Select media types to advertise

Synopsis:

```
int MDI_SetAdvert_Extended  ( mdi_t * mdi,
                              int PhyAddr,
                              int Media)
```

Arguments:

mdi — A pointer to the mdi_t structure obtained from the *MDI_Register_Extended()* function call.

PhyAddr — The physical address of the physical layer device (PHY).

Media — Values for the advertised media-type. Valid values are:

- MDI_10bT — 10 Base T, half-duplex
- MDI_10bTFD — 10 Base T, full-duplex
- MDI_100bT — 100 Base T, half-duplex
- MDI_100bTFD — 100 Base T, full-duplex
- MDI_1000bT — 1000 Base T, half-duplex
- MDI_1000bTFD — 1000 Base T, full-duplex.

The above-mentioned media values are flags. They may be OR'd together.

Description:

The *MDI_SetAdvert_Extended()* function selects the media types to advertise to the PHY's link partner.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_SetSpeedDuplex()

Force the link-state setting

Synopsis:

```
int MDI_SetSpeedDuplex  (mdi_t * mdi,
                         int PhyAddr,
                         intSpeed,
                         int Duplex)
```

Arguments:

mdi	A pointer to the mdi_t structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY) for which the link is to be forced.
Speed	The bit rate, in megabits per second, at which the PHY should operate.
Duplex	The speed at which to operate. Choices are: • half-duplex speed — specify 0 • full-duplex speed — specify 1.

Description:

The *MDI_SetSpeedDuplex()* function forces the link-state to a specific setting instead of allowing link auto-negotiation to occur.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

MDI_SyncPhy()

Synchronize the PHY

Synopsis:

```
int MDI_SyncPhy  (mdi_t * mdi,
                  int PhyAddr)
```

Arguments:

mdi	A pointer to the mdi_t structure obtained from the *MDI_Register_Extended()* function call.
PhyAddr	The physical address of the physical layer device (PHY).

Description:

The *MDI_SyncPhy()* function synchronizes the PHY. Synchronization is necessary after a reset occurs.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

nic_calc_crc_be()

Generate CRC32 checksums for big-endian mode

Synopsis:

```
uint32_t  nic_calc_crc_be  ( char * buf,
                             int len)
```

Arguments:

buf A pointer to the buffer containing mulitcast packet addresses.

len The byte-length of the multicast packet addresses.

Description:

The *nic_calc_crc_be()* function generates Cycle Redundancy Check (CRC32) checksums across the data buffer. Typically, the checksums are used for multicast packet filtering to determine which bit in a hash table corresponds to a given multicast address. The *nic_calc_crc_be()* function generates the CRC by shifting the bits from right to left.

Returns:

The computed 32-bit CRC value.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

`nic_calc_crc_le`

nic_calc_crc_le()

Generate CRC32 checksums for little-endian mode

Synopsis:

```
uint32_t  nic_calc_crc_le  ( char * buf,
                             int len)
```

Arguments:

buf A pointer to the buffer containing mulitcast packet addresses.

len The byte-length of the multicast packet addresses.

Description:

The *nic_calc_crc_le()* function generates Cycle Redundancy Check (CRC32) checksums across the data buffer. Typically, the checksums are used for multicast packet filtering to determine which bit in a hash table corresponds to a given multicast address. The *nic_calc_crc_le()* function generates the CRC by shifting the bits from left to right.

Returns:

The computed 32-bit CRC value.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

`nic_calc_cre_be`

nic_config_t

Structure used to store device configuration information

Synopsis:

```
typedef struct _nic_config_t
    uint32t      revision;
    uint32t      flags;
    uint32t      mtu;
    uint32t      mru;
    uint32t      verbose;
    uint32t      lan;
    uint32t      permanent_address[8];
    uint32t      current_address[8];
    uint32t      mac_length;
    uint32t      connector;
    int32t       phy_addr;
    uint32t      media;
    int32t       media_rate;
    int32t       duplex;
    uint32t      bus_type;
    uint32t      vendor_id;
    uint32t      device_id;
    uint32t      device_index;
    uint32t      device_revision;
    uint32t      serial_number;
    uint32t      num_mem_windows;
    uint32t      num_io_windows;
    uint32t      num_irqs;
    uint32t      num_dma_channels;
    uint64t      mem_window_base[8];
    uint64t      mem_window_size[8];
    uint64t      io_window_base[8];
    uint64t      io_window_size[8];
    uint64t      rom_base;
    uint64t      rom_size;
    uint32t      irq[8];
    uint32t      dma_channel[8];
    uint8t       device_description[64];
    uint8t       up_type[16];
    int32t       iftype;
    uint32t      priority;
} nic_config_t;
```

Description:

The **nic_config_t** structure contains device configuration information and stores information parsed from the driver-option string.

The members are defined as follows:

revision — Should be set to NIC_CONFIG_REVISION.

flags — Valid values are defined by the **nic_flags_t** enumerated types in **<hw/nicinfo.h>**. The following flags are currently defined:

- NIC_FLAG_MULTICAST — multicast packet reception is enabled.
- NIC_FLAG_PROMISCUOUS — the device is currently in promiscuous mode.
- NIC_FLAG_BROADCAST — the device can receive broadcast packets.
- NIC_FLAG_LINK_DOWN — the link is known to be down. Packets can't currently be transmitted or received on the medium.

mtu — Maximum packet size that the device can accept for transmission (including the Ethernet header).

mru — Maximum packet size that the device can successfully receive from the medium (including the Ethernet header).

verbose — The current verbosity level. The higher the verbosity level, the more debug information the driver will emit to the system logger.

lan — The instance (LAN) number of the interface.

permanent_address[8] — The unique station address (MAC address) that the manufacturer assigns to this device (usually read from EEPROM by the driver).

current_address[8]

The station address (MAC address) that the device is currently operating with. This is usually, but not necessarily, the same as the device's permanent address.

mac_length

The length of the device's MAC address, in bytes. This length is six for an Ethernet device.

connector

The type of physical connector that's used to connect the device to the medium. This may be one of the *nic_connector_types* enumerated types, defined in **<hw/nicinfo.h>**. The following values are currently defined:

- NIC_CONNECTOR_UNKNOWN — the driver can't determine the connector type.
- NIC_CONNECTOR_UTP — the device is connected to a UTP (unshielded twisted-pair) cable.
- NIC_CONNECTOR_BNC — the device is connected to a coaxial cable, via a BNC connector.
- NIC_CONNECTOR_FIBER — the device is connected to an optical-fiber cable.
- NIC_CONNECTOR_AUI — the device is connected to a tranceiver via the AUI (Attachment Unit Interface).
- NIC_CONNECTOR_MII — the device is connected to a physical layer (PHY) device, via the MII (Media Independent Interface).
- NIC_CONNECTOR_STP — the device is connected to an STP (shielded twisted-pair) cable.

phy_addr

The address used to communicate with the PHY device, in order to access its internal registers.

media Specifies the type of the medium over which the device communicates. This may be one of the *nic_media_types* enumerated types, defined in **<hw/nicinfo.h>**. The following types are currently defined:

- NIC_MEDIA_802_3 — the medium is that defined by the IEEE 802.3 standard (CSMA/CD, Ethernet).
- NIC_MEDIA_802_5 — the medium is that defined by the IEEE 802.5 standard (Token Ring).
- NIC_MEDIA_FDDI — the medium is that defined by the ISO 9314 standard– FDDI (Fiber Distributed Data Interface).
- NIC_MEDIA_ATM — the medium is ATM (Asynchronous Transfer Mode).
- NIC_MEDIA_802_11 — the medium is that defined by the IEEE 802.11 standard (WiFi).

media_rate The current media rate, in Kbits per second, at which the device is operating. If the current operation rate is unknown, set this field to -1.

duplex The current duplex at which the device is operating. A value of 0 means half-duplex; a value of 1 means full-duplex. If the current duplex setting is unknown, set this field to -1.

bus_type The type of bus through which the device is connected to the host system. See the defined values in **<drvr/common.h>**.

vendor_id Specifies the PCI Vendor ID for a PCI device that was assigned to the vendor of the device, which is readable from the PCI configuration space.

device_id Specifies the PCI Device ID for a PCI device that was assigned by the vendor of the device, which is readable from the PCI configuration space.

device_index For a PCI device, this is used to uniquely identify a particular instance of the device, in conjunction with *vendor_id* and *device_id*. Where there are multiple instances of a device in the system with identical Vendor and Device IDs, these devices are each assigned a unique number. The numbers that are assigned are sequential, beginning at zero. For a non-PCI device, this is an index that addresses an instance of the device in the system. The mapping from index to device is driver-dependent.

device_revision For a PCI device, this is the device revision that is readable from the PCI configuration space. For other devices, the meaning of the revision number is driver-dependent.

serial_number Specifies a driver-dependent serial number.

num_mem_windows

Specifies the number of memory-mapped apertures that are used to access the device.

num_io_windows Specifies the number of I/O-mapped apertures that are used to access the device.

num_irqs Specifies the number of interrupt vector numbers that the driver attaches to in order to receive interrupt events from the device.

num_dma_channels

Specifies the number of DMA channels used to transfer data between the device and memory.

mem_window_base[8]

This array contains the physical base-addresses of the device's memory-mapped apertures.

mem_window_size[8]

This array contains the sizes, in bytes, of the device's memory-mapped apertures.

io_window_base[8]

This array contains the base addresses, in I/O address space, of the device's I/O-mapped apertures.

io_window_size[8]

This array contains the sizes, in bytes, of the device's I/O-mapped apertures.

rom_base If the device has a memory-mapped ROM associated with it, this specifies the physical address of the ROM.

rom_size If the device has a memory-mapped ROM associated with it, this specifies the size, in bytes, of the ROM.

irq[8] This array contains the interrupt-vector numbers that the driver attaches to in order to receive interrupt events from the device.

dma_channel[8] This array contains the DMA channels that transfer data between the device and memory.

device_description[64]

This is a NULL-terminated user-readable string describing the device. It should describe the make and model of the device.

uptype[16] This is a NULL-terminated string which describes the type of interface the device presents. For an Ethernet device, is should be set to *en*. This string indicates to higher-level software, how the data packets going to and from the driver will be formatted.

iftype	This is one of the interface types from **<net/if_types.h>**. For an Ethernet device, it should be set to IFT_ETHER.
priority	Specifies the priority at which the driver's event-handling thread should run. The default recommended value is 21.

Classification:

QNX Neutrino

nic_dump_config()

Output configuration information

Synopsis:

```
void  nic_dump_config  ( nic_config_t * cfg)
```

Arguments:

cfg A pointer to the structure containing configuration information.

Description:

The *nic_dump_config()* function sends stored generic configuration information to the system logger.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

nic_ethernet_stats_t

Ethernet statistics for a **nicinfo** *command*

Synopsis:

```
typedef struct nic_ethernet_stats {
    uint32t     valid_stats;
    uint32t     align_errors;
    uint32t     single_collisions;
    uint32t     multi_collisions;
    uint32t     fcs_errors;
    uint32t     tx_deferred;
    uint32t     late_collisions;
    uint32t     xcoll_aborted;
    uint32t     internal_tx_errors;
    uint32t     no_carrier;
    uint32t     internal_rx_errors;
    uint32t     excessive_deferrals;
    uint32t     length_field_outrange;
    uint32t     oversized_packets;
    uint32t     sqe_errors;
    uint32t     symbol_errors;
    uint32t     jabber_detected;
    uint32t     short_packets;
    uint32t     total_collision_frames;
    uint32t     dribble_bits;
} nic_ethernet_stats_t;
```

Description:

This structure holds Ethernet-specific statistics.

Your driver must fill in the following members:

valid_stats A set of flags that indicate what Ethernet-specific statistics the driver keeps track of. The following flags are defined:

- NIC_ETHER_STAT_ALIGN_ERRORS — the *align_errors* field is valid.
- NIC_ETHER_STAT_SINGLE_COLLISIONS — the *single_collisions* field is valid.

- NIC_ETHER_STAT_MULTI_COLLISIONS — the *multi_collisions* field is valid.
- NIC_ETHER_STAT_FCS_ERRORS — the *fcs_errors* field is valid.
- NIC_ETHER_STAT_TX_DEFERRED — the *tx_deferred* field is valid.
- NIC_ETHER_STAT_LATE_COLLISIONS — the *late_collisions* field is valid.
- NIC_ETHER_STAT_XCOLL_ABORTED — the *xcoll_aborted* field is valid.
- NIC_ETHER_STAT_INTERNAL_TX_ERRORS — the *internal_tx_errors* field is valid.
- NIC_ETHER_STAT_NO_CARRIER — the *no_carrier* field is valid.
- NIC_ETHER_STAT_INTERNAL_RX_ERRORS — the *internal_rx_errors* field is valid.
- NIC_ETHER_STAT_EXCESSIVE_DEFERRALS — the *excessive_deferrals* field is valid.
- NIC_ETHER_STAT_LENGTH_FIELD_MISMATCH — the *length_field_mismatch* field is valid.
- NIC_ETHER_STAT_LENGTH_FIELD_OUTRANGE — the *length_field_outrange* field is valid.
- NIC_ETHER_STAT_OVERSIZED_PACKETS — the *oversized_packets* field is valid.
- NIC_ETHER_STAT_SQE_ERRORS — the *sqe_errors* field is valid.
- NIC_ETHER_STAT_SYMBOL_ERRORS — the *symbol_errors* field is valid.
- NIC_ETHER_STAT_JABBER_DETECTED — the *jabber_detected* field is valid.

- NIC_ETHER_STAT_SHORT_PACKETS — the *short_packets* field is valid.
- NIC_ETHER_STAT_TOTAL_COLLISION_FRAMES — the *total_collision_frames* field is valid.
- NIC_ETHER_STAT_DRIBBLE_BITS — the *dribble_bits* field is valid.

align_errors The number of frames received that aren't an integral number of bytes in length. Corresponds to the AlignmentErrors attribute defined by the 802.3 spec.

single_collisions The number of frames that experienced a single collision upon transmission, but were subsequently transmitted successfully. Corresponds to the SingleCollisionFrames attribute defined by the 802.3 spec.

multi_collisions The number of frames that experienced more than one collision upon transmission, but were subsequently transmitted successfully. Corresponds to the MultipleCollisionFrames attribute defined by the 802.3 spec.

fcs_errors The number of frames received that are an integral number of bytes in length, but had an incorrect Frame Check Sequence field. Corresponds to the FrameCheckSequence attribute, defined by the 802.3 spec.

tx_deferred The number of frames that experienced a delay during transmission because the medium was busy. Corresponds to the FramesWithDeferredXmissions attribute, defined by the 802.3 spec.

late_collisions The number of frames that experienced a late collision (out-of-window collision) upon

transmission. Corresponds to the LateCollisions attribute defined by the 802.3 spec.

xcoll_aborted The number of frame transmissions that were aborted due to excessive collisions. Corresponds to the FramesAbortedDueToXSColls attribute defined by the 802.3 spec.

internal_tx_errors

The number of transmits that failed due to an internal error in the NIC device. The most common type of internal transmit error that occurs with a typical device implementation is a FIFO underrun. Corresponds to the FramesLostDueToIntMACXmitErrors attribute defined by the 802.3 spec.

no_carrier The number of times a carrier signal wasn't detected during frame transmission. Corresponds to the CarrierSenseErrors attribute defined by the 802.3 spec.

internal_rx_errors

The number of packet reception attempts that failed due to an internal error in the NIC device. The most common type of internal receive error that occurs with a typical device implementation is a FIFO overrun. Corresponds to the FramesLostDueToIntMACRcvErrors attribute defined by the 802.3 spec.

excessive_deferrals

The number of frame transmissions that were aborted due to excessive deferral. Corresponds to the FramesWithExcessiveDeferral attribute defined by the 802.3 spec.

length_field_mismatch

The number of received packets where the type/length field in the Ethernet header is in the valid range to specify a packet length, but this length doesn't match the number of bytes that were actually received. Corresponds to the InRangeLengthErrors attribute defined by the 802.3 spec.

length_field_outrange

The number of received packets where the type/length field in the Ethernet header isn't in the valid range to specify a type, but the value is greater than the maximum Ethernet packet size. Corresponds to the OutOfRangeLengthField attribute defined by the 802.3 spec.

oversized_packets

The number of received packets whose length is greater than the maximum Ethernet packet size. Corresponds to the OutOfRangeLengthField attribute defined by the 802.3 spec.

sqe_errors The number of signal-quality errors detected on the medium. Corresponds to the SQETestErrors attribute defined by the 802.3 spec.

symbol_errors The number of invalid symbols detected on the medium while a carrier signal was present. Corresponds to the SymbolErrorDuringCarrier attribute defined by the 802.3 spec.

jabber_detected The number of times a 10Mbit transceiver entered the jabber state. Corresponds to the Jabber attribute defined by the 802.3 spec.

short_packets The number of packets received that were below the minimun Ethernet frame size. Corresponds to the Runts attribute defined by the 802.3 spec.

total_collision_frames
: The total number of packets that experienced one or more collisions during transmission.

dribble_bits
: The total number of packets received where extraneous bits were present on the media at the end of the frame, but the frame was otherwise valid.

Classification:

QNX Neutrino

nic_get_syspage_mac()

Retrieve a stored MAC address

Synopsis:

```
int nic_get_syspage_mac ( char *mac)
```

Arguments:

mac A pointer to the Media Access Control (MAC) address stored in the system page.

Description:

The *nic_get_syspage_mac()* function retrieves, during system startup, a MAC address that was stored in the system page. If a driver is unable to determine the MAC address that was manufacturer-assigned to the interface (e.g. by reading it from an SROM), it should use this function instead.

If this function fails, you'll need to specify a MAC address on the command line, otherwise, the driver should send an error message to the system logger and the instantiation of the interface should fail.

Returns:

0 if a MAC address was successfully retrieved from the syspage; -1 on failure.

Classification:

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

nic_parse_options()

Parse an option string for a network driver

Synopsis:

```
int nic_parse_options  ( nic_config_t *cfg,
                         char *option)
```

Arguments:

cfg A pointer to the members of the **nic_config_t** structure that the option string updates.

Based on the contents of the option string, this function updates various members of the **nic_config_t** structure, which the *cfg* argument points to. The various options that are recognized by this function affect the structure as follows:

ioport	The specified value is stored in the next free element (the element indexed by the *num_io_windows* field) of the *io_window_base* array. The *num_io_windows* field is incremented.
irq	The specified value is stored in the next free element (the element indexed by the *num_irqs* field) of the *irq* array. The *num_irqs* field is incremented.
dma	The specified value is stored in the next free element (the element indexed by the *num_dma_channels* field) of the *dma_channel* array. The *num_dma_channels* field is incremented.
vid	The specified value is stored in the *vendor_id* field.
did	The specified value is stored in the *device_id* field.

pci	The specified value is stored in the *device_index* field.
mac	The specified string is converted to an array of bytes, and stored in the *current_address* array.
lan	The specified value is stored in the *lan* field.
mtu	The specified value is stored in the *mtu* field.
mru	The specified value is stored in the *mru* field.
speed	The specified value is stored in the *media_rate* field.
duplex	The specified value is stored in the *duplex* field.
media	The specified value is stored in the *media* field.
promiscuous	This option doesn't need an argument. If specified, the NIC_FLAG_PROMISCUOUS flag is set in the flags field.
nomulticast	This option doesn't need an argument. If specified, the NIC_FLAG_MULTICAST flag is cleared in the flags field.
connector	The specified value is stored in the *connector* field.
deviceindex	The specified value is stored in the *device_index* field.
phy	The specified value is stored in the *phy_addr* field.
memrange	The specified value is stored in the next free element (i.e. the element indexed by the *num_mem_windows* field) of the *mem_window_base* array, and the *num_mem_windows* field is incremented. If a window size was specified in addition to a

	base (the base value was followed by a colon, followed by the length value), the element of the *mem_window_size* array at the corresponding array index is updated.
iorange	The specified value is stored in the next free element (i.e. the element indexed by the *num_io_windows* field) of the *io_window_base* array, and the *num_io_windows* field is incremented. If a window size was specified in addition to a base (the base value was followed by a colon, followed by the length value), the element of the *io_window_size* array at the corresponding array index is updated.
verbose	If specified *without* an argument, the verbose field is incremented. If specified *with* an argument, the verbose field is set to the specified value.
iftype	The specified value is stored in the *ifype* field.
uptype	The specified string value is copied into the *uptype* array.
priority	The specified value is stored in the *priority* field.

The driver typically initializes the fields of the **nic_config_t** structure with default values, before it parses the options. This allows the user to override the default values via driver options. In some cases, the driver should initialize a field with an invalid value.

For example, if the driver sets the speed and duplex values to -1, it will be able to tell, after the options have been parsed, whether the user attempted to explicitly set the speed and/or duplex to specific values. Then the driver can determine whether to force the link configuration, or to allow link autonegotiation/autodetection to take place.

The driver should set NIC_FLAG_MULTICAST in the *flags* field *before* parsing the options. If the *nomulticast* option is specified, this flag is subsequently cleared.

option The options to parse.

Description:

The *nic_parse_options()* function assists in parsing a driver network string. This function parses standardized options. Drivers can parse their driver-specific option strings with the *getsubopt()* function. Standardized options have a well-defined behavior that's consistent across all network drivers.

If *getsubopt()* doesn't recognize an option as being driver-specific, the option should then be passed to the *nic_parse_option()* function. It will try to interpret the option; if it can't, the **nic_config_t** structure will be updated appropriately. If the driver uses the *nic_parse_options()* function for option parsing, the **nic_config_t** structure stores the results.

CAUTION:

The *getsubopt()* function modifies the option string that's passed to it, by changing commas to spaces. You should have the driver make a copy of the option string before using *getsubopt()* to parse it.

Examples:

Here's an example of how the fictitious "toad" driver, which has two driver-specific options, would parse its options. In the example, the **`toad_device_t`** structure is a driver-specific structure the driver uses to store its internal state.

```
#include  <sys/slog.h>
#include  <sys/slogcodes.h>
#include  <string.h>
#include  <stdlib.h>
#include  <errno.h>
#include  <drvr/nicsupport.h>
int
toad_parse_options(toad_device_t *toad,
    const char *optstring, nic_config_t *cfg)
{
    char        *value;
    int     opt;
    char        *options, *freeptr;
    char        *c;
    int     err = EOK;

       static char *toad_opts[] = {
               "receive",
               "transmit",
               NULL
       };
       enum {
               TOADOPT_RECEIVE = 0,
               TOADOPT_TRANSMIT
       };

       if (optstring == NULL)
           return 0;

       /* getsubopt() is destructive */
       freeptr = options = strdup(optstring);
```

```
    while (options && *options != '\0') {
        c = options;
        if ((opt = getsubopt( & options, toad_opts, & value)) == -1) {
            if (nic_parse_options(cfg, value) == EOK)
                continue;
            nic_slogf(_SLOGC_NETWORK, _SLOG_WARNING,
               "devn-toad: unknown option %s", c);
           err = EINVAL;
           break;
       }

       switch (opt) {
           case TOADOPT_RECEIVE:
               if (toad != NULL)
                   toad->num_rx_descriptors =
                       strtoul(value, 0, 0);
               continue;
           case TOADOPT_TRANSMIT:
               if (toad != NULL)
                   toad->num_tx_descriptors =
                       strtoul(value, 0, 0);
               continue;
           default:
               /* Impossible */
       }
   }

   free(freeptr);
    errno = err;

    return (err == EOK) ? 0 : -1;
}
```

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes

continued...

Safety	
Thread	Yes

See also:

`nic_config_t`

nic_slogf()

Output messages and debug information

Synopsis:

```
int nic_slogf  ( int opcode,
                 int severity,
                 const char *fmt... )
```

Arguments:

opcode	A combination of a *major* and *minor* code.
severity	The severity of the log message.
fmt	A standard *printf()* string followed by *printf()* arguments.

Description:

The *nic_slogf()* function outputs error messages, informational messages, or debug information to the device. This function is a cover for *slogf()*.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

slogf()

nic_stats_t

Statistical information for devices

Synopsis:

```
typedef struct nic_stats {
    uint32t revision;
    uint32t media;
    union {
        nic_ethernet_stats_t estats;
        nic_wifi_stats_t wstats;
    } un;
    uint32t valid_stats;
    uint32t txed_ok;
    uint32t rxed_ok;
    uint64t octets_txed_ok;
    uint64t octets_rxed_ok;
    uint32t txed_multicast;
    uint32t rxed_multicast;
    uint32t txed_broadcast;
    uint32t rxed_broadcast;
    uint32t tx_failed_allocs;
    uint32t rx_failed_allocs;
};
```

Description:

The **net_stats_t** structure is used when a module wants to keep track of mandatory and non-mandatory statistical information. Higher-level software may query the driver's statistical counter by issuing the DCMD_IO_NET_GET_STATS *devctl()*. The results from the **devctl** are stored in **net_stats_t**.

The members include:

revision — Set this field to NIC_STATS_REVISION.

media — Describes the device medium. It must be set to one of the nic_media_types enumerated types defined in **<hw/nicinfo.h>**. It's important to set this field correctly, since it affects how the rest of the structure will be interpreted.

un.estats — If the media type is NIC_MEDIA_802_3, this structure should be filled with Ethernet-specific statistics. See **nic_ethernet_stats_t** structure for a description of the Ethernet-specific statistics structure.

un.wstats — If the media type is NIC_MEDIA_802_11, this structure should be filled with wireless-specific statistics. See **nic_wifi_stats_t** for a description of the wireless-specific statistics structure.

valid_stats — A set of flags that indicate which generic statistics the driver keeps track of. The following flags are defined:

- NIC_STAT_TXED_MULTICAST — the *txed_multicast* field is valid.
- NIC_STAT_RXED_MULTICAST — the *rxed_multicast* field is valid.
- NIC_STAT_TXED_BROADCAST — the *txed_broadcast* field is valid.
- NIC_STAT_RXED_BROADCAST — the *rxed_broadcast* field is valid.
- NIC_STAT_TX_FAILED_ALLOCS — the *tx_failed_alloced* field is valid.
- NIC_STAT_RX_FAILED_ALLOCS — the *rx_failed_alloced* field is valid.

txed_ok — This mandatory statistic counts the number of packets transmitted successfully.

rxed_ok — This mandatory statistic counts the number of packets received successfully.

octets_txed_ok — This mandatory statistic counts the number of bytes transmitted successfully.

octets_rxed_ok This mandatory statistic counts the number of bytes received successfully.

txed_multicast Counts the number of multicast packets the interface transmits.

rxed_multicast Counts the number of multicast packets the interface receives.

txed_broadcast Counts the number of broadcast packets the interface transmits.

rxed_broadcast Counts the number of broadcast packets the interface receives.

txed_failed_allocs

Counts the number of dropped packets that couldn't be transmitted because of a failed attempt to allocate memory.

rxed_failed_allocs

Counts the number of dropped packets that couldn't be received because of a failed attempt to allocate memory.

Classification:

QNX Neutrino

nic_strtomac()

Convert a MAC address

Synopsis:

```
int nic_strtomac  ( const char *s,
                    unsigned char *mac)
```

Arguments:

s A pointer to the MAC address string.

mac A pointer to the MAC address to convert.

Description:

The *nic_strtomac()* function converts a MAC address from a string form to a numeric form. The string may be any of the following forms:

- **xxxxxxxxxxxx** — where "x" is a hexidecimal digit in one of the folllowing ranges:
  - 0-9
  - a-z
  - A-Z
- **xxxxxxxxxxxx** — where "x" is a hexidecimal digit in one of the folllowing ranges:
  - 0-9
  - a-z
  - A-Z

The *nic_strtomac()* function assumes that the MAC address is six-bytes long.

Returns:

non-zero if the MAC address is invalid; 0 if it's valid.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

nic_wifi_dcmd_t

Get WiFi-specific parameters

Synopsis:

```
typedef struct {
    uint32t       subcmd;
    uint32t       size;
    union         ssid;
    union         bss_type;
    union         station_name;
    union         channel;
    union         auth_type;
    union         crypto_type;
    union         wep_key;
    union         wep_cfg;
    union         bss_cfg;
    union         signal_info;
    } un;
}nic_wifi_dcmd_t;
```

Description:

When the driver receives a DCMD_IO_NET_WIFI devctl, it's passed a pointer to a structure of **nic_wifi_dcmd_t**. This *devctl()* either gets or sets various WiFi-specific parameters.

The members are defined as follows:

subcmd
: The WiFi-specific parameter that's to be read or configured. Either the DEVDIR_TO or DEVDIR_FROM flag will be set in this field (these flags are defined in **<devctl.h>**.)

  If DEVDIR_TO is set, the **devctl** is supplying information to the driver so it can set a particular property. If DEVDIR_FROM is set, the **devctl** is querying the current value of a particular property.

  To determine which property is to get or set, the driver should logically AND the value of this field with NIC_WIFI_SUBCMD_MASK, defined in **<hw/nicinfo.h>**.

size Currently unused.

ssid Stores the SSID (Service Set Identifier), when the NIC_WIFI_SUBCMD_SSID property is specified in the *subcmd* field.

bss_type Stores the BSS (Basic Service Set) type, when the NIC_WIFI_SUBCMD_BSS_TYPE property is specified in the subcmd field.

Valid values for this field are:

- NIC_WIFI_BSS_TYPE_BSS — Basic Service Set
- NIC_WIFI_BSS_TYPE_IBSS — Independant Basic Service Set
- NIC_WIFI_BSS_TYPE_ADHOC — Ad-hoc mode
- NIC_WIFI_BSS_TYPE_AP — Access point

station_name Stores the name of the base station, when the NIC_WIFI_SUBCMD_STATION_NAME property is specified in the *subcmd* field.

channel Stores the communication channel, when the NIC_WIFI_SUBCMD_CHANNEL property is specified in the *subcmd* field.

auth_type Stores the authentication type, when the NIC_WIFI_SUBCMD_AUTH_TYPE property is specified in the *subcmd* field. Valid values are:

- NIC_WIFI_AUTH_TYPE_OPEN
- NIC_WIFI_AUTH_TYPE_SHARED_KEY

crypto_type Stores the encryption type, when the NIC_WIFI_SUBCMD_CRYPTO_TYPE property is specified in the *subcmd* field. Valid values are:

- NIC_WIFI_CRYPTO_TYPE_NONE — no encryption

- NIC_WIFI_CRYPTO_TYPE_WEP — Wired Equivalent Privacy

wep_key

Stores the encryption key, when the NIC_WIFI_SUBCMD_CRYPTO_DATA property is specified in the *subcmd* field.

This field is a structure, for which the following fields are defined:

- *num* — a key identifier, which is a number between one and four, inclusive.
- *length* — the length of the key, in bytes.
- *data* — the actual key.

wep_cfg

Stores encryption configuration information, when the NIC_WIFI_SUBCMD_CRYPTO_CFG property is specified in the *subcmd* field.

This field is a structure, for which the following field is defined:

- *active_key* — selects which key is currently active. This value may be zero (disables encryption), or a key identifier between one and four, inclusive.

bssid_cfg

Stores BSS (Basic Service Set) configuration information, when the NIC_WIFI_SUBCMD_BSSID property is specified in the *subcmd* field.

This field is a structure, for which the following fields are defined:

- *macaddr* — stores the 6-byte station (MAC) address.
- *channel* — stores the communication channel, which may also be configured or read via the NIC_WIFI_SUBCMD_CHANNEL sub-command.

signal_info Stores information about the carrier signal, when the NIC_WIFI_SUBCMD_BSSID property is specified in the *subcmd* field. This is a *read-only* property.

This field is a structure, for which the following fields are defined:

- *radio_freq* — specifies the frequency of the carrier signal in hundreds of megahertz, i.e. a value of 24 means 2.4 gigahertz.
- *tx_rate* — specifies the data transfer bit-rate, in hundreds of kilobits per second, i.e. a value of 55 is 5.5 Mbits/sec.
- *quality*, *signal_level*, *noise_level* — these values give percentages, rounded to the nearest decimal point, which indicate the signal quality, signal level, and noise level of the carrier signal, respectively.

Classification:

QNX Neutrino

nic_wifi_stats_t

Get WiFi-specific statistics

Synopsis:

```
typedef struct _nic_wifi_stats {
    uint32t     valid_stats;
    uint32t     tx_gragment;
    uint32t     tx_multicast;
    uint32t     tx_failed;
    uint32t     tx_retry;
    uint32t     tx_multi_retry;
    uint32t     rts_success;
    uint32t     rts_failure;
    uint32t     ack_failure;
    uint32t     duplicate;
    uint32t     rx_fragment;
    uint32t     rx_multicast;
    uint32t     fcs_errors;
} nic_wifi_stats_t;
```

Description:

This structure holds WiFi-specific statistics.

Your driver must fill in the following members:

valid_stats A set of flags that indicate what WiFi-specific statistics the driver keeps track of. The following flags are defined:

- NIC_WIFI_STAT_TX_FRAGMENT — the *tx_fragment* field is valid.
- NIC_WIFI_STAT_TX_MULTICAST — the *tx_multicast* field is valid.
- NIC_WIFI_STAT_TX_FAILED — the *tx_failed* field is valid.
- NIC_WIFI_STAT_TX_RETRY — the *tx_retry* field is valid.
- NIC_WIFI_STAT_TX_MULTI_RETRY — the *tx_multi_retry* field is valid.

- NIC_WIFI_STAT_RTS_SUCCESS — the *rts_success* field is valid.
- NIC_WIFI_STAT_RTS_FAILURE — the *rts_failure* field is valid.
- NIC_WIFI_STAT_ACK_FAILURE — the *ack_failure* field is valid.
- NIC_WIFI_STAT_DUPLICATE — the *duplicate* field is valid.
- NIC_WIFI_STAT_RX_FRAGMENT — the *rx_fragment* field is valid.
- NIC_WIFI_STAT_RX_MULTICAST — the *rx_multicast* field is valid.
- NIC_WIFI_STAT_FCS_ERRORS — the *fcs_errors* field is valid.

tx_fragment The number of data or management fragments that were transmitted successfully. This number corresponds to the TransmittedFragmentCount attribute defined by the IEEE 802.11 specification.

tx_multicast The number of multicast frames that were transmitted sucessfully. This number corresponds to the MulticastTransmittedFrameCount attribute defined by the IEEE 802.11 specification.

tx_failed The number of frame transmissions that were aborted because they exceeded the retry limits. This number corresponds to the FailedCount attribute defined by the IEEE 802.11 specification.

tx_retry The number of frames that were successfully transmitted, after one or more retries. This number corresponds to the RetryCount attribute defined by the IEEE 802.11 specification.

tx_multi_retry The number of frames that were successfully transmitted, after more than one retry. This number

corresponds to the MultipleRetryCount attribute defined by the IEEE 802.11 specification.

rts_success The number of times a clear to send (CTS) was received in response to a request to send (RTS). This number corresponds to the RTSSuccessCount attribute defined by the IEEE 802.11 specification.

rts_failure The number of times a CTS was not received in response to an RTS. This number corresponds to the RTSFailureCount attribute defined by the IEEE 802.11 specification.

ack_failure The number of times an unexpected ACK was received. This number corresponds to the AckFailureCount attribute defined by the IEEE 802.11 specification.

duplicate The number of times a duplicate frame was received. This number corresponds to the FrameDuplicateCount attribute defined by the IEEE 802.11 specification.

rx_fragment The number of data or management fragments that were successfully received. This number corresponds to the ReceivedFragmentCount attribute defined by the IEEE 802.11 specification.

rx_multicast The number of multicast frames that were successfully received. This number corresponds to the MulticastReceivedFrameCount attribute defined by the IEEE 802.11 specification.

fcs_errors The number of received frames that had frame check-sequence errors. This number corresponds to the FCSErrorCount attribute defined by the IEEE 802.11 specification.

Classification:

QNX Neutrino

npkt_t

Data structure describing a packet

Synopsis:

```
typedef struct _npkt {
    net_buf      buffers;
    npkt_t       *next;
    void         *org_data;
    uint32_t     flags;
    uint32_t     framelen;
    uint32_t     tot_iov;
    uint32_t     csum_flags;
    uint32_t     ref_cnt;
    uint16_t     req_complete;
    union {
        void     *p;
        unsigned char c [16];
    } inter_module;
} npkt_t;
```

Description:

A packet consists of an **npkt_t** structure, which has data buffers associated with it. If the driver wants to create a packet to send upstream, it should call *alloc_up_npkt()*.

A data buffer is described by a structure of type **net_buf_t**, as defined in **<sys/io-net.h>**. The data in a buffer is comprised of one or more contiguous fragments. Each fragment is described by a **net_iov_t** structure (also defined in **<sys/io-net.h>**) that contains a pointer to the fragment's data, the size of the fragment, and the physical address of the fragment. Note that packets being sent upstream must consist of a single fragment.

The **npkt_t** structure is defined in **<sys/io-net.h>**.

The **npkt_t** structure is the main data structure for a packet. The following fields of the **npkt_t** structure are of importance to the network driver:

buffers Points to a queue of data buffers. The buffer queues can be manipulated and traversed by a set of macros

defined in **<sys/queue.h>**. See the examples below for the kind of operations a driver would need to perform on buffer queues.

next — Used for chaining packets into a linked list. The last item in the list is set to NULL.

org_data — For the sole use of the originator of the packet. The driver should only modify or interpret this field if the driver was the originator of the packet.

flags — The logical OR of zero or more of the following:

If you're using the new lightweight Qnet, a network driver developed with releases prior to 6.3 could malfunction because the assignment of the bits in the *flags* field of the **npkt_t** structure has changed. See _NPKT_ORG_MASK and _NPKT_SCRATCH_MASK in **<sys/io-net.h>**.

- _NPKT_NOT_TXED — if the driver couldn't transmit a packet, for whatever reason, it should set this flag before calling **tx_done** to indicate the packet is known to have been dropped.
- _NPKT_UP — should be set for packets originating from the driver.
- _NPKT_MSG — indicates that the packet doesn't contain data, but rather contains a message. A driver will set this flag when it sends a capabilities message upstream.
- _NPKT_PROMISC — the upper 12 bits of the *flags* field are reserved for the driver's internal purposes.

  The driver can use the eight most significant bits while it's processing a packet. The driver shouldn't make assumptions about the state of these bits when it receives a packet from the upper layers.

The next four most significant bits are for the use of the originator of a packet. The driver can use these flags for packets being sent upstream. If a packet didn't originate with the driver, the driver must not alter these flags.

framelen The total size of the packet data, in bytes, including the Ethernet header.

tot_iov The total number of fragments that comprise the packet data. This number must be one for packets being sent upstream.

csum_flags Used for hardware checksum offloading. See the "Hardware checksum offloading" section in the Writing a Network Driver chapter for more details.

ref_cnt For packets originating from the driver, this should be set to one.

req_complete For packets originating from the driver, this should be set to zero.

net_buf_t

A queue of structures of type **net_buf_t** is used to describe the data fragments that are associated with the packet.

```
typedef struct _net_buf {
    TAILQ_ENTRY (_net_buf)  ptrs;
    int         niov;
    net_iov_t  *net_iov;
} ;
```

The members of this structure are as follows:

ptrs Used by the queue manipulation macros to create queues of buffers.

niov The number of data fragments associated with the buffer.

net_iov Points to an array of data structure descriptors.

net_iov_t

The **net_iov_t** structure is used to describe the data fragment descriptors associated with the packet.

```
typedef struct _net_iovec {
    void   *iov_base;
    paddr_t iov_phys;
    size_t  iov_len;
} ;
```

The members of this structure are as follows:

iov_base Points to the data fragment.

iov_phy The physical address of the data fragment.

iov_len The size of the data fragment, in bytes.

Classification:

QNX Neutrino

Glossary

802.3

A standard defined by the IEEE that defines the operation of a type of network, often referred to as “Ethernet”.

802.11

A standard defined by the IEEE that defines the operation of a type of wireless network, often referred to as “WiFi”.

Big-endian

Describes a layout by which numeric values are stored in memory. The most-significant bytes of numeric values are stored at the lower addresses.

Bus-mastering

A hardware mechanism whereby a device other than the CPU can directly transfer data to or from memory.

DDK

Device Driver Kit.

DMA

Direct Memory Access. A hardware mechanism whereby data can be transferred between a device other than the CPU and memory, without the CPU being involved in the memory access cycles.

DLL

Dynamically Loadable Library.

Ethernet

A type of network that involves the transmission of data packets across a physical medium (see 802.3).

IEEE

The Institute of Electrical and Electronics Engineers.

LAN

Local Area Network.

Little-endian

Describes a layout by which numeric values are stored in memory. The least-significant bytes of numeric values are stored at the lower addresses.

MAC

Media Access Control. The protocol used by devices on a network to arbitrate access to the media.

MII

Media Independant Interface. An interface, defined by the 802.3 standard, for connecting a MAC device to a PHY device.

MOST bus

Media-Oriented Systems Transport bus. An optical bus designed for use in vehicles.

NIC

Network Interface Controller.

PCI

Peripheral Component Interconnect. A popular, high-bandwidth bus used for connecting peripheral devices to a computer system.

PHY

A physical-layer device that deals with the details of signalling on the media.

TCP/IP

Transmission Control Protocol over Internet Protocol.

WiFi

A type of network that involves the transmission of data packets using radio or infrared signals (see 802.11).

Index

E

F

H

I

L

M

N

O

P

Q

R

S

T

U