Network Drivers#

The networking stack supports three variety of drivers. These are:

- Native io-pkt drivers: Drivers which have been developed from scratch for use with the io-pkt stack
- io-net drivers: Drivers which were written for the legacy networking stack io-net
- NetBSD Ported drivers: Driver source which was taken from the NetBSD source tree and ported to io-pkt

The native and NetBSD drivers all hook directly into the stack in a similiar manner. io-net drivers interface through a "shim" layer which converts the io-net binary interface into the compatible io-pkt interface. We have a special driver (devnp-shim.so) which is automatically loaded up when you want to start an io-net driver.

Differences between ported NetBSD drivers and native io-pkt drivers#

There's a fine line between what a "native" and ported driver is. If you do more than the initial "make it run" port, the feature set of a ported driver and a native driver won't really be any difference. From a source point of view, a ported driver has a very different layout than a "started from scratch" native io-pkt driver. The native driver source (under sys/dev_qnx) looks quite similar in terms of content and files to what an io-net driver looks like and have all of the source for a particular driver under one directory. The NetBSD driver source (under sys/dev) is quiet different in layout with source for a particular driver spread out under a specific driver directory, ic, pci, usb and other directories depending on the driver type and bus that it's on.

- NetBSD ported drivers don't allow the stack to run in multi-threaded mode. NetBSD drivers don't have to worry about rx / tx threads running simultaneously when run inside of the NetBSD operating system so there isn't a need to have close attention paid to appropriate locking issues between rx and tx. For this reason, a configuration flag is, by default, set to indicate that the driver doesn't support multi-threaded access. This will result in the entire stack running in a single threaded mode of operation (if one driver isn't multi-threaded capable, no drivers will run with multiple threads). This flag can be changed once the driver has been carefully examined to ensure that there are no locking issues.

- NetBSD drivers don't include full support for Neutrino specific utilities such as "nicinfo". If any driver does not have support for the nicinfo devctls, it falls through to the generic stack routines, which provide for very rudimentary nicinfo support which does not include media error counts.

- Unless otherwise indicated, we will provide source and allow you to build NetBSD drivers that we've ported, but, unless we have full documentation from the silicon vendors, we won't be able to classify the device as supported. All supported drivers will be included as binaries as part of our GA release. Unsupported drivers will be included in binary format for Foundry27 builds, but will not be included as part of our GA release.

Differences between io-net drivers and other drivers#

- io-net drivers export a name space entry (/dev/enx). Native drivers do not.
- You can unmount an io-net driver (umount /dev/enx). With a native driver, you have to destroy it (ifconfig tsec0 destroy).
- io-net drivers are all prefixed with "en". Native drivers have different prefixes for different hardware (e.g. "tsec" for Freescales TSEC devices).
- io-net drivers support the io-net devctl commands. Native drivers do not.
- io-net drivers will be slower than native drivers since they use the same threading model as that used in io-net.
- io-net driver DLLs are prefixed by "devn-". Core Networking drivers are prefixed by "devnp".

Minor point:io-net drivers used the "speed" and "duplex" command line options to override the auto-negotiated link defaults once. Often the use of these options caused more problems than they fixed. Native (and hopefully most NetBSD ported drivers) allow their speed and duplex to be determined at runtime via a device ioctl, which ifconfig uses to get and set the link speed and duplex. See "ifconfig -m" and "ifconfig mediaopt". This is a big improvement over the io-net drivers.

How do I load / unload a driver?#

Drivers can be loaded into the stack from the command line just as in io-net.

- io-pkt-v4-hc -di82544

This command line invocation will work regardless of whether or not the driver is a native driver or an io-net style driver. The stack automatically detects the driver type and will load the devnp-shim.so binary in the event that the driver is an io-net driver. You need to make sure that all drivers are located in a directory that can be resolved by the LD_LIBRARY_PATH environment variable if you don't want to have to specify the fully qualified name of the device in the command line.

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

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

The io-net option is also supported to provide backwards compatibility with existing scripts.

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

The standard way to remove a driver from the stack is ifconfig <iface> destroy

- ifconfig wm0 destroy

How can I troubleshoot a driver?#

For native drivers and io-net drivers, the nicinfo utility is usually the first debug tool that's used (aside from ifconfig) when problems with networking occur. This will let you know whether or not the driver has properly negotiated at the link layer and whether or not it is sending and receiving packets. Also, ensure that the "slogger" daemon is running, and then after the problem occurs, run the "sloginfo" utility to see if the driver has logged any diagnositic information. You can increase the amount of diagnostic information that a driver will log by specifying the "verbose" command line option to the driver. Many drivers support various levels of verbosity - you might even try specifying "verbose=10".

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

Shared Interrupts - Problems?#

Different devices sharing a hardware interrupt is kind of a neat idea, but unless you really need to do it - because you've run out of hardware interrupt lines - it generally doesn't help you much. In fact, it can cause you trouble. For example, if your driver doesn't work - e.g. no received packets - check and see if it is sharing an interrupt with another device, and if so, re-configure your board so it does not. Most of the times, when shared interrupts are configured, there is no good reason for it - i.e. you haven't really run out of interrupts - and this can decrease your performance, because when the interrupt fires, ALL of the devices sharing the interrupt need to run and check and see if it is for them. If you check the source, you can see that some drivers do the "right thing" which is to read registers in their interrupt handlers to see if the interrupt is really for them, and ignore it if not. But many, many drivers do not - they schedule their thread-level event handlers to check their hardware, which is inefficient and reduces performance. If you are using PCI bus, use the "pci -v" utility to check interrupt allocation. You may be surprised as to what you see. Another point worth making is that sharing interrupts can vastly increase interrupt latency, depending upon exactly what each of the drivers do. Remember that after an interrupt fires, it will NOT be re-enabled by the kernel until ALL driver handlers tell the kernel that they have completed handling. So, if one driver takes a long, long time servicing a shared interrupt which is masked, if another device on the same interrupt causes an interrupt during that time period, processing of that interrupt can be delayed for an unknown duration of time. Bottom line is that interrupt sharing can cause problems, and reduce performance, increase cpu consumption, and seriously increase latency. Unless you really need to do it, don't. If you must share interrupts, make sure your drivers are doing the "right thing". Face it - shared interrupts are the new "trans fats" :-)

How do I port a NetBSD Driver?#

A rough and ready guide to how a driver was ported from NetBSD is available in the source tree under /trunk/lib/io-pkt/sys/dev/doc. The NetBSD drivers make heavy use of libnbdrvr.so, the NetBSD porting library. This library abstracts out the NetBSD OS calls and replaces them with the appropriate Neutrino OS calls. While mostly complete, we're expecting that this library will need updates and tuning as time goes on, especially when we start porting more USB devices over (at this stage, USB support in the porting library is best characterized as "experimental". It works, but we haven't exercised it to a large degree).

How do I write a new Driver?#

A tech note covering how to write a native driver is available in the source tree under /trunk/lib/io-pkt/sys/dev_qnx/doc. Sample driver code is also available under the /trunk/lib/io-pkt/sys/dev_qnx/sample directory.

How do I write a non-PCI Driver?#

On a board with a PCI bus, you can rely on the pci server being there to tell you about the ethernet hardware - how many interfaces there are, and their hardware parameters. But what about on a board with no PCI bus? A good example of this is the devnp-mpc85xx.so ethernet driver. Over time, board-specific information (such asn interrupts, etc) crept in the driver, and that's not good, because every new board required that the driver binary be altered - yuck. So, there was an initiative to beef up the existing HWINFO infrastructure. This required changed to both the startup - to create the hwi entries - and the driver - to use the hwi entries. But at the end of the day, the result is that by using the hwi infrastructure, a new driver binary is not required for every board. Take a look at the devnp-mpx85xx.so source - specifically, the mpc85xx/detect.c source file, and the ppc85xx_get_syspage_params() function, which fishes out all the required hardware parameters. Now, you will notice that devnp-mpc85xx.so still had command line options, which allow a user to override the auto-detected hwi values - eg during development - but ideally, with a properly-written startup, this should not be necessary. For more information on the hwi stuff, click here.

How do I debug a driver using gdb?#

First you have to make sure that you're source is compiled with debugging information included. With you're driver code in the correct place in the sys tree (dev_qnx or dev), you can do the following

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

Now that you have a debug version, you can start gdb and set a breakpoint at main in the io-pkt binary (don't forget to have your driver specified in the arguments and make sure that the PATH and LD_LIBRARY_PATH environment variables are properly set up). After hitting the breakpoint in main(), do a 'sharedlibrary' command in gdb. You should see libc loaded in. Set a breakpoint in dlsym(). When that's hit your driver should be loaded in but io-pkt hasn't done the first callout into it. Do a 'set solib-search-path' and add the path to your driver and then do a 'sharedlibrary' again. The syms for your driver should now load and you can set a break point where you want your debugging to start.

How do I get the stack's 802.11 layer to dump debug information?#

This is kind of cool. The debug information dump is compiled in and enabled / disabled using sysctls. If you do:

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

with a WiFi driver, you'll see net.link.ieee80211.debug and net.link.ieee80211.vap0.debug sysctls.

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

turns on the debug output and now when you type "sloginfo" you'll see the debug log output being generated.

Are Jumbo Packets and Hardware Checksumming Supported?#

Jumbo packets are packets that carry more payload than the normal 1500 bytes. For Jumbo packets to work, you need all of the protocol stack, the drivers, and the network switches (yes, really) to support jumbo packets. Even the definition of a jumbo packet is unclear - different people will use different lengths. Anyways, the io-pkt (hardware-independent) stack DOES support jumbo packets. Not all network hardware supports jumbo packets (generally, newer gige nics do). The native drivers for io-pkt DO support jumbo packes. For example, devnp-i82544.so is a native io-pkt driver for PCI and it supports jumbo packets. So does the devnp-mpc85xx.so for MPC 83xx/85xx. If you can use jumbo packets with io-pkt, you can see substantial performance gains because more data can be moved per packet header processing overhead. How do you configure to operate with jumbo packets? Easy! Do this:

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

Note that for maximum performance, we also turned on hardware packet checksumming (for both transmit and receive) and I have very arbitrarily chosen a jumbo packet mtu of 8100 bytes. A little detail: io-pkt by default allocates 2K byte clusters for packet buffers. This works well for 1500 byte packets, but for example when an 8K jumbo packet is received, we end up with 4 linked clusters. We can improve performance by telling io-pkt (when we start it) that we are going to be using jumbo packets, like this:

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

If we pass the "pagesize" and "mclbytes" cmd line options to the stack, we tell it to allocate contiguous 8K buffers (which may end up being two adjacent 4K pages, which works fine) for each 8k cluster to use for packet buffers. This will reduce packet processing overhead, which will improve throughput and reduce CPU utilization.

What about Transmit Segmentation Offload?#

Transmit Segmentation Offload (TSO) is a capability provided by some modern NIC cards (see, for example, http://en.wikipedia.org/wiki/Large_segment_offload). Essentially, instead of the stack being responsible for breaking a large IP packet into MTU sized packets, the driver can do this instead. This greatly offloads the amount of CPU required to transmit large amounts of data. You can tell if a driver supports TSO by typing "ifonfig" and looking at the capabilities section of the interface output. It will have "tso4" and/or "tso6" marked as one of it's capabilities. How do you configure the driver to use? For example, with devnp-i82544.so, do this:

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

CPU Utilization?#

Above we have talked about reducing CPU utilization. It is possible to drastically reduce CPU utilization during heavy packet throughput, by using a hardware feature to reduce the number of (received packet) interrupts, which allows the stack to amortize the cost of the interrupt servicing over more received packets, and hence increase efficiency. For example, devnp-i82544.so, the "native" driver for io-pkt for the intel pro series of gige nics, utilizes the "Interrupt Throttling Rate" hardware feature of these nics. By default, the driver programs the hardware to allow a maximum of 7,500 interrupts per second. This number was empirically chosen to drastically reduce CPU utilitization of x86 boxes during heavy throughput, without affecting the throughput. This looks like a free lunch, doesn't it? This default configuration works very well at freeing up CPU during heavy throughput, but there is a catch: increased packet latency. There are some customers who are greatly concerned with round trip time, for their applications. They don't care how much CPU they burn, they NEED low packet latency, measured in microseconds. If you are one of those people, you want to specify "irq_threshold=0" to devnp-i82544.so to turn off the default receive interrupt throttling. So in summary, devnp-i82544.so has the default configuration of conserving CPU utilization, without affecting throughput under heavy load. If you don't care about CPU utilization, and wish to spend it on providing absolute minimum latency, specify "irq_threshold=0" to devnp-i82544.so.

Wired Drivers#

Ported NetBSD Drivers#

- *devnp-bge.so*: Broadcom 570x 10/100/1000 chipset (supported)
- *devnp-msk.so*: Marvel Yukon2 chipset 10/100/1000 (x86 only)
- *devnp-fxp.so*: Intel 10/100 chipsets

Native Drivers#

- *devnp-i82544.so*: Intel 10/100/1000 chipsets (supported)
- *devnp-speedo.so*: Intel 10/100 chipsets (supported)
- *devnp-mpc85xx.so*: Freescale 10/100/1000 (enhanced) Triple Speed Ethernet Controller (TSEC) on MPC85xx, MPC83xx chips (supported)
- *devnp-bcm1250.so*: Broadcom 12xx 14xx 10/100/1000 Ethernet controller on the BCM12xx, BCM14xx chips (supported)

Wireless Drivers#

Ported NetBSD Drivers#

The ported drivers that have source publicly available are not currently considered to be supported. While we will make every effort to help you out, we can't guarantee that we will be able to rectify problems that may occur with these drivers. In many cases, multi-CPU support (x86, ppc,sh,arm, mips) either isn't possible or not tested

- *devnp-ral.so*: PCI bus RAL Wi-Fi chipset
- *devnp-ural.so*: USB 2.0 - ~Wi-Fi dongle based on the Ralink RT2500USB chipset
- *devnp-rum.so*: USB 2.0 - ~Wi-Fi dongle based on the Ralink RT2501USB and RT2601USB chipsets
- *devnp-ath.so*: Atheros chipsets (x86 only). This driver is not currently built (you have to remove the Makefile.dnm file in the directory) since it requires a uudecode tool for windows builds (there's a readme in the directory explaining why). Our official driver will not be built from this location given that the open source HAL is improperly built for our implementation on CPU variants other than x86.
- *devnp-wpi.so*: Intel 3945 chipsets. Note that this driver requires a firmware binary which is NOT licensed for use on Neutrino (please see the README file in the driver directory)

Native Drivers#

Due to licensing restrictions, we will not be able to provide either source code or binaries on Foundry27. These will be included as part of the official Beta / GA releases though (the difference being that you have to specifically agree to an EULA as part of the Beta / GA installation process).

The initial release of Core Networking will include official support for the following Wi-Fi chip sets

- Atheros 5006x
- Broadcom 43xx
- Marvel 8686

Drivers FAQ#

What's the difference between delay and DELAY in the NetBSD ported drivers?#

The NetBSD drivers have two different delay functions, both of which take an argument of microseconds. From the NetBSD docs, DELAY() is reentrant (doesn't modify any global kernel or machine state) and is safe to use in interrupt or process context. HOWEVER, Neutrino's version of delay is in milliseconds so this could result in very long timeouts if used directly as is in the drivers. We've defined DELAY to do the appropriate thing for a microsecond delay, so all NetBSD ported drivers should macro delay to DELAY.

How can I reduce jitter in the drivers?#

Drivers are continually allocating and freeing packet buffers called mbufs. io-pkt maintains a per-thread cache of mbufs, the default size of which is 32. So, if you are continually allocating and freeing more than 32 mbufs, you will incur excessive overhead during the mbuf freeing and allocating which can cause packet jitter. There are two ways to deal with this: (1) decrease the size of your transmit descriptor ring, or (2) use the "mbuf_cache=x,pkt_cache=x" options to io-pkt to increase the maximum allowable size of the per-thread mbuf cache, where x is a number larger than the default 32. Note that decreasing the size of the transmit descriptor ring can actually have a beneficial effect in some circumstances (eg fast-forwarding, bridging) because it can reduce page thrashing in the CPU cache.

How do I build drivers under 6.4.0 without downloading all of the io-pkt source base?#

We've changed the driver building under 6.4.1 so that headers that were previously available only as part of the source package are now installed as part of the system headers. These headers will be included in an upcoming

6.4.0.1 maintenance patch. In the interim, the appropriate headers are available in the attachment at the bottom of this page.

Under windows, run bash to get a bash shell first. On linux and self-hosted, make sure that you've got root access

```
cd $(QNX_TARGET)/usr/include
```

```
gzip -d _path_to_/iopkt_build_headers.tar.gz
tar -xvf _path_to/iopkt_build_headers.tar
```

(Under bash with windows, if you want to specify the full path including the disk name, use /c/ to prefix the directory name)

These headers should be installed under your QNX_TARGET/usr/include directory (you'll end up with a QNX_TARGET/usr/include/io-pkt subdirectory containing all of the header files).

A build directory can be created by placing the Makefile and prodroot.mk files from trunk/sys/dev_qnx into a top llevel directory and adding driver sub-directories below it as follows:

```
top_dir/
   Makefile
   prodroot.mk
   driver_1/
       driver_1 source, Makefile, cpu variants, etc.
   driver_2/
       driver_2 source, Makefile, cpu variants, etc.
```