Frequently Asked Questions#

What is io-pkt?#

io-pkt is a new networking infrastructure (protocols, drivers, utilities) to replace the existing io-net networking infrastructure. While io-net was loosely based on the NetBSD stack source, io-pkt much more closely tracks the NetBSD stack source. For example, the lowest level of (native) packet buffer in io-net was the "npkt", while in io-pkt the native packet buffer is an "mbuf". As a result of this, protocols, drivers and utilities should port much easier to QNX io-pkt - there is considerable source leverage.

Also, io-pkt gets rid of a thread context switch during packet receive, which improves receive performance as compared to io-net. Transmit performance is about the same.

Also, io-pkt has much different (and we think, better) support for 802.11 wireless. The hardware-independent portion of the 802.11 protocols is now located in the stack instead of in each driver. More (ported) wireless drivers are supported.

Also, io-pkt has support for runtime-loadable cryptographic libraries, which can offload security ops (eg IPSec) if the hardware supports it.

How do I get io-pkt?#

io-pkt is available in source code from here. io-pkt is actually in two different repositories: there is the public one, which contains 99% of the source, which you can download and compile. There is a second repository (sys/dev_nda) which contains source code for drivers which we legally cannot give out because we have obtained it under non-disclosure agreements. Well, we could give it out, but our lawyers would hunt us down like vermin and string us up by our thumbs. Currently the only stuff in the NDA repository is wireless drivers. For more information, you can also look at the Source Guide

How do I compile io-pkt?#

SeanB has done a marvellous job of this. I actually told him one day that I thought his work with make was more impressive than his work with the procotols :-) All you have to do is type "make install" at the root of the source tree, and all source for all CPU architectures will be compiled, in the correct order, and all the various libraries will be installed. You can either run with a stage (most people at QNX do) but I like to live on the edge and run without a stage, so it is possible. If you run without a stage, don't forget that you'll be overwriting your "official" installation with experimental binaries (!!!) For more information, you can also look at the Source Guide.

How do I install io-pkt on a machine with a hard disk?#

When io-pkt is part of a GA release, I am sure there will be all sort of magical file system soft links to make is work. If you're a do-it-yourselfer, and you've downloaded and compiled the code, the easiest and safest way to run io-pkt (assuming you don't want to get rid of your existing io-net stuff) is to put all the binaries in one directory on your hard disk - say /root/io-pkt

What binaries do you need? Good question. At a minimum, I might suggest copying the following binaries (appropriate for your CPU architecture!) from your io-pkt source tree (where they were compiled) to your directory e.g. /root/io-pkt:

io-pkt-v4-hc - from sys/target/CPU/o.v4
io-pkt-v6-hc - from sys/target/CPU/o.v6
libsocket.so - from lib/socket/CPU/so
devnp-shim.so - from sys/dev_qnx/shim/CPU/dll
devnp-i82544.so - from sys/dev_qnx/i82544/CPU/dll
ifconfig - from utils/i/ifconfig/CPU/o

netstat - from utils/n/netstat/CPU/o
nicinfo - from utils/u/nicinfo/CPU/o
dhcp.client from utils/d/dhcp.client/CPU/o

Obviously, choose whichever network driver you wish to run. Remember that you DON'T have to run an io-pkt driver - you can run an existing binary io-net driver - io-pkt will automatically load the shim for you. Pretty neat, huh?

Now, create an executable text shell script file, which looks like this:

```
export LD_LIBRARY_PATH=/root/io-pkt:$LD_LIBRARY_PATH
export PATH=/root/io-pkt:$PATH
```

When you run the above script, it will cause the shell to look FIRST in /root/io-pkt for the io-pkt executables and libraries.

How do I put io-pkt on a board WITHOUT a hard disk?#

Typically in this situation, you are going to build an image and then somehow transfer that image to your target (eg via tftp by the board monitor).

What I do is create a directory for each different target that I am going to be building an image for. For example, I might have a directory /root/8560 where I build an image for an MPC 8560. In that directory, I would have files that looked like:

```
8560.build
8560.img
devc-serppc8260
startup-85x0ads
```

To build the image I would execute:

```
mkifs -vvv 8560.build 8560.img ; cp 8560.img /xfer
```

And I am running "inetd -d", and I have uncommented the "tftp" line in the "/etc/inetd.conf" file.

To add io-pkt to the above, I would add the following to the 8560.build file:

```
./libsocket.so
./devnp-mpc85xx.so

[data=c]

./io-pkt-v4-hc
./io-pkt-v6-hc
./ifconfig
./netstat
./nicinfo
./ping
./ftp
getconf
setconf
```

When I use dot-slash above, that refers to executables that I would copy from my io-pkt source tree, to my /root/8560 directory. Generic executables, suitable for the CPU architecture, can be added normally (eg getconf utility) and will be pulled from the right place by the mkifs utility.

I might uncomment any automatic invocation of io-net in the build file. To save space, you could get rid of the io-net executables - but remember, if you wish to run an io-net network driver binary (eg devn-mpc85xx.so) be sure to leave it in the build file, and also add ./devnp-shim.so to the build file.

How do I run io-pkt?#

Let's say you've done the above. To run io-pkt, now type the following:

```
# io-pkt-v6-hc -d i82544 -p tcpip
```

io-pkt will load the devnp-i28544.so from the /root/io-pkt directory. If you run

```
# ifconfig
```

you will see a "wm0" interface - NOT an en0 interface. This tells you that you have a native io-pkt driver running. The interface name that you see (wm0) will be unique to a driver (for example, the speedo driver will give you an fxp0 interface). This allows you to see at a glance which card type the interface corresponds to. Note that you can also run

```
# ifconfig -v wm0
```

to see more detailed information about the i82544 ethernet interface, including link speed, duplex and flow control. Using ifconfig, with a native io-pkt driver, you can change the link speed and duplex at runtime - something you couldn't do with an io-net driver.

Note that if you type:

```
# nicinfo
```

But let say you started io-pkt like this instead:

```
# io-pkt-v6-hc -d speedo -p tcpip
```

This will work, too - since devnp-speedo.so has not been copied to /root/io-pkt (you could have) io-pkt automatically locates the io-net binary devn-speedo.so in your LD_LIBRARY_PATH and then quietly loads devnp-shim.so for you, and points the shim at the io-net speedo driver it found, which is awesomely cool to the verge of being incredibly confusing. If you then run:

```
# ifconfig
```

you will see an "en0" interface - that tells you that you have an io-net driver running, which will work fine, but incurs the costs of a thread switch during packet receive.

Can I run two independent stacks at the same time on one machine?#

Definitely. This can be done with either two versions of the same stack (io-pkt and io-pkt, for example), or two different stacks (io-net and io-pkt). With different stack instantiations, completely independent routing tables and other stack information are maintained which allows the stacks to use exactly the same addresses (for example). Note that this does require that you have a separate network interface for each stack that's run.

I have two network cards (easiest is different types - eg one speedo and one i82544) and io-net uses one network card, and io-pkt uses the other.

In order to start the stack while registering a separate entry in the namespace, you need to add the "prefix=..." option to the stack as such:

```
# io-pkt-v6-hc -d i82544 -p tcpip prefix=/alt
```

Notice the /alt I stuck in above. This causes the stack to prefix its

namespace entries with "/alt" (do an

```
# ls /alt
```

to see how this works).

This configuration requires that the SOCK environment variable be set so that libsocket can identify which stack instance to use. This can be done either by exporting the environment variable:

```
# export PATH=/root/io-pkt:$PATH
```

or by setting the environment variable before every command:

```
# SOCK=/alt ifconfig
# SOCK=/alt dhcp.client
```

Using "SOCK=/alt" causes any executable you run in the future to talk to io-pkt instead of io-net.

Why is the io-pkt binary so much larger than io-net?#

There are two things that affect the size of io-pkt as compared to io-net. Firstly, io-net does not include any protocol support by itself. It simply acts as middleware, connecting a to be specified driver with a to be specified protocol. io-pkt has the full IP stack compiled into the single binary and therefore, when comparing binary sizes, you need to look at the combination of both io-net + npm-tcpip.so sizes. Secondly, io-pkt contains numerous enhancements that inevitably increase the size of the binary. It is for this reason that we provide a stack variant (io-pkt-v4) which has a reduced feature set for systems which have tighter memory constraints.

How do I use the Final 6.4.0 Milestone build posted in the Project Downloads section?#

After downloading the file, it must be unzipped and untarred. If you're running self-hosted (i.e. running on a Neutrino box), $ gzip -d patch-632-CN-Milestone-Fn.tar.gz $ tar -xvf patch-632-CN-Milestone-Fn.tar

This puts the Core Networking build under the directory patches/632-CN-Milestone-Fn/target/qnx6 directory.

If you want to run things from there, you can use a script file (named iopkt4.sh) that contains the following to update the PATH and LD_LIBRARY_PATH environment vars

```
 #Execute with ". ./iopktfn.sh"
 #!/bin/sh
 export LD_LIBRARY_PATH=$HOME/patches/632-CN-Milestone-Fn/target/qnx6/x86/lib:$HOME/patches/632-CN-Milestone
$HOME/patches/632-CN-Milestone-Fn/target/qnx6/x86/usr/lib:$LD_LIBRARY_PATH
 export PATH=$HOME/patches/632-CN-Milestone-Fn/target/qnx6stage/x86/bin:$HOME/patches/632-CN-Milestone-Fn/target/
$HOME/patches/632-CN-Milestone-Fn/target/qnx6/x86/usr/bin:$HOME/patches/632-CN-Milestone-Fn/target/qnx6/x86/usr/sbi
```

On a Linux machine, unzip and untar as above. You can then select the appropriate binaries from the patch directory that's been created in your home directory when you build your OS image.

On Windows, you can either use WinZip to extract the files or the same commands as above in to a directory of your choosing.

What about running io-pkt on Neutrino 6.3.2?#

We do not officially recommend or support running io-pkt on a Neutrino 6.3.2 system. While we do make efforts to ensure that the code base will continue to compile and run on 6.3.2, we can not guarantee this for future revisions of the code.