# The io_pkt Stack#

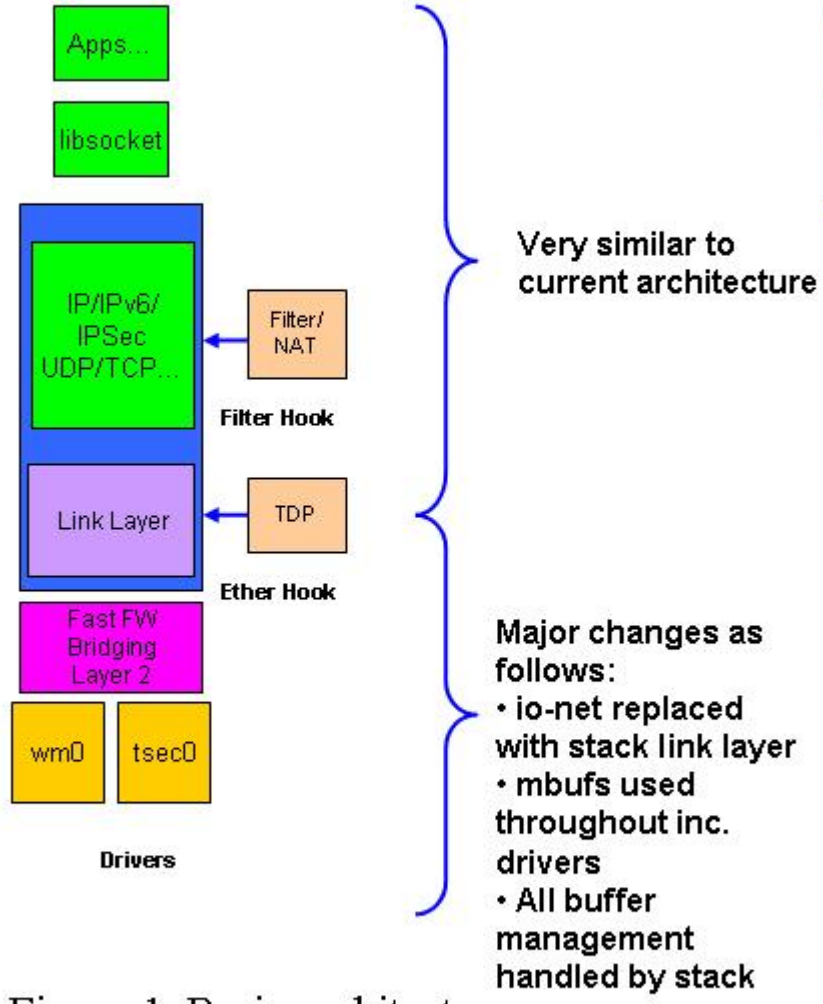



Figure 1: Basic architecture

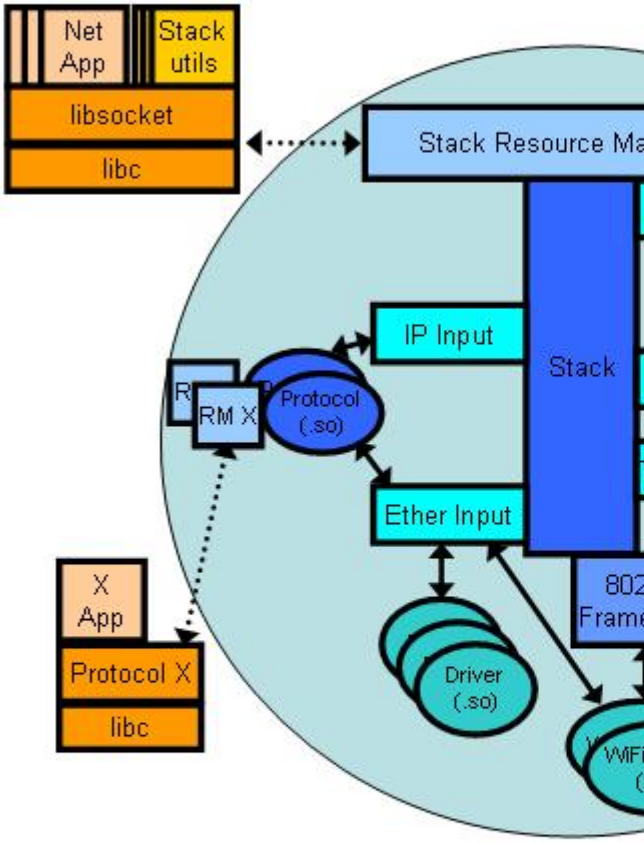



Figure 2: Detail

The io-pkt stack is very similar in architecture to other component subsystems inside of the Neutrino operating system. At the bottom layer (see Figure 1), drivers exist which provide the mechanism for passing data to and receive data from hardware. The drivers hook into a multi-threaded layer 2 component (that also provides fast forwarding and bridging capability) that ties them together and provides a unified interface into the layer 3 component which then handles the individual IP and upper layer protocol processing components (TCP and UDP). In Neutrino, a resource manager is layered on top of the stack. The resource manager acts as the message passing intermediary between the stack and user applications. It provides a standardized open/read/write/ioctl type of interface using a message stream to communicate with networking applications. Networking applications written by the user link with the socket library. The socket library converts the message passing interface exposed by the stack into a standard BSD style socket layer API which is the standard for most networking code today.

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

In addition to the socket level API, there are also other, programmatic interfaces into the stack that are provided for other protocols or filtering to occur. These interfaces (used directly by TDP (aka QNET)) are very different from those provided by io-net so anyone using similar interfaces to these in the past will have to re-write them for io-pkt.

A more detailed view of the io-pkt architecture is shown in Figure 2.

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

APIs providing connection into the data flow at either the Ethernet or IP layer allow protocols to co-exist within the stack process. Protocols (such as QNET) are also built as DLLs. A protocol would link directly into either the IP or Ethernet layer and would run within the stack context. They are prefixed with "lsm" (loadable shared module) and are loaded into the stack using the "-p" option. The "tcpip" protocol (-ptcpip) is a special option that is recognized by the stack but it doesn't result in a protocol module being linked in (since the IP stack is already present). You still use the "-ptcpip" option to pass additional parameters to the stack that apply to the IP protocol layer (e.g. -ptcpip prefix=/alt to get the IP stack to register /alt/dev/socket as it's resource manager name).

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

In addition to drivers and protocols, hooks for packet filtering and egress traffic shaping are also included in the stack. For filtering, there are two main interfaces supported. These are the Berkeley Packet Filter (BPF) interface (which is a socket level interface) and the PF (packet filter) interface (for more information, see the packet filtering page).

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

The other standard interface that's available for users is the "packet filtering" (PF) interface> It provides a read / write / modify / block interface. This gives complete control over which packets are received by or transmitted from the upper layers and is more closely related to the io-net "filter" API.

**Threading Model#**

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

The stack uses a thread pool to service events that are generated from other parts of the system. These events may be time outs, ISR events or other things generated by the stack or protocol modules. All of these threads

are equivalent and operate in a floating priority mode (i.e. the thread priority matches that of the client application thread accessing the stack resource manager). Once a thread receives an event, it examines the event type to see if it's a HW event, stack event or "other" event. Given a HW event, the HW is serviced and, for a receive packet, the thread determines whether bridging or fast-forwarding is required and, if so, performs the appropriate lookup to determine to which interface the packet should be queued and then takes care of transmitting it. If the packet is meant for the local stack, it queues the packet on the stack queue. The thread then goes back and continues checking and servicing hardware events until there are no more events. The thread will then check and see if there is currently a stack thread running to service stack events that may have been generated as a result of it's actions. If there is no stack thread running, it becomes the stack thread and will loop processing stack events until there are none remaining. It then returns back to the "wait for event" state in the thread pool. This capability of having a thread transition directly from a HW servicing thread to the stack thread removes the context switching inherent in the io-net model and greatly improves the receive performance for locally terminated IP flows.

**Threading priorities#**

There are a couple of options available to change the priority of the threads responsible for receiving packets from the hardware. The default thread priority can be configured with the "rx_pulse_prio" option passed to the stack.

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

This will result in all receive threads running at priority 50. The current default for these threads is priority 21.

The second mechanism allows you to change the priority on a **per interface** basis. This is an option passed to the driver and, as such, will only be supported if the driver supports it. When the driver registers for its receive interrupt, it can specify a priority for the pulse that is returned from the ISR. This pulse priority is what the thread will use when running. Sample code from the mpc85xx Ethernet driver is shown.

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

There is one thing to note with the use of driver specific thread priorities. These priorities are assigned on a _per interface_ basis. The stack normally creates one thread _per cpu_ to allow the stack to scale appropriately in terms of performance on an SMP system. Once you use an interface specific parameter with multiple interfaces, you must get the stack to create one thread _per interface_ in order to have that option picked up and used properly by the stack. This is handled with the "-t" option to the stack. So, as an example, to have the stack start up and receive packets on one interface at priority 20 and on a second interface at priority 50 on a uni-processor system, the following command line option would be used.

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

A sloginfo warning entry will be generated if more interfaces than threads are present and a per interface priority is specified. If there are insufficient threads present then the per interface priority is ignored (the rx_pulse_prio default option is still honoured though).

The actual options for setting the priority and selecting an individual card are device driver dependent. Please refer to your driver documentation for specific option information.

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