

Wind River® Network Stack

PROGRAMMER'S GUIDE
Volume 3: Interfaces and Drivers

6.8

Copyright © 2010 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

www.windriver.com/company/terms/trademark.html

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:
`installDir\product_name\3rd_party_licensor_notice.pdf`.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

Toll free (U.S.A.): 800-545-WIND
Telephone: 510-748-4100
Facsimile: 510-749-2010

For additional contact information, see the Wind River Web site:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

Contents

Changes in This Edition	ix
1 Overview	1
1.1 Introduction	1
1.2 About This Manual	1
1.2.1 About the IP Addresses Used in This Manual	3
1.3 Additional Documentation	3
Wind River Documentation	4
Online Resources	4
Books	4
1.4 RFCs	4
2 Configuring and Managing Memory	7
2.1 Introduction	7
2.2 Setting the Memory Limit	8
Finding the Current Limit	8
Evaluating Memory Needs	9
2.3 netBufLib Buffer Pools	10
2.3.1 Tuples	11
2.3.2 Creating netBufLib Pools	12
netPoolCreate()	13
netPoolInit()	16
Memory Requirements Routines	18
2.4 Legacy Network Stack Pools	20
2.4.1 Legacy Tuple Pool for IPNET-Native Devices	21

3	Working with Network Drivers and Devices	23
3.1	Introduction	23
3.2	Overview of the MUX	23
	OSI Network Model	24
	MUX Layer	24
3.3	Working with Network Driver Instances	25
3.3.1	Attaching a Service to a Network Interface	28
3.3.2	Configuring a Network Interface with an Address	29
	Retrieving Interface Information	29
	Configuring an Interface	29
	Creating a Pseudo-Interface	30
	Creating Unnumbered Interfaces	31
3.3.3	Fixing Interfaces That Have Erroneous Addresses	31
3.3.4	Assigning a Host Name to an Address	31
3.3.5	Bringing the Device Up for Protocol Communication	32
3.3.6	Determining the Device Link Status	32
4	Working with Tunneling and Shared Memory	33
4.1	Introduction	33
4.2	Working with IPv4 and IPv6 Tunneling	33
4.2.1	Configuring VxWorks for Tunneling	34
	GIF Tunnel Interface Driver	34
	GRE Tunnel Interface Driver	35
	6over4 Tunnel Interface Driver	35
	6to4 Tunnel Interface Driver	35
	SIT Tunnel Interface Driver	36
4.2.2	Creating 6to4 Tunnels for IPv6 Packets	36
4.2.3	Creating RFC 2893-Style Configured Tunnels	37
4.2.4	Tunnel Example	39
4.3	Using the Shared-Memory Network	43
4.3.1	Backplane Shared-Memory Region	43
	Backplane Processor Numbers	43
	Shared-Memory Network Master	44
	Shared-Memory Anchor	45
	Shared-Memory Heartbeat	45
	Shared-Memory Location	46
	Shared-Memory Size	46
	Test-and-Set to Shared Memory	46
4.3.2	Interprocessor Interrupts	47
4.3.3	Sequential Addressing	48

4.4	Shared-Memory Network Configuration	50
4.4.1	Configuration Example	51
4.4.2	Troubleshooting	54
5	Integrating a New Network Device Driver	57
5.1	Introduction	57
5.1.1	MUX Network Driver and Network Service Styles	58
5.1.2	Components of the MUX	60
5.1.3	Wrapper Conversion Components	61
5.2	MUX Programming Interface with Network Drivers	63
5.3	Loading and Unloading Device Instances	64
5.3.1	VxBus vs. Non-VxBus Drivers	65
5.3.2	Unloading a Device from the MUX	66
5.4	Driver Implementations of the NET_FUNCS Interface	68
5.4.1	xStart()	70
5.4.2	xStop()	71
5.4.3	xIoctl()	72
	Arguments to xIoctl()	73
	Ioctl Commands	77
5.4.4	xSend()	79
5.4.5	xMCastAddrAdd()	81
5.4.6	xMCastAddrDel()	82
5.4.7	xMCastAddrGet()	83
5.4.8	xPollSend()	83
5.4.9	xPollRcv()	85
5.4.10	xFormAddress()	86
5.4.11	xPacketDataGet()	87
5.4.12	xAddrGet()	88
5.4.13	xEndBind()	89
5.5	Driver Implementation	90
5.5.1	IPNET-Native Style Driver Implementation	90
5.5.2	M_BLK-Oriented Devices	90
5.5.3	M_BLK-Oriented Driver Implementation	91
5.5.4	MUX Receive Routine	91
5.6	How VxWorks Launches and Uses Your Driver	92
5.6.1	Service-to-MUX Interface	92

5.6.2	Data-Link-to-MUX Interface	94
5.6.3	Polled Mode—for Debugging Only	96
5.7	Driver Interface with the MUX	96
	Data Structures Shared by the Driver and the MUX	96
5.8	MUX Routines for Network Drivers	97
5.8.1	Receive Routine	98
5.8.2	Transmit Restart Routine	98
5.8.3	Notifying the MUX of Device Events	99
5.9	Queueing Work to the Network Job Queues	100
5.10	Collecting and Reporting Packet Statistics	102
5.10.1	Calling the Driver Routines	103
6	Integrating a New Network Service	107
6.1	Introduction	107
6.2	Implementing the MUX/Network Service Interface	107
6.2.1	Initializing the Interface	108
6.2.2	Binding to a Network Device	109
6.2.3	Using MUX/Service Interface Routines	111
	Sending Packets	111
	Constructing Link Level Headers	113
	Shutting Down an Interface	113
6.2.4	Service Routines Registered when Binding to a Device	114
	xStackShutdownRtn()	114
	xStackRcvRtn()	115
	xStackErrorRtn()	117
	xStackRestartRtn()	118
6.3	Adding a Socket Interface to Your Service	118
6.3.1	Process Overview	119
6.3.2	Registering a Socket Back End	120
6.3.3	Socket Functional Interface	121
	xSocketRtn()	122
	xAcceptRtn()	123
	xIoctlRtn()	124
6.3.4	Memory Validation and Socket Ioctls	124
7	Working with the 802.1Q VLAN Tag	127
7.1	Introduction	127

7.2	Adding VLAN Support	128
7.3	About the 802.1Q VLAN Tag Header	128
7.4	MUX Extensions for Layer 2 VLAN Support	129
7.4.1	Enabling VLAN Support for a Port	130
7.4.2	Disabling VLAN Support for a Port	131
7.4.3	MUX-L2 Ingress Rules	131
7.4.4	MUX-L2 Egress Rules	132
7.4.5	Accessing the MUX L2 Control Routines	133
7.5	Current MUX-L2 Limitations	133
7.6	VLAN Management	134
7.6.1	MUX-L2 VLAN Management	134
7.6.2	Subnet-Based VLAN Management	135
	Consequences of Changing the VID	136
	Example of Subnet-Based VLAN Management	136
7.6.3	Socket-Based VLAN Management	138
7.7	Using the MUX-L2 Show Routines	140
8	Quality of Service	143
8.1	Introduction	143
8.2	Differentiated Services	144
8.2.1	Including DiffServ in a Build	144
8.2.2	Using DiffServ	144
	Adding a Filter Rule for a Meter/Marker Entity	145
	Deleting a Filter Rule from a Meter/Marker Entity	146
	Creating a Meter/Marker Entity	146
	Deleting a Meter/Marker Entity	146
	Mapping a Filter to a Meter/Marker Entity	146
	Removing a Filter-to-Meter/Marker Entity Mapping	147
8.2.3	Classes	147
8.2.4	Creating New Meter/Marker Entity Varieties	149
8.2.5	Using Existing Meter/Marker Entity Varieties	150
	SimpleMarker	151
	Single-Rate Three-Color Marker	151
8.3	Network Interface Output Queues	152
8.3.1	Operations	154
	Adding an Interface Output Queue	154
	Getting an Object that Describes an Interface Output Queue	154
	Adding a Filter Rule to a Container Queue	155

	Deleting a Filter Rule from a Container Queue	156
8.3.2	Leaf Queues	156
8.3.3	Container Queues	159
	Available Container Queues	159
8.3.4	Adding a New Queue Type	161
8.3.5	Example—Reserving Bandwidth for an Application	163
	Specifying the burst Parameter	164
9	Ingress Traffic Prioritization	167
9.1	Introduction	167
9.2	Factors to Consider Before Using Ingress Filtering	168
	Systems with Multiple Interfaces for Incoming Traffic	168
	Traffic Congestion and Fairness	168
	Driver Variety	169
9.3	Building VxWorks to Include Ingress Traffic Prioritization	169
9.4	Implementing an Ingress Filter Routine	170
9.4.1	Registering an Ingress Filter Routine	171
A	Networking Shell Commands	173
A.1	Introduction	173
A.2	Networking Shell Commands	173
	ifconfig	173
	qc	178
	qos	180
	route	182
	slab	185
	sysctl	187
	sysvar	187
Index	189

Changes in This Edition

This edition of *Wind River Network Stack Programmer's Guide, 6.8, volume 3*, contains documentation corrections only. There are no feature updates for VxWorks 6.8 and VxWorks Platforms 3.8, Update Pack 2.

1

Overview

1.1 Introduction	1
1.2 About This Manual	1
1.3 Additional Documentation	3
1.4 RFCs	4

1.1 Introduction

The Wind River Network Stack is a dual IPv4/IPv6 TCP/IP stack that is designed for use in modern, embedded real-time systems. It includes many services and protocols that you can use to build networking applications.

This is the third volume of the *Wind River Network Stack Programmer's Guide*. For information on the following topics, see the *Overview* chapter of the *Wind River Network Stack Programmer's Guide, Volume 1*:

- an overview of the Wind River Network Stack
- a list of features unique to Wind River platforms
- a guide to relevant additional documentation
- where to get the latest release information

1.2 About This Manual

The following is an overview of the information you will find in this manual. See [1.3 Additional Documentation](#), p.3 to learn about the other two volumes that describe the Wind River Network Stack, and additional documentation that you may find helpful.

1. Overview

This chapter.

2. Configuring and Managing Memory

This chapter describes the following:

- Configuring the memory limit at build time ([2.2 Setting the Memory Limit](#), p.8).
- Creating and using **netBufLib** pools ([2.3 netBufLib Buffer Pools](#), p.10).
- The legacy network stack data pool and network stack system pool, used by previous versions of the network stack and sometimes still required by particular applications ([2.4 Legacy Network Stack Pools](#), p.20).

3. Working with Network Drivers and Devices

In addition to drivers supplied for physical network interfaces, the Wind River Network Stack also includes drivers for the creation of GIF, GRE, SIT, 6to4, and 6over4 devices—over IPv4, IPv6, or both. This chapter provides instructions and some background information on how to create and configure device instances associated with the network stack.

4. Working with Tunneling and Shared Memory

This chapter describes the following tunneling over IPv4 or IPv6, and using a shared-memory network driver to allow multiple processors to communicate over their common backplane.



NOTE: The tunneling feature is available only in the Wind River Platforms builds of the network stack. The Wind River General Purpose Platform, VxWorks Edition, does not support tunneling.

5. Integrating a New Network Device Driver

This chapter describes how to integrate a new network interface driver with Wind River Network Stack. For this, use the MUX, which is an interface that insulates network services from the particulars of network interface drivers, and vice versa.

6. Integrating a New Network Service

A network service is an implementation of the network and transport layers of the OSI network model. Under the Wind River Network Stack, network services communicate with the data link layer through the MUX interface. This chapter describes how to integrate a new network service with the MUX and, thus, with the network stack.

7. Working with the 802.1Q VLAN Tag

This chapter describes the implementation of 802.1Q VLAN tagging for VxWorks and tells you how to configure VxWorks to include this feature.



NOTE: The 802.1Q VLAN tagging feature is available only in the Wind River Platforms builds of the network stack. The Wind River General Purpose Platform, VxWorks Edition, does not support 802.1Q VLAN tagging.

8. Quality of Service and 9. Ingress Traffic Prioritization

These chapters describe the network stack's Quality of Service (QoS) capability, in which the stack treats some network traffic to better service than others. The Wind River Network Stack implements the Differentiated Services (DiffServ) model of QoS, which classifies traffic entering a network and conditionalizes it before treating it in an appropriate manner. Similarly, the ingress traffic prioritization feature allows you to assign priorities to the packets arriving at an interface and have the stack process higher-priority packets before lower-priority packets.



NOTE: The QoS feature is available only in the Wind River Platforms builds of the network stack. The Wind River General Purpose Platform, VxWorks Edition, does not support QoS.

The Wind River Network Stack does not support ingress filtering in symmetric multiprocessing (SMP) builds.

A. Networking Shell Commands

This appendix contains reference entries for the common networking shell commands.

1.2.1 About the IP Addresses Used in This Manual

When working with the examples in this manual, you may find it convenient to cut and paste example text into source code or to a command line. To avoid disrupting the use of IPv4 or IPv6 addresses that are, or might be, put into service, the examples in this manual restrict themselves to the following address spaces:

- 10/24 – one part of the private address space
- 127.0/8 – loopback addresses
- 169.254/16 – link local addresses
- 172.16/12 – another part of the private address space
- 192.0.2/24 – test and documentation addresses
- 192.168/16 – another part of the private address space
- 2001:DB8::/32 – test and documentation addresses (RFC 2849)
- FE80::/10 – link local addresses

1.3 Additional Documentation

The following sections describe additional documentation about the technologies described in this book.

Wind River Documentation

The Wind River Network Stack is described in the three volumes of the *Wind River Network Stack Programmer's Guide*:

- Volume 1 has an overview with general information about the network stack, and describes the network and transport layers.
- Volume 2 describes application-layer protocols and socket programming.
- Volume 3 (this volume) describes network services, drivers, and the MUX, which is an abstraction layer between drivers and services.

The user's guide for your Platform includes instructions on how to build a component or product into VxWorks, either through the Workbench **Kernel Configuration Editor** or the **vxprj** utility.

For information on using Workbench to create a VxWorks image project and to include build components, see *Wind River Workbench by Example*. For information on using the **vxprj** command-line utility, see the *Wind River VxWorks Command-Line Tools User's Guide*.

The *Wind River VxWorks Platforms Migration Guide* details how to migrate from an earlier release of the network stack.

For information on host-side network diagnostic tools, see the *Wind River Workbench User's Guide*.

Online Resources

Online resources are as follows:

- The Internet Engineering Task Force, <http://www.ietf.org>

Books

Additional documentation is as follows:

- *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*, Douglas E. Comer.
- *UNIX Network Programming, Volume 2, Second Edition* by W. Richard Stevens

1.4 RFCs

The Wind River Network Stack interfaces and drivers comply with the IETF RFCs listed in this section. The RFCs are accessible from the following Web site:

<http://www.ietf.org>

- RFC 1700: *Assigned Numbers*
- RFC 1853: *IP in IP Tunneling*

- RFC 2002: *IP Mobility Support*
- RFC 2473: *Generic Packet Tunneling in IPv6 Specification*
- RFC 2529: *Transmission of IPv6 over IPv4 Domains without Explicit Tunnels*
- RFC 2674: *Definitions of Managed Objects for Bridges with Traffic Classes, Multicast Filtering and Virtual LAN Extensions*
 - Although the MUX-L2 implements selected RFC 2674 static VLAN objects, the VLAN configuration methodology is not compatible with the RFC 2674 MIB. RFC 2674 VLAN management is VLAN-centric and requires a port list bitmap specifying the ports belonging to a VLAN. The VLAN management for the MUX-L2 is port-centric and achieves VLAN configuration on a per-port basis.
- RFC 2697: *A Single Rate Three Color Marker*
- RFC 2784: *Generic Routing Encapsulation (GRE)*
- RFC 2893: *Transition Mechanisms for IPv6 Hosts and Routers*
- RFC 3056: *Connection of IPv6 Domains via IPv4 Clouds*

This section lists RFCs relevant to the Wind River Network Stack interfaces and drivers:

- RFC 894: *A Standard for the Transmission of IP Datagrams over Ethernet Networks*
- RFC 1213: *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*
- RFC 2233: *The Interfaces Group MIB using SMIv2*
- RFC 2849: *The LDAP Data Interchange Format (LDIF) - Technical Specification*

Configuring and Managing Memory

2.1	Introduction	7
2.2	Setting the Memory Limit	8
2.3	netBufLib Buffer Pools	10
2.4	Legacy Network Stack Pools	20

2.1 Introduction

The Wind River Network Stack and the network drivers that work with it use several types of memory pools for memory allocation. Memory allocated by the IP network stack comes ultimately from the system heap. The stack creates multiple “slab memory caches” based upon the system heap; each cache provides the stack with buffers of a fixed size, sometimes partially initialized as objects of a particular kind. These memory caches can grow dynamically, as needed, with a limit on total allocation, by the stack, across all the memory caches. If the maximum is reached, a limited form of “garbage collection” may occur, which returns any completely free memory slabs held in a memory cache to the system pool. The stack allocates both packet buffers and various fixed-size control structures from these memory caches. The stack allocates several other control structures directly from the system heap; these direct allocations do not have an explicit bound.

To receive network packets, IPNET-native network drivers allocate packet buffers and packet control structures from the memory caches that make up the stack’s packet pool. This use is also bounded by the overall limit on allocation into the stack’s memory caches.

Traditional **M_BLK**-oriented VxWorks network drivers use **netBufLib** pools to allocate buffers into which they receive packets. Such drivers create these pools when the MUX loads a network interface (usually at initialization time). Other protocols, such as TIPC, also use **netBufLib** pools. The memory used in these pools comes ultimately from the system heap. However, these pools do not generally grow after creation.

This chapter describes the following:

- What to consider when configuring the memory limit [2.2 Setting the Memory Limit](#), p.8.
- How to create and use **netBufLib** pools ([2.3 netBufLib Buffer Pools](#), p.10).
- How to configure the legacy network stack data pool and network stack system pool, used by previous versions of the network stack and still required by some applications ([2.4 Legacy Network Stack Pools](#), p.20).

2.2 Setting the Memory Limit

When configuring memory, it is not required (nor possible) to configure the number or size of packets. The network stack is given a maximum amount of memory, which it uses to create the packet pool layout, and the layout of the packet pool changes dynamically to adapt to the current workload.

For example, if an application suddenly starts sending or receiving very large UDP datagrams, the stack adjusts the packet pool to contain more large (that is, 10KB or 64KB) packets and decreases the number of 1500-byte packets, if it needs to stay within the total memory limit.

The maximum memory usage is configurable at build time by setting **IPNET_MEMORY_LIMIT**, which is defined in `installDir/ip_net2-6.x/ipnet2/config/ipnet_config.h`, as shown below:

```
/*
 *=====
 *                               IPNET_MEMORY_LIMIT
 *=====
 * Defines the maximum amount of memory that the stack is allowed to use.
 * The limit must be large enough to allow enough packet buffers, packet
 * headers, sockets, timers to support the workload the system has.
 * Most memory will be used to packet buffers. Each socket can queue up to
 * (send buffer size + receive buffer size) buffer octets. Each device
 * will need up to TX + RX descriptors of MTU sized packets.
 */
#define IPNET_MEMORY_LIMIT (1024 * 1024 * 20)
```

This parameter specifies the number of bytes the stack is allowed to allocate. The default value is 16 MB.

Finding the Current Limit

To find the current value of **IPNET_MEMORY_LIMIT**, you can run either of the following commands:

```
-> ipnet_conf_memory_limit
ipnet_conf_memory_limit = 0x4385bc: value = 16777216 = 0x1000000
```

You can also use the **slab** command—see [slab](#), p.185.

Evaluating Memory Needs

When setting the memory limit value, keep in mind that the memory must be enough for all sockets, packet headers, packet buffers, timers, events, TCP segment control blocks (one per non-packed TCP segment), IPv4/IPv6 address blocks, and poll signals—used to implement `select()` and blocking calls to `sendmsg()` and `recvmsg()`.

As a reference, [Table 2-1](#) lists the size of each, which are designated for 32-bit CPUs and can vary between releases. The actual sizes can be obtained by using the `slab` shell command.

Table 2-1

Memory Sizes

Item	Size (bytes)
Each socket	544
Each packet header	672
Each packet buffer	takes 1682, 3182, 10182, 65718 (or 1500, 3000, 10 KB and 64 KB buffers) ^a
Each timer	20
Each event	24
Each TCP segment control block	32
Each IPv4 address block	112
Each IPv6 address block	140
Each poll signal	72

a. A packet that can hold 1500 bytes of L3 header + L4 header + L4 payload will occupy 1682 bytes of memory. The packet has to be larger than the maximum number of L3+L4 data since there has to be room for L2 headers and also room for putting the data at any specific alignment that the stack might want.

The sizes of the packets were chosen for the following reasons:

- 1500 bytes is, by far, the most commonly used size. This matches the MTU of Ethernet and loopback devices.
- 3000 bytes was chosen because a PPP frame can, in the worst case, can grow to twice its original size during TX framing. Thus, 3000 represents 2 times 1500.
- 10000 bytes is big enough to hold most fragmented UDP datagrams. There are a number of protocols that generate UDP datagram up to 8192 bytes.
- 65535 bytes is the largest packet supported by IPv4 and IPv6, without IPv6 jumbogram extension header.

The total amount of memory assigned to the stack must never be so small that it cannot fill:

- the send and receive buffer of at least one socket
- all RX and TX descriptors of at least one network interface

For example, a socket with 64KB send and receive buffers using the **gei** driver would need about 800KB for just the buffer and header. This does not include any memory for socket, timers, and so on.

Thus, the minimum amount of memory would be 1MB for the system to function properly when using one socket under high load with no fragmentation, or using a few (10 or so) sockets under light load.

The default value of 16MB should work for most systems that are using the default VxWorks limit of 50 open file descriptors in the system, which also limits the number of sockets to 50. In most cases, even 10MB should be sufficient.

For more information on calculating the memory requirement for your application, see the *Wind River VxWorks Platforms User's Guide*.

2.3 netBufLib Buffer Pools

The upper layers of the VxWorks network stack do not use **netBufLib** buffer pools, nor do IPNET-native network drivers; but traditional **M_BLK**-oriented network device drivers do, as does the TIPC protocol. If you modify or exchange data directly with **M_BLK**-oriented drivers, you may need to be familiar with the **netBufLib** library and its interfaces. You may also find this library useful if your application needs a flexible, standalone buffer management implementation.

The **netBufLib** library facilitates creation and management of pools of buffers (called *clusters*—see also [2.3.1 Tuples](#), p. 11), along with the control structures—**M_BLKs** and **CL_BLKs**—that link clusters into chains and (in some cases) share clusters between different code paths. The **netBufLib** library presents a high-level interface that depends upon particular back-end implementations, which allocate and free pool resources. There are three different **netBufLib** back ends currently implemented in the Wind River Network Stack:

netBufPool

The default, and most full-featured pool implementation. It supports multiple cluster pools of different sizes, and allows you to allocate either separate **M_BLKs**, **CL_BLKs**, and clusters, or to allocate all three together, in coordinated tuples. To use this pool back end, include the **INCLUDE_NETBUFPOOL** component in your build.

linkBufPool

A pool implementation specialized to provide optimized allocation of tuples of a single cluster size. Wind River recommends that non-IPNET-native network drivers use this back end for the device receive pools that they create. This back end fuses together the **M_BLK** and **CL_BLK** control structures into a single contiguous **M_LINK** structure. You cannot allocate unattached clusters, **M_BLKs**, or **CL_BLKs** when you use **linkBufPool**; you may, however, create a **linkBufPool** without attached clusters and allocate **M_LINK** structures from it

that are not attached to clusters. To use this pool back end, include the `INCLUDE_LINKBUFPOOL` component in your build.

vxmux_null_buf

This pool back end is not for application use. It is only for internal use by the stack.

It is a single-purpose back end implementation, which the stack uses when it passes packets to device drivers for them to transmit. Since the network device drivers expect packets to be described by `M_BLK`/`CL_BLK`/cluster tuples (see [2.3.1 Tuples](#), p. 11, for more on tuples), but the stack does not use this format, the stack must “repackage” packets that it passes to the driver transmit routine, so that they appear as tuples. The stack does this efficiently by using the `nullBufPool` back end. IPNET-native style network device drivers use the stack-native packet format to avoid even the minimal overhead of the `nullBufPool` wrapping.

To enable the `netBufLib` library, include the `INCLUDE_NETBUFLIB` component in your image. You can call display routines for `netBufLib` pools, if you include the `INCLUDE_NETPOOLSHOW` component in your image. Some less frequently used routines in the `netBufLib` API are in a separate library, `netBufAdvLib`, to which you can gain access if you include the `INCLUDE_NETBUFADVLIB` component. The capabilities of this library are described briefly in the section on creating `netBufLib` pools, see [2.3.2 Creating netBufLib Pools](#), p. 12 and the reference entry for `netBufAdvLib` for more information.

2.3.1 Tuples

The `netBufLib` API describes a packet by a *tuple* or by a chain of tuples. The tuple is a construct that consists of an `M_BLK` structure, a `CL_BLK` structure, and a cluster buffer.

- The `M_BLK` is similar in nature to the `mbuf` used in the BSD network stack. Among other members, the `M_BLK` has a `pCIBlk` field, which is a pointer to the `CL_BLK`.
- The `CL_BLK` in turn holds a pointer to the cluster buffer. The cluster buffer is the DMA buffer. The `M_BLK` also has a pointer into the cluster buffer but this pointer can be modified by software to add or subtract offsets. The cluster buffer pointer in the `CL_BLK` always points to the base of the cluster buffer. The `CL_BLK` also maintains a reference count for the cluster and, optionally, a function pointer and function arguments to be used when freeing an external cluster (one not allocated from a `netBufLib`-style pool).
- The access path to the start address of a cluster buffer in a tuple is `pMblk->pCIBlk->clNode.pCIBuf`.



CAUTION: The IP stack, as well as other protocols that attach to a network device through the MUX, may pass a segmented packet described by a chain of more than one `M_BLK` tuple to a network driver for transmission. Thus, a network driver transmit routine must be able to handle such multi-segment packets. However, the IP stack will not handle a received multi-segment packet that a network device passes it via the MUX; the IP stack expects that all received packets that are delivered to it be described by a single tuple with a single contiguous data block in the tuple's cluster. Network drivers must abide by this restriction.



CAUTION: Certain fields within an **M_BLK** that previous versions of the Wind River Network Stack used might not be used by the current stack version, or might possibly be used for different purposes. Such fields include the **rcvif**, **header**, **aux**, and **altq_hdr** members of the **M_PKT_HDR** substructure in each **M_BLK**. Applications should not assume, however, that these members are available for their own use.

Allocating a Tuple

Use the **netTupleGet()** routine to allocate a tuple.



NOTE: If you are using the **netBufPool** back end, you can allocate a bare cluster using **netClusterGet()**, a bare **CL_BLK** using **netCIBlkGet()**, and a bare **M_BLK** using **netMblkGet()**, then join the cluster and cluster block with **netCIBlkJoin()** and join the **M_BLK** to the cluster block/cluster pair combination using **netMblkCIBlkJoin()**. But it is simpler and more efficient for you to call **netTupleGet()** for this purpose.

Freeing a Tuple Chain

To free a tuple chain linked through the **mBlkHdr.mNext** field, call **netMblkCIBlkChainFree()**. To free only the first tuple of such a chain and return a pointer to the next, call **netMblkCIBlkFree()**.

Copying a Tuple Chain

To construct a copy of a tuple chain (or part of a chain) which shares references to the clusters in the original chain, and hence does not copy bulk data, call **netMblkChainDup()**. To copy a tuple chain's data into a (sufficiently large) buffer, call **netMblkToBufCopy()**.

There are various other routines available for manipulating, allocating, and freeing tuples or bare clusters, and control structures; see the **netBufLib** reference manual entry, or the source code at *installDir/components/ip_net2-6.x/vxmx/src/mem/netBufLib.c*.

2.3.2 Creating netBufLib Pools

To create **netBufLib** pools, call either the older **netPoolInit()** routine or the newer **netPoolCreate()** routine. Wind River recommends that you call the **netPoolCreate()** routine, since it frees you from having to allocate memory for the clusters, **CL_BLKs**, and **M_BLKs** making up a **netBufLib** pool.

Pools that you create by calling **netPoolCreate()** have the following additional capabilities that are not available in pools that you create by calling **netPoolInit()**:

- By calling **netPoolRelease()** you can safely free those pools that you created with **netPoolCreate()**. This routine puts a pool into a release state; when all the holders of buffers belonging to the pool have returned them to the pool, the pool is automatically freed. A driver can use this routine to free a network device's receive pool when the device is unloaded.
- By calling the **netPoolIdGet()** routine you can look up by name a pool that you created with **netPoolCreate()**. You can obtain the name of such a pool by calling **netPoolNameGet()**.

- Several agents (network interfaces, protocols, and so forth) may share a pool that you create with **netPoolCreate()**. An agent that wants to use such a pool may call **netPoolAttach()** to look up a pool by name and attach to it; this increments a count that prevents the pool from being released until all agents that have attached to it detach from it by calling **netPoolDetach()**.
- You can associate a set of attributes with pools that you create with **netPoolCreate()**—including shareability, buffer alignment, and the memory partitions out of which the buffers and control structures that **netPoolCreate()** allocates at pool creation time (see *pNetBufCfg Parameter*, p.14).
- You can bind a pool that you create with **netPoolCreate()** to another pool, called its parent pool, by calling the **netPoolBind()** routine. When an agent attempts to allocate a packet from a pool, but that pool does not have sufficient resources, the attempt will repeat in the pool's parent pool. When the agent later frees the packet, the packet is returned to whichever pool it was originally allocated from. A parent pool may be the parent of several child pools, and provides a shared back-up supply for the child pools, which are usually private to one agent. You cannot successfully release a parent pool while there are still children bound to it; you must first unbind its child pools by calling **netPoolUnbind()**. You must configure a parent pool to have the same pool **attributes** as any child pools that you attach to it, and each of these pools must be sharable (see *attributes*, p.14).

To enable the pool attachment, pool binding, and pool attributes capabilities, include the component **INCLUDE_NETBUFADVLIB** in your image. The **netPoolRelease()** capability, and pool look-up by name, are available for pools that you create with **netPoolCreate()** even if you do not include the **INCLUDE_NETBUFADVLIB** component.

Pools that you create with **netPoolInit()** lack the above capabilities. However, **netPoolInit()** allows (and requires) that you create a pool using pre-allocated memory for the clusters and control structures. If your code needs to create a pool in this manner, it should call the **netPoolInit()** routine rather than **netPoolCreate()**.

netPoolCreate()

To create a memory pool, call **netPoolCreate()**:

```
NET_POOL_ID netPoolCreate
(
    NETBUF_CFG * pNetBufCfg, /* Configuration Structure */
    POOL_FUNC * pFuncTbl     /* Optional plug in function table */
)
```

This routine takes two parameters, *pFuncTbl* and *pNetBufCfg*.

pFuncTbl Parameter

The *pFuncTbl* parameter is a pointer to a table of function pointers that specifies which **netBufLib** back end implementation governs the new pool (see *2.3 netBufLib Buffer Pools*, p.10). Set this parameter to one of the following values:

_pNetPoolFuncTbl

to use the **netBufPool** back end with a backward-compatible memory requirements routine that guarantees only four-byte alignment for both clusters and control structures¹

NULL

to use the **netBufPool** back end with a memory requirements routine (**_netMemReqDefault()** in **netBufLib.c**) that yields more stringent alignment, which may yield marginally better performance than the backwards-compatible memory requirements routine chosen when **_pNetPoolFuncTbl** is explicitly passed

_pNetPoolFuncTbl

to use the **linkBufPool** back end

pNetBufCfg Parameter

The *pNetBufCfg* parameter is a **NETBUF_CFG** structure that you have filled in to indicate what sort of pool you want to create. **NETBUF_CFG** is defined in **netBufLib.h** as:

```
typedef struct netBufCfg
{
    char *          pName;           /* Pool Name */
    UINT32          attributes;      /* pool attributes */
    void *          pDomain;         /* RTP ID or NULL for kernel */
    int             ctrlNumber;      /* # of ctrl structures to pre-allocate */
    PART_ID         ctrlPartId;      /* Mem Partition for Control structures */
                                /* NULL = use Kernel partition */
    int             bMemExtraSize;   /* Additional memory for runtime buffers */
    PART_ID         bMemPartId;      /* Mem Partition for buffers */
                                /* NULL = default for kernel or RTP */
    NETBUF_CL_DESC * pClDescTbl;     /* desired cluster sizes and count */
    int             clDescTblNumEnt; /* num of entries in cluster table */
} NETBUF_CFG;
```

The members of this structure are as follows:

pName

A string of length less than **NET_POOL_NAME_SZ** (this is 16 bytes for most architectures). **netPoolCreate()** copies this name into the **NET_POOL** structure that it returns.

pDomain and **bMemExtraSize**

These members are ignored at present. Set them to **NULL** and 0 (zero) respectively.

ctrlPartId and **bMemPartId**

Set these to the memory partitions from which the pool is to allocate memory for control structures (**M_BLKs** and **CL_BLKs**) and for cluster buffers, respectively. Set these to **NULL** if you want to allocate this memory from the kernel system heap.

ctrlNumber

Set this to the number of **M_BLKs** the pool allocates; the pool will allocate the same number of **CL_BLKs** as well.

attributes

The pool's nominal cluster alignment and whether the pool can be shared (see [2.3.2 Creating netBufLib Pools](#), p.12 for a discussion of pool sharing). Set this to one of the following values:

- **ATTR_AI_SH_ISR** – integer-aligned; shareable
- **ATTR_AC_SH_ISR** – cache-line-aligned; shareable

1. As opposed to, for example, cache-line alignment.

Example 2-1 **Establishing a Network Driver Pool with `netPoolCreate()` and `_pLinkPoolFuncTbl`**

A network driver could create a network tuple pool using the `linkBufPool` backend by calling a routine like the following:

```
NET_POOL_ID myPoolCreate
(
    int    tupleCnt, /* how many tuples? */
    int    clSize,   /* how big is each cluster? */
    char * poolName /* name for network pool; commonly NULL */
)
{
    NETBUF_CFG    netBufCfg;
    NETBUF_CL_DESC clDescTbl;
    NET_POOL_ID   pPool;

    if (tupleCnt <= 0 || clSize < 0)
        return (NULL);

    bzero ((char *)&netBufCfg, sizeof(netBufCfg));
    bzero ((char *)&clDescTbl, sizeof(clDescTbl));

    netBufCfg.pName = poolName;
    netBufCfg.attributes = ATTR_AC_SH_ISR;
    netBufCfg.ctrlNumber = tupleCnt;

    if (size > 0)
    {
        netBufCfg.clDescTblNumEnt = 1;
        netBufCfg.pClDescTbl = &clDescTbl;
        clDescTbl.clNum = tupleCnt;
        clDescTbl.clSize = clSize;
    }

    pPool = netPoolCreate (&netBufCfg, _pLinkPoolFuncTbl);

    return (pPool);
}
```

The driver must specify a cluster size big enough for the maximum receivable frame. The `netPoolCreate()` call will round up the specified cluster size to a multiple of `NETBUF_ALIGN` (64), and return clusters aligned on `NETBUF_ALIGN`-byte boundaries.

In the current release, if size is at least 1500, then `netPoolCreate()`:

- Adds the default cluster offset, specified by the `NETBUF_LEADING_CLSPACE_DRV` parameter of component `INCLUDE_NETBUFLIB`, to the requested cluster size.
- Adjusts the `mBlkHdr.mData` pointers, for the tuples allocated from the pool, to point to that same offset after the start of the cluster.

This function also supports the much less common case of creating a pool with only bare `M_LINKs` and no clusters, by passing zero for `clSize`.

netPoolInit()

If for some reason you cannot use `netPoolCreate()` to create a network pool, you can call the `netPoolInit()` routine to initialize a `netBufLib` network pool for which you first allocate the memory yourself. You must allocate memory for the `NET_POOL` structure, as well as the clusters, `M_BLKs`, and `CL_BLKs`.

Pools that you create with `netPoolInit()` do not support some administrative capabilities of pools that you create using `netPoolCreate()` (see the discussion of

these capabilities in [2.3.2 Creating netBufLib Pools](#), p.12). However, pools created with **netPoolInit()** and **netPoolCreate()** are equivalent in regard to pool back end support and basic allocation/freeing of **M_BLKs**, **CL_BLKs**, and clusters.

```
STATUS netPoolInit
(
    NET_POOL_ID    pNetPool,          /* pointer to a net pool */
    M_CL_CONFIG *  pMclBlkConfig,     /* pointer to a mBlk configuration */
    CL_DESC *      pClDescTbl,        /* pointer to cluster desc table */
    int            clDescTblNumEnt,    /* number of cluster desc entries */
    POOL_FUNC *    pFuncTbl           /* pointer to pool function table */
)
```

The parameters that you pass to **netPoolInit()** are as follows:

pNetPool

A pointer to a **NET_POOL** structure that describes the pool to initialize.²

pMclBlkConfig

An **M_CL_CONFIG** structure that specifies the number of **M_BLKs** and **CL_BLKs** and which memory buffer for **netPoolInit()** to carve them from. This structure is defined in **netBufLib.h** as:

```
typedef struct
{
    int    mBlkNum;          /* number of mBlks */
    int    clBlkNum;         /* number of cluster Blocks */
    char * memArea;          /* pre allocated memory area */
    int    memSize;          /* pre allocated memory size */
} M_CL_CONFIG;
```

When you use the **linkBufPool** back end, **netPoolInit()** ignores the **clBlkNum** member; in such a pool, the number of **CL_BLKs** is always equal to the number of **M_BLKs**, since **linkBufPool** joins the two control structures into a contiguous **M_LINK** structure.

When you use the **netBufPool** back end, you usually will choose the number of cluster blocks to be equal to the total number of clusters in all cluster pools, and choose the number of **M_BLKs** to be at least this large, or larger if you anticipate cluster sharing. One exception to this general guideline is that if you primarily intend to allocate bare clusters (rather than tuples), you need not have as many control structures as clusters in the pool.

You must specify a memory region (**memArea**, **memSize**) sufficiently large for the number of control structures, considering also the alignment of the structures that the back end in use requires. Each **M_BLK** structure has, preceding it, a hidden pointer to the **NET_POOL** it comes from, and you must account for the space for these hidden pointers in **memSize**. For the **netBufPool** back end, the alignment requirement for both **M_BLKs** and **CL_BLKs** is just the size of a pointer (4 bytes); but for the **linkBufPool** back end, **M_LINKs** must have an alignment of **NETBUF_ALIGN**.

An easy way to find the memory required for these structures is to call the memory requirements routine **pFuncTbl->pMemReqRtn** (see [Memory Requirements Routines](#), p.18 for more information).

pClDescTbl

An array of **clDescTblNumEnt** **CL_DESC** structures, each of which describes a single cluster pool within the network buffer pool. The **CL_DESC** structure is defined in **netBufLib.h** as:

2. The **NET_POOL** structure is internal and, thus, is not documented.

```
typedef struct clDesc
{
    int    clSize;      /* cluster type */
    int    clNum;       /* number of clusters */
    char * memArea;     /* pre-allocated memory area */
    int    memSize;     /* size of pre-allocated memory size */
} CL_DESC;
```

Such a cluster pool is characterized by the number of clusters within it, and the (usable) size of each cluster within the pool. Note that when using the **linkBufPool** back end, only one cluster size is allowed. When using the **netBufPool** back end, the same restrictions on cluster sizes mentioned for **netPoolCreate()** apply (see [pClDescTbl](#), p.15).

Specify a region of available memory (**memArea**, **memSize**) from which **netPoolInit()** carves the clusters. If you specify a **memSize** value that is too small for the number of clusters in the pool, **netPoolInit()** fails, returning **ERROR**.

If you instruct **netPoolInit()** to use the **netBufPool** back end, when calculating **memSize**, account for the presence of a hidden **CL_POOL** pointer preceding each cluster. For the **linkBufPool** back end, while there is no hidden cluster pool pointer, the alignment requirements of each cluster are more stringent: you must round up each cluster size to a multiple of **NETBUF_ALIGN**, and add an additional **NETBUF_ALIGN** to allow for the whole block to align correctly. An easy way to calculate the memory needs in either case is to call the memory requirements routine described in [Memory Requirements Routines](#), p.18.

When using the **linkBufPool** back end, if you specify any clusters at all, you must specify the same number of clusters as **M_BLKs**, since **linkBufPool** permanently joins **M_BLKs**, **CL_BLKs**, and clusters into tuples. You cannot allocate bare **M_BLKs**, **CL_BLKs**, or clusters from such a pool.

clDescTblNumEnt

The number of structures in **pClDescTbl**.

pFuncTbl

The back end's table of function pointers; set this to **_pNetPoolFuncTbl** for the **netBufPool** back end, or **_pLinkPoolFuncTbl** for the **linkBufPool** back end.

Memory Requirements Routines

Call the memory requirements routines to determine the amount of memory you need for a particular number of **M_BLKs**, **CL_BLKs**, or clusters of a particular size. You can also call memory requirements routines to determine the required alignment of each single **M_BLK**, **CL_BLK**, or cluster.

Each of the two back ends **netBufPool** and **linkBufPool** provides its own memory requirements routine, and **netBufLib** also provides a default memory requirements routine, **_netMemReqDefault()**, that it uses when the second argument to **netPoolCreate()** is **NULL**. Alternatively, if you provide a custom **POOL_FUNC** back end function table, **netBufLib** obtains its memory requirements routine from the **pMemReqRtn** member of the **POOL_FUNC**, or uses **_netMemReqDefault()** if that member is **NULL**.

The prototype of a **netBufLib** memory requirements routine is defined in **netBufLib.h** as follows:

```
int memoryRequirementsRoutine
(
    int type, /* NB_BUFTYPE_[CLUSTER|M_BLK|CL_BLK] */
    int num, /* number of clusters or control structures */
    int size /* Cluster size (ignored for control structures) */
)
```

The arguments to this call are as follows:

type

What type of memory the caller wants to size, one of the following:

- **NB_BUFTYPE_CLUSTER** – cluster memory
- **NB_BUFTYPE_M_BLK** – **M_BLK** memory
- **NB_BUFTYPE_CL_BLK** – **CL_BLK** memory

num

The number of items; when this is zero, the routine returns the required alignment for a single **M_BLK**, **CL_BLK**, or cluster of the specified size.

size

For clusters only, this indicates the cluster size.

For instance, **netPoolCreate()** would make the following call to find out how much memory is needed for 200 clusters of size 1518 (**pMemReq** points to the appropriate memory requirements routine):

```
size = pMemReq (NB_BUFTYPE_CLUSTER, 200, 1518);
```

To find the alignment required for each **M_BLK**, it makes the following call:

```
align = pMemReq (NB_BUFTYPE_M_BLK, 0, 0);
```

pMemReq() returns a size such that a block of that size is sufficient to hold the specified number of properly aligned items, no matter the alignment of the block. This means that the memory requirements routine adds some extra size to guarantee correct alignment of the first block. To disregard this extra size and find the memory space used by each aligned item, use an expression such as the following:

```
oneItem = (pMemReq (NB_BUFTYPE_CL_BLK, 2, 0) -
           pMemReq (NB_BUFTYPE_CL_BLK, 1, 0));
```

If for some reason you need to modify the alignments that clusters or control structures use, one way to do this is to copy the **POOL_FUNC** table from the appropriate back end, and replace the **pMemReqRtn** member in this copy of the table with a pointer to your own memory requirements routine, and then pass the pointer to the copied **POOL_FUNC** table as the *pFuncTbl* argument to either **netPoolCreate()** or **netPoolInit()**.

For more information, see the reference entry for **netPoolInit()**.

2.4 Legacy Network Stack Pools

Previous versions of the Wind River Network Stack made use of two special **netBufLib** pools:

- the network stack data pool
- the network stack system pool

The stack used the data pool for packets sent to the network and for data in socket send buffers; it used the system pool for control structures such as sockets, route entries, protocol control blocks, socket addresses, and so on.

The network stack no longer uses **netBufLib** pools internally, except when it communicates with network device drivers. It does not require the legacy network stack data and system pools, and so the component **INCLUDE_NET_POOL** that includes and configures these pools is not present in the default VxWorks build. However, there may be certain cases in which you need these legacy pools.

For example, you may need the network stack data pool if you must prefix a link-layer header to a packet that a non-network-stack protocol sends, but there is insufficient leading space in the packet's head cluster to prefix the header. This may occur, for instance, in code that calls **muxAddressForm()** to prefix a link header to a datagram before sending it using **muxSend()**. The **muxAddressForm()** routine (or the device-specific **formAddress()** routine that it calls) uses the macro **M_PREPEND()** to prefix space for the link header to the packet. This macro, defined in *installDir/components/ip_net2-6.x/vxcoreip/include/net/mbuf.h*, adjusts pointers and lengths if there is sufficient leading space in the head cluster (and if the head cluster is not shared); otherwise, it calls the routine **m_prepend()**, which attempts to allocate a 128-byte tuple from the network stack data pool, to prefix to the existing chain and hold the link header. If the network stack data pool does not exist, this allocation fails (gracefully), and the attempt to send the packet fails.

Another example is an application or protocol that uses the **muxTkSend()** routine to send a packet to an END device, specifying a non-NULL destination MAC address. This routine calls the END's **formAddress()** routine in this case also.

If your application or protocol calls **muxAddressForm()** or **muxTkSend()** in this way and relies upon **M_PREPEND()** to successfully allocate a tuple, you may need to include the component **INCLUDE_NET_POOL** in your VxWorks image, and configure the data pool with at least one pool of clusters of size 128-bytes or larger, along with **M_BLKs** and **CL_BLKs**. (An alternative is to create a pool of your own for this purpose, and set the **NET_POOL** pointer **_pNetDpool** to point to this pool.)

For reference, here is a brief description of the parameters of the **INCLUDE_NET_POOL** component, used to configure the network stack system pool and network stack data pool. Note that both of these pools use the **netBufPool** back end.

NUM_SYS_MBLKS

The number of **M_BLK** structures in the system pool.

NUM_SYS_CLBLKS

The number of **CL_BLK** structures in the system pool.

PMA_SYSPPOOL

The address of a pre-allocated memory buffer that the system pool carves its **M_BLKs** and **CL_BLKs** from. To allow the initialization code to allocate this memory buffer, set this parameter and **PMS_SYSPPOOL** to zero.

PMS_SYSPPOOL

The size in bytes of the pre-allocated buffer at **PMA_SYSPPOOL**.

NUM_SYS_n

SIZ_SYS_n

PMA_SYS_n

PMS_SYS_n

These parameters, with *n* being one of 16, 32, 64, 128, 256, 512, 1024, or 2048, configure a cluster pool within the system pool. The value of **SIZ_SYS_n** specifies the usable size in bytes of each cluster in the pool, and must be at least *n* but less than 2 times *x*. **NUM_SYS_n** is the number of clusters in the cluster pool. **PMA_SYS_n** is the address of a pre-allocated buffer of length **PMS_SYS_n** bytes, which **netPoolInit()** carves into the clusters for the pool. To allow the initialization code to allocate memory itself for the cluster pool, set both **PMA_SYS_n** and **PMS_SYS_n** to zero.

NUM_DAT_MBLKS

The number of **M_BLK** structures in the data pool.

NUM_DAT_CLBLKS

The number of **CL_BLK** structures in the data pool.

PMA_DATPOOL

Address of a pre-allocated memory buffer to carve for the data pool's **M_BLKs** and **CL_BLKs**. To allow the initialization code to allocate the memory, set this parameter and **PMS_DATPOOL** to zero.

PMS_DATPOOL

The size in bytes of the pre-allocated buffer at **PMA_DATPOOL**.

NUM_DAT_n

PMA_DAT_n

PMS_DAT_n

These parameters, with *n* being one of 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536, configure a cluster pool within the data pool. The value of *n* is the usable size in bytes of each cluster in the pool; unlike the system pool, the data pool's cluster sizes are hard-coded as powers of two.

NUM_DAT_n is the number of clusters in the cluster pool. **PMA_DAT_n** is the address of a pre-allocated buffer of length **PMS_DAT_n** bytes, which **netPoolInit()** carves into the clusters for the pool. To allow the initialization code to allocate memory itself for the cluster pool, set both **PMA_DAT_n** and **PMS_DAT_n** to zero.

2.4.1 Legacy Tuple Pool for IPNET-Native Devices

The **INCLUDE_END2_LINKBUFPOOL** (Legacy **linkBufPool** for **END2** devices) component creates a single **linkBufPool** to be shared by all IPNET-native devices for the benefit of **M_BLK**-oriented protocols that allocate tuples out of an **END** device's private network pool. Note that such allocation is not a recommended practice, but some legacy protocols engage in it. [Table 2-2](#) lists the parameters for this component.

Table 2-2 **INCLUDE_END2_LINKBUFPOOL Parameters**

Parameter and Description	Default Value and Data Type
END2_LINKBUFPOOL_NTUPLES Specifies the number of tuples in END2 legacy linkBufPool .	8 ^a int
END2_LINKBUFPOOL_CLSIZE Specifies the size of tuple clusters in END2 legacy linkBufPool . Note that multiple devices, possibly with different MTUs, share the same pool; so choose a size sufficient for the maximum MTU value.	1600 uint

a. This value may increase to approximately 40.

3

Working with Network Drivers and Devices

- 3.1 Introduction 23
- 3.2 Overview of the MUX 23
- 3.3 Working with Network Driver Instances 25

3.1 Introduction

This chapter begins with an overview of the MUX model, which provides an interface between the data-link and TCP/IP protocol layers. It then describes how to do the following tasks:

- create, configure, and bring up network interface devices
- add and delete route table entries
- configure router advertisement and solicitation
- add automatic IPv4 interface configuration
- use a reverse ARP (RARP) client

3.2 Overview of the MUX

In the Wind River Network Stack, network interface drivers pass information up in the network stack through the mediation of an interface layer known as the MUX. The MUX insulates network services (protocols) from the specifics of network interface drivers and vice versa. This decoupling lets you add new network drivers (not necessarily Ethernet-based) without needing to alter the network service. Likewise, the decoupling lets you add a new network service without needing to modify the existing MUX-based network interface drivers.

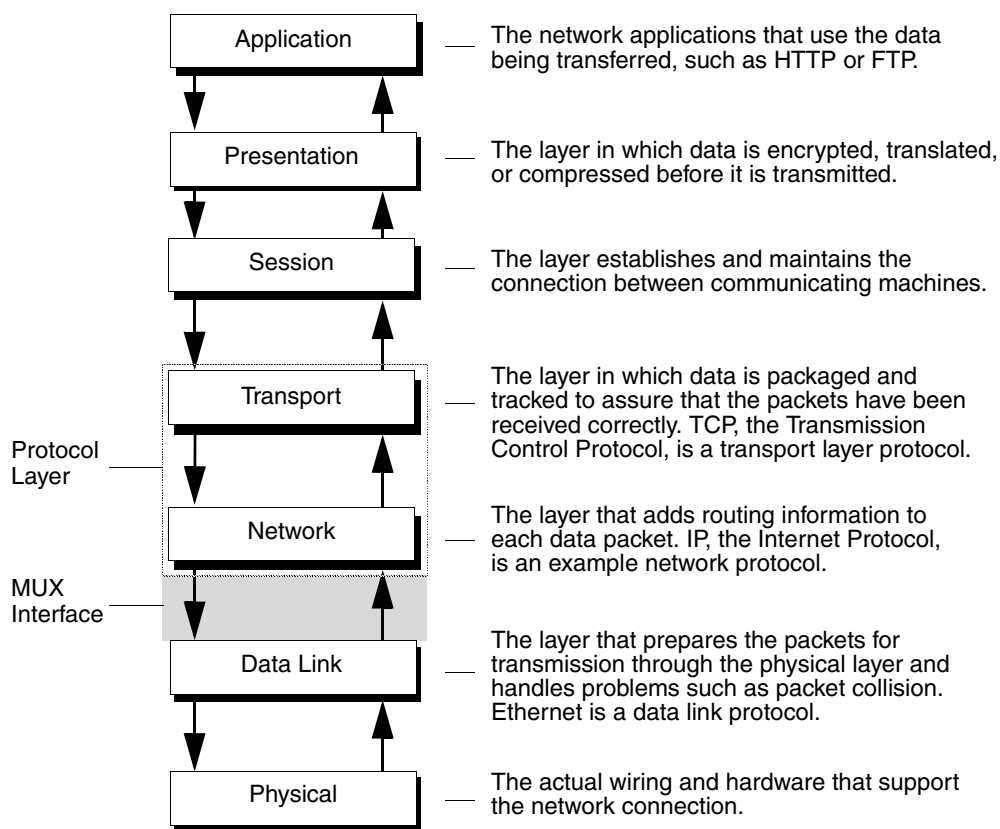
OSI Network Model

The OSI Network Model describes seven layers through which data passes when it is transmitted from an application on one machine to a peer on a remote machine.

Starting in the application layer, data passes down through each layer of the stack to the physical layer, which handles the physical transmission to the remote machine. After arriving on the remote machine, data passes up through each layer from the physical to the application.

In the abstract, each layer in the stack is independent of the other layers. When a protocol in one layer communicates with a peer protocol in the same layer on a remote machine, it passes its message to the layer immediately below it. The protocol itself is not responsible for passing messages further than the adjacent layer.

Figure 3-1 **OSI Network Model and the MUX**



MUX Layer

In practice, network stacks that implement each layer with perfect independence are rare. Within TCP/IP, the protocols that manage the transport and network layer routines are sufficiently coupled that they are sometimes referred to as the protocol layer. The MUX is an interface between the data link layer and this protocol layer.

The MUX is not a new layer. There are no MUX-level protocols that communicate with peers in the MUX of a remote machine. The MUX concerns itself solely with standardizing communication between the protocol and data link layers of a single stack. Because of the MUX, a protocol layer service and network driver do not need direct knowledge of the other's internal implementation details.

For example, when a network driver needs to pass along a packet it receives, the driver does not directly access any structure or routine within a protocol layer service implementation. Instead, the driver calls a MUX routine that handles the details. The MUX does this by calling the receive routine that the network service registered with the MUX. This design lets any MUX-compatible network service use any MUX-compatible network driver.

3.3 Working with Network Driver Instances

You can configure the Wind River Network Stack at either build time or run time to automatically load, start, and configure multiple network interfaces.



NOTE: This section covers real (hardware) network interface devices and their device drivers. While “pseudo” or “virtual” network interfaces can be loaded in the MUX (and may be specific to a particular network service, above the MUX), such virtual interfaces are not included in this discussion.

The required network drivers should already have been built into the VxWorks image. Although it is possible to dynamically download network drivers—if one has an alternate download path not depending on the driver itself—it is rarely done.

Network drivers in VxWorks are either:

- legacy pre-VxBus network drivers
- VxBus network drivers

Both types are frequently referred to as END drivers (meaning Enhanced Network Driver drivers), and their device instances are END devices.

From the point of view of network configuration, the two types of drivers differ primarily in how one causes the network devices that they control to be loaded into the MUX and started. The *VxWorks Device Driver Developer's Guide* contains detailed information on how VxBus drivers are registered with the VxBus system and how their device instances are initialized, loaded into the MUX, and started; this is done under control of the VxBus system, and the VxBus network drivers.



WARNING: You should not call `muxDevLoad()` to manually load a device controlled by a VxBus network driver.

Step 1: Loading Legacy Devices

Devices controlled by legacy network drivers are loaded in the MUX and started in one of two ways:

- Devices with entries listed in the **endDevTbl** array, usually defined in the BSP's **configNet.h** file, are loaded and started at system boot by network initialization code.¹ Devices may be entered in the **endDevTbl** array either statically or, in some cases, by BSP-specific code that scans a bus (usually a PCI bus) for network device instances.
- Devices not listed in the **endDevTbl** array can be manually (or programmatically) loaded into the MUX and started later, by calling **muxDevLoad()** and then **muxDevStart()**.²

You can see what devices are already loaded in the MUX by calling the **muxShow()** function from the C-interpreter shell (either the target-resident version or the host shell).

To manually add a device (controlled by a legacy network driver) into the MUX, call **muxDevLoad("deviceName", deviceUnit)**, specifying the appropriate device name and unit number. For example, to load the "fei2" device controlled by the legacy fei driver, you could use this command:

```
-> pFei2End = muxDevLoad ("fei", 2)
```

If successful, **muxDevLoad()** returns an END device cookie, which is really a pointer to the device's **END_OBJ** structure, although most code should treat the pointer as opaque. If unsuccessful, **muxDevLoad()** returns NULL. (You can also get the device cookie using the **endFindByName()** routine.)

Step 2: Starting Legacy Devices

Having successfully loaded the device, you can start it (that is, enable packet transmission and reception by the device) by calling **muxDevStart()** with the device cookie:

```
-> muxDevStart (pFei2End)
```

muxDevStart() returns OK if successful, **ERROR** if it fails. To stop a successfully started device, call **muxDevStop()** with the device cookie. as follows:

```
-> muxDevStop (pFei2End)
```

Note that starting or stopping an END device in this way affects all network services that bind to the device through the MUX; such services may handle whether a device is administratively "up" or "down" differently, but if a device is stopped, it is effectively down for all services bound to it.

Step 3: Attaching to a Device

Whether an END device is managed by a VxBus driver or a legacy END driver, once the END device is loaded in the MUX and started, you can then bind network services to the device.

To attach the Wind River network stack to an END device, call the **ipcom_drv_eth_init()** routine. For example, to bind the IP stack to the **fei2** device, call:

```
-> ipcom_drv_eth_init ("fei", 2, 0)
```

1. The code is the **usrNetEndLibInit()** function in **usrNetEndLib.c** (for VIP builds) or in **usrNetwork.c** (for legacy BSP builds).
2. Some such devices will require special BSP setup, which we do not discuss here.

The first two arguments are the device name and unit number. The last argument, 0, represents the NULL string. It is possible to make the network stack use an alias of its own for the device name; for example, if one called instead:

```
-> ipcom_drv_eth_init ("fei", 2, "eth3")
```

In this case, the interface would be known within the network stack (for instance, in **ifconfig** output) as "eth3". However, the MUX would still refer to the device as 'fei2'. If the third argument to **ipcom_drv_eth_init()** is zero (NULL), the stack uses the same name as the MUX for the device.

As a shortcut method, **ipAttach()** can attach the network stack to the device in the same way as a call to **ipcom_drv_eth_init()** with NULL as the third argument. Thus, running the command:

```
-> ipAttach (2, "fei")
```

has the same effect as the first **ipcom_drv_eth_init()** call above.

The **ipAttach()** function is available if the component **INCLUDE_IPWRAP_IPPROTO** is included in the VxWorks image.

Step 4: Displaying and Configuring Devices

Once you have attached the network stack to the device, you can display it and configure it as required for each network service. To configure the device, use the **ifconfig** command. The most convenient way to call this command is using the command interpreter shell (in the component **INCLUDE_SHELL_INTERP_CMD**). For example:

```
# ifconfig -a
```

lists all the network interfaces in the network stack, while:

```
# ifconfig fei2 10.21.0.1/24 up
# ifconfig fei2 inet6 add feed:1::cafe/64
```

adds an IPv4 and an IPv6 address to the **fei2** network interface.

If the command interpreter shell is not available, **ifconfig** is also available as a wrapper function **ifconfig()** (include the component **INCLUDE_IPWRAP_IFCONFIG**). You can call this routine from the C interpreter shell. It takes a single string argument, so it would be used as follows:

```
-> ifconfig "-a"
-> ifconfig "fei2 10.21.0.1/24 up"
```

The **ifconfig()** routine can be called programmatically, but the **ifconfig** command is also available programmatically as the routine:

```
IP_PUBLIC int ipnet_cmd_ifconfig(int argc, char **argv);
```

Step 5: Adding a Default Route

Other elementary network configuration includes adding a default route (in the example below, using the **route** command) and adding static host name entries to the host table:

```
# route add default 10.21.0.254      # add 10.21.0.254 as the default router
# route show                        # display routing table, also netstat -r
# C                                # change to C interpreter shell
-> hostAdd ("peer1", "10.21.0.11")    # add "peer1" as static host name
                                     # for 10.21.0.11
-> hostShow                          # display host table
```

For more information, see the *Wind River Network Stack Programmer's Guide, Vol. 1: Adding Routing Support*.

3.3.1 Attaching a Service to a Network Interface

A protocol or service, such as IPv4, must attach itself to one or more network interfaces to communicate with remote peers. When a service attaches to an interface, packets addressed to that interface can flow up to the service. This also lets the service transmit packets out through the interface.

If you want the IP stack to automatically attach to and configure up to four network interfaces, include the build configuration components **INCLUDE_IPNET_IFCONFIG_1**, **INCLUDE_IPNET_IFCONFIG_2**, **INCLUDE_IPNET_IFCONFIG_3**, and **INCLUDE_IPNET_IFCONFIG_4**, and set the value of the configuration parameters **IFCONFIG_1**, **IFCONFIG_2**, **IFCONFIG_3**, and **IFCONFIG_4** (where *n* is 1, 2, 3, or 4). By default, only **INCLUDE_IPNET_IFCONFIG_1** is in the image, so only one interface is attached and configured, according to the **IFCONFIG_1** parameter.

Set the **IFCONFIG_1** parameter to a series of strings, each of which begins with one of the following keywords:

ifname

The name that the IP stack uses for the network interface, for example: **"ifname eth0"**. If you simply specify **"ifname"** without an interface name, the network stack will use the same name as the MUX uses for the interface, which is given by the **devname** keyword.

devname

The name of the device to attach to, as it appears in the MUX, for example **"devname fei0"**. The default is **"devname driver"**, which indicates that the stack gets the name of the device, to attach itself to, from the "boot device" entry in the boot parameters.³

inet

The interface IPv4 address and subnet, for example: **"inet 10.1.2.100/24"**.

You may use one of the following keywords in place of the IPv4 address:

"inet driver"

The stack should read the interface IP address and mask from the **"inet on ethernet"** entry in the boot parameters. This is the default setting for the **inet** keyword, but only one interface should use this setting.

"inet dhcp"

The interface should retrieve its address and mask from a DHCP server. Depending on the DHCP server configuration, the interface might also retrieve its gateway from the server.

"inet rarp"

The interface should retrieve its address and mask from a RARP server.

gateway

The default IPv4 gateway, for example: **"gateway 10.1.2.1"**. You may specify the default gateway for only one of the statically configured interfaces. If you specify **"gateway driver"**, the stack uses the gateway from the **"gateway inet"** address in the boot parameters.

3. At most one of the statically configured interfaces should use this default.

inet6

The interface IPv6 address and subnet, for example: "**inet6 3ffe:1:2:3::4/64**". You can insert the **tentative** keyword before the address to instruct the stack to check for address duplication before it assigns the address to the interface, for example: "**inet6 tentative 3ffe:1:2:3::4/64**".

gateway6

The default IPv6 gateway. You may specify only one default gateway.

3.3.2 Configuring a Network Interface with an Address

When a network service attaches to an interface, packets addressed to that interface can flow up to the service. You must assign an address to the interface to route packets to it. When you assign an address to an interface, you also need to assign a netmask (IPv4) or prefix (IPv6) that determines whether the interface can access a particular network.

You can call **ifconfig()** to do either of the following tasks:

- Retrieve configuration information on an interface.
- Assign an address and a netmask or prefix to a network interface.

You can call the **ifconfig()** routine in the C-interpretter and in the command-interpretter. Examples of calls in the C-interpretter are prefaced with **->**. Examples in the command-interpretter are prefaced with **#**. To use **ifconfig**, include the **INCLUDE_IFCONFIG** configuration component in your build. For a detailed description of all possible flags and parameter values, see [ifconfig](#), p.173.



NOTE: See also the **ifLib** and **if6Lib** reference entries.

Retrieving Interface Information

Use the following **ifconfig()** syntax to retrieve information about a network interface:

```
-> ifconfig "[flags] [interfaceName] [command]"
# ifconfig [flags] [interfaceName] [command]
```

For the *interfaceName* parameter, the generic format of a network interface name is *nameNumber*—for example: **fei0**. The format of a logical interface name is *typeUserDefinedName*. For example, a PPPoE interface name would be **pppoeUserDefinedName**; a VLAN interface name would be **vlanUserDefinedName**. *UserDefinedName* can be any string that does not exceed **IFNAMSIZ** (16 characters).

For example, to retrieve information on the **fei0** interface, use the following command:

```
-> ifconfig "fei0"
```

To retrieve information on all IPv6 interfaces, use the following command:

```
-> ifconfig "-a"
```

Configuring an Interface

Use the following **ifconfig()** syntax to assign information to a network interface:

```
-> ifconfig "interfaceName" [protocol] command"
# ifconfig interfaceName [protocol] command
```

For the *protocol* parameter, specify either **inet** or **inet6**.

inet

The **inet** protocol value has special syntax because you use it both as a protocol selector and as a command to change the primary IPv4 address.

For example, to change the primary IPv4 address to 192.0.2.64, issue the following command:

```
-> ifconfig "fei0" inet 192.0.2.64
```

To configure the **fei0** network interface to add another IPv4 address, of 172.16.0.100, use the following command:

```
-> ifconfig "fei0" inet add 172.16.0.100
```

Because the call specifies no mask for what would have been a class B address pre-CIDR, the command assumes a default class B netmask value of 0xffff0000. To override the default netmask associated with an IPv4 address, use the CIDR slash notation as follows:

```
-> ifconfig "fei0" inet add 172.16.0.100/24
```

Alternatively, you can specify the mask using the dot-notation as follows:

```
-> ifconfig "fei0" inet add 172.16.0.100 netmask 255.255.255.0
```

These last two commands each specify a netmask of 24 bits, or 0xffff0000. The stack uses the netmask value when it creates an interface entry in the system route table.

inet6

To use **ifconfig()** to configure IPv6 addresses, use the **inet6** protocol value. For example:

```
-> ifconfig "fei0" inet6 add 2002:C000:0240::66/16
```

The **ifconfig()** call above configures the **fei0** network interface to have an IPv6 address of 2002:C000:240::66.



NOTE: As there is no concept of a primary IPv6 address, you only use **inet6** as a protocol selector.

In the command above, the string **/16** indicates a prefix length of 16. If you do not specify a prefix length, **ifconfig()** assumes a default prefix of 64 bits. Alternatively, you can use the **prefixlen** option as follows:

```
-> ifconfig "fei0" inet6 add 2002:C000:0240::66 prefixlen 16
```

This example address is a 6to4 IPv6 address with a local IPv4 tunnel address of 192.0.2.64. The IPv4 notation uses base 10; the IPv6 notation uses base 16. Thus, 192.0.2.64 is 0xC0000240, which is written as C000:0240 in IPv6 notation.

Creating a Pseudo-Interface

You can call the **ifconfig()** routine to create pseudo-interfaces (also called *logical interfaces*) such as VLAN interfaces and tunnels.

For example, to create a VLAN interface that puts the VLAN tag 1234 on all traffic sent to the 10.0.0.0/8 network, that is assigned the address 10.1.2.3, and that uses **fei0** as the underlying physical interface, issue the following series of commands:

```
-> ifconfig "vlan10 create"  
-> ifconfig "vlan10 vlan 1234 vlanif fei0"  
-> ifconfig "vlan10 inet 10.1.2.3"  
-> ifconfig "vlan10 up"
```

You can also chain these commands together:

```
-> ifconfig "vlan10 create vlan 1234 vlanif fei0 inet 10.1.2.3 up"
```

Creating Unnumbered Interfaces

While most stacks restrict unnumbered interfaces to point-to-point interface types, the Wind River Network Stack allows you to turn any type of interface into an unnumbered interface simply by assigning it the same IP address as another interface (for example, the main interface).

As an example, suppose Ethernet interface **gei0** is assigned the address **1.2.3.4**. You can turn interface **ppp0** into an unnumbered interface by assigning the same address to it as follows:

```
-> ifconfig ppp0 inet 1.2.3.4
```

Note that this is exactly the same command you use to assign an address to an interface; the only difference is that, in this case, it is the same as the Ethernet interface, which thus turns **ppp0** into an unnumbered interface.

3.3.3 Fixing Interfaces That Have Erroneous Addresses

When you call **ifconfig()**, you create a local entry in the route table. Local entries in the route table identify network interfaces on the local host.

To reconfigure a network interface using **ifconfig()**, issue one of the following commands:

```
-> ifconfig "interfaceName inet delete oldIPv4address add newIPv4address"  
-> ifconfig "interfaceName inet6 delete oldIPv6address add newIPv6address"
```

If you are changing the primary IPv4 address, issue the following command:

```
-> ifconfig "interfaceName inet newIPv4address"
```

To restore all the local network route entries to the route table, issue the following command:

```
-> ifconfig "interfaceName down up"
```

For more information, see [3.3.2 Configuring a Network Interface with an Address](#), p.29.

3.3.4 Assigning a Host Name to an Address

It is often easier to refer to hosts and interfaces by names instead of by IP addresses. To add host names to the local host table, call **hostAdd()**. For example:

```
-> hostAdd "myIPv4Interface", "192.0.2.64"  
-> hostAdd "myIPv6Interface", "2002:C000:240::66"
```

The host table stores one entry per Internet address, but you can associate multiple names with an address, as aliases.

When specifying IPv6 link or site local addresses, you must supply a scope delimiter, %, and corresponding interface ID. For more information, see the **hostAdd()** reference entry.

3.3.5 Bringing the Device Up for Protocol Communication

To start and enable a network interface, call **ifconfig()** to bring up the protocol state of the device. You can start or stop transmission by bringing this state up or down. To bring up IPv4 and IPv6 functionality, use the **up** modifier. For example:

```
-> ifconfig "fei0 up"
```

Similarly, to bring down IPv4 and IPv6 functionality, use the **down** modifier. For example:

```
-> ifconfig "fei0 down"
```

3.3.6 Determining the Device Link Status

The network stack's **IFF_RUNNING** and **IFF_UP** flags indicate the current status of a link. For example, an Ethernet network interface sets the **IFF_RUNNING** flag as long as one end of a network cable is plugged into the device and the other end is plugged into another device (using a cross-cable) or into a hub or switch. Use the **SIOCGIFFLAGS ioctl()** call to determine the state of this flag, as follows:

```
struct ifreq ifp;
int fd = socket descriptor for the link ;
ioctl (fd, SIOCGIFFLAGS, &ifp);
if (ifp.ifr_flags & IFF_RUNNING)
{
    // interface is running
}
if (ifp.ifr_flags & IFF_UP)
{
    // interface is UP
}
```

Working with Tunneling and Shared Memory

- 4.1 Introduction 33
- 4.2 Working with IPv4 and IPv6 Tunneling 33
- 4.3 Using the Shared-Memory Network 43
- 4.4 Shared-Memory Network Configuration 50

4.1 Introduction

This chapter describes tunneling and shared-memory.

In this document, *tunneling* through an IPv4 Internet refers to the encapsulation of data (such as an IPv4 or an IPv6 packet for a VPN connection) in an IPv4 packet that is then transmitted to an IPv4-addressed destination. At the destination, the data is retrieved from the IPv4 packet and processed. If the data retrieved is an IPv6 packet, and the receiving stack is a dual IPv4/IPv6 stack, the stack can forward the IPv6 packet out onto the IPv6 Internet.

By using a shared-memory network driver, multiple processors can communicate over their common backplane as if they were communicating over a network, through a MUX-capable network driver. The second half of this chapter covers shared memory.



NOTE: The tunneling feature is available only in the Wind River Platforms builds of the network stack. The Wind River General Purpose Platform, VxWorks Edition, does not support tunneling.

4.2 Working with IPv4 and IPv6 Tunneling

A tunnel attaches to the stack as a network interface. Create tunneling interfaces with the `ifconfig` command.



NOTE: IPsec does its own IP tunneling for associations running in tunneling mode; therefore, an IPsec tunnel will not show up as a regular network interface.

Tunnels can be either *configured* or *automatic*. A configured tunnel determines the endpoint addresses by configuration information on the encapsulating node. An automatic tunnel determines the IPv4 endpoints from the addresses of the embedded IPv6 datagram.

IPv4 multicast tunneling determines the endpoints through Neighbor Discovery. See the *Wind River Network Stack Programmer's Guide, Volume 1* for more on Neighbor Discovery.



NOTE: Although this chapter focuses on host-to-host tunnels, you can also set up host-to-router tunnels, router-to-router tunnels, and other permutations.

4.2.1 Configuring VxWorks for Tunneling

The Wind River Network Stack includes the following tunneling components:

- 6over4 Tunnel Interface Driver (RFC 2529)
- 6to4 Tunnel Interface Driver (RFC 3056)
- GIF Tunnel Interface Driver (RFC 1853, RFC 2473)
- GRE Tunnel Interface Driver (RFC 2002, RFC 2784)
- SIT Tunnel Interface Driver

The `INCLUDE_IPNET_USE_TUNNEL` (**Tunnel Interface support**) component enables these.

GIF Tunnel Interface Driver

The `INCLUDE_IPNET_USE_TUNNEL` (**Tunnel Interface support**) component pulls in modules that implement the **gif** tunneling pseudo-device for IPv4 and IPv6. GIF can tunnel IPv4 and IPv6 over IPv4 or IPv6. GIF tunneling is configured, not automatic, which means you must specify the endpoints when the tunnel is created, rather than relying on the endpoints being extracted from the addresses of the protocol being tunneled; you can configure endpoints per route entry.¹

The GIF component has no configuration parameters and is automatically included when you enable tunneling support.

Use **ifconfig** to create GIF tunnel interfaces. Interfaces for GIF tunnels must begin with "gif". For example:

```
_-> ifconfig "gif0 create"
```

To set tunnel endpoints on a GIF tunnel, use **ifconfig**. For details, see [ifconfig](#), p.173. For example, to set the endpoints on a GIF tunnel over IPv4:

```
-> ifconfig "gif0 inet tunnel 192.168.0.10 10.1.2.3"
```

1. Meaning, after having created the tunnel network interface you can route commands to direct traffic to any number of various destination hosts or subnets to use the tunnel interface.

To set the endpoints on a GIF tunnel over IPv6:

```
-> ifconfig "gif0 inet6 tunnel 2001:10::10 2001:20::1"
```

GRE Tunnel Interface Driver

The `INCLUDE_IPNET_USE_TUNNEL (Tunnel Interface support)` component pulls in support for the `gre` tunneling pseudo-devices for IPv4 and IPv6. GRE can tunnel IPv4 and IPv6 over IPv4 or IPv6. Like GIF tunneling, GRE tunnels are *configured* as opposed to *automatic* (which is to say that you must specify the endpoints when the tunnel is created, rather than relying on the endpoints being extracted from the addresses of the protocol being tunneled; you can configure endpoints per route entry. GRE has a version field set to 0 and provides an optional payload checksum.

GRE can be run in "minimal encapsulation" when tunneling IPv4 over IPv4 (described in RFC 2002).

Use `ifconfig` to create tunnel interfaces. Interfaces for GRE tunnels must begin with "`gre`". For example:

```
-> ifconfig "greTunnel create"
```

To set tunnel endpoints on a GRE tunnel, use `ifconfig`:

```
ifconfig "interfaceName [inet | inet6] tunnel localAddress remoteAddress"
```

For example:

```
-> ifconfig "greTunnel inet6 tunnel 2001:10::10 2001:20::1"
```

6over4 Tunnel Interface Driver

The `INCLUDE_IPNET_USE_TUNNEL (Tunnel Interface support)` component pulls in support for the `6over4` tunneling pseudo-device for IPv6.

A `6over4` tunnel is an IPv4 multicast tunnel that requires a fully functional IPv4 multicast infrastructure.

`6over4` requires the `INCLUDE_IPCOM_USE_INET6` component and has no configuration parameters.

Use `ifconfig` to create tunnel interfaces. Interfaces for `6over4` tunnels must begin with "`6over4`". For example:

```
-> ifconfig "6over4Lan create"
```

To set a local IPv4 address for a `6over4` tunnel, use `ifconfig`:

```
-> ifconfig "interfaceName inet tunnel localAddress remoteAddress"
```

As shown above, `ifconfig` requires a peer address, even though it does not use it in this case; provide a dummy peer address, such as the following:

```
-> ifconfig "6over4Lan inet tunnel 192.168.0.10 0.0.0.0"
```

6to4 Tunnel Interface Driver

The `INCLUDE_IPNET_USE_TUNNEL (Tunnel Interface support)` component pulls in support for the `6to4` tunneling pseudo-device for IPv6. Unlike GIF and GRE,

6to4 is an *automatic* tunnel: tunnel endpoints are extracted from the encapsulated IPv6 datagram, and so you do not need to configure them manually.

6to4 tunnels use a prefix of the form "2002:tunnelIPv4address::/48" (for instance, "2002:a01:203::1") to tunnel IPv6 traffic over IPv4 (see RFC 3056). Routers advertise a prefix of the form "2002:[IPv4]:xxxx/64" to IPv6 clients.

This component requires the **INCLUDE_IPCOM_USE_INET6** component and has no configuration parameters.

Use **ifconfig** to create tunnel interfaces. Interfaces for 6to4 tunnels must begin with "6to4". For example:

```
-> ifconfig "6to4tun create"
```

SIT Tunnel Interface Driver

Like 6to4, the SIT (Simple Inter-site Tunnel) tunneling device for IPv6 is an automatic tunnel; tunnel endpoints are extracted from the encapsulated IPv6 datagram.

SIT uses IPv4-compatible IPv6 addresses (for instance, "::10.1.2.3") to tunnel IPv6 traffic over IPv4. The IPv4 address is in the 32 least-significant bits in the IPv6 address. Such an address uses the prefix "::/96".

Use **ifconfig** to create tunnel interfaces. Interfaces for SIT tunnels must begin with "sit". For example:

```
-> ifconfig "sit0 create"
```



NOTE: The IETF has deprecated the use of this tunnel type.

4.2.2 Creating 6to4 Tunnels for IPv6 Packets

In the Wind River Network Stack, a 6to4 pseudo-device is an automatic tunnel and one of the many IPv6 transition mechanisms.

Figure 4-1 **6to4 Addresses**

2002	IPv4 Address	SLA ID	Interface ID
16 bits	32 bits	16 bits	64 bits

Consider the following setup code for automatic tunneling on a Wind River Network Stack dual stack.

```
/* code for 6to4 tunnel setup */
#include "vxWorks.h"
#include "net/utils/ifconfig.h"
#include "net/utils/routeCmd.h"

void tunnel6to4Test ( )
{
    /* Create and attach the tunnel */
    ifconfig ("6to4tun create");
    /* Add IPv6 address to the tunnel */
    ifconfig ("6to4tun inet6 add 6to4AddressForLocalInterface prefixlen 128");
    /* Bring up the tunnel */
    ifconfig ("6to4tun up");
}
```

```
/* Route all packets to the 2002::/16 network through
 * the "6to4tun" tunnel */
routecl ("add -inet6 -net -dev 6to4tun -prefixlen 16 2002::");
}
```

The second `ifconfig()` call creates a route table entry that catches packets with IPv6 destinations that match the first 128 bits of the *6to4AddressForLocalInterface*. This is the IPv4 address of the local network interface prefixed with an IPv6 6to4 prefix. This 6to4 IPv6 address implies a local IPv6 address space of 80 bits.

By convention, the 16 bits just after the IPv4 segment of the address are interpreted as a Site-Level Aggregation (SLA) ID, which you can set to any value. You may use this to organize the local space into several (up to 65,536) subnets. Because the site as a whole is identified to other sites by the first 48 bits of the address, you can use the remaining bits at your convenience without affecting the ability of remote machines to communicate with the site.

If you intend the systems at the tunnel ends to function as routers, make sure that the network stacks at those tunnel ends are configured to forward IPv6 packets. In the Wind River Network Stack, call `sysctl()` for this purpose:

```
-> sysctl "net.inet6.ip6.forwarding=1"
```

For more information on `sysctl`, see [sysctl](#), p.187.

4.2.3 Creating RFC 2893-Style Configured Tunnels

The RFC 2893-style configured tunnels based on `gif` devices are point-to-point links. They are similar to point-to-point links over a serial cable except that the transmission medium is the Internet. Through the mediation of a `gif` device instance, you can use a configured tunnel as a direct connection to an endpoint in the IPv6 address space. Unlike with `stf`-based tunnels, you do not need to define the tunnel endpoints in terms of 6to4 IP addresses. Both endpoints need to support dual IPv4/IPv6 stacks, and both tunnel endpoints need an IPv4 address in addition to whatever IPv6 addresses you associate with the tunnel endpoints.

You set a `gif`-based configured tunnel in much the same way as you set up an `stf`-based automatic tunnel. To create a `gif` device instance and bind it to the IPv6 stack, use code modeled after the following:

```
/* code for gif tunnel-over-IPv4 setup. The tunnel will be
 * configured with an IPv6 address, but it is possible to add an IPv4 * address
 * as well */
#include "vxWorks.h"
#include "net/Utils/ifconfig.h"

void tunnelGifTest ( )
{
    /* Create and attach the tunnel */
    ifconfig ("gif0 create");
    /* Configure the tunnel to tunnel over IPv4 */
    ifconfig ("gif0 inet tunnel localIPv4Address remoteIPv4Address");
    /* Add IPv6 address to the tunnel */
    ifconfig ("gif0 inet6 add IPv6AddressForTunnel");
    /* Bring up the tunnel */
    ifconfig ("gif0 up");
}
```

Packets sent to this `gif` device are always transmitted from the specified local interface to the specified remote interface.

Note that the IPv6 addresses you supply in the code, like the example above, need not be 6to4 addresses. They can be any valid IPv6 addresses (of proper scope) validly associated with the tunnel endpoints.

To direct IPv6 packets to this device by default, add an IPv6 default entry to the route table. For example:

```
-> route "add -inet6 default tunnelEndpointIPv6Address"
```

Alternatively, you can direct only some IPv6 packets to the interface. For example, the following entry captures traffic destined for 2001:DB8:1::/64:

```
-> route "add -inet6 2001:DB8:1:: tunnelEndpointIPv6Address -prefixlen 64"
```

Example 4-1 Configuration Example

In this example, assume the following:

- you own the IPv4 address 10.1.0.1 and the IPv6 address 2001:DB8:1234::1
- the owner of IPv4 address 10.2.0.1 has agreed to route your IPv6 traffic for 2001:DB8:3333::/32
- the IPv4 node at 10.2.0.1 is a dual IPv4/IPv6 stack with an IPv6 address of 2001:DB8:5678::1

Create a GIF tunnel between 10.1.0.1 and 10.2.0.1 as follows:

```
-> ifconfig "gif0 create up"
-> ifconfig "gif0 inet tunnel 10.1.0.1 10.2.0.1"
```

Then configure the interfaces and route table as follows:

```
-> ifconfig "fei0 inet6 2001:DB8:1234::1"
-> ifconfig "gif0 inet6 2001:DB8:1234::1 2001:DB8:5678::1 prefixlen 128"
-> route "add -inet6 2001:DB8:3333:: 2001:DB8:5678::1 -prefixlen 48"
```

Your local node now supports the following devices and route table entries:

Table 4-1 Devices and Route Table Entries

	Addresses	Configured
fei0	10.1.0.1	
fei0	2001:DB8:1234::1	
gif0	2001:DB8:1234::1 -> 2001:DB8:5678::1	10.1.0.1 ->10.2.0.1
	Routes	
	10.1.0.1/8	fei0
	2001:DB8:1234::/64	fei0
	2001:DB8:3333::/32	gif0

If you negotiate for routing services from more than one remote IPv6 router that supports a dual IPv4/IPv6 stack, you can create additional **gif** devices to manage configured tunnels through those routers. You would also want to set up your route table to control which IPv6 packets go to which router.

For example, if the second remote dual IPv4/IPv6 stack is at 10.3.0.1 IPv4 and 2001:DB8:9999::1 IPv6, use the following setup code:


```
-> ifconfig "gif0 create up"
-> ifconfig "gif1 create up"
-> ifconfig "gif0 inet tunnel 10.1.0.1 10.2.0.1"
-> ifconfig "gif1 inet tunnel 10.1.0.1 10.3.0.1"
-> ifconfig "fe10 inet6 add 2001:DB8:1234::1"
-> ifconfig "gif0 inet6 add 2001:DB8:1234::1 prefixlen 128"
-> ifconfig "gif1 inet6 add 2001:DB8:1234::1 prefixlen 128"
```

This sets up the following devices:

Table 4-2 Addresses

Devices	Addresses	Configured
gif0	2001:DB8:1234::1 -> 2001:DB8:5678::1	10.1.0.1 -> 10.2.0.1
gif1	2001:DB8:1234::1 -> 2001:DB8:9999::1	10.1.0.1 -> 10.3.0.1

To add a route over **gif0** use 5678::1 in a **route** command. For example:

```
-> route "add -inet6 2001:DB8:1:: 2001:DB8:5678::1 -prefixlen 64"
```

To add a route over **gif1** use 2001:DB8:9999::1 in a **route** command. For example:

```
-> route "add -inet6 2001:DB8:4444 2001:DB8:9999::1 -prefixlen 48"
```

Thus, the route table would include the following entries:

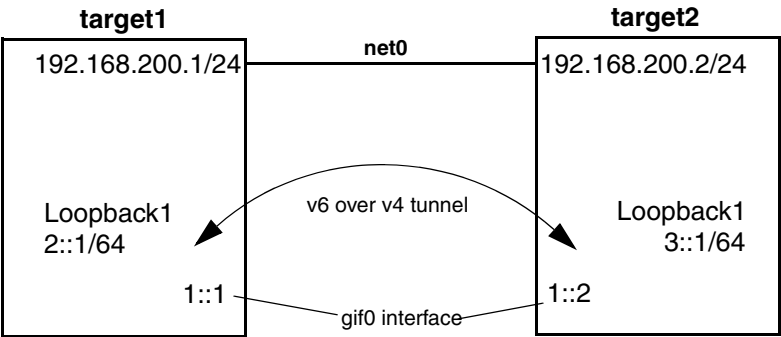
Table 4-3 Route Entries

Routes	Device
2001:DB8:9999:1::/64	gif0
2001:DB8:4444::/48	gif1

4.2.4 Tunnel Example

This example shows how to establish a GIF tunnel to tunnel IPv6 traffic over an IPv4 connection. In this example, there are two targets, connected as shown in [Figure 4-2](#).

Figure 4-2 Two Targets Establish a GIF Tunnel



Step 1: Establish the Network

To create this topology, do the following:

1. On **target1**, issue the following command:

```
# ifconfig gif0 create inet tunnel 192.168.200.1 192.168.200.2 inet6 add 1::1 up
```

2. On **target2**, issue the following command:

```
# ifconfig gif0 create inet tunnel 192.168.200.2 192.168.200.1 inet6 add 1::2 up
```

This creates a new interface called **gif0**: a virtual interface that connects endpoint 192.168.200.1 to 192.168.200.2 on **target1** and 192.168.200.2 to 192.168.200.1 on **target2**.

You can verify that the tunnel's endpoints are IPv4 by looking at the interface type in the output from the **ifconfig gif0** command:

```
# ifconfig gif0
gif0    Link type:Tunnel  Queue:none  IPv[4|6]-over-IPv4 192.168.200.2 -->
192.168.200.1  ttl:64
        inet 224.0.0.1  mask 240.0.0.0
        inet6 unicast FE80::C0A8:C802%gif0  prefixlen 64  automatic
        inet6 unicast 1::2  prefixlen 64
        inet6 unicast FE80::%gif0  prefixlen 64  anycast
        inet6 unicast 1::  prefixlen 64  anycast
        inet6 multicast FF02::1:FF00:2%gif0  prefixlen 16
        inet6 multicast FF02::1:FF00:0%gif0  prefixlen 16
        inet6 multicast FF02::1%gif0  prefixlen 16  automatic
        inet6 multicast FF02::1:FFA8:C802%gif0  prefixlen 16
        UP RUNNING SIMPLEX POINTOPOINT MULTICAST NOARP
        MTU:1480  metric:1  VR:0
        RX packets:0 mcast:0 errors:0 dropped:0
        TX packets:10 mcast:10 errors:0
        collisions:0 unsupported proto:0
        RX bytes:0  TX bytes:688
```

Notice that the inet6 address of **gif0** is **1::2 (inet6 unicast)**, which means that the tunnel will forward all packets destined to network **1::/64**.

From **target2**, ping the IPv6 address of the **target1** interface (**1::1**):

```
# ping6 ::1

Pinging ::1 (::1) with 64 bytes of data:
Reply from ::1 bytes=64 time=0ms hlim=64
Reply from ::1 bytes=64 time=0ms hlim=64
Reply from ::1 bytes=64 time=0ms hlim=64
Reply from ::1 bytes=64 time=0ms hlim=64

--- ::1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 4080 ms rtt
min/avg/max = 0/0/0 ms
```

Step 2: Add Loopback Interfaces and Default Routes.

Since the default loopback interface (**lo0**) is used by many system calls and socket applications you will add a new loopback interface (**lo1**). This interface represents a different network.

In most cases a tunnel is used to connect networks that cannot be directly connected. This example shows how to add a loopback interface on each target and apply an IPv6 address to each interface. Then it shows how to use the **route** command to tell the network device about remote networks and how to reach those networks using a default gateway.

On **target1**, issue the following command to add a loopback interface:

```
# ifconfig lo1 create inet6 add 2::1 up
```

Verify the new interface:

```
# ifconfig lo1
lo1      Link type:Local loopback  Queue:none
         inet 224.0.0.1  mask 240.0.0.0
         inet6 unicast FE80::1%lo1  prefixlen 64  automatic
         inet6 unicast 2::1  prefixlen 64
         inet6 multicast FF02::1:FF00:1%lo1  prefixlen 16
         inet6 multicast FF02::1%lo1  prefixlen 16  automatic
         UP RUNNING LOOPBACK MULTICAST
         MTU:1500  metric:1  VR:0
         RX packets:3 mcast:0 errors:0 dropped:3
         TX packets:3 mcast:3 errors:0
         collisions:0 unsupported proto:0
         RX bytes:168  TX bytes:168
```

On **target2**, issue the following command to add a loopback interface:

```
# ifconfig lo1 create inet6 add 3::1 up
```

Verify the new interface:

```
# ifconfig lo1
lo1      Link type:Local loopback  Queue:none
         inet 224.0.0.1  mask 240.0.0.0
         inet6 unicast FE80::1%lo1  prefixlen 64  automatic
         inet6 unicast 3::1  prefixlen 64
         inet6 multicast FF02::1:FF00:1%lo1  prefixlen 16
         inet6 multicast FF02::1%lo1  prefixlen 16  automatic
         UP RUNNING LOOPBACK MULTICAST
         MTU:1500  metric:1  VR:0
         RX packets:3 mcast:0 errors:0 dropped:3
         TX packets:3 mcast:3 errors:0
         collisions:0 unsupported proto:0
         RX bytes:168  TX bytes:168
```

From **target2**, ping the loopback interface of **target1** (2::1/64):

```
# ping6 2::1
Pinging 2::1 (2::1) with 64 bytes of data:
Echo request operation failed: Network is unreachable (51)
```

You cannot ping interfaces on the network 2::/64 since you do not have a route to that network. Look at the routing table of **target2** by issuing the **route show** command:

```
# route show

INET route table - vr: 0, table: 254
Destination      Gateway          Flags Use    If    Metric
0.0.0.0/0        192.168.200.254 UGS    0    net0    0
127.0.0.0/8      localhost        UR     0    lo0     0
localhost        localhost        UH    12    lo0     0
192.168.200.0/24  link#2           UC     1    net0    0
192.168.200.1    7a:7a:c0:a8:c8:01 UHL   10    net0    1
target2          link#1           UH    10    lo0     0

INET6 route table - vr: 0, table: 254
Destination      Gateway          Flags Use    If    Metric
::              link#1           UHRS   0    lo0     0
::1             ::1             UH    48    lo0     0
1::/64          link#4           U     0    gif0    0
1::2           link#1           UH     0    lo0     0
3FFE:1:2:3::/64 link#2           UC     0    net0    0
3FFE:1:2:3::4   link#1           UH     0    lo0     0
FE80::%lo0/64   link#1           UC     0    lo0     0
FE80::%net0/64   link#2           UC     0    net0    0
FE80::%gif0/64   link#4           U     0    gif0    0
FE80::1%lo0     link#1           UH     0    lo0     0
```

Or you can use the `-inet6` flag to view only IPv6 route configuration:

```
# route show -inet6

INET6 route table - vr: 0, table: 254
Destination      Gateway          Flags Use    If    Metric
::               link#1           UHRS  0      lo0    0
::1              ::1              UH    48     lo0    0
1::/64           link#4           U      0      gif0   0
1::2             link#1           UH     0      lo0    0
3FFE:1:2:3::/64  link#2           UC     0      net0   0
3FFE:1:2:3::4    link#1           UH     0      lo0    0
FE80::%lo0/64    link#1           UC     0      lo0    0
FE80::%net0/64   link#2           UC     0      net0   0
FE80::%gif0/64   link#4           U      0      gif0   0
FE80::1%lo0      link#1           UH     0      lo0    0
```

Step 3: Add a Default Gateway

Notice that IPv6 does not have a default gateway (a default gateway is notated by the letter "G" in the flags field). Since we have only one exit point from the device, we can configure the tunnel as a default gateway for any unknown IPv6 destination.

Issue the following command to add a default gateway to `target2`:

```
# route add -inet6 default 1::1
add net ::: netmask ::: gateway 1::1
```

Now, look at the route table again:

```
# route show -inet6

INET6 route table - vr: 0, table: 254
Destination      Gateway          Flags Use    If    Metric
::               link#1           UHRS  0      lo0    0
::/0             1::1            UGS   0      gif0   0
::1              ::1              UH    48     lo0    0
1::/64           link#4           U      0      gif0   0
1::2             link#1           UH     0      lo0    0
3FFE:1:2:3::/64  link#2           UC     0      net0   0
3FFE:1:2:3::4    link#1           UH     0      lo0    0
FE80::%lo0/64    link#1           UC     0      lo0    0
FE80::%net0/64   link#2           UC     0      net0   0
```

Add a default gateway to `target1`:

```
# route add -inet6 default 1::2
add net ::: netmask ::: gateway 1::2
```

Step 4: Test the Tunnel

With a default gateway and a v6-over-v4 tunnel you can ping the loopback interfaces. From `target1`:

```
# ping6 3::1
Pinging 3::1 (3::1) with 64 bytes of data:
Reply from 3::1 bytes=64 time=0ms hlim=64
Reply from 3::1 bytes=64 time=0ms hlim=64
Reply from 3::1 bytes=64 time=0ms hlim=64
Reply from 3::1 bytes=64 time=0ms hlim=64

--- 3::1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 4080 ms rtt
min/avg/max = 0/0/0 ms
```

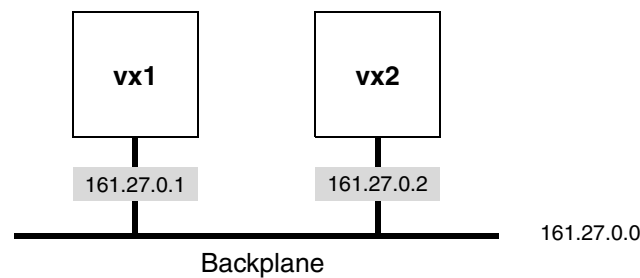
This shows ping-v6 packets encapsulated in IPv4 packets and carried from `target1` to `target2` over an IPv4 link.

4.3 Using the Shared-Memory Network

The **smEnd** shared-memory network driver allows multiple processors to communicate over their common backplane as if they were communicating over a network, through a MUX-capable network driver.

A multiprocessor backplane bus is an Internet network with its own network / subnet number. Each processor that is a host on this network has its own unique IP address. In the example shown in Figure 4-3, two processors are on a backplane. The Internet address for the shared-memory network is 161.27.0.0. Each processor on the shared-memory network has a unique Internet address, 161.27.0.1 for **vx1** and 161.27.0.2 for **vx2**.

Figure 4-3 Shared-Memory Network



Processors can communicate with other processors on the same backplane by means of an instance of the **smEnd** driver. This driver behaves as any other network driver, and so a variety of network services may communicate through it.

4.3.1 Backplane Shared-Memory Region

This simulation of driver communication takes place in a contiguous memory region that all processors on the backplane can access through instantiations of the **smEnd** driver.²

Backplane Processor Numbers

Assign each processor on the backplane a unique *backplane processor number* starting with 0. With the exception of processor #0, which by convention and by default is the shared-memory network master (described below), these numbers are arbitrary and you may set them to whatever you find convenient.

Set the processor numbers in the boot-line parameters that you pass to the boot image. You can burn these parameters into ROM, set them in the processor's NVRAM (if available), or enter them manually.

2. The backplane is a type of bus. In this document, the terms "backplane" and "bus" are used interchangeably.



NOTE: You can set up two shared memory networks on a single backplane with the **smEnd** driver, with a single processor acting as a node on each of the two networks. However, if you use VxMP, you can set up only one shared memory network over the backplane. In this case, the processor number of the master node is zero.

Shared-Memory Network Master

One processor on the backplane is the *shared-memory network master*. The shared-memory network master has the following responsibilities:

- Initialize the shared-memory region and the *shared-memory anchor*.
- Maintain the *shared-memory heartbeat*.
- Function (usually) as the gateway to the external network.
- Allocate the shared-memory region (on some boards the master statically reserves the shared-memory region; on others it allocates this region from the kernel heap).

No processor can use the shared-memory network until the shared-memory network master initializes it. However, the master processor is *not* involved in the actual transmission of packets on the backplane between other processors. After the shared-memory network master initializes the shared-memory region, all of the processors, including the master, are peers.

Set the processor number of the master with the **shared memory master CPU number (SM_MASTER)** build configuration parameter. A node that knows the master node's processor number can determine at run time whether it is the master node by comparing this processor number with the one that you assigned to the node in the boot parameters.



NOTE: You configure the maximum number of processors at build time with the **max # of cpus for shared network (SM_CPUS_MAX)** configuration parameter. The largest processor number that you can use is one less than this total maximum processor count.

Typically, the master boots from the external network directly. The master has two Internet addresses in the system: its Internet address on the external network, and its address on the shared-memory network. (See the reference entry for **usrConfig**.)

The other processors on the backplane can boot indirectly over the shared-memory network, using the master as the gateway. They need only have an Internet address on the shared-memory network. These processors specify the shared-memory network interface, **sm**, as the boot device in the boot parameters.

For more information and an example, see [4.4 Shared-Memory Network Configuration](#), p.50.

Shared-Memory Anchor

The location of the shared-memory region depends on the system configuration. All processors on the shared-memory network must be able to access the shared-memory region within the same bus address space as the anchor.

The shared-memory anchor serves as a common point of reference for all processors. You may place the anchor structure and the shared-memory region in the dual-ported memory of one of the participating boards (the master by default) or in the memory of a separate memory board.³



NOTE: Some BSPs allow you to put the anchor and shared-memory regions on a participating non-master board.

The anchor contains an offset to the actual shared-memory region. The master sets this value during initialization. The offset is relative to the anchor itself. Thus, the anchor and pool must be in the same address space so that the offset is valid for all processors.

Set the anchor bus address by setting configuration parameters or by setting boot parameters. For the shared-memory network master, you assign the anchor bus address in the master's configuration at the time you build the system image.

Set the shared-memory anchor bus address, *as seen by the master*, during configuration with the configuration parameter **SM_ANCHOR_ADRS**.

For the other processors on the shared-memory network, you can assign a default anchor bus address during configuration in the same way. However, this requires that you burn boot ROMs with that configuration, because the other processors must, at first, boot from the shared-memory network. For this reason, you can specify the anchor bus address in the boot parameters if the shared-memory backplane network interface is the boot device.

The format of the boot line is *bootDeviceName=localAddress*. For example:

```
sm=0x10010000
```

This is the address of the anchor as seen by the processor you are booting.

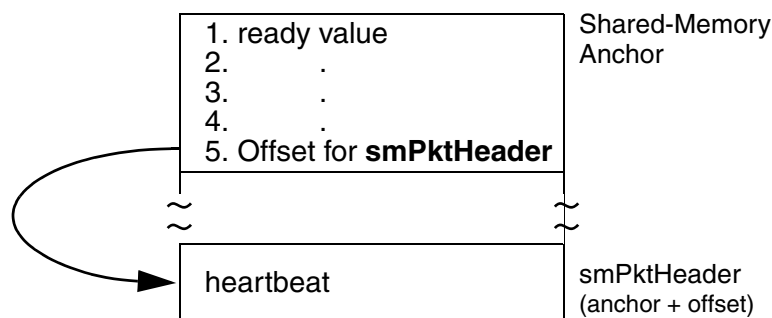
Shared-Memory Heartbeat

The processors on the shared-memory network cannot communicate over that network until the shared-memory network master finishes initializing the shared-memory region. To let the other processors know when the backplane is "alive," the master maintains a *shared-memory heartbeat*. This heartbeat is a counter that the master increments once per second. Processors on the shared-memory network determine that the shared-memory network is alive by watching the heartbeat for a few seconds.

The shared-memory heartbeat is located in the first 4-byte word of the shared-memory packet header. The offset of the shared-memory packet header is the fifth 4-byte word in the anchor, as shown in [Figure 4-4](#).

3. This is board-specific and handled by the board designer. For shared memory residing on the master, VxWorks has definitions for the anchor location. These are architecture-specific.

Figure 4-4 Shared-Memory Heartbeat



Shared-Memory Location

The compiler puts the shared-memory region in a fixed location with a fixed size. You set this location (which is BSP-specific) with the `SM_MEM_ADRS` parameter in the `INCLUDE_SM_COMMON` component for that board.⁴ Because all processors on the backplane access the shared-memory region, you must configure that memory as non-cacheable or use a cache coherency mechanism.

The shared-memory region (not including the anchor) can also be allocated at run time if you set `SM_MEM_ADRS` to `NONE`. In this case, a region of size `SM_MEM_SIZE` is allocated and made non-cacheable. If used this way, be wary that shared memory is allocated from the kernel heap. Thus, the whole heap must be mapped on the backplane.

Shared-Memory Size

Set the size of the shared-memory network area by setting the build configuration parameter `SM_MEM_SIZE`. A related area, the shared-memory object area, used by VxMP, is governed by the configuration parameter `SM_OBJ_MEM_SIZE`.

The size you will need for the shared-memory network area depends on the number of processors and on how much traffic you expect. There is less than 2KB of overhead for data structures. After that, the shared-memory network area is divided into 2KB packets. Thus, the maximum number of packets available on the backplane network is $(areaSize - 2KB) / 2KB$. A reasonable minimum is 64KB. A configuration with a large number of processors on one backplane and many simultaneous connections can require as much as 512KB. Reserving a backplane network memory area that is too small will slow network communication.

Test-and-Set to Shared Memory

To prevent more than one processor from simultaneously accessing certain critical data structures of the shared-memory region, the `smEnd` driver uses an indivisible test-and-set (TAS) instruction to obtain exclusive use of a shared-memory data structure. This translates into a *read-modify-write* (RMW) cycle on the backplane bus.⁵

4. The BSP does provide default values, which are correct.



NOTE: The shared-memory network driver does not support the specification of TAS operation size. This size is architecture dependent.

The selected shared memory must support the RMW cycle on the bus and must guarantee the indivisibility of such cycles. This can be problematic if the memory is dual-ported (that is, it resides on the master and can be accessed there as local RAM, while existing also as shared memory on the bus accessible by the slave boards), as the memory must then lock out one port during a RMW cycle on the other.

Some processors do not support RMW indivisibly in hardware, but do have software hooks to provide the capability. For example, some processor boards have a flag that you can set to prevent the board from releasing the backplane bus, after you acquire the flag, until you clear that flag. You can implement this test-and-set technique for a processor in the **sysBusTas()** routine of the system-dependent library **sysLib**. The **smEnd** driver calls this routine to set up mutual exclusion on shared-memory data structures.



CAUTION: Configure the shared-memory test-and-set type (configuration parameter: **SM_TAS_TYPE**) to either **SM_TAS_SOFT** or **SM_TAS_HARD**. If even one processor on the backplane lacks hardware test-and-set, you must configure *all* processors in the backplane to use software test-and-set (**SM_TAS_SOFT**).

4.3.2 Interprocessor Interrupts

Each processor on the backplane has a single *input queue* for packets that it receives from other processors. To attend to its input queue, a processor can either poll or rely on interrupts (either bus interrupts or mailbox interrupts). When polling, the processor examines its input queue at fixed intervals. When using interrupts, the sending processor notifies the receiving processor that the receiving processor's input queue contains packets.

Processors that communicate by means of either bus interrupts or mailbox interrupts are more efficient than those that use polling because they invest fewer cycles in communication (although at a cost of greater latency). Unfortunately, the bus interrupt mechanism can handle only as many processors as there are interrupt lines available on the backplane (for example, VMEbus has seven). In addition, not all processor boards are capable of generating bus interrupts.

As an alternative to bus interrupts, you can use *mailbox interrupts*, also called *location monitors* because they monitor the access to specific memory locations. A mailbox interrupt specifies a bus address that, when a processor writes to it or reads from it, causes a specific interrupt on the processor board. You can set hardware jumpers or software registers to set each board to use a different address for its mailbox interrupt.

To generate a mailbox interrupt, a processor accesses the specified mailbox address and performs a configurable read or write of a configurable size. Because each interrupt requires only a single address on the bus, there is effectively no limit on the number of processors that can use mailbox interrupts. Most modern processor boards include some kind of mailbox interrupt.

5. Or a close approximation to it. Some hardware cannot generate RMW cycles on the VME bus and the PCI bus does not support them at all.

Each processor must tell the other processors which notification method it uses. Each processor enters its *interrupt type* and up to three related parameters in the shared-memory data structures. The shared-memory network drivers of the other processors use this information when sending packets.

Specify the interrupt type and parameters for each processor by setting the build configuration parameters **SM_INT_TYPE** and **SM_INT_ARG n** . The possible values are defined in the header file **smLib.h**. [Table 4-4](#) summarizes the available interrupt types and parameters.

Table 4-4 **Backplane Interrupt Types**

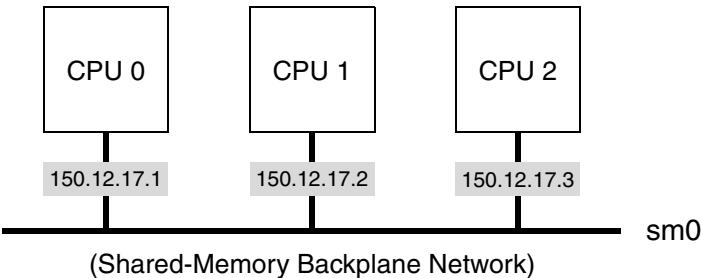
Type	Arg 1	Arg 2	Arg 3	Description
SM_INT_NONE	-	-	-	Polling
SM_INT_BUS	level	vector	-	Bus interrupt
SM_INT_MAILBOX_1	address space	address	value	1-byte write mailbox
SM_INT_MAILBOX_2	address space	address	value	2-byte write mailbox
SM_INT_MAILBOX_4	address space	address	value	4-byte write mailbox
SM_INT_MAILBOX_R1	address space	address	-	1-byte read mailbox
SM_INT_MAILBOX_R2	address space	address	-	2-byte read mailbox
SM_INT_MAILBOX_R4	address space	address	-	4-byte read mailbox
SM_INT_USER_1	user defined	user defined	user defined	first user-defined method
SM_INT_USER_2	user defined	user defined	user defined	second user-defined method

4.3.3 Sequential Addressing

Sequential addressing is a method for a target to assign its own IP address based on its processor number. Target processors assign their IP addresses in ascending order, with the master having the lowest address, as shown in [Figure 4-5](#).

Using sequential addressing, a target on the shared-memory network can determine its own IP address. You need specify only the master's IP address. All other processors on the backplane determine their IP address by adding their processor number to the master's IP address.

Figure 4-5 Sequential Addressing



Sequential addressing makes it easier for you to configure your network. When you explicitly assign an IP address to the master processor, you implicitly assign IP addresses to other processors. This makes it easier for you to set up the boot parameters. Thus, when you set up a shared-memory network with sequential addressing, choose a block of IP addresses and assign the lowest address in this block to the master.

When the master initializes the shared-memory network driver, the master passes in its IP address as a parameter. The shared-memory backplane network stores this information in the shared-memory region. If you specify any other address in the **inet on backplane (b)** boot parameter, the specified address overrides the sequential address.

To determine the starting IP address for an active shared-memory network, use **smNetShow()**.

In the following example, the master's IP address is 150.12.17.1.

```
[vxKernel] -> smNetShow
```

The following output displays on the standard output device:

```
Anchor Local Addr: 0x4100, Hard TAS
Sequential addressing enabled.
Master IP address: 150.12.17.1   Local IP address: 150.12.17.2


heartbeat = 56, header at 0xe0025c, free pkts = 57.

cpu int type      arg1          arg2          arg3          queued pkts
-----
0  mbox-1         0xd 0xfb000000 0x80          0
1  mbox-1         0xd 0xfb001000 0x80          2

          PACKETS          ERRORS
      Unicast      Brdcast
      Input   Output   Input   Output   +   Input   Output
=====
      26      27      2       2   |   0       1
value = 0 = 0x0
[vxKernel] ->
```

With sequential addressing, when booting a slave, the backplane IP address and gateway IP boot parameters are not necessary. The default gateway address is the address of the master. You may specify another address if this is not the desired configuration.

```
[vxWorks Boot] : p
boot device      : sm=0x800000
processor number : 1
file name       : /folk/fred/wind/target/config/bspName/vxWorks
host inet (h)   : 150.12.1.159
```

```
user (u)          : moose
flags (f)         : 0x0
[vxWorks Boot]    : 

boot device       : sm
unit number      : 0
processor number  : 1
host name        : host
file name        : /folk/fred/wind/target/config/bspName/vxWorks
inet on backplane (b): 150.12.17.2:ffffff00
host inet (h)     : 150.12.1.159
user (u)         : moose
flags (f)        : 0x0
target name (tn)  : t207-2

Attaching to SM net with memory anchor at 0x10004100...
SM address: 150.12.17.2
Attached TCP/IP interface to esm0.
Gateway inet address: 150.12.17.1
Attaching interface lo0...done
Loading /folk/fred/wind/target/config/bspName/vxWorks/boot.txt

sdm0=/folk/fred/wind/target/config/bspName/vxWorks/vxKernel.sdm
0x000d8ae0 + 0x00018cf0 + 0x00011f70 + (0x0000cc0c) + 0x00000078 + 0x0000015c
```

You enable sequential addressing during configuration. The relevant component is `INCLUDE_SM_SEQ_ADDR`.

4.4 Shared-Memory Network Configuration

For UNIX, configure the host to support a shared-memory network with the same process outlined elsewhere for other types of networks. In particular, a shared-memory backplane network requires the following:

- That you have put all shared-memory network host names and addresses in `/etc/hosts`.
- That you have put all shared-memory network host names in `.rhosts` in your home directory or in `/etc/hosts.equiv` if you are using RSH.
- That you have put an entry in the host's routing table that specifies the master's Internet address on the external network as the gateway to the shared-memory backplane network.
- If you use proxy arp, the master masquerades as the other boards on the SM network, on the ethernet network. That way, there is no need for an entry on the host's routing table that tell to go through the master to reach the slaves. If you do not use proxy arp, the master must act as a gateway. In order for that to work, the host's routing table requires an entry telling the host that to reach the slaves it has to go through the master.
- That you have turned on the IP forwarding parameter on the node functioning as gateway.

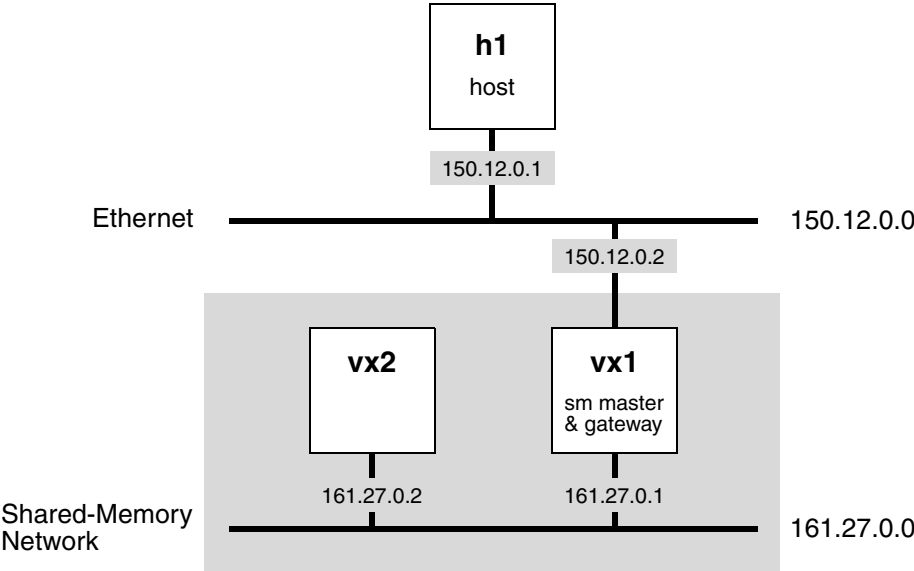
For Windows hosts, the steps required to configure the host are determined by your version of Windows and the networking software you are using. See that documentation for details.

4.4.1 Configuration Example

This section presents an example of a simple shared-memory network. The network contains a single host and two target processors on a single backplane. In addition to the target processors, the backplane includes a separate memory board for the shared-memory region, and an Ethernet controller board. The additional memory board is not essential, but provides a configuration that is easier to describe.

Figure 4-6 illustrates the overall configuration. The Ethernet network is assigned network number 150, subnet 12.0, and the shared-memory backplane network is assigned network number 161, subnet 27.0. The host **h1** is assigned the Internet address 150.12.0.1.

Figure 4-6 Example Shared-Memory Network



The shared-memory master is **vx1**, and functions as the gateway between the Ethernet and shared-memory networks. It therefore has two Internet addresses: 150.12.0.2 on the Ethernet network and 161.27.0.1 on the shared-memory network.

The other backplane processor is **vx2**; it has the shared-memory network address 161.27.0.2. It has no address on the Ethernet because it is not directly connected to that network. However, it can communicate with **h1** over the shared-memory network, using **vx1** as a gateway. All gateway use is handled by the IP layer and is completely transparent to the user. Table 4-5 shows the example address assignments.

Table 4-5 Network Address Assignments

Name	inet on Ethernet	inet on Backplane
h1	150.12.0.1	—
vx1	150.12.0.2	161.27.0.1
vx2	—	161.27.0.2

To configure the UNIX system for our example, you must add the Internet address and name of each system to the `/etc/hosts` file. Note that the backplane master has two entries. The second entry, `vx1.sm`, is not actually necessary because the host system never accesses that system with that address, but you should include it in the file to ensure that some other device does not use the address.⁶

The entries in `/etc/hosts` are as follows:

```
150.12.0.1    h1
150.12.0.2    vx1
161.27.0.1    vx1.sm
161.27.0.2    vx2
```

To allow remote access from the target systems to the UNIX host, add the names of the target systems to the `.rhosts` file in your home directory (or to the file `/etc/hosts.equiv`):

```
vx1
vx2
```

To inform the UNIX system of the existence of the Ethernet-to-shared-memory network gateway, add the following line to the file `/etc/gateways` before you start the route daemon **routed**.

```
net 161.27.0.0 gateway 150.12.0.2 metric 1 passive
```

Alternatively, you can add the route manually (effective until the next reboot) with the following UNIX command:

```
% route add net 161.27.0.0 150.12.0.2 1
```

To prepare a run-time image for `vx1`, the backplane master shown in [Figure 4-6](#), include the following configuration components:

- **INCLUDE_SM_NET** – includes the shared memory network
- **INCLUDE_SM_COMMON** – includes configuration parameters common to memory sharing utilities
- **INCLUDE_SM_NET_SHOW** – includes the `smNetShow()` routine

Within these components, you can set the parameters as shown in [Table 4-6](#).

Table 4-6 Shared-Memory Build Parameters

Workbench Description and Parameter Name	Default Value & Data Type
is the shared memory off board? SM_OFF_BOARD	FALSE BOOL
Shared memory is on a separate board.	
shared memory anchor offset from start of phys memory SM_ANCHOR_OFFSET	0x600 uint
You may define the shared-memory anchor address may relative to this value.	

6. For user configuration on UNIX, the file to be modified must be located in the user's home directory or the user must have administrator (root) privileges.

Table 4-6 Shared-Memory Build Parameters (cont'd)

Workbench Description and Parameter Name	Default Value & Data Type
shared memory anchor address SM_ANCHOR_ADRS Address of anchor as seen by local CPU.	((int)sysSmAnchorAdrs) uint
shared memory address, NONE = allocate local memory SM_MEM_ADRS	SM_ANCHOR_ADRS + SM_ANCHOR_SIZE
shared memory size SM_MEM_SIZE Size of the shared-memory network area, in bytes.	0x00020000 - SM_ANCHOR_SIZE uint
shared memory object pool size SM_OBJ_MEM_SIZE Size of the shared-memory object area, in bytes.	0x00010000 uint
shared memory interrupt type SM_INT_TYPE Interrupt targets with 1-byte write mailbox, see Table 4-4 .	SM_INT_BUS uint
shared memory interrupt type – argument 1 SM_INT_ARG1 Mailbox in short I/O space, see Table 4-4 .	sysSmLevel uint
shared memory interrupt type – argument 2 SM_INT_ARG2 Mailbox at: 0xc000 for vx1, 0xc002 for vx2 (see Table 4-4).	((int) BUS_INT uint
shared memory interrupt type – argument 3 SM_INT_ARG3 Write 0 value to mailbox, see Table 4-4 .	0 uint
Shared memory packet size SM_PKTS_SIZE Shared-memory packet size.	0 uint
max period in ticks to wait for master to boot SM_MAX_WAIT Slave nodes wait this long for the master to boot and establish shared memory before trying to use this memory.	3000 uint
shared memory master CPU number SM_MASTER The address of the master board on the backplane (this will always be 0 unless you are using two smEnd devices).	0 uint

Table 4-6 Shared-Memory Build Parameters (cont'd)

Workbench Description and Parameter Name	Default Value & Data Type
max # of cpus for shared network SM_CPUS_MAX Maximum number of CPUs for the shared network.	DEFAULT_CPUS_MAX uint
shared memory test-and-set type SM_TAS_TYPE Either SM_TAS_SOFT or SM_TAS_HARD . If even one processor on the backplane lacks hardware test-and-set, <i>all</i> processors in the backplane must use the software test-and-set (SM_TAS_SOFT).	SM_TAS_HARD

When booting the backplane master, **vx1**, specify boot line parameters such as the following:

```
boot device           : gn
processor number      : 0
host name             : h1
file name             : /usr/wind/target/config/bspName/vxWorks
inet on ethernet (e)  : 150.12.0.2
inet on backplane (b) : 161.27.0.1:ffffff00
host inet (h)         : 150.12.0.1
gateway inet (g)      :
user (u)              : thoreau
ftp password (pw) (blank=use rsh) :
flags (f)             : 0
```



NOTE: To determine which boot device to use, see the BSP documentation.

The other target, **vx2**, would use the following boot parameters:⁷

```
boot device           : sm
processor number      : 1
host name             : h1
file name             : /usr/wind/target/config/bspName/vxWorks
inet on ethernet (e)  :
inet on backplane (b) : 161.27.0.2
host inet (h)         : 150.12.0.1
gateway inet (g)      : 161.27.0.1
user (u)              : thoreau
ftp password (pw) (blank=use rsh)†:
flags (f)             : 0
```

4.4.2 Troubleshooting

Getting a shared-memory network configured for the first time can be tricky. If you have trouble, use the following troubleshooting procedures—taking one step at a time:

7. You do not need to set the parameters **inet on backplane (b)** and **gateway inet (g)** if you have configured your target to use sequential addressing (because the values for these parameters will be established automatically), but you can use these parameters to override the values established automatically through sequential addressing.

1. Boot a single processor in the backplane without any additional memory or processor cards.
2. Power off and add the memory board, if you are using one. Power on and boot the system again. Using the boot ROM commands for display memory (**d**) and modify memory (**m**), verify that you can access the shared memory at the address you expect, with the size you expect.
3. Rebuild the system and manually fill in the **inet on backplane** boot parameter (do not rely on sequential addressing). This initializes the shared-memory network. The following message appears during the reboot:

```
Backplane anchor at anchorAddress...Attaching network interface sm...done.
```

4. After the system boots, display the state of the shared-memory network with the **smNetShow()** routine, as follows:

```
-> smNetShow ["interfaceName"] [, 1]
value = 0 = 0x0
```

The interface parameter is **sm** by default. Normally, **smNetShow()** displays cumulative activity statistics to the standard output device; specifying 1 (one) as the second argument resets the totals to zero.

5. Test the host connection to the shared-memory master by pinging both of its IP addresses from the host. On the host console, type:

```
ping 150.12.0.2
```

This should succeed and produce a message something like:

```
150.12.0.2 is alive
```

Then type:

```
ping 161.27.0.1
```

This should also succeed. If either ping fails, the host is not configured properly, or the shared-memory master has incorrect boot parameters.

6. Power off and add the second processor board. Do not configure the second processor as the system controller board. Power on and stop the second processor from booting by typing any key to the boot ROM program. Boot the first processor as you did before.
7. If you have trouble booting the first processor with the second processor plugged in, you have some hardware conflict. Check that only the first processor board is the system controller. Check that there are no conflicts between the memory addresses of the various boards.
8. On the second processor's console, use the **d** and **m** boot ROM commands to verify that you can see the shared memory from the second processor. This is either the memory of the separate memory board (if you are using the off-board configuration) or the dual-ported memory of the first processor (if you are using the on-board configuration).
9. Use the **d** command on the second processor to look for the two-part shared-memory anchor (bus address space and anchor location within that space). You can also look for the shared-memory heartbeat; see [Shared-Memory Heartbeat](#), p.45.
10. After you have found the anchor from the second processor, enter the boot parameter for the boot device with that two-part anchor bus address:

```
boot device: sm=0x10010000
```

Enter the other boot parameters and try booting the second processor.

11. If the second processor does not boot, you can use **smNetShow()** on the first processor to see if the second processor is correctly attaching to the shared-memory network. If not, then you have probably specified the anchor bus address incorrectly on the second processor or have a mapping error between the local and backplane buses. If the second processor is attached, then the problem is more likely to be with the gateway or with the host system configuration.
12. You can use host system utilities, such as **arp**, **netstat**, and **ping**, to study the state of the network from the host side.
13. If all else fails, call your technical support organization.

5

Integrating a New Network Device Driver

5.1	Introduction	57
5.2	MUX Programming Interface with Network Drivers	63
5.3	Loading and Unloading Device Instances	64
5.4	Driver Implementations of the NET_FUNCS Interface	68
5.5	Driver Implementation	90
5.6	How VxWorks Launches and Uses Your Driver	92
5.7	Driver Interface with the MUX	96
5.8	MUX Routines for Network Drivers	97
5.9	Queueing Work to the Network Job Queues	100
5.10	Collecting and Reporting Packet Statistics	102

5.1 Introduction

As described in [3.2 Overview of the MUX](#), p.23, the MUX interface insulates network services from the particulars of network interface drivers, and vice versa. The MUX provides the following support to the VxWorks system:

- The MUX maintains and manages a collection of network device instances. The MUX provides routines to load a device into the MUX, to unload it from the MUX, to show the currently loaded devices, or to obtain a list of the loaded devices programmatically.¹
- For each network device that the MUX knows about, the MUX maintains a list of network services (frequently also called 'protocols') bound to the device. These are the services to which the MUX may deliver packets received by the network device, according to the network type of the received packet and the manner and order in which each service bound to the device.

1. VxBus MAC drivers take responsibility for loading and their device instances into the MUX, and unloading the devices when the driver is unregistered or the device instance is released by the driver.

- The MUX also maintains a chain of services bound as 'output protocols' to each device; such services may receive (and possibly filter) packets sent to the device for transmission. Use of such output protocols is not common.
- The MUX notifies the services bound to a device of certain events relating to that device. The device driver often initiates these notifications when it detects an event of interest for one of its device instances, but in some other cases the notifications result from an action of one of the bound services, or of a system administrator.
- The MUX provides APIs for managing and sending packets to its individual network devices. Usually these APIs are called by protocols bound to the device, but in some cases they may be called by other software. The MUX implements these routines with help from the network driver managing the device instance.
- The MUX provides some routines called by network device drivers for devices loaded in the MUX. These include routines to deliver received packets to the appropriate bound services, to notify such services of various events, to request restart of packet transmission after a transmit stall, as well as various utility functions.

This chapter provides an overview of the programming interfaces supported by (and required by) the MUX, for network device drivers, network services, and other software. It does not describe, in any detail, how to write either network services or network device drivers, apart from their interaction with the MUX. The *VxWorks Device Driver Developer's Guide* provides the most complete available documentation on writing VxBus network device drivers for VxWorks.

5.1.1 MUX Network Driver and Network Service Styles

The Wind River Network Stack supports several driver styles and several protocols styles (or services).

Driver Styles

The Wind River Network Stack supports three types of drivers:

- IPNET-native drivers are those that use the **Ipcom_pkt** packet format, native to the current IPNET network stack. IPNET-native drivers are VxBus-enabled drivers². Wind River recommends that you write new network drivers according to this model.
- **M_BLK**-oriented VxBus drivers. These drivers support VxBus and use the **M_BLK** tuple packet format native to the **coreip** network stack of VxWorks 6.4 and earlier.
- **M_BLK**-oriented legacy drivers. These drivers also use the **M_BLK** tuple packet format, but do not support VxBus. Support for these drivers is provided for the maintenance of existing legacy drivers only. This driver style should not be used for new development.

2. Although there is no reason why a non-VxBus driver could not be converted to be IPNET-native without also converting it to a VxBus-enabled driver, Wind River strongly recommends that the VxBus driver model be followed for all new drivers.

From the point of view of the MUX, there are only two relevant driver styles (really device styles): IPNET-native and **M_BLK** (whether VxBus or legacy). The MUX determines the style of the driver by calling the **EIOCGSTYLE** **END** ioctl

- An IPNET-native device driver supports this ioctl and returns **END_STYLE_END2** via the ioctl argument.
- An **M_BLK**-oriented driver usually does not support **EIOCGSTYLE**, but if it supports it, the ioctl returns **END_STYLE_END**.

Note that in many places in the source code and in APIs, **end2** or **END2** is a synonym for IPNET-native, while 'END-style' refers to **M_BLK**-oriented drivers. However, both types of devices are generically referred to as END devices, and are represented in the MUX by an **END_OBJ** structure.

Previous versions of the MUX supported a variant of the **M_BLK**-oriented driver, called an NPT driver, with a different interface to the MUX than the original **END**-style driver. VxWorks 6.7 and later versions of VxWorks do not support the NPT driver style.

The *VxWorks Device Driver Developer's Guide* explains how to structure and write VxBus network drivers, using either the IPNET-native or the **M_BLK**-oriented style. It also discusses how to integrate such drivers into VxWorks images.

Protocol Styles

The Wind River Network Stack also supports three styles of network services. These styles are identified by the function that the service (protocol) uses to bind to a network device:

- Protocols that bind with **mux2Bind()**. These protocols use **Ipcom_pkt** packet format; they send packets using **mux2Send()**. The **INCLUDE_MUX2** component contains support for this style. The protocols of the IPNET stack are of this style.
- Protocols that bind with **muxBind()**. These protocols use the **M_BLK** packet format, and generally send packets using **muxSend()**. The **INCLUDE_MUX** component supports this protocol style.
- Protocols that bind with **muxTkBind()**. These protocols use the **M_BLK** packet format, and generally send packets using **muxTkSend()**. The **INCLUDE_MUXTK** component supports this style.

The interface between the MUX and the protocol differs for each style. For instance, the signatures of the callback functions that the protocol registers with the MUX when it binds to a device differ between the styles.

The **mux2Bind()** protocols work natively with IPNET-native network drivers; the **muxBind()** protocols work natively with **M_BLK**-oriented network drivers; and the **muxTkBind()** protocols work natively only with the no-longer supported NPT style drivers. However, wrapper code handles conversions between the differing APIs and packet formats when necessary so that all three styles of protocol binding work with both IPNET-style and **M_BLK**-oriented devices. For more information, see [5.1.3 Wrapper Conversion Components](#), p.61.

5.1.2 Components of the MUX

[Table 5-1](#) lists the components used to support the three protocol styles, and the IPNET-native and M_BLK-oriented drivers.

In most cases, these components are included or excluded from your VxWorks image automatically, based on the network drivers and protocols that you include in your image. However, if you create a new network driver, you may need to record an explicit dependency upon **INCLUDE_END2** (for IPNET-native drivers) or **INCLUDE_END** (for M_BLK drivers) using a **REQUIRES** line in the driver's component description (.cdf file).

Table 5-1 Basic MUX and END Components

Component	Description
INCLUDE_MUX_COMMON	The MUX common support component pulls in the muxCommon module that contains MUX code common to all network service and device styles.
INCLUDE_MUX2	The MUX mux2Bind() service component pulls in the vxmux_mux2 module that provides mux2Bind() and other support for protocols using Ipcom_pkt packets.
INCLUDE_MUX	The MUX muxBind() service component pulls in muxLib and provides support for muxBind() protocols and their APIs.
INCLUDE_MUXTK	The MUX muxTkBind() service component pulls in muxTkLib and provides support for muxTkBind() protocols and their APIs.
INCLUDE_END_COMMON	The common Enhanced Network Device support component pulls the endCommon module, which contains support routines that are common to both IPNET-native and M_BLK-oriented network devices..
INCLUDE_END2	The END2-style interface support component pulls in the vxmux_end2 module that supports for IPNET-style network drivers and devices..
INCLUDE_END	The END-style interface support component pulls in endLib , which provides support routines used by all M_BLK-oriented network devices and drivers.

Table 5-1 Basic MUX and END Components (cont'd)

Component	Description
INCLUDE_END_ETHER_HDR	The M_BLK ethernet/802.3 component provides routines registered by END-style ethernet device drivers and used by some M_BLK -oriented MUX code to parse or build Ethernet II and 802.3 headers.
INCLUDE_END_POLLED_STATS	The END driver polled statistics support component enables polled mode statistics collection for devices managed by network drivers that support it. Note that most gigabit device drivers and IPNET-native drivers fall into this category.

5.1.3 Wrapper Conversion Components

Table 5-2 lists the components that provide support for conversion between **Ipcom_pkt** and **M_BLK** packets, and certain other API translations used with different combinations of network protocol and driver styles.

For the most part, these components are included in the VxWorks image automatically, based upon which network drivers and network services are included. However, for VIP builds, the dependencies bringing the wrapper components into the image are implemented using **INCLUDE_WHEN** directives; for example, **INCLUDE_MUX2_OVER_END** declares:

```
INCLUDE_WHEN INCLUDE_MUX2 INCLUDE_END
```

Such **INCLUDE_WHEN** directives can be overridden by manual exclusion of the component. For the MUX wrapper code, such a manual exclusion would usually be a mistake; if done, the component would need to be manually included again to make the stack function correctly for the relevant combination of protocol and driver style.

Table 5-2 Conversion Components

Component	Description
INCLUDE_MUX2_OVER_END	Provides support for Ipcom_pkt -oriented protocols that bind to M_BLK -oriented devices using mux2Bind() . This component is automatically included when the INCLUDE_MUX2 and INCLUDE_END components are included.
INCLUDE_MUX_OVER_END2	Provides support for M_BLK -oriented protocols that bind to IPNET-native devices using muxBind() . This component is automatically included when the INCLUDE_MUX and INCLUDE_END2 components are included.

Table 5-2 Conversion Components (cont'd)

Component	Description
INCLUDE_MUXTK_OVER_END	Provides support for M_BLK -oriented protocols that bind to END-style devices using muxTkBind() . This component is automatically included when the INCLUDE_MUXTK and INCLUDE_END components are included.
INCLUDE_MUXTK_OVER_END2	Provides support for M_BLK -oriented protocols that bind to IPNET-native devices using muxTkBind() . This component is automatically included when the INCLUDE_MUXTK and INCLUDE_END2 components are included.
INCLUDE_VXMUX_MBLK	Provides some underlying routines used by the various INCLUDE_x_OVER_y components listed in this table to convert between M_BLK and Ipcom_pkt packets.

The modules that contain the wrapper code are listed below:

- The module **vxmux_mux2_over_end.c** contains wrapper code needed for protocols bound to **M_BLK**-oriented devices using **mux2Bind()**.
- The module **vxmux_mux_over_end2.c** contains wrapper code needed for protocols bound to IPNET-native devices using **muxBind()**.
- The module **muxTkOverEnd.c** contains wrapper code needed for protocols bound to **M_BLK**-oriented devices using **muxTkBind()**.
- The module **vxmux_muxtk_over_end2.c** contains wrapper code needed for protocols bound to IPNET-native devices using **muxTkBind()**.

Only the initialization routines in these modules are globally visible. The initialization routines install pointers to the other wrapper functions needed by the MUX. Since the MUX does not access the functions directly; image size can be decreased when not all the wrapper routines are needed in the image. For example, if a production image includes an IPNET-native driver and the IPNET stack, but no **M_BLK**-oriented drivers or services (such as the WDB agent), then all of the wrapper functions, as well as all support for **M_BLK**-oriented drivers and protocols, may be omitted from the image.



NOTE: One limitation applies in the current release; a polled mode send and receive by an **Ipcom_pkt** protocol over an END-style device is not implemented. Polled-mode send and receive are intended to support the WDB agent, which is at present **M_BLK**-oriented, and so does not need the polled **Ipcom_pkt**-oriented routines.



NOTE: The module **muxTkLib.c** contains code to support protocols that bind using **muxTkBind()** and related APIs that such protocols use. The NPT-driver style was native for **muxTkBind()** protocols, and is no longer supported. Protocols that call **muxTkBind()** to bind to **M_BLK**-oriented or IPNET-native devices require additional wrapper support, from **muxTkOverEnd.c** or **ipcom_muxtk_over_end2.c**, respectively.

5.2 MUX Programming Interface with Network Drivers

The programming interface between the MUX and network drivers consists of several parts:

- Every network driver provides a “load” function that either the driver itself (for a VxBus driver) or the stack startup code (for a legacy driver) passes as an argument to the **muxDevLoad()** routine, once for each device instance. **muxDevLoad()** calls the driver's load function twice for each instance, once to obtain the driver name root and once to obtain the **END_OBJ** that represents the device instance to the MUX. **muxDevLoad()** completes initialization of the **END_OBJ** structure and loads the device into the MUX.
- The **END_OBJ** representing a device contains a pointer to a table of functions that the driver provides for use by the MUX. For an **M_BLK**-oriented driver, this table is a **NET_FUNCS** structure. For an IPNET-native driver, it is a slightly larger **END2_NET_FUNCS** structure containing a **NET_FUNCS** substructure as its first member. The **END_OBJ**, **NET_FUNCS**, and **END2_NET_FUNCS** structures are declared in the header file:

installDir/vxworks-6.x/target/h/end.h.

For some of these function pointers, the actual function calling signature depends upon the driver style.

- **muxDevLoad()** installs certain other function pointers directly in the **END_OBJ** structure. The most visible of these is the **receiveRtn** member; the driver calls this function pointer (using the macro **END_RCV_RTN_CALL()** or **END2_RCV_RTN_CALL()** provided by **endLib.h**) in order to deliver a received packet to the MUX, to be ultimately passed to the appropriate protocol bound to the device instance that received the packet. The receive routine must be called only in the context of the device instance's job queue task. The calling signature of the receive routine depends on the driver style.
- The MUX provides a few functions specifically for use by network drivers. These include **muxTxRestart()**, **muxError()**, and the utility routines **muxLinkUpNotify()** and **muxLinkDownNotify()**.
- The MUX provides additional functions that while not reserved only to device drivers, may in some cases be called by device drivers. For instance, VxBus network drivers call **muxDevLoad()** and **muxDevUnload()** for their own device instances.³ VxBus network drivers also call **muxDevStart()** and

3. Only the VxBus driver should call **muxDevLoad()** or **muxDevUnload()** for a VxBus network device instance.

muxDevStop() to start and stop device instances (enabling or disabling reception and transmission). It is also possible for management code to call **muxDevStart()** or **muxDevStop()** for a network device.

- Various libraries (**endCommon**, **endLib**, **vxmux_end2**, **endEtherHdr**, **etherMultiLib**, and so on) provide APIs used by network drivers to help them implement the functionality required by the MUX.
- The **jobQueueLib** APIs provide a high-performance mechanism for 'posting' work to a specific job queue, to be executed by a specific task. While not strictly part of the MUX, these APIs are often associated with the MUX. They are used by the network stack, by the MUX, and by network drivers, and are available to other software as well. In particular, network driver interrupt service routines post jobs to the network device's job queue in order to defer all time consuming work to task level and keep the ISR's execution time as short as possible.
- The MUX provides other functions that are intended primarily for network services, such as **mux2Send()**, **muxIoctl()**, and so on. A driver is not forbidden from calling these, but it would rarely need to.

Of the MUX routines that are expected to be called by network drivers, those that access the list of protocols bound to a device instance must be called only as a job on the network job queue associated with the network device instance. Due to the mutual exclusion architecture of the IPNET stack in this release, this job queue must be the one executed by the **tNet0** task, if IPNET is bound to the network device.⁴ The functions that must execute as network jobs on the device job queue include the MUX-installed receive routine, **muxTxRestart()**, **muxError()**, **muxLinkUpNotify()**, and **muxLinkDownNotify()**.

5.3 Loading and Unloading Device Instances

The routine **muxDevLoad()** adds a network device instance to the MUX.

```
IMPORT DEV_COOKIE muxDevLoad
(
    int unit,
    END_OBJ* (*endLoad) (char *, void*),
    char *initString,
    BOOL loaning,
    void* pBSP
);
```

The *unit* parameter is the unit number of the device instance; for example, the second device instance managed by the **gei** driver would be unit number 2, also known as **gei2**. The *endLoad* parameter points to a function provided by the device driver. The **muxDevLoad()** routine calls this function twice during the execution of **muxDevLoad()**. The first call obtains the driver name (for example, **gei**) from the driver. On this call, the first argument to **endLoad** points to a buffer of length **END_NAME_MAX** whose first byte is zero; the driver's **endLoad()** routine copies the driver name into the buffer, and returns NULL.

4. Then network job queue executed by **tNet0** has job queue identifier **netJobQueueId**; this should be the default network job queue chosen by any network driver.

The **muxDevLoad()** routine calls **endLoad()** a second time with its second argument equal to the *pBSP* argument to **muxDevLoad()**, and its first argument pointing to a non-empty string that starts with the decimal representation of the unit number, followed by a colon, followed by a copy of the *initString* string passed to **muxDevLoad()**. The *initString* copy, prefixed with the unit number string and colon, must fit in a buffer of size **END_INIT_STR_MAX** (255). This second call is where the driver does most of the work to allocate (for non-VxBus drivers) and partially initialize an **END_OBJ** structure, which invariably occurs as the first member of the driver-specific control structure for the device instance. The return value from the second **endLoad()** call is a pointer to the partially initialized **END_OBJ** structure. The **muxDevLoad()** routine returns this pointer as a “device cookie” after it completes initializing the **END_OBJ**. As part of the initialization process, **muxDevLoad()** calls the **EIOCGSTYLE** **END** ioctl function to determine the driver style (**M_BLK**-oriented or **IPNET**-native). It sets various function pointers in the **END_OBJ**, notably the **receiveRtn** member, but also others used internally by the **MUX**, according to the results.

5.3.1 VxBus vs. Non-VxBus Drivers

The **muxDevLoad()** routine is called by different code for VxBus network drivers than for non-VxBus drivers. A VxBus driver allocates the control structure for one of its device instances in its **devInstanceInit2** function. The VxBus system code calls that routine for each of the driver's device instances discovered dynamically by the VxBus system, or configured explicitly in the BSP's **hwconf.c** file. The control structure contains the **END_OBJ** structure, so far uninitialized. Later, the stack startup code executes the **muxDevConnect** method (for **M_BLK**-oriented drivers) or **mux2DevConnect** method (for **IPNET**-native drivers) for all device instances for which the driver provide one of these methods. Either of these methods will call **muxDevLoad()**, followed by **muxDevStart()** if the **muxDevLoad()** call was successful. VxBus drivers typically pass **NULL** as the *initString* argument to **muxDevLoad()**, and pass the VxBus device ID for the instance as the *pBSP* argument. That gives the driver all the information it needs to identify the device instance and find its control structure.

A legacy (non-VxBus) driver does not call **muxDevLoad()** itself. Instead, it provides its load routine as a global symbol. Entries for each device instance are added to the **endDevTbl[]** array in the BSP's **configNet.h** file, either statically at build time by the system designer, or dynamically at run time by BSP-specific bus-scanning code. Each of these entries includes the device unit number, a pointer to the driver's load routine (or sometimes, to a BSP-specific wrapper function for the driver load routine), a device initialization string to be passed as the *initString* argument to **muxDevLoad()**, a BSP-specific value to be passed as the *pBSP* argument, an (unused) flag indicating whether the device 'loans' packets to the **MUX**, and a flag to indicate whether the entry has already been processed. The network start-up code iterates through the **endDevTbl[]** array and calls the **muxDevLoad()** function for each element in the array that has not already been processed, and marks the element as done as it processes it.

A VxBus network driver divides its instance initialization work between its **devInstanceInit2()** routine and its **xLoad()** routine. The division is somewhat arbitrary, but basically anything that depends upon **endLib** or the **MUX** may be held off to the **xLoad()** routine. A non-VxBus driver has no **devInstanceInit2()** routine, and so all the work needs to be done in its **xLoad()** routine. The following

work is done in either the **devInstanceInit2()** routine, or in the **xLoad()** routine's second pass:

1. Allocate, zero out, and then populate the drivers's control structure for the device instance.
2. For IPNET-native ethernet compatible drivers only, initialize the *formAddress*, *packetDataGet*, and *addrGet* **NET_FUNCS** members with the values of **_func_endEtherAddressForm**, **_func_endEtherPacketDataGet**, and **_func_endEtherPacketAddrGet()**.⁵
3. For IPNET-native drivers only, initialize the *hdrParse* and *formLinkHdr* members of the **END_OBJ** structure. IPNET-native ethernet compatible drivers may set these members to the **end2EtherHdrParse()** and **end2EtherIIFormLinkHdr()** functions.
4. Call **END_OBJ_INIT()** to initialize the **END_OBJ** core. Among other things, this sets the driver description, and the pointer to the driver's **NET_FUNCS** table.
5. Parse and process the initialization string (non-VxBus drivers only).
6. Identify and reset the device, putting it into a quiescent state.
7. Initialize any necessary private structures.
8. Determine the device's MAC address. This is done in a driver-specific way, and sometimes in a BSP-specific way.
9. Allocate memory for transmit and receive descriptors. Generic VxBus drivers may call **vxbDmaBufLib** support routines to create the necessary tags and maps.
10. For **M_BLK**-oriented drivers only, allocate a network buffer tuple pool. The pool has tuples large enough to hold the maximum size frame the driver supports, and the driver uses it primarily for receiving frames, but may also use it to coalesce fragmented transmits. The best way to create such a pool is to call the **endLib** utilities **endPoolCreate()** or **endPoolJumboCreate()** (for jumbo frames). These in turn call **netPoolCreate()** using the **linkBufPool** back end. IPNET-native drivers should merely set the **pNetPool** member of the **END_OBJ** structure to the value **_end2_linkBufPool**.
11. Call **endM2Init()** to initialize MIB interface statistics structures.
12. Initialize the driver's hardware offload capabilities structure (if the driver and hardware support such capabilities).
13. Initialize polled statistics structures (if the driver supports this). IPNET-native drivers should support polled-mode statistics collection.

5.3.2 Unloading a Device from the MUX

While it is not common to do so, it is sometimes possible to unload a network device from the MUX using the **muxDevUnload()** routine:

```
STATUS muxDevUnload  
(
```

-
5. IPNET-native drivers do not refer to the **etherAddressForm()**, **endEtherPacketDataGet()**, or **endEtherPacketAddrGet()** functions directly, to avoid requiring these functions when no **M_BLK**-oriented protocols that need them are included in the image.

```
char* pName,  
int unit  
);
```

When **muxDevUnload()** is called, it checks if the specified device is still up. If so, **muxDevUnload()** calls the driver's **stop()** routine for the device. Next, it calls the shutdown routines for each service that bound to the device; the service shutdown routine must in turn call **muxUnbind()** to unbind itself from the device. **muxDevUnload()** pends until all services bound to the device have unbound from it, and any further references to the device acquired using the **muxDevAcquire()** have also been released. After all references to the device have been released, **muxDevUnload()** continues and calls the driver's **unload()** routine.

The MUX calls your driver's **xUnload()** when a system application calls **muxDevUnload()**. A VxBus network driver may itself call **muxDevUnload()** in response to a call to its **vxbDrvUnlink()** method, asking it to unlink an instance.

When **muxDevUnload()** is called, it checks if the specified device is still up. If so, **muxDevUnload()** calls the driver's **xStop()** routine for the device. Next, it calls the shutdown routines for each service that bound to the device; the service shutdown routine must in turn call **muxUnbind()** to unbind itself from the device. Finally, **muxDevUnload()** calls the **xUnload()** routine for the device. (See [Figure 5-4](#).)

The driver's **unload()** routine has the signature:

```
LOCAL STATUS xUnload  
(  
    END_OBJ * pEnd  
)
```

(*x* indicates is a driver-specific name prefix).

In its **unload()** routine, your driver is responsible for doing whatever it takes to release all resources associated with the device that were created or allocated during the driver's **xLoad()** routine. (For non-VxBus network drivers, this would include all resources associated with the device. For VxBus network drivers, resources allocated in the **devInstanceInit2()** routine, before the **xLoad()** routine is called, do not need to be freed yet.) These resources may include memory; the device network pool and all its buffers for **M_BLK** drivers; kernel objects such as semaphores associated with the device, and so on.

The driver control structure for the device, which starts with an embedded **END_OBJ**, is a special case:

- If the driver's **xUnload()** routine returns **OK**, **muxDevUnload()** will itself free:

pEnd->devObject.pDevice

which normally points at the **END_OBJ** (**pEnd*) itself. This frees the whole driver control structure, unless the **devObject.pDevice** value has been changed.

- If the driver's **xUnload()** routine returns any other value, **muxDevUnload()** does not attempt to free the **END_OBJ**; that becomes the driver's responsibility. If there is no error condition, but the driver wishes to free the **END_OBJ** itself, the **unload()** routine should return **EALREADY**. Any return value other than **OK** or **EALREADY** indicates an error condition, and an error message will be logged.

- Generally, VxBus network driver **unload()** routines should return **EALREADY**, since the driver control structure is typically needed in the driver's **vxbDrvUnlink()** method after the call to **muxDevUnload()**.

For **M_BLK**-oriented drivers, the **unload()** routine must free the device network buffer pool:

- For pools created with **endPoolCreate()** or **endPoolJumboCreate()**, this is done by calling **endPoolDestroy()**.
- Pools created using **netPoolCreate()** may be freed by calling **netPoolRelease()**. Calling **netPoolRelease()** causes the system to free a pool after the stack releases all network pool resources that it is holding from that pool.
- For any pools that the driver created using **netPoolInit()**, there is no such safe pool release routine, and the driver must ensure that all tuples have been returned to the driver pool before it returns successfully from **unload()**. If it cannot do so, the driver does not properly support unloading the device.
- Wind River recommends that drivers use **endPoolCreate()**, **endPoolJumboCreate()**, or **netPoolCreate()**, instead of **netPoolInit()**, to create driver memory pools.



NOTE: IPNET-native drivers allocate their network buffers out of a shared global pool, usually the IPNET packet pool. The **stop()** routine for such a driver has already freed any packet resources that the driver held, so there is no additional work to do in this regard in the unload routine.

The driver's **unload()** function is a device-specific call. If the driver has any resources that it shares among all of its device instances, it must not free these shared resources until the MUX calls the driver's **unload()** routine for each of these devices.

Note that VxBus drivers expect **muxDevUnload()** to be called for any of the driver's device instances only from the driver's **{vxbDrvUnlink}** method. **muxDevUnload()** should not be called from other code for a VxBus network device, or instability may result.

For **muxDevUnload()** to work as expected, any network services bound to the device in the MUX must either have registered a working shutdown routine, or must be manually unbound from the device before calling **muxDevUnload()**. In particular, since the WDB agent does not currently support unbinding from an END device, it is not possible to unload a device to which the WDB agent is bound.

5.4 Driver Implementations of the NET_FUNCS Interface

Table 5-3 lists the entry points of the **NET_FUNCS** interface that drivers implement and expose to the MUX. These routines are driver-specific and are prefixed with a driver identifier, such as "In7990" for the Lance Ethernet driver.

Table 5-3 NET_FUNCS Interface Routines

Routine	Description
<i>xStart</i> ()	Enable reception and transmission for the device.
<i>xStop</i> ()	Deactivate the network device.
<i>xUnload</i> ()	Release a device, or a port on a device, from the MUX. For information on unload (), see 5.3 Loading and Unloading Device Instances , p.64.
<i>xIoctl</i> ()	Support various ioctl commands.
<i>xSend</i> ()	Accept data from the MUX and send it on to the physical layer.
<i>xMCastAddrAdd</i> ()	Add a multicast address to the list of those registered for the device.
<i>xMCastAddrDel</i> ()	Remove a multicast address from those registered for the device.
<i>xMCastAddrGet</i> ()	Retrieve a list of multicast addresses registered for a device.
<i>xPollSend</i> ()	Send frames in polled mode rather than interrupt-driven mode. (Poll mode should only be used for debugging.)
<i>xPollRcv</i> ()	Receive frames in polled mode rather than interrupt-driven mode. Poll mode should only be used for debugging. For details, see 5.6.3 Polled Mode—for Debugging Only , p.96.
<i>xFormAddress</i> ()	Add addressing information to a packet.
<i>xPacketDataGet</i> ()	Separate the addressing information and data in a packet.
<i>xAddrGet</i> ()	Extract the addressing information from a packet.
<i>xEndBind</i> ()	Exchange data with the network service at bind time. (Optional)

IPNET-native drivers use a slightly larger function table, an **END2_NET_FUNCS** structure that contains an **END2_NET_FUNCS** structure as its first element. However, the MUX calls the **send**(), **pollSend**(), and **pollRecv**() functions that an IPNET-native driver stores in its **END_FUNCS** substructure with different arguments than suggested by the explicit type of these members in the **NET_FUNCS** structure, passing **Ipcom_pkt*** arguments instead of **M_BLK_ID** arguments. **END2_NET_FUNCS** provides also (at present) one additional member after the **END_FUNCS** substructure, shown in [Table 5-4](#).

Table 5-4 Additional functions for IPNET-native drivers

Routine	Description
llhiComplete	Parse a link header in an Ipcom_pkt packet into an LL_HDR_INFO structure for benefit of network services that bind using muxBind (). May be NULL for ethernet drivers.

5.4.1 xStart()

The driver's **Start()** routine is called by **muxDevStart()**. It does whatever is necessary to make the device instance active and available.

1. Register your device driver's ISR by calling **sysIntConnect()**, if this has not been done earlier in the load routine or in **devInstanceInit2()**.
2. VxBus network drivers should reread parameters that might be expected to change between device stops and starts, for instance the parameter specifying the network job queue to which the driver posts work for the device.
3. Configure the device for packet reception and transmission.
4. Populate the receive DMA descriptor ring and the parallel ring of buffer pointers (which might be **M_BLK_IDs** or **Ipcom_pkt** pointers or raw buffer pointers, depending on the driver).
5. Initialize the transmit ring.
6. Enable device interrupts at the board/interrupt controller level as well as at the device-specific level.
7. Appropriately set the device registers that finally enable reception and transmission.
8. Set the PHY to the desired mode.
9. IPNET-native drivers must allocate an **Ipcom_pkt** with attached buffer for use in polled-mode sends, and store a pointer to the **Ipcom_pkt** in the **pollPkt** member of the **END_OBJ** structure.

For VxBus network drivers, the driver's **muxDevConnect()** method calls **muxDevStart()** after calling **muxDevLoad()**; for other network drivers, the start-up code calls **muxDevStart()** after **muxDevLoad()**. In either case, **muxDevStart()** passes **xStart()** the unique interface identifier that the driver's **xLoad()** routine returned.

As with **xLoad()**, the MUX makes this call for each port that it activates within the driver.

An example template for the **xStart()** routine follows:

```
STATUS xStart
(
    END_OBJ * pEnd, /* END object */
)
{
    x_DRV_CTRL * pDrvCtrl = (x_DRV_CTRL *) pEnd;
    /*
     * Some drivers may require additional mutual exclusion beyond the
     * transmit semaphore. If so, be sure to observe proper mutex ordering.
     */

    END_TX_SEM_TAKE (&pDrvCtrl->end, WAIT_FOREVER);

    if ((pDrvCtrl->end.flags & IFF_UP) == 0)
    {
        /*
         * - Reread selected device parameters, such as the job queue ID.
         * - Connect the driver's ISRs, if not already done.
         * - Initialize RX descriptor ring; set each descriptor to point to
         *   a buffer from a tuple from the device network buffer pool.
         * - Initialize TX ring as needed.
         * - Configure the device according to current settings.
         * - Enable device interrupts.
        */
    }
```



```

        * - Enable transmission and reception.
        * - Set desired PHY mode.
        */
        pDrvCtrl->end.flags |= (IP_IFF_UP | IP_IFF_RUNNING);
    }
    END_TX_SEM_GIVE (&pDrvCtrl->end);
    return (OK);
}

```

Write this routine to return **OK**, or **ERROR** in which case it should set **errno** appropriately.

5.4.2 xStop()

The driver's **Stop()** routine is called by **muxDevStop()** or **muxDevStopAll()**. This routine halts a network device, putting it into a quiescent state in which it does not generate interrupts. It also does the following:

1. Waits for any network jobs which may be outstanding for the device to complete, and arranges that more network jobs will not be posted nor will device interrupts be reenabled.
2. Disconnects any driver ISRs that the **xStart()** routine connected.
3. Frees outstanding transmitted packets that have not been returned to the stack, and returns tuples associated with the device's receive ring to the device network buffer pool. For IPNET-native drivers in particular, all buffers allocated for the device from the shared global packet pool (the IPNET stack packet pool, usually) should be returned to the pool, so that they can be used by the stack as well as other IPNET-native devices while the current device is stopped. (This is done so that if the buffer pool is shared between multiple devices, the other devices have access to the buffers while the current device is stopped.) Otherwise, the **xStop()** routine does not release data structures that were allocated in the **xLoad()** routine or in **devInstanceInit2()**.
4. IPNET-native drivers must free the **Ipcom_pkt** pointed to by the **pollPkt** member of the **END_OBJ** structure. This packet was allocated in the driver's **start()** routine, and is used for polled-mode sends.

The MUX passes **xStop()** the **END_OBJ** pointer returned by the driver's **xLoad()** routine. **xStop()** is considered a synchronous routine, that is, it should not return until the device has been fully quiesced.

An **xStop()** template follows:

```

STATUS xStop
(
    END_OBJ * pEND, /* END object */
)
{
    x_DRV_CTRL * pDrvCtrl = (x_DRV_CTRL *) pEnd;

    END_TX_SEM_TAKE (&pDrvCtrl->end, WAIT_FOREVER);
    if (pDrvCtrl->end.flags & IFF_UP)
    {
        pEND->flags &= ~(IFF_UP | IFF_RUNNING);
        /*
         * - Prevent any jobs in progress from reenabling interrupts
         * - Disable device interrupts
         * - Wait for any outstanding jobs to complete; ensure no others are
         *   posted.
         * - Disable packet reception and transmission.
         * - Clean the transmit ring, freeing any packets to the stack.
        */
    }
}

```

```
    * - Clean RX tuple ring, returning tuples to network buffer pool.  
    * - If using the recycle cache, call endMCacheFlush().  
    * - If the driver ISRs were connected in xStart(), disconnect them  
    */  
    }  
    END_TX_SEM_GIVE (&pDrvCtrl->end);  
    return (OK);  
}
```

Write this routine to return **OK**.

5.4.3 xioctl()

Network drivers are not installed into the VxWorks I/O system using **iosDrvInstall()**, and so they do not directly support the **ioctl()** function, which passes an integer “file descriptor” as its first argument. However, the MUX supports an ioctl-like interface, **muxIoctl()**, that a caller can use to request miscellaneous device-specific services from network drivers. The ioctl command codes in this interface begin with “EIOC” and are listed in the file **target/h/endCommon.h**.



NOTE: Wind River assigns MUX ioctl command codes according to the scheme defined in **target/h/sys/ioctl.h**, using the macros **_IOR()**, **_IOW()**, **_IORW()**. The **n** argument to these macros becomes the low-order byte of the command code. Low-order byte values in the range 0-127 are reserved for Wind River use. Choose **n** values in the range 128-255 to avoid possible conflict with internal codes.

Some protocol stacks may translate certain socket ioctl commands into other “EIOC” ioctl codes that they then pass to the MUX. However, the translation need not be one-to-one, and there is not any protocol stack-independent way to call MUX ioctl commands using a socket file descriptor. Applications that need to call MUX ioctls should bind as a service to a network driver, and pass the binding cookie as the first argument to **muxIoctl()**; or failing that, you may use the device cookie returned by **muxDevLoad()** or by **endFindByName()** as a fake binding cookie when calling **muxIoctl()**. Some MUX ioctl calls are handled by the MUX itself, but most are passed down by **muxIoctl()** to the network driver’s **xIoctl()** routine.

Any variety of network driver may need to support MUX ioctl commands, particularly if it is to interface with the existing IP network service sublayer. See [Table 5-5](#) for a list of commonly used ioctl commands.

The MUX no longer supports the NPT driver model. A network driver must return **EINVAL** if it receives the **EIOCGNPT** ioctl command (as any driver should do when it receives a MUX ioctl that it does not understand).

An IPNET-native driver ioctl routine must support the **EIOCGSTYLE** ioctl function, and must store the value **END_STYLE_END2** as an **int** value at the address pointed to by the *data* argument. An **M_BLK** oriented driver may simply omit to support the **EIOCGSTYLE** ioctl, although it could alternatively support it and store the value **END_STYLE_END**.



WARNING: The `muxIoctl()` routine handles the multicast address add, delete, and list-get ioctl commands (`EIOCMULTIADD`, `EIOCMULTIDEL`, `EIOCMULTIGET`) by calling the corresponding `mCastAddrAdd()`, `mCastAddrDel()`, or `mCastAddrGet()` functions from the driver `NET_FUNCS` structure. Do not be tempted to support only the multicast table management ioctls and not the corresponding `NET_FUNCS` functions.

Arguments to `xioctl()`

The MUX passes the `xIoctl()` routine three arguments:

- the `END_OBJ` pointer that the driver returned from `xLoad()`
- the ioctl command being issued (for instance, one from [Table 5-5](#))
- an additional argument, often a pointer to a **data** buffer for additional data given in the command or for data to be returned on completion of the command

While this argument is prototyped as a `caddr_t` (the equivalent of a `char *`), the actual type passed depends upon the particular MUX ioctl command. For the commands `EIOCSADDR`, `EIOCGADDR`, `EIOCMULTIADD`, and `EIOCMULTIDEL` that pass link-layer addresses, the lengths of these addresses must be implicitly known to the driver and to the attached services that call `muxIoctl()`. For Ethernet drivers, the addresses are 6 bytes long.

Example 5-1 Template Example

The following `xIoctl()` template example is modeled after the `geiEndIoctl()` routine of the `gei825xxVxbEnd.c` driver:

```
LOCAL int xIoctl
(
    END_OBJ * pEnd,      /* END Object */
    int      command,    /* ioctl command */
    caddr_t  data        /* holds response from command */
)
{
    MY_DRV_CTRL * pDrvCtrl;
    END_MEDIALIST * mediaList;
    END_CAPABILITIES * hwCaps;
    END_MEDIA * pMedia;
    INT32 value;
    int error = OK

    pDrvCtrl = (MY_DRV_CTRL *) pEnd;
    if (command != EIOCPOLLSTART && command != EIOCPOLLSTOP)
        semTake (pDrvCtrl->xDevSem, WAIT_FOREVER);

    switch (command)
    {
        /*****
        /* IPNET-native drivers must support EIOCGSTYLE */
        case EIOCGSTYLE:
            if (data == NULL)
                error = EINVAL;
            else
                *(int *)data = END_STYLE_END2;
            break;
        *
        *****/
        case EIOCSADDR:
            if (data == NULL)
```

```
        error = EINVAL;
    else
        bcopy ((char *)data, (char *)pDrvCtrl->ethAddr,
                ETHER_ADDR_LEN);
    /* Set the receive configuration so that device receives
       packets destined for the new station address, rather than
       the old one. */
    xEndRxConfig (pDrvCtrl);
    break;

case EIOCGADDR:
    if (data == NULL)
        error = EINVAL;
    else
        bcopy ((char *)pDrvCtrl->ethAddr, (char *)data,
                ETHER_ADDR_LEN);
    break;

case EIOCSFLAGS:
    value = (INT32) data;
    if (value < 0)
    {
        value = ~value;
        END_FLAGS_CLR (pEnd, value);
    }
    else
        END_FLAGS_SET (pEnd, value);
    /* Set receive configuration according to new flags */
    xEndRxConfig (pDrvCtrl);
    break;

case EIOCGFLAGS:
    if (data == NULL)
        error = EINVAL;
    else
        *(long *)data = END_FLAGS_GET(pEnd);
    break;

case EIOCMULTIADD:
    error = xMCastAddrAdd (pEnd, (char *) data);
    break;

case EIOCMULTIDEL:
    error = xMCastAddrDel (pEnd, (char *) data);
    break;

case EIOCMULTIGET:
    error = xMCastAddrGet (pEnd, (MULTI_TABLE *) data);
    break;

case EIOCPOLLSTART:
    pDrvCtrl->polling = TRUE;
    /*
     * Note that this command is called with interrupts locked.
     *
     * - Save the current interrupt mask to be restored when exiting
     *   polled mode.
     * - Disable device interrupts
     * - Empty and clean the transmit ring buffer; either return all
     *   TX packet resources, or save them to be returned when polled
     *   mode is exited. The latter avoids the possibility that
     *   cluster free routines will call functions that shouldn't be
     *   called with interrupts locked.
     */
    break;

case EIOCPOLLSTOP:
    pDrvCtrl->polling = FALSE;

    /*
     * - Reenable device interrupts as they were when polled mode was
     *   entered.
     */
}
```

```

        */

        break;

case EIOCGMIB2233:
case EIOCGMIB2:
    error = endM2Ioctl (&pDrvCtrl->xEndObj, cmd, data);
    break;

case EIOCGPOLLCONF:
    if (data == NULL)
        error = EINVAL;
    else
        *((END_IFDRVCONF **)data) = &pDrvCtrl->xEndStatsConf;
    break;

case EIOCGPOLLSTATS:
    if (data == NULL)
        error = EINVAL;
    else
    {
        /* Retrieve current statistics from the hardware: */
        error = xEndStatsDump(pDrvCtrl);
        if (error == OK)
            *((END_IFCOUNTERS **)data) = &pDrvCtrl->xEndStatsCounters;
    }
    break;

case EIOCGMEDIALIST:
    if (data == NULL)
    {
        error = EINVAL;
        break;
    }
    if (pDrvCtrl->xMediaList->endMediaListLen == 0)
    {
        error = ENOTSUP;
        break;
    }

    mediaList = (END_MEDIALIST *)data;
    if (mediaList->endMediaListLen
        < pDrvCtrl->xMediaList->endMediaListLen)
    {
        mediaList->endMediaListLen =
            pDrvCtrl->xMediaList->endMediaListLen;
        error = ENOSPC;
        break;
    }

    bcopy((char *)pDrvCtrl->xMediaList, (char *)mediaList,
          sizeof(END_MEDIALIST) + (sizeof(UINT32) *
            pDrvCtrl->xMediaList->endMediaListLen));
    break;

case EIOCGIFMEDIA:
    if (data == NULL)
        error = EINVAL;
    else
    {
        pMedia = (END_MEDIA *)data;
        pMedia->endMediaActive = pDrvCtrl->xCurMedia;
        pMedia->endMediaStatus = pDrvCtrl->xCurStatus;
    }
    break;

case EIOCSIFMEDIA:
    if (data == NULL)
        error = EINVAL;
    else
    {
        pMedia = (END_MEDIA *)data;

```

```
        /* Assumes a VxBus driver using miiBus : */
        miiBusModeSet (pDrvCtrl->xMiiBus, pMedia->endMediaActive);
        /* Read new link state, update MAC and MIB state accordingly,
         * send END_ERR_LINKUP or END_ERR_LINKDOWN muxError( ) events
         * if needed; if link comes up, call muxTxRestart( ) : */
        xLinkUpdate (pDrvCtrl->xDev);
        error = OK;
    }
    break;

case EIOCGIFCAP:
    hwCaps = (END_CAPABILITIES *)data;
    if (hwCaps == NULL)
    {
        error = EINVAL;
        break;
    }
    hwCaps->csum_flags_tx = pDrvCtrl->xCaps.csum_flags_tx;
    hwCaps->csum_flags_rx = pDrvCtrl->xCaps.csum_flags_rx;
    hwCaps->cap_available = pDrvCtrl->xCaps.cap_available;
    hwCaps->cap_enabled = pDrvCtrl->xCaps.cap_enabled;
    break;

case EIOCSIFCAP:
    hwCaps = (END_CAPABILITIES *)data;
    if (hwCaps == NULL)
    {
        error = EINVAL;
        break;
    }
    pDrvCtrl->xCaps.cap_enabled = hwCaps->cap_enabled;
    break;

case EIOCGIFMTU:
    if (data == NULL)
        error = EINVAL;
    else
        *(INT32 *)data = pEnd->mib2Tbl.ifMtu;
    break;

case EIOCSIFMTU:
    value = (INT32)data;
    if (value <= 0 || value > pDrvCtrl->xMaxMtu)
    {
        error = EINVAL;
        break;
    }
    pEnd->mib2Tbl.ifMtu = value;
    if (pEnd->pMib2Tbl != NULL)
        pEnd->pMib2Tbl->m2Data.mibIfTbl.ifMtu = value;
    break;

case EIOCGRCVJOBQ:
    if (data == NULL)
    {
        error = EINVAL;
        break;
    }

    qinfo = (END_RCVJOBQ_INFO *)data;
    nQs = qinfo->numRcvJobQs;
    qinfo->numRcvJobQs = 1;
    if (nQs < 1)
        error = ENOSPC;
    else
        qinfo->qIds[0] = pDrvCtrl->xJobQueue;
    break;

default:
    error = EINVAL;
    break;
}
```

```
if (cmd != EIOCPOLLSTART && cmd != EIOCPOLLSTOP)
    semGive (pDrvCtrl->xDevSem);

return (error);
}
```

`xIoctl()` should return **OK** if successful, and an **errno.h**-style error code in case of failure. The routine generally returns **EINVAL** both for unsupported ioctl codes as well as for invalid arguments to a supported ioctl. The routine may occasionally return other particular codes such as **EIO**, **ENOSPC**, **ENOTSUP**, or **ENOBUFS**.

ioctl Commands

Table 5-5 lists the `mutexIoctl()` commands and associated data types:

Table 5-5 MUX ioctl Commands and Data Types

Command	Purpose	Data Type
EIOCGFLAGS	Get device flags.	int *
EIOCSFLAGS	Set device flags. See EIOCSFLAGS , p.78.	int
EIOCGIFCAP / EIOCSIFCAP	Get/set device capabilities.	END_CAPABILITIES *
EIOCGIFMEDIA	Get current PHY “media.” Return the active media mode and link status into the data structure.	END_MEDIA *
EIOCGMEDIALIST	Get supported media list. Return the device’s supported PHY media list into the data structure.	END_MEDIALIST *
EIOCGADDR / EIOCSADDR	Get/set device address. data points to a buffer for the link-layer station address.	char *
EIOCMULTIADD	Add multicast address. data points to a multicast address to add to the multicast list (and enable reception for).	char *
EIOCMULTIDEL	Delete multicast address. data points to a multicast address to remove from the multicast list (and no longer receive).	char *
EIOCMULTIGET	Get multicast list. data is a pointer to MULTI_TABLE * a table that the driver fills with the multicast addresses in its multicast reception list.	
EIOCPOLLSTART	Put device in polled mode.	NULL

Table 5-5 MUX ioctl Commands and Data Types (cont'd)

Command	Purpose	Data Type
EIOCPOLLSTOP	Put device in interrupt mode (exit polled mode).	NULL
EIOCGMTU	Get the link MTU.	INT32 *
EIOCSIFMTU	Set the link MTU.	INT32
EIOCGRCVJOBQ	Get the queue ID of the job queue the device uses to post work.	END_RCVJOBQ_INFO *
EIOCQUERY	Retrieve the bind routine (see EIOCQUERY , p.79).	END_QUERY *
EIOCGHDRLEN	Get the size of the datalink header (if int * this is not supported, you can assume a 14-byte header).	
EIOCGMIB2	Get RFC 1213 MIB information from the driver. muxIoctl() calls the driver's registered END ioctl function to handle this command. The driver's ioctl function in turn calls endM2Ioctl() .	M2_INTERFACETBL *
EIOCGMIB2233	Get RFC 2233 MIB information from the driver. muxIoctl() calls the driver's registered END ioctl function to handle this command. The driver's ioctl function in turn calls endM2Ioctl() .	M2_ID **
EIOCGPOLLCONF	Get statistics polling configuration.	END_IFDRVCONF **
EIOCGPOLLSTATS	Get the current poll statistics counts.	END_IFCOUNTERS **
EIOCGSTYLE	Get driver style. Only IPNET-native drivers need support this command.	int *

EIOCSFLAGS

The **EIOCSFLAGS** MUX ioctl is called by upper layers of the stack to set a small number of device flags (values defined in `installDir/components/ip_net2-6.x/vxcoreip/include/net/if.h`, such as **IFF_PROMISC** or **IFF_ALLMULTI**, that may be administratively controllable. (**IFF_UP** is not directly administratively controllable; a network device should be brought up or down by calling **muxDevStart()** or **muxDevStop()**. Other flags like **IFF_BROADCAST**, **IFF_SIMPLEX**, **IFF_MULTICAST** that refer to general characteristics of the device are likewise not administratively controllable.)

The caller may use the **EIOCSFLAGS** ioctl either to clear or to set bit flags. If the most significant bit of the integer argument to the ioctl is clear, so that the argument appears non-negative as a signed integer, the intent is to set any of the other bits that are on. On the other hand, if the most significant bit in the argument is set, so that the argument appears negative, the intent is to clear bits: specifically, to clear

the bits that are set in the ones-complement of the argument. For example, to clear **IFF_PROMISC**, the argument would be **~IFF_PROMISC**, while to set **IFF_PROMISC** and **IFF_ALLMULTI**, the argument would just be **(IFF_PROMISC | IFF_ALLMULTI)**. The use of the most significant bit to determine whether to set or clear means that you cannot define this bit as a flag with a different purpose.

EIOCQUERY

In the case where **command** is **EIOCQUERY**, **data** points to an **END_QUERY** structure. The caller sets the **query** field of this structure to the type of query (for instance, **END_BIND_QUERY**), and the **queryLen** field to the size of the **queryData** buffer. Upon receipt of an **EIOCQUERY** command, your **xIoctl()** routine should either copy data into this **queryData** buffer, or return an error value such as **EINVAL**.

Your driver is not required to support **EIOCQUERY** queries.

5.4.4 xSend()

The MUX calls the driver's **send()** routine when a network service issues a send request, by calling **muxSend()**, **muxTkSend()**, or **mux2Send()**. The effective signature with which this routine is called differs depending upon the driver style. For **M_BLK** style drivers, the routine has prototype:

```
LOCAL int xSend (END_OBJ * pEnd, M_BLK_ID pMblk);
```

whereas for IPNET-native drivers, the routine has prototype

```
LOCAL int xSend (END_OBJ * pEnd, Ipcom_pkt * pkt);
```

The *pMblk (pkt)* argument points at the lead **M_BLK (Ipcom_pkt)** of a chain of one or more **M_BLKs (Ipcom_pkt's)** describing a complete single packet. The **send()** routine is responsible for sending this data over the device. For detailed information on how to write a driver **send()** routine, see the *VxWorks Device Driver Developer's Guide*.



NOTE: The current release of the Wind River Network Stack's IP stack normally passes only packets consisting of a single **M_BLK** tuple or **Ipcom_pkt**, that is, a single contiguous segment, to the MUX for transmission. However, when zbufs sockets are enabled, the IP stack may pass multi-segment packets to the driver send routine; and other **M_BLK** oriented services may also pass multi-segment packets. So, network drivers must continue to support transmission of packets described by a chain of more than one **M_BLK** tuple or **Ipcom_pkt**.

The return values from this routine also differ between driver styles. The following are the valid return codes:

OK (M_BLK-oriented or IPNET-native)

The normal return when the send succeeds. The driver takes ownership of the packet and arranges to free it when the transmit completes.

END_ERR_BLOCK (M_BLK-oriented)

-IP_ERRNO_EWOULDBLOCK (IPNET-native)

The normal return when the send cannot proceed due to lack of resources (usually lack of space in the TX DMA ring). For this return value only, the caller maintains ownership of the packet. The driver considers the device to have entered a 'TX stalled' state, and arranges to call **muxTxRestart()** at

a later time when more TX resources are available, for instance after transmits in progress complete and space opens up in the TX ring. Currently, drivers should also return this value when the device's link to the media is inactive.

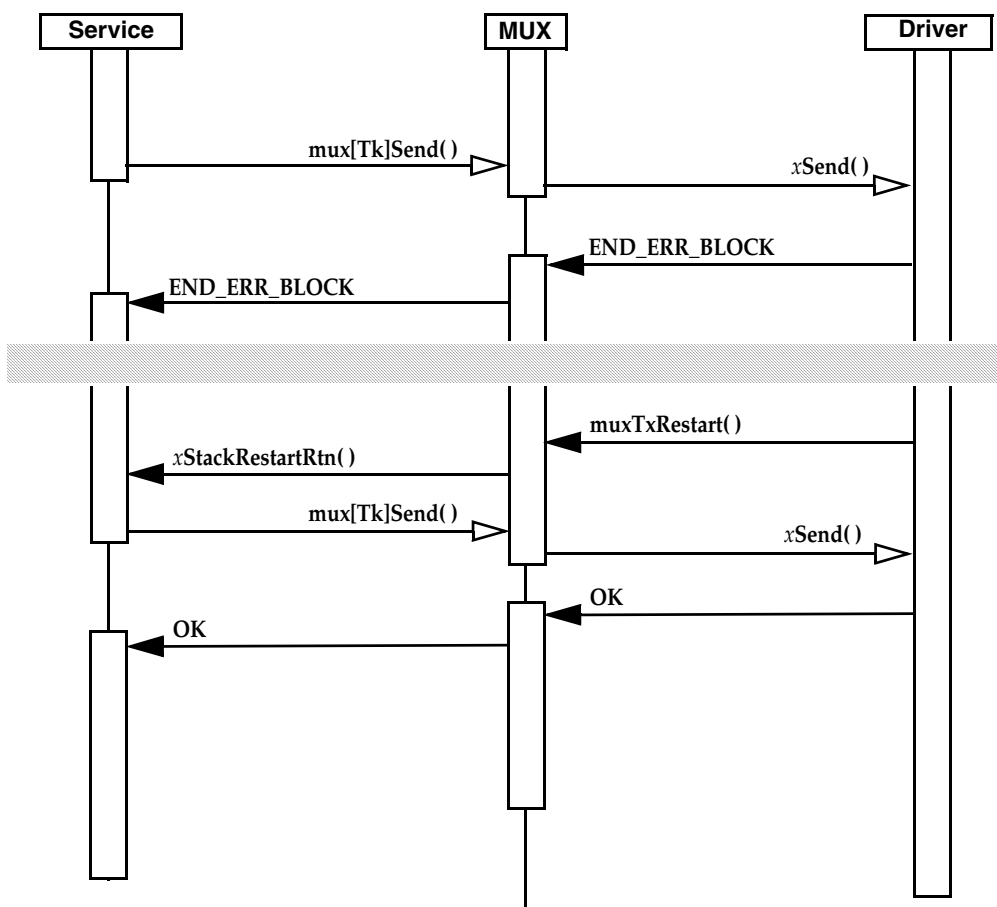
-IP_ERRNO_ENETDOWN (IPNET-native only)

The packet cannot be sent because the device's link to the transmission media is inactive. At present, this return value should not be used; IPNET-native drivers should return **-IP_ERRNO_EWOULDBLOCK** instead.

ERROR (M_BLK-oriented)

Negative error code other than the two listed above (IPNET-native) The packet could not be sent for some other reason. This case should be exceedingly rare; it probably indicates a bug in the driver or the sending protocol. The driver must free the packet in this case if it can.

Figure 5-1 Implementing Flow Control



For END drivers, the data in **pPkt** is a link-level frame; the needed link header is already present.

The prototype of the **xSend()** routine in an END is:

```

STATUS xSend
(
    END_OBJ * pEND,    /* END object */
    M_BLK_ID pPkt      /* M_BLK chain containing the frame */
)
    
```

5.4.5 xMCastAddrAdd()

muxMCastAddrAdd() calls this routine to tell the driver to configure the device so that it will receive packets destined for a particular link-layer multicast address. The driver must also maintain a full list of the multicast addresses so added; Ethernet devices may use the **etherMultiLib** library APIs to do so.

A typical Ethernet **xMCastAddrAdd()** routine looks like this:

```
STATUS xMCastAddrAdd
(
    END_OBJ *   pEnd,          /* driver's control structure */
    char *      pAddress       /* buffer containing multicast address */
)
{
    int retVal;
    x_DRV_CTRL * pDrvCtrl;

    pDrvCtrl = (x_DRV_CTRL *) pEnd;

    semTake (pDrvCtrl->devSem, WAIT_FOREVER);

    retVal = etherMultiAdd (&pEnd->multiList, pAddr);

    if (retVal == ENETRESET)
    {
        pEnd->nMulti++;
        if (pEnd->flags & IFF_UP)
            xEndHashTblPopulate (pDrvCtrl);
        retVal = OK;
    }

    if (retVal != OK)
    {
        errnoSet (retVal);
        retVal = ERROR;
    }

    semGive (pDrvCtrl->devSem);
    return (retVal);
}
```

The **etherMultiAdd()** routine does the following:

1. checks that the specified address in the buffer at **pAddr** is in fact a valid Ethernet multicast address
 - if not, returns **EINVAL**
2. checks whether the address already belongs to the specified list **pEnd->multiList**
 - if so, increments a reference count associated with the address, and returns 0 (zero)
3. attempts to allocate a buffer for the new address
 - if unsuccessful, returns **ENOBUFS**
 - if successful, adds the new address to the list with reference count 1 and returns **ENETRESET**, which lets the driver know that it should increment the multicast address count and reconfigure the hardware to receive packets destined to the new multicast address

In this example code, the driver does this by calling its routine **xEndHashTblPopulate()**.⁶ Different devices have different multicast filtering

6. This routine is very device-specific.

capabilities, but generally you should program your device to receive packets for every multicast address in **pEnd->multiList**, and for as few others as possible. (An exception is that if the **IFF_ALLMULTI** flag has been set, the device should receive packets destined to any multicast address.)

A driver for a device that is not multicast capable should clear **IFF_MULTICAST** in its **flags**, and should also provide dummy **mCastAddrAdd()**, **mCastAddrDel()**, and **mCastAddrGet()** routines in its **NET_FUNCS** interface that simply set **errno** to **ENOTSUP** and return **ERROR**.

Drivers for multicast-capable devices using non-Ethernet MAC addresses cannot use **etherMultiLib**, and will have to implement their own methods to manage multicast address lists.

5.4.6 xMCastAddrDel()

This routine removes a previously registered multicast address from the list that the driver maintains. It does the reverse of **xMCastAddrAdd()**.

A typical Ethernet **xMCastAddrDel()** implementation looks like this:

```
STATUS xMCastAddrDel
(
    END_OBJ * pEnd,      /* END object */
    char *    pAddress /* buffer with the multicast address to be removed */
)
{
    int retVal;
    x_DRV_CTRL * pDrvCtrl;

    pDrvCtrl = (x_DRV_CTRL *) pEnd;

    semTake (pDrvCtrl->devSem, WAIT_FOREVER);

    retVal = etherMultiDel (&pEnd->multiList, pAddr);

    if (retVal == ENETRESET)
    {
        pEnd->nMulti--;
        if (pEnd->flags & IFF_UP)
            xEndHashTblPopulate (pDrvCtrl);
        retVal = OK;
    }

    if (retVal != OK)
    {
        errnoSet (retVal);
        retVal = ERROR;
    }

    semGive (pDrvCtrl->devSem);
    return (retVal);
}
```

The routine is very similar to **xMCastAddrAdd()**, except that it calls **etherMultiDel()** instead of **etherMultiAdd()**, and decrements **pEnd->nMulti** rather than incrementing it, if **etherMultiDel()** returns **ENETRESET**.

etherMultiDel() checks the specified list **pEnd->multiList** for the specified link-layer address at **pAddr**. If the address is not present in the list, **etherMultiDel()** returns **ENXIO**. Otherwise, **etherMultiDel()** decrements the reference count associated with the address.

If the reference count is still nonzero, **etherMultiDel()** simply returns **OK**. Otherwise, it removes the address from the list and frees the buffer that **etherMultiAdd()** allocated to hold it, and returns **ENETRESET**. An **ENETRESET** return value indicates to the driver that the address list has changed, and the device must be reconfigured to receive the new, smaller set of multicast addresses. The driver typically does this using the same routine **xEndHashTblPopulate()** that it provides and calls from **xMCastAddrAdd()**.

5.4.7 xMCastAddrGet()

This routine retrieves a list of all multicast addresses that are currently active for reception on the device. It does not need to touch the device hardware at all.

It takes as arguments a pointer to the **END_OBJ** returned by **xLoad()**, and a pointer to a **MULTI_TABLE** structure into which the list will be put.

An Ethernet **xMCastAddrGet()** routine can use **etherMultiLib** and looks like this template:

```
STATUS xMCastAddrGet
(
    END_OBJ *      pEnd,          /* END object */
    MULTI_TABLE *  pMultiTable /* container for address list */
)
{
    int retVal;
    x_DRV_CTRL *  pDrvCtrl;

    pDrvCtrl = (x_DRV_CTRL *) pEnd;

    semTake (pDrvCtrl->devSem, WAIT_FOREVER);

    retVal = etherMultiGet (&pEnd->multiList, pMultiTable);

    semGive (pDrvCtrl->devSem);
    return (retVal);
}
```

The **MULTI_TABLE** structure specifies the address and length of a buffer, into which the driver should write (in any convenient order) as many of the addresses in its multicast reception list as will fit. Although conventions for non-Ethernet addresses have not been well established, for Ethernet the addresses are written with no padding or separators, so addresses are effectively assumed to be of fixed length known to the driver and the caller. After the addresses have been written to the buffer, the driver should rewrite the **len** member of the **MULTI_TABLE** with the actual number of bytes taken up by the addresses written.

Write this routine to return **OK**. It should always be successful, unless the driver does not support multicast, in which case the routine should return **ERROR** and set **errno** to **ENOTSUP**.

5.4.8 xPollSend()

When using the **WDB_COMM_END** communications type, the external WDB debug agent calls **muxTkPollSend()** with interrupts locked when it wants to send a packet during system mode debugging. **muxTkPollSend()** in turn calls the network device's **xPollSend()** routine.

Packets may also be sent over an interface in polled mode using the **muxPollSend()** or **mux2PollSend()** APIs.

The effective prototype of the driver's **pollSend()** routine depends upon the driver style. For **M_BLK**-oriented drivers, it is

```
LOCAL int xPollSend (END_OBJ * pEnd, M_BLK_ID pMblk);
```

whereas for IPNET-native drivers, it is:

```
LOCAL int xPollSend (END_OBJ * pEnd, Ipcom_pkt * pkt);
```

An IPNET-native driver (but not an **M_BLK** oriented driver) may assume that the packet passed is nonsegmented, described by a single **Ipcom_pkt** rather than possibly a chain. The MUX wrapper code that handles **muxPollSend()** or **muxTkPollSend()** calls to an IPNET-native device takes care of coalescing the (possibly multi-segment) **M_BLK** chain passed to one of these routines into the single **Ipcom_pkt** buffer allocated in the IPNET-native driver's **start()** routine and stored in the **pollPkt** member in the device's **END_OBJ**. Any calls to **mux2PollSend()** are required to pass only a single-segment packet.



NOTE: Polled mode transmission is a low-performance interface intended to support debugging. For details, see [5.6.3 Polled Mode—for Debugging Only](#), p.96.



WARNING: When the MUX calls your driver's **xPollSend()** routine, the system is probably in a mode that cannot service kernel calls. Therefore, this routine must not perform any kernel operations, such as taking a semaphore or allocating memory. Likewise, this routine must not block or delay because the entire system might halt.

This routine may be called only after the device has been put in to polled mode using the **EIOCPOLLSTART** ioctl. The **xPollSend()** routine should immediately return **ERROR** if the device is not in polled mode when it is called. Otherwise, it must either transfer a packet to the device for transmission, or return **EAGAIN** if not ready to do so.

Since the device is in polled mode, it may not rely upon the transmit interrupt (which is disabled) to schedule clean-up of the transmit ring. Further, the caller maintains ownership of the **M_BLK** chain or (single) **Ipcom_pkt**, which implies that either the routine itself must wait until the transmit completes before returning, or that it must make a copy of the data to be transmitted.

Typically, an **M_BLK**-oriented driver maintains a single tuple used for polled-mode sends on the device, and copies data from the provided **M_BLK** chain to the tuple's cluster buffer using **netMblkToBufCopy()**. This avoids the complications of dealing with multisegment **M_BLKs**, and avoids also requiring much additional memory to support the low-performance polled mode. An IPNET-native driver normally does not need to copy data in this way since any packet passed to its **pollSend()** routine is guaranteed to be single-segment. But if the device has transmit data alignment restrictions not met by a packet passed via a **mux2PollSend()** call, the IPNET-native driver may copy that packet's data into the **pollPkt** packet, which is always available, being allocated and set with proper alignment in the driver's **start()** routine.⁷ The **xPollSend()** routine may use the

7. Note that all drivers assume that any calls to the **pollSend()** routine are externally serialized. Usually the WDB agent is the only client of the polled send and polled receive routines, so that this is automatic.

driver's ordinary transmit encapsulation routine to queue the single copy tuple to its transmit ring and enable transmission, but then should busy-wait for transmission to complete and must re-clean the transmit ring before returning, to avoid reusing the copy tuple too soon. When transmit is complete, the routine returns **OK**.

The **xPollSend()** routine and the **xSend()** routine share the same transmit descriptors and the same transmit queue. Therefore, **xPollSend()** should treat the transmit queue and descriptors in the same manner as the **xSend()** routine.

Wind River recommends that your **EIOCPOLLSTART** ioctl-handling code clean the transmit ring before returning, so that polled-mode sends start with an empty transmit ring. In principle, it is possible that the **xPollSend()** routine interrupts the normal **xSend()** routine, for instance if the naive user sets a system-level breakpoint within the driver **xSend()** code. This can potentially corrupt the transmit ring. There is not much that can be done about this, other than using non-END WDB communication type when debugging network drivers.

Write **xPollSend()** to return **OK**, **EAGAIN**, or **ERROR** (it should *not* set **errno** under any circumstances):

OK

Indicates that the packet is successfully sent.

EAGAIN

Indicates that the driver or device is not ready to transmit a frame, and the caller should try again in a little while. The **xPollSend()** routine may wait for short periods of time for the hardware to reach a state where another transmission is possible, although it is preferable to return **EAGAIN** and let the caller drive the polling.

ERROR

Indicates an argument error by the caller or a fatal condition which prevents the provided packet from ever being sent. (Note that the WDB agent, typically the only user of the polled-mode routines, may treat any nonzero return, including **ERROR**, as equivalent to **EAGAIN**, causing the error to repeat.)

The **xPollSend()** routine has the same prototype as the **xSend()** routine. See [5.4.4 xSend\(\)](#), p.79 for additional discussion of the parameters.

5.4.9 xPollRcv()

The external WDB agent, when using the **WDB_COMM_END** communications type, calls **muxTkPollReceive()** with interrupts locked during system mode debugging, to poll for availability of a received packet, retrieving the packet if one is available. **muxTkPollReceive()** in turn calls the driver's **xPollRcv()** routine.

The driver's **pollRcv()** routine may in principle also be called by the **muxPollReceive()** or **mux2PollReceive()** routines, although since normally the WDB agent is the only client of polled mode receives, this is uncommon.

The effective prototype of the driver's **pollRcv()** routine depends upon the driver style. For **M_BLK**-oriented drivers, it is:

```
LOCAL int xPollRcv (END_OBJ * pEnd, M_BLK_ID pMblk);
```

whereas for IPNET-native drivers, it is

```
LOCAL int xPollRcv (END_OBJ * pEnd, Ipcom_pkt * pkt);
```

In both cases, the caller passes in a single-segment packet buffer into which the routine copies a received packet, if one is available.

This routine receives frames using polling instead of an interrupt-driven model. The MUX passes this routine a pointer to the device's `END_OBJ`, and a pointer to an `M_BLK` tuple (or `Ipcom_pkt` with buffer) in which to place the frame. The routine checks the next descriptor in its receive ring, and if a frame has been received, retrieves the frame and copies it into the packet buffer, adjusting the `M_BLK` or `Ipcom_pkt` to specify the packet length. Since the WDB agent currently assumes 4-byte alignment for IP headers, a driver that may execute on architectures with alignment restrictions may need to make small adjustments upwards to the starting address (`pMblk->mBlkHdr.mData` or `&pkt->data[pkt->start]`) at which it copies the data, to ensure that any IP header starts at an address that is a multiple of 4. For instance, ethernet drivers should align the start of the ethernet header on an address congruent to 2 modulo 4. The adjustment is done by modifying `pMblk->mBlkHdr.mData` or `pkt->start`, if necessary. If no frame is immediately available, it returns `EAGAIN`.



NOTE: Polled mode reception is a low-performance interface intended to support debugging. For details, see [5.6.3 Polled Mode—for Debugging Only](#), p.96.



WARNING: When the system calls your `xPollRcv()` routine, it is probably in a mode that cannot service kernel calls. Therefore, this routine must not perform any kernel operations, such as taking a semaphore or allocating memory. Likewise, this routine must not block or delay because the entire system might halt.

It is an error to call `muxTkPollReceive()` when the network device is not in polled mode. It is also an error to call those routines specifying an `M_BLK` which is not attached to a cluster. The `xPollRcv()` routine is encouraged to check for these error conditions, although they are not expected to occur.

For an `M_BLK`-oriented driver, the available space in the provided packet buffer for packet data is `pMblk->mBlkHdr.mLen` bytes starting at address `pMblk->mBlkHdr.mData`. For an IPNET-native driver, the available space is `pkt->maxlen - pkt->start` bytes starting at address `pkt->data + pkt->start`. It is not necessarily an error if the provided packet buffer is not large enough to hold the frame received from the wire, as the caller may be only interested in frames that it knows will be short. If a frame is received that is too large to fit in the provided buffer (adjusting for any required alignment padding), the driver simply drops it and returns `EAGAIN`, as though the frame had not been received at all. (During system mode debugging, when the system is suspended and the WDB agent is in control running with interrupts locked out in polled mode, any packets that are received that are not for the WDB agent itself, apart from certain ARP requests, are dropped.)

5.4.10 `xFormAddress()`

The `formAddress()` routine is an intrinsically `M_BLK` oriented routine, and all `M_BLK`-oriented drivers must provide it. Its support by IPNET-native drivers is optional. However, in order to support non-IPNET, `M_BLK` oriented protocols that send datagrams (without pre-constructed link headers) over an IPNET-native device using `muxTkSend()`, or protocols that call the `muxAddressForm()` or `muxLinkHeaderCreate()` APIs to construct link headers, an IPNET-native driver

must provide a **formAddress()** routine. An IPNET-native ethernet driver (or such a driver for any device using ethernet headers) should in its **xLoad()** routine set the **funcs.formAddress** member in its **END2_NET_FUNCS** table to the value of **_func_endEtherAddressForm**, or to **_func_end8023AddressForm** if IEEE 802.3 headers are desired. Note that the IPNET stack does not ordinarily make use of the **formAddress()** functionality for ethernet drivers, since it constructs ethernet headers itself.

The **xFormAddress()** routine generates a link-level header, prepends it to the **M_BLK** chain containing outgoing data, and adjusts the **mBlk.mBlkHdr.mLen** and **mBlk.mBlkHdr.mData** members accordingly.

If the incoming **M_BLK**'s cluster does not have enough available leading space to contain the added header information, the routine creates an additional **M_BLK/CL_BLK**/cluster tuple for this purpose and inserts it at the beginning of the **M_BLK** chain.

The **xFormAddress()** routine provides support for the **muxFormAddress()** and **muxLinkHeaderCreate()** functions. The current version of the IP network stack does not call either function to create link headers; instead, it creates them internally for supported link types. Other protocols and services may still rely upon the **xFormAddress()** routine, however, so **ENDs** still need to provide it. Ethernet or similar IEEE 802.3 drivers may use one of the implementations in **endLib**, **endEtherAddressForm()** or **end8023AddressForm()**.

The **xFormAddress()** prototype is:

```
M_BLK_ID xFormAddress
(
    M_BLK_ID  pData          /* M_BLK chain containing outgoing data */
    M_BLK_ID  pSrc,          /* source address, in an M_BLK */
    M_BLK_ID  pDst,          /* destination address, in an M_BLK */
    BOOL      bcastFlag      /* use link-level broadcast ? */
)
```

The source and destination link-level addresses are present in memory at the locations specified by **pSrc->mBlkHdr.mData** and **pDst->mBlkHdr.mData**, respectively. The caller also provides the network service type (in network byte order) in the **pDst->mBlkHdr.reserved** field.

If **bcastFlag** is **TRUE**, the driver should construct a link-level broadcast header. That is, it should ignore the destination address at **pDst->mBlkHdr.mData**, substituting the link-level broadcast address.

All the **M_BLK_ID** arguments correspond to **M_BLKs** owned by the caller; the routine should not attempt to free them.

The **xFormAddress()** routine returns a pointer to the first **M_BLK** of the resulting chain; this would be the original **M_BLK** if it was not necessary to prefix a new one.

5.4.11 xPacketDataGet()

Various **M_BLK**-oriented MUX routines call **xPacketDataGet()** to parse the link-level header in a frame represented by an **M_BLK** chain. The primary use is in the receive routine (**muxReceive()**) installed for **M_BLK**-oriented **END** drivers, so all **M_BLK**-oriented drivers must provide a **packetDataGet()** function. However, there are other uses such as output filter protocols and the **muxPacketDataGet()** API that require the driver's **packetDataGet()** function. Some of these uses apply to **M_BLK**-oriented protocols that may be bound over IPNET-native drivers. Thus,

IPNET-native drivers also are encouraged to provide a **packetDataGet()** function. The **xLoad()** routine of an IPNET-native driver for a device that uses ethernet or IEEE 802.3 link headers should set the **funcs.packetDataGet** member of its **END2_NET_FUNCS** structure to the value of the **_func_endEtherPacketDataGet** variable.

Besides the **M_BLK_ID** specifying the frame, the MUX passes this routine a pointer to an **LL_HDR_INFO** structure that the routine must fill out.

The **xPacketDataGet()** routine sets the members of the structure specifying the byte offset and byte size of both the destination and source addresses within the link header; as well as the network service type of the packet, and the byte offset to the network-level header (the same as the link header size). The **ctrlAddrOffset** and **ctrlSize** members are currently unused. The link header is not guaranteed to be contained all in the first **M_BLK** tuple of the chain, although in practice it almost always is. The **M_BLK** chain should not be modified.

The routine should return **OK** unless there is an error in the packet which prevents parsing the link header (for instance, if the packet is too short to contain a full link header), in which case it should return **ERROR**. It should not free the packet.

Drivers for devices using ethernet or IEEE 802.3 MAC headers may use the **endEtherPacketDataGet()** implementation in **endLib** rather than implementing their own version of this routine.

The **xPacketDataGet()** prototype is:

```
STATUS xPacketDataGet
(
    M_BLK_ID      pPkt,      /* M_BLK chain containing packet */
    LL_HDR_INFO * pLHInfo    /* structure to hold header info */
)
```

5.4.12 xAddrGet()

This routine is called only by the rarely used (and somewhat ill-defined) function **muxPacketAddrGet()**. The routine is expected to extract up to four link-level addresses from the link header of a frame specified as an **M_BLK** chain. The four addresses are identified as “local” or “immediate” source and destination addresses, and “ultimate” or “end” or “remote” source and destination addresses. **endLib** implements a version for ethernet, called **endEtherPacketAddrGet()**; this version treats the local/immediate and remote/end/ultimate addresses identically.

The **xAddrGet()** prototype is:

```
STATUS xAddrGet
(
    M_BLK_ID pPkt,      /* M_BLK chain containing frame */
    M_BLK_ID pSrc,      /* local source address, in an M_BLK */
    M_BLK_ID pDest,     /* local destination address, in an M_BLK */
    M_BLK_ID pESrc,     /* end source address, in an M_BLK */
    M_BLK_ID pEDest     /* end destination address, in an M_BLK */
)
```

This routine retrieves the address values for an incoming frame that the MUX provides in the **M_BLK** chain **pPkt**. The link header may be assumed to be present entirely in the first tuple of the chain.

For each of the additional **M_BLK** parameters that are not **NULL**, this routine calls **netMblkDup()** to duplicate the **pPkt M_BLK** to this other **M_BLK** parameter; it

then adjusts the other **M_BLK**'s **mBlkHdr.mData** and **mBlkHdr.mLen** fields to indicate the location and size of the desired address within the header. In the case of **endEtherPacketAddrGet()**, the **mBlkHdr.mLen** fields would all be set to 6, and the **mBlkHdr.mData** members for **pSrc** and **pESrc** would be adjusted 6 bytes into the header, while those of **pDest** and **pEDest** would be left equal to **pPkt->mBlkHdr.mData**.

This routine should return a status of **OK**, or **ERROR** in which case **errno** should be set appropriately.

An ethernet-compatible IPNET-native driver should, in its load routine, set the **funcs.addrGet** member of its **END2_NET_FUNCS** structure to the value of the **_func_endEtherPacketAddrGet** variable.

5.4.13 **xEndBind()**

The **xEndBind()** routine is an optional driver routine that gives your driver the ability to respond to service bind events. In **xEndBind()**, your driver can support the exchange of information between a service and a driver whenever the service binds to a device managed by that driver (provided the binding service also supports this exchange, and the service and the driver agree on the format of the information exchanged).

The MUX calls your driver's **xEndBind()** routine (if any), when a service binds to your driver. To get a reference to a driver's **xEndBind()** routine, the MUX first sends an **EIOCQUERY** ioctl message with the **END_BIND_QUERY** type to the driver's **xIoctl()** routine, and that routine must respond appropriately with a pointer to its **xEndBind()** routine in the structure it returns or the MUX will not invoke that routine. The MUX does not use the **endBind** function pointer in the **NET_FUNCS** interface for this purpose.

The **xEndBind()** prototype is:

```
STATUS xEndBind
(
    END_OBJ *   pEND,           /* END object */
    void *      pNetSvcInfo,    /* info provided by the network service */
    void *      pNetDrvInfo,    /* template for network driver info */
    long        type            /* network service type of binding service */
)
```

The **pNetSvcInfo** and **pNetDrvInfo** arguments are the same as the last two arguments provided in the call to **muxTkBind()**; if **muxBind()** was used to bind the protocol, **NULL** is passed for both arguments. Wind River has established no convention for the use of these arguments; its protocols (when they call **muxTkBind()**) pass **NULL** for both. The **xEndBind()** routine is therefore likely useful only when a custom driver and a custom network service are developed together with knowledge of each other.

Write the **xEndBind()** routine to return **OK**, or else return **ERROR** and set **errno**. If it returns **ERROR**, the MUX denies the attempt to bind the network service.

5.5 Driver Implementation

This section covers drive implementation of both IPNET-native and **M_BLK**-oriented devices.

5.5.1 IPNET-Native Style Driver Implementation

The module **vxmux_mux2.c** contains code necessary for protocols binding with **mux2Bind()**, and some MUX APIs that such protocols use. These APIs describe packets using **Ipcom_pkt** structures, rather than the **M_BLK** structures.

For **mux2Bind()** protocols to work with IPNET-native devices, only the **muxCommon** and **ipcom_mux2** modules are needed.

For **mux2Bind()** protocols to work with END-style devices, additional wrapper routine modules are needed.

mux2Lib contains the following public functions, all new:

```
void mux2LibInit (void);

void * mux2Bind
(
    char * pName,
    int    unit,
    BOOL   (*stackRcvRtn) (void * callbackArg, Ipcom_pkt * pkt),
    STATUS (*stackShutdownRtn) (PROTO_COOKIE cookie, void * callbackArg),
    STATUS (*stackTxRestartRtn) (void * callbackArg),
    void   (*stackErrorRtn)
        (void * callbackArg, END_ERR * err),
    unsigned short type,
    char * pProtoName,
    void * callbackArg
);

int mux2Send (END_OBJ * pEnd, Ipcom_pkt * pkt);
int mux2PollSend (END_OBJ * pEnd, Ipcom_pkt * pkt);
int mux2PollReceive (END_OBJ * pEnd, Ipcom_pkt * pkt);
```

The **mux2Bind()** routine creates a binding between a network service and a network device. Network service uses this routine to bind to a network device specified by the *pName* and *unit* arguments.

This routine can be used to bind to either IPNET-native style or **M_BLK-oriented** devices. Optimal performance is obtained when this routine binds to IPNET-native style devices. However, if the device is **M_BLK-oriented**, wrapper routines are inserted by **mux2Bind()** to convert between the **M_BLK**-oriented packet model (which uses **M_BLK** tuples) and the network service packet model, which uses **Ipnet_pkt** structures. Translation wrappers can decrease performance.

5.5.2 M_BLK-Oriented Devices

The module **muxLib.c** contains code that is needed by protocols bound using **muxBind()**, and some MUX APIs that such protocols use. Only the **muxCommon** and **muxLib** modules are needed for **muxBind()** protocols to work with **M_BLK**-oriented devices. To run **muxBind()** protocols over IPNET-native style devices, additional wrapper routines from **muxOverEnd2.c** are needed.

5.5.3 M_BLK-Oriented Driver Implementation

An M_BLK-oriented driver is frame-oriented. It is organized around the **END_OBJ** and the **NET_FUNCS** structures. The entry points for the **NET_FUNCS** interface of and **END** are:

- **start()** – enable device interrupts and activate the interface
- **stop()** – stop or deactivate a network device or interface
- **unload()** – release a device, or a port on a device, from the MUX
- **ioctl()** – support various ioctl commands
- **send()** – accept data from the MUX and send it on towards the physical layer
- **mCastAddrAdd()** – add a multicast address to those registered for a device
- **mCastAddrDel()** – delete a multicast address registered for a device
- **mCastAddrGet()** – get a list of multicast addresses registered for a device
- **pollSend()** – send packets in polled mode rather than interrupt-driven mode
- **pollRcv()** – receive frames in polled rather than interrupt-driven mode
- **formAddress()** – add addressing information to a packet
- **packetDataGet()** – separate the addressing information and data in a packet
- **addrGet()** – extract the addressing information from a packet

If you write an M_BLK-oriented driver that does not run over Ethernet, you need to implement these entry points explicitly. These drivers running over Ethernet (using either 802.3 or DIX header formats) can set these interface members to the **endLib** implementations of these routines: **endEtherAddressForm()** (**end8023AddressForm()** to construct 802.3-style headers), **endEtherPacketDataGet()**, and **endEtherPacketAddrGet()**.

5.5.4 MUX Receive Routine

During the **muxDevLoad()** call, the MUX sets the **receiveRtn** member of the device's **END_OBJ**-derived **DRV_CTRL** structure to point to the routine that the device should call to pass received data up the stack. An **END** calls this routine with two parameters, as follows:

```
device -> receiveRtn ( device, packet );
```

device

The **END_OBJ** pointer that describes the device that is calling the routine.

packet

An M_BLK pointer that describes the packet being received.

The header file **endLib.h** has a macro, **END_RCV_RTN_CALL()**, that the driver can use to call the MUX receive routine.

5.6 How VxWorks Launches and Uses Your Driver

The task **tUsrRoot** is the first task started during system boot. It initializes all portions of the operating system, including the network stack. Part of network stack initialization consists of initializing at least one network job queue, and spawning a task (such as **tNet0**) to process items on each network job queue.

To load your network device into the MUX, **tUsrRoot** calls **muxDevLoad()**. As input to the call, **tUsrRoot** specifies your driver's **xLoad()** entry point, and the **muxDevLoad()** routine calls this entry point.

The **xLoad()** routine handles any device-specific initialization and returns an object that derives from the **END_OBJ** class (see [Driver Implementations of the xLoad\(\) Routine](#), p.97). The **xLoad()** routine does not enable the device to transmit and receive data (the **xStart()** routine does this when it is called by **muxDevStart()**).

After control returns from **xLoad()** to **muxDevLoad()**, the MUX completes the **END_OBJ** object by adding to it a pointer to a routine your driver can call to pass packets up to the MUX. The MUX then adds this returned **END_OBJ** to a list of **END_OBJ** structures. This list maintains the state of all currently active network devices on the system. After control returns from **muxDevLoad()**, your driver is loaded and ready to use.

5.6.1 Service-to-MUX Interface

To attach to a previously loaded network device, a service calls **muxBindCommon()**. The network service supplies pointers to routines that the MUX can call to:

- shut down the service
- pass an error message to the service
- pass a packet to the service
- restart transmission by the service

The prototypes of these callback routines differ depending on whether you used **mux2Bind()**, **muxBind()**, or **muxTkBind()**.

If you call **muxTkBind()** to bind to an **M_BLK**-oriented device, this imposes an additional layer of translation. It might be better to provide **muxBind()**-style callbacks and use **muxBind()** to bind to **END** devices and avoid the performance impact of this additional translation work.

The **muxBindCommon()** routine returns a cookie that identifies the binding of the service to the specified device. The service uses this cookie in several other MUX calls to refer to this binding instance.

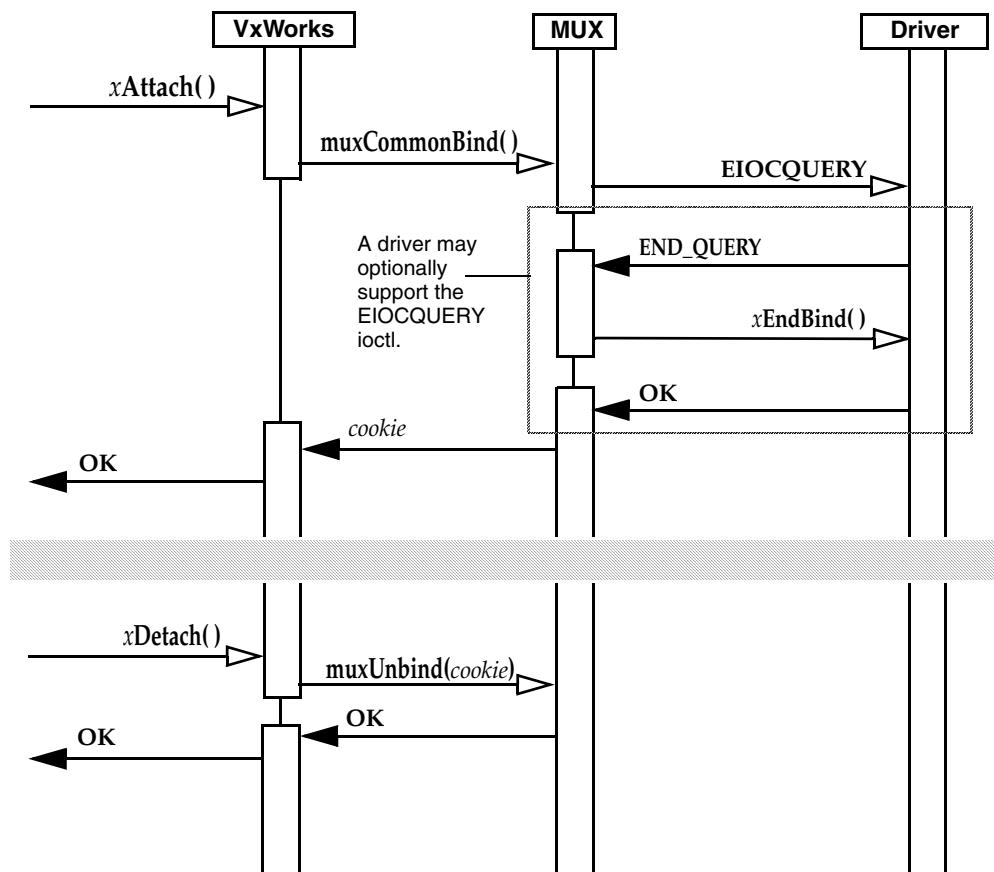
After the service binds itself to a driver through the MUX, it can then call MUX-supplied routines, such as **mux[Tk]Send()**, to transmit a packet or request other MUX services.



NOTE: You may find that you need to call a device-specific MUX routine such as **muxTkSend()** or **muxIoctl()** without first binding to the device. To do this, you must obtain a cookie that describes the device. Call **muxTkCookieGet()** to obtain such a cookie for a specified device name and unit number. This routine allocates no memory, and the value it returns does not describe a real binding, so do not call **muxUnbind()** with this cookie. Also, note that the cookie is valid only while the network device remains loaded in the MUX.

The Wind River Network Stack attaches its protocols to the network boot interface and to network interfaces corresponding to any **INCLUDE_IPNET_IFCONFIG_***n* components you have enabled. If there are additional network interfaces to which the stack should be attached (perhaps interfaces that your application discovers dynamically), your application code must do this attachment itself by calling either **ipcom_drv_eth_init()** or the legacy function **ipAttach()**. The **ipAttach()** routine calls **mux[Tk]Bind()** (see Figure 5-2)

Figure 5-2 Binding and Unbinding a Stack and an Interface



A protocol that binds itself to a loaded device using **mux[Tk]Bind()** may unbind itself using **muxUnbind()**. To cause the Wind River Network Stack to unbind all of its protocols from an interface to which it is attached, use the command-line tool **ifconfig** to first bring the interface down, then detach it. For example:

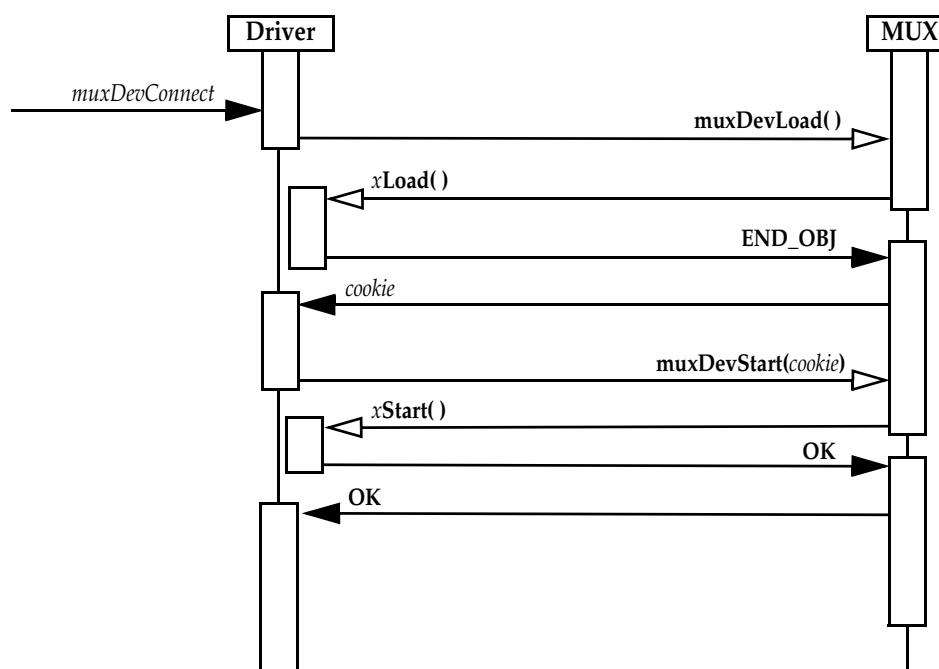
```
-> ifconfig motetsec1 down
-> ifconfig motetsec1 detach
```

ifconfig can be accessed programmatically as the function **ipnet_cmd_ifconfig()**.

5.6.2 Data-Link-to-MUX Interface

VxBus network drivers typically initialize and load to the MUX all of the devices they manage that are discovered dynamically by the VxBus system, or that are listed explicitly in the BSP's **hwconf.c** file. For legacy network drivers on the other hand, a device must have an entry in the **endDevTbl[]** array in the BSP's **configNet.h** file. (Particular BSPs may be able to add dynamically discovered devices to available empty slots in this array.) Stack initialization code causes the **{muxDevConnect}()** method of all VxBus network drivers to run, then iterates through the **endDevTbl[]** array, loading and starting any legacy devices with entries there. For both sorts of devices, **muxDevLoad()** is called first, then **muxDevStart()** (see Figure 5-3).

Figure 5-3 Loading and Starting a VxBus Network Device



NOTE: The **{muxDevConnect}()** method that gets called is the driver's corresponding method handler routine.

The value returned by **muxDevLoad()** identifies the device. You can use this identifier in a subsequent call to **muxDevStart()**, **muxDevStop()**, or **muxDevUnload()**. In some previous versions of the stack, this value could also be passed as an argument to **muxIoctl()** or **muxTkSend()**, but that no longer works—these routines expect an interface binding cookie of the sort that is returned from **mux[Tk]Bind()**, or a pseudo-bind cookie of the sort returned by **muxTkCookieGet()** (see the discussion of **muxTkCookieGet()** in [5.6.1 Service-to-MUX Interface](#), p.92).

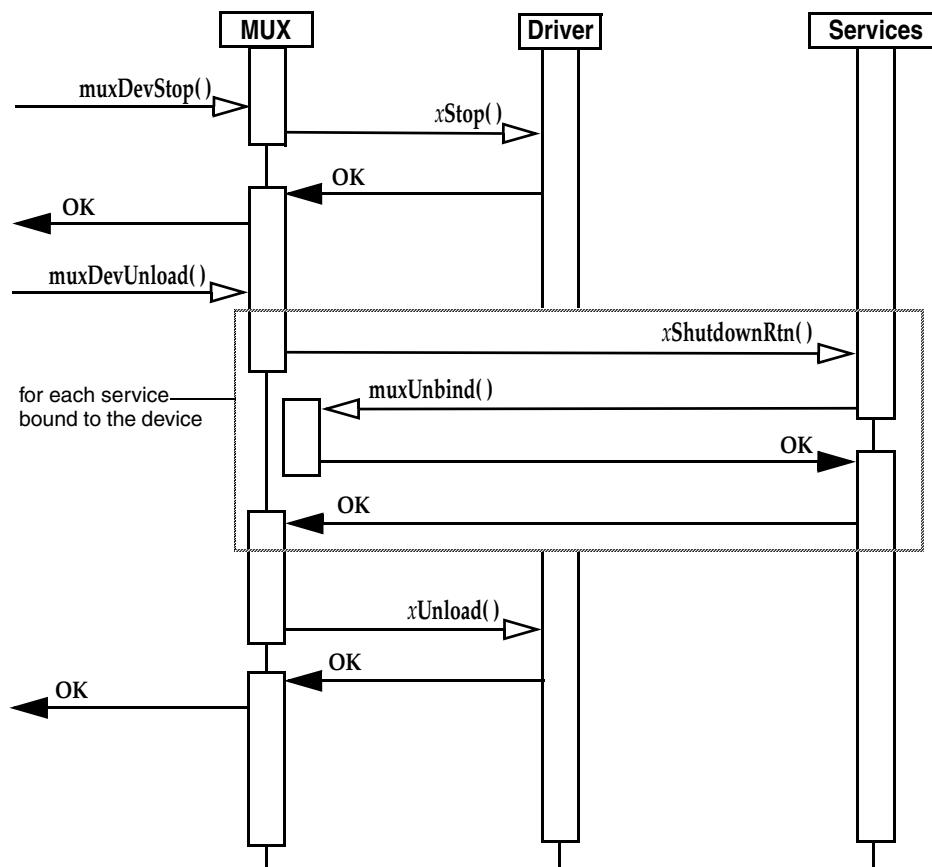
Your driver's load routine creates a **DRV_CTRL** structure, derived from an **END_OBJ** structure, and its **NET_FUNCS** interface. The **END_OBJ** structure provides the MUX with a description of the device, and the **NET_FUNCS** interface provides the MUX with pointers to the driver's implementations of the standard MUX routines: **start()**, **stop()**, **receive()**, **ioctl()**, and so on.

The **muxDevStart()** call enables transmission and reception over an END device. After a device starts, it can pass packets up to the MUX by calling the **receiveRtn** function pointer that the MUX set in the driver's **END_OBJ** structure. The MUX delivers these packets to the appropriate bound services. If no bound services match the packet type, the MUX discards the packet.



NOTE: The Wind River Network Stack expects to borrow the buffers it receives and thus avoid data copying. If a device cannot transfer incoming data directly into clusters, the driver must explicitly copy the data from private memory into a cluster in sharable memory before passing it in an **M_BLK** up to the MUX. The driver must describe a packet destined for the stack as a single **M_BLK/CL_BLK/cluster** tuple (see [Tuples](#), p. 11).

Figure 5-4 Stopping and Unloading a Device



When the driver passes the MUX receive routine a packet with a network service type that matches a service bound to the device, the MUX calls the service's receive routine that the service registered when it called **mux[Tk]Bind()**. If the service receive routine returns **TRUE** (or any nonzero value), the service consumed the packet. Otherwise, the MUX checks if any other bound service can accept the packet, and if not, discards it, freeing the associated **M_BLK** tuple. When a service consumes a packet, the service is responsible for freeing the packet.

To disable transmission and reception on a device, call **muxDevStop()**. Call **muxDevUnload()** to remove the network interface from the MUX. Note that **muxDevUnload()** forcibly shuts down any services that are bound to the device;

the service's shutdown routine must in turn call **muxUnbind()** to unbind the service from the device.

5.6.3 Polled Mode—for Debugging Only

Drivers ordinarily operate in an interrupt-driven mode. During debugging, it can be convenient to have a driver operate in polled mode.

This section discusses routines that drivers implement to support polled-mode, and MUX routines that applications or services can call to transfer packets in polled mode.

The network stack itself does not use polled mode. Currently, only the WDB agent **COMM_END** back end in system mode uses polled mode. During system mode debugging, the WDB agent calls **muxPollSend()** directly in order to pass packets to the driver in polled mode.

While it is possible for an application to use a driver that has implemented the necessary APIs, and call **muxPollSend()**—just as the WDB agent does—the network stack itself does not support sending data, such as IP traffic, in polled mode. The stack will not detect that a driver is in polled mode and, consequently, will not call **muxPollSend()**.

Also, the polled mode interface copies whole packets for both transmission and reception.

Thus, you cannot use polled mode routines as a high-performance polling mechanism for increasing forwarding network performance. Wind River recommends that you use polled mode only for system mode debugging over WDB.

5.7 Driver Interface with the MUX

This subsection describes the driver entry points and the shared data structures that comprise an driver's interface with the MUX.

Data Structures Shared by the Driver and the MUX

The core data class for an END driver is the END object, or **END_OBJ**. This structure is defined in **target/h/end.h**. The driver's **xLoad()** routine returns a pointer to an object derived from the **END_OBJ** that it allocates and partially populates. This object supplies the MUX with information that describes the driver as well as a pointer to a **NET_FUNCS** interface that the driver implements.

Although the driver's **xLoad()** routine populates much of the **END_OBJ** object, the MUX sets some of this object's members when a service binds to the device. Specifically, the MUX maintains the array of services bound to the **END_OBJ**. When the driver calls the receive routine that the MUX registered by setting the **receiveRtn** member of the driver's **END_OBJ** object, this routine in turn calls the bound service receive routines appropriate for the received packet.

Driver Implementations of the `xLoad()` Routine

A driver must implement an `xLoad()` routine. In a VxBus network driver, since the driver itself passes the address of this routine to `muxDevLoad()`, the `xLoad()` routine need not be public; however, for a legacy END driver, the MUX accesses the `xLoad()` routine directly, outside of the driver, and it must be globally visible. It is usually the only globally visible routine for such a driver.

Before the stack can use a network interface to send and receive frames, it must load the appropriate network device into the MUX, attach services to the network driver, and configure the interface at the service level (for example, by assigning IP addresses). The `tUsrRoot` task loads network devices into the MUX by calling the `muxDevConnect()` method for any VxBus network drivers or `muxDevLoad()` for all the (non-VxBus) network drivers that are in `endDevTbl[]`. The entries in this table provide all the information needed to call `muxDevLoad()`. This includes a reference to the driver's `xLoad()` routine (or in some cases, a wrapper that the BSP provides that in turn calls the driver's `xLoad()` routine).

As input, the `xLoad()` routine takes an initialization string, as well as an optional argument provided by the BSP or VxBus driver.

Write your driver's `xLoad()` routine as a two-pass algorithm. The MUX calls it twice during the load procedure. In the first pass, the initialization string argument points to a buffer starting with a zero byte. The `xLoad()` routine is expected to check for this empty string and overwrite it with a string containing the prefix name of the device (such as "fei" or "emac"). This informs `muxDevLoad()` about the driver-specific interface name prefix.

The MUX then calls your `xLoad()` routine a second time. This time the initialization string starts with a decimal unit number string, followed by a colon, followed by the contents of the initialization string from the `endDevTbl[]` that `tUsrRoot` passed to `muxDevLoad()`. Your `xLoad()` routine must then return a pointer to the `END_OBJ`-derived `DRV_CTRL` object that it creates, or a `NULL` if the load fails.

VxBus network drivers typically pass `NULL` to `muxDevLoad()` as the initialization string, and pass the instance's `VXB_DEVICE_ID` as the optional argument; the `xLoad()` routine gets all the information it needs from the device ID and the VxBus parameter system. Older network drivers pass an initialization string that is typically a driver-specific colon-separated sequence of driver configuration parameters, and the `xLoad()` routine must tokenize and parse this string, which generally contains such things as the address of memory mapped registers for the device, the number of receive and transmit descriptors to use, the PHY address, and special device flags or options.

5.8 MUX Routines for Network Drivers

The MUX provides a small number of routines intended specifically for use by network drivers. This section discusses these functions. It only considers routines provided by the MUX proper; libraries such as `netBufLib`, `ipnet_pkt_pool`, `jobQueueLib`, `etherMultiLib`, `endEtherHdr`, and others provide many additional support routines usable by network drivers.

All of the routines discussed in this section must be called only in the context of the network job queue used by the network device. If the IPNET stack is bound to the device, that network job queue must be the one with ID *netJobQueueId*, which is serviced by the *tNet0* task. Executing the routines in any other task would lead to mutual exclusion failure and possible instability.

5.8.1 Receive Routine

The first routine that the MUX provides is provided as the *receiveRtn* function pointer installed by **muxDevLoad()** in the device's **END_OBJ** structure. The routine installed by the MUX is either **muxReceive()** (for **M_BLK**-oriented drivers) or **mux2Receive()** (for IPNET-native drivers); however, the driver should never call these routines directly, but only via the **END_RCV_RTN_CALL()** or **END2_RCV_RTN_CALL()** macros provided by *endLib.h* for **M_BLK**-oriented and IPNET-native drivers respectively. The macros make the call using the *receiveRtn* function pointer. This allows other software to replace the *receiveRtn* function pointer to intercept receive calls from a device and apply alternate processing.

An **M_BLK**-oriented driver calls the receive routine as:

```
END_RCV_RTN_CALL (pEnd, pMblk);
```

where *pEnd* is a pointer to the device's **END_OBJ** structure, and *pMblk* is an initialized **M_BLK** tuple describing the received packet; while an IPNET-native driver calls the receive routine as:

```
END2_RCV_RTN_CALL (pEnd, pkt);
```

Here *pEnd* is as above, but *pkt* is a pointer to an initialized **Ipcom_pkt** describing the received packet.

In either case, the IPNET stack requires that the received packet be described as a single contiguous data segment. Note that IPNET requires only 2-byte alignment for the network-layer header.

See the *VxWorks Device Driver Developer's Guide* for more detailed information on what **M_BLK** or **Ipcom_pkt** members need to be set to describe the received packet.

5.8.2 Transmit Restart Routine

The MUX provides the routine:

```
void muxTxRestart(void * pEnd);
```

that a driver is expected to call some time after its send routine has returned indicating a transmit 'stall', that is a return value of **END_ERR_BLOCK** for an **M_BLK** oriented driver, or of **-IP_ERRNO_EWOULDBLOCK** for an IPNET-native driver. The driver calls this routine when it determines that a new send may succeed; usually when an in-progress transmit completes and space opens up after the driver cleans the transmit ring. This routine informs all network services bound to the device (and that provided a protocol TX restart callback function) that they may attempt to send again. Protocols may use this signal to implement transmit flow control.

A driver should call **muxTxRestart()** only from a job on the device's network job queue. The driver's transmit mutex should not be held across the call to **muxTxRestart()**, to avoid possible misorderings with respect to protocol-level

mutual exclusion held across calls to the driver's **send()** routine. Note that the driver's **send()** routine may well be called several times as part of the call to **muxTxRestart()**.

5.8.3 Notifying the MUX of Device Events

The MUX provides several functions that let a network driver notify the MUX about various events affecting a device. The MUX code notifies the protocols bound to the device. The fundamental routine is:

```
void muxError
(
    void *    pCookie,                /* END_OBJ */
    END_ERR * pError                  /* Error structure. */
)
```

The driver must call this routine from within a network job posted to the device's job queue. The driver passes **muxError()** a pointer to the device's **END_OBJ**, and a pointer to an **END_ERR** structure, defined in the **target/h/end.h** header file:

```
typedef struct end_err
{
    INT32 errCode;                    /* Error code, see above. */
    char* pMesg;                      /* NULL terminated error message */
    void* pSpare;                     /* Pointer to user-defined data. */
} END_ERR;
```

The **end.h** header lists several 'error codes', really just event codes, that may be set in the first member of this structure. Not all of these error codes are used by network drivers. The MUX uses the **muxError()** notification mechanism internally for per-END-device events that it generates, or that protocol actions generate. The events most commonly used by network drivers are the following:

END_ERR_LINKDOWN

the device's connection to the media has gone down

END_ERR_LINKUP

the device's connection to the media has become active

END_ERR_NO_BUF

the device could not deliver a received packet to the stack because it has run out of packet buffers

Note that the codes **END_ERR_DOWN**, **END_ERR_UP**, and **END_ERR_FLAGS** are handled internally by the MUX; drivers should not generate these events. The *pMesg* argument should point to a short, static, human readable string describing the error. (This message is not automatically logged.) The *pSpare* argument is a placeholder for a pointer to additional event-specific data, but no conventions have been established for its use. Protocols attached to the device generally only look at the **errCode** member, will ignore any events with error code values that they do not recognize or do not care about. More convenient functions exist for the cases of **END_ERR_LINKDOWN** and **END_ERR_LINKUP**.

```
void muxLinkUpNotify (END_OBJ * pEnd);
void muxLinkDownNotify (END_OBJ * pEnd);
```

These events are often generated from a VxBus driver's **{miiMediaUpdate}()** method (also known as the 'link update' method), which is called by the MII monitor task if it detects a change in link state. The driver posts **muxLinkUpNotify()** as a network job to the device job queue if the link state has

changed from inactive to active, or posts **muxLinkDownNotify()** if the link state has changed from active to inactive.

5.9 Queueing Work to the Network Job Queues

The network stack initialization code spawns (by default) a single network job queue to handle network-related work, primarily for network interface drivers. The job queue is serviced by a single VxWorks task named **tNet0**. While it is possible to configure the network stack startup code to create multiple job queues and spawn a task for each job queue, in VxWorks 6.7 only one of these queues, that serviced by the **tNet0** task, may execute IPNET stack protocol code, and so only that job queue is suitable for use as the job queue associated with any network device to which the IPNET stack will bind. A job queue is identified by its **JOB_QUEUE_ID**, which drivers should treat as an opaque value. The **JOB_QUEUE_ID** for the job queue serviced by the **tNet0** network daemon task is available in the variable **netJobQueueId**, declared in **netLib.h**.

The best way to post work to a network job queue is to call the **jobQueuePost()** routine:

```
STATUS jobQueuePost
(
    JOB_QUEUE_ID jobQueueId,
    QJOB *        pJob
)
```

The job to be done is represented by a **QJOB** object, the structure of which is declared in **jobQueueLib.h** as:

```
typedef struct _QJOB
{
    struct _QJOB * pNext;
    UINT32         priInfo; /* modify with QJOB_SET_PRI () */
    QJOB_FUNC      func;    /* function to execute */
} QJOB;
```

The **QJOB_FUNC** is declared as:

```
typedef void (* QJOB_FUNC) (void *);
```

The members of the **QJOB** class are as follows:

pNext

jobQueueLib uses the **pNext** member internally for queueing jobs. The driver may ignore this field.

priInfo

The **priInfo** member records the job priority (0 through 31), as well as some other flags used internally by **jobQueueLib**. The network daemon that is running the job queue services queued jobs in strict priority order, treating 31 as the highest priority. You should set the **priInfo** member to a priority value between 0 and 31; this will ensure that the bits used internally by **jobQueueLib** start out cleared. Unless the driver has a specific reason for doing otherwise, it should choose the default priority **NET_TASK_QJOB_PRI** (16) as defined in *installDir/components/ip_net2-6.x/vxmux/include/netLib.h*

func

The **func** member is the routine that is executed by the task that services the job queue when this job runs. That task passes this routine a single argument, which is a pointer to the **QJOB**.

A driver is responsible for allocating its own **QJOBS**. A driver needs one **QJOB** for each type of work it would like to post from an ISR. A driver that uses a single ISR to handle all device interrupts might use only a single **QJOB**, while another driver might use three separate **QJOBS** for receive work, transmit cleanup, and error handling, respectively. But your driver needs only a small, finite number of **QJOBS**, and typically embeds them in its driver control structure for the device.

A driver initializes its **QJOB** members at initialization time, typically in its load or start routine, and usually leaves them unchanged after that. The following code snippet shows how a hypothetical “quik” driver might initialize three **QJOB** members embedded in its driver control structure:

```
pDrvCtrl->quikTxJob.func      = quikTxHandle;
pDrvCtrl->quikTxJob.priInfo  = NET_TASK_QJOB_PRI;
pDrvCtrl->quikRxJob.func     = quikRxHandle;
pDrvCtrl->quikRxJob.priInfo  = NET_TASK_QJOB_PRI;
pDrvCtrl->quikErrJob.func    = quikErrHandle;
pDrvCtrl->quikErrJob.priInfo = NET_TASK_QJOB_PRI;
```

Here is how the receive interrupt might post the **quikRxJob** to execute **quikRxHandle()**:

```
jobQueuePost (pDrvCtrl->jobQueueId, &pDrvCtrl->quikRxJob);
```

Here is how the **quikRxHandle()** job handler routine might recover the driver control structure from its argument, which is a pointer to **pDrvCtrl->quikRxJob**:

```
LOCAL void quikRxHandle
(
    QJOB * pJob
)
{
    QUIK_DRV_CTRL * pDrvCtrl = member_to_object (pJob, QUIK_DRV_CTRL,
                                                quikRxJob);

    ...
    /*
     * Do a bounded amount of RX work, then requeue the job if there is
     * still more work to do; otherwise, reenale RX interrupts, and
     * return.
     */
    ...
}
```

The **member_to_object()** macro shown above, declared in **jobQueueLib.h**, converts the address of a member of a structure to the address of the structure itself.

The **jobQueuePost()** routine enqueues the specified job onto the specified job queue, and if necessary unpendes the task that services the job queue. That task services any queued jobs in strict priority order. As it services each queued job, it dequeues the job object just before calling the routine specified by the job object’s **func** member. While the job is enqueued, attempts to requeue it by calling **jobQueuePost()** again will have no effect, but you should avoid such calls. Certainly the driver should not modify a **QJOB** object while it is enqueued; during this time it is considered to be owned by the job queue itself. To avoid the ISR’s reposting the job when it is enqueued, it may be sufficient for the device to lock device interrupts, when shared interrupts are not a factor. But if other devices can share the interrupt line that the device uses, then it may be necessary for the device to maintain a separate atomic flag to indicate whether the device has already

posted the job. For some devices, the device interrupt mask register can play the role of this flag. The job handler routine must take care when it clears the flag and reenables interrupts, so as to avoid races that could either prevent the driver from receiving any further packets, or that could occasionally cause those packets that it receives to languish without the driver servicing them until the arrival of a subsequent packet.

It is possible (and convenient) for a **QJOB** to repost itself from the **QJOB**'s handler routine. You cannot cancel a job you have already queued. This may become relevant when an interface shuts down; the driver must use some other mechanism (such as an atomic flag) to wait until the jobs in the queue execute, and then prevent further queuing of the jobs.

Network job queues are usually shared by multiple network interfaces, and by network protocol code also. It is possible to design a driver that starves the network stack and other drivers. When a driver uses **taskDelay()**, or any other delay mechanism, in code that executes in the context of a network job queue daemon, the delay prevents the task from processing packets from other interfaces. For this reason, you must carefully consider using delays in the driver. Consider rescheduling the job with another **jobQueuePost()** call instead of delaying. This allows other interfaces, as well as the network stack, to perform other work while the driver is waiting.

Because interrupts are relatively costly in terms of overall system performance, one recommended goal of network drivers is to process many packets before reenabling interrupts. However, to avoid starvation of other interfaces, the driver should enforce a cap on the number of packets that it processes at any one time. If additional packets are available when the cap is reached, the driver can reschedule the receive routine with another call to **jobQueuePost()**.

5.10 Collecting and Reporting Packet Statistics

This section describes the interfaces by which drivers collect and report packet statistics to attached network services. Note that services have the option to maintain interface statistics on their own, ignoring the statistics that are collected by the driver and device. However, making use of the statistics collected by the driver, in particular when polled-mode statistics is used with hardware support, may be more efficient.

There are two interfaces which a driver can use to report packet statistics to the higher levels. **M_BLK**-oriented drivers may use the **endM2Packet()** API, which allows software collection of interface statistics on a per-packet basis. Alternatively, **M_BLK**-oriented or IPNET-native drivers can collect (often with hardware support) statistics which may be polled (at fairly low frequency) by the upper levels. Enabling this polling requires including the components **INCLUDE_END_POLLED_STATS** and **INCLUDE_MIB2_IF** in the VxWorks image.

In the present release, statistics collected by the driver in either of these two ways are reported only via the **m2IfLib** MIB-II interface library used by the SNMP agent. The IPNET stack maintains separate per-device statistics in software, some of which may be queried using the **ifconfig** command.

5.10.1 Calling the Driver Routines

A driver's load routine should call **endM2Init()** to provide needed interface information to the stack. This includes the interface type, the MAC address and its length, the MTU, the interface's data rate (that is, wire speed), and interface flags.

The **endM2Init()** routine will initialize the MIB interface data structures and store this information as appropriate to either RFC 1213 or RFC 2233, whichever is configured into the VxWorks image. For example:

```
endM2Init(&pDrvCtrl->endObj, M2_ifType_ethernet_csmacd,
        (u_char *) &pDrvCtrl->enetAddr, 6, ETHERMTU, MOT_TSEC_PHY_SPEED_1000,
        IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST | IFF_SIMPLEX);
```

The driver's unload routine calls **endM2Free()** to release any MIB-related data structures for which memory was allocated by **endM2Init()**.

```
endM2Free (&pDrvCtrl->endObj);
```

If a driver wishes to support the polled statistics mode, it adds two members to its device control structure:

```
END_IFDRVCONF endStatsConf;
END_IFCOUNTERS endStatsCounters;
```

The driver load routine should initialize these members after calling **endM2Init()**:

```
bzero ((char *)&pDrvCtrl->endStatsCounters, sizeof(END_IFCOUNTERS));
pDrvCtrl->endStatsConf.ifPollInterval = sysClkRateGet();
pDrvCtrl->endStatsConf.ifEndObj = &pDrvCtrl->endObj;
pDrvCtrl->endStatsConf.ifValidCounters = (END_IFINUCASTPKTS_VALID
| END_IFINMULTICASTPKTS_VALID | END_IFINBROADCASTPKTS_VALID
| END_IFINOCETTS_VALID | END_IFOUTOCETTS_VALID
| END_IFOUTUCASTPKTS_VALID | END_IFOUTMULTICASTPKTS_VALID
| END_IFOUTBROADCASTPKTS_VALID);
```

The **ifValidCounters** member is a set of bit flags indicating which statistics the driver supports, that is: which of the **END_IFCOUNTERS** members the driver will fill in. The **ifPollInterval** member sets the period (in system clock ticks) at which the stack polls statistics.

The **ifEndObj** merely points to the **END_OBJ** structure for the interface.

The remaining members of the **END_IFDRVCONF** structure are initialized by code outside the driver.

The driver's **xIoctl()** routine should implement a few MIB-related commands. The following code comes from the **motTsecEnd.c** driver (and some additional comments have been added):

```
/*
 * Drivers should support EIOCGMIB2233 and EIOCGMIB2 by calling endM2Ioctl().
 */
case EIOCGMIB2233:
case EIOCGMIB2:
    /* These commands retrieve the interface statistical
     * data structures used for RFC 2233 or RFC 1213, respectively.
     * Note that the driver doesn't access these directly.
     */
    error = endM2Ioctl (&pDrvCtrl->endObj, cmd, data);
    break;

/*
 * The EIOCGPOLLCNF and EIOCGPOLLSTATS commands are implemented
 * only if the driver wishes to support polled statistics retrieval.
 * EIOCGPOLLCNF retrieves a pointer to the polling configuration
 * structure (as initialized by the driver load routine).
 * EIOCGPOLLSTATS first collects the statistics into the
 * END_IFCOUNTERS structure, retrieving them from the hardware if
```

```
* necessary, and the stores a pointer to that structure at the
* indicated address.
*/
case EIOCGPOLLCNF:
    if ((data == NULL))
        error = EINVAL;
    else
        *((END_IFDRVCONF **)data) = &pDrvCtrl->endStatsConf;
    break;

case EIOCGPOLLSTATS:
    if ((data == NULL))
        error = EINVAL;
    else
    {
        /* retrieve the statistics from the hardware */
        error = motTsecEndStatsDump(pDrvCtrl);
        if (error == OK)
            *((END_IFCOUNTERS **)data) = &pDrvCtrl->endStatsCounters;
    }
    break;
```

A VxBus network driver enables polled-mode statistics collection in its **{muxDevConnect}()** or **{mux2DevConnect}()** method by means of code like the following:

```
if (_func_m2PollStatsIfPoll != NULL)
    endPollStatsInit (pDrvCtrl->geiMuxDevCookie,
                     _func_m2PollStatsIfPoll);
```

The **_func_m2PollStatsIfPoll** function pointer is used to avoid a direct reference to the **m2PollStatsIfPoll()** function, for cases when **INCLUDE_MIB2_IF** is not included in the image. For legacy network drivers, the network stack initialization code takes care of calling **endPollStatsInit()** for each device in the **endDevTbl** array, if **INCLUDE_END_POLLED_STATS** is included in the image.

Polling takes place by calling **muxIoctl()** with the **EIOCGPOLLSTATS** command for the desired network device. The handler for that command should retrieve from the hardware the supported statistics and store them in the appropriate members of the driver's **END_IFCOUNTERS** structure (**endStatsCounters** in the example above). The counts stored should be the counts accumulated since the last polling call, or (on the first call only) those accumulated since the interface was started.

A driver which does not support the polled mode statistics collection should not implement the **EIOCGPOLLCNF** and **EIOCGPOLLSTATS** MUX ioctl commands. Instead, it accumulates statistics per packet by calling the **endM2Packet()** routine, as follows.

For successfully transmitted packets, the driver calls:

```
endM2Packet (&pDrvCtrl->endObj, pMblk, M2_PACKET_OUT);
```

For packets which could not be transmitted due to a resource limitation (not for normal TX stalls), the driver should call:

```
endM2Packet (&pDrvCtrl->endObj, NULL, M2_PACKET_OUT_DISCARD);
```

For packets which the driver detected a transmission error (not a resource limitation or a normal TX stall), the driver calls:

```
endM2Packet (&pDrvCtrl->endObj, NULL, M2_PACKET_OUT_ERROR);
```

For successfully received packets, the driver calls:

```
endM2Packet (&pDrvCtrl->endObj, pMblk, M2_PACKET_IN);
```

For packets received with errors, the driver calls:

```
endM2Packet (&pDrvCtrl->endObj, NULL, M2_PACKET_IN_ERROR);
```

(In such a case, the driver does not ordinarily deliver the packets to the stack). For packets which could not be received due to resource limits, the driver should call:

```
endM2Packet (&pDrvCtrl->endObj, NULL, M2_PACKET_IN_DISCARD);
```

Note that collection of such failure statistics could be a best-effort activity.

Integrating a New Network Service

- 6.1 Introduction 107
- 6.2 Implementing the MUX/Network Service Interface 107
- 6.3 Adding a Socket Interface to Your Service 118

6.1 Introduction

This chapter describes how to integrate a new network service with the Wind River Network Stack. By a network service, or equivalently a network protocol, we mean here an implementation of the network layer (layer 3) of the OSI network model. The service may also implement the transport or higher levels of a network stack; however it is the network layer that interfaces with the MUX.



NOTE: By using “network service” in this sense, we are explicitly excluding TCP or UDP socket applications running on top of the Wind River Network Stack. Such services do not interact directly with the MUX; the Wind River Network Stack’s IP protocols (and related protocols such as ARP), are what we would consider the ‘network services’ in this case.

Figure 3-1 shows how network services communicate with the data link layer through the MUX interface. Part of porting a new network service to the network stack is porting its data link layer access code to use the MUX interface. Everything specific to the network interface is handled in the drivers of the data link layer, which are described in [5. Integrating a New Network Device Driver](#).

6.2 Implementing the MUX/Network Service Interface

A network service sends and receives packets through the MUX interface. At minimum, to work with the MUX your service must implement an initialization routine and routines that support packet transfer and error reporting.

6.2.1 Initializing the Interface

Besides loading a network device into the MUX and starting it, you usually want to attach a network stack—that is, a collection of network protocols—to the device. A stack must provide a means of attaching itself to a network device. For example, you can attach the current version of the Wind River Network Stack to a network device by calling either of the following routines:

- **ipcom_drv_eth_init** (const char * drvname, Ip_u32 drvunit, const char * ifname)
- **ipAttach** (int drvunit, char * drvname)

The arguments are in slightly different formats in these routines.

ipcom_drv_eth_init() is the lowest-level function, and **ipAttach()** is provided for compatibility with previous versions of the Wind River Network Stack.

The *drvname* and *drvunit* arguments are the device name and unit number, respectively, as seen by the MUX and the network driver. If you call **ipcom_drv_eth_init()**, you can optionally specify an interface name, to be used by the network stack, that differs from the driver-level name. For example, the following call attaches to the **fei2** network interface, but calls it **eth0** at the stack level:

```
ipcom_drv_eth_init ("fei", 2, "eth0");
```

On the other hand, the following call (which may be less confusing) attaches the network stack to the interface and uses the same interface name, **fei2**, in the stack that it uses at the driver level:

```
ipcom_drv_eth_init ("fei", 2, IP_NULL);
```

An **ipAttach()** counterpart to this second call would be:

```
ipAttach (2, "fei")
```

Calling **ipcom_drv_eth_init()** or **ipAttach()** makes the stack aware of the interface, and causes the stack to bind its protocols to the device. For example, for an IPv4-capable stack, at least the IPv4 and ARP protocols would be bound to the device, and possibly others.

Whether you write a stack of several protocols, or simply a single network service, you need to provide a similar routine that informs your service (or stack) of the network device, and causes it to bind to that device. For convenience, assume this routine is called **xAttach()** and that *x* is replaced with your service's name. You can also modify the startup code to insert calls to **xAttach()** for particular devices. There is no standard interface to accomplish this. However Wind River recommends that you create a component descriptor file (.cdf file) and a configlet, so that your service can be handled by the Workbench **Kernel Configuration Editor**. For more information, see the *VxWorks Kernel Programmer's Guide*.

Although it is not required, it is recommended practice to also provide an **xDetach()** to unbind your service from a network device and release internal service resources that refer to that device.

6.2.2 Binding to a Network Device

A network service must bind to a network device (that has been loaded into the MUX) before it can send and receive packets through it. A service binds to a loaded network device by calling one of the three routines **muxBind()**, **muxTkBind()**, or **mux2Bind()**. The first two of these routines are for **M_BLK** oriented protocols, while **mux2Bind()** is provided to bind an **Ipcom_pkt**-oriented protocol.



NOTE: The IPNET portion of the Wind River Network Stack binds its protocols using **mux2Bind()**. **M_BLK** oriented protocols that bind using **muxBind()** or **muxTkBind()** can bind to IPNET-native network devices; and **Ipcom_pkt**-oriented protocols that bind using **mux2Bind()** can bind to **M_BLK**-oriented device. However, in both of these cases, there is some packet format translation overhead incurred on each send and receive. There is also some minimal API translation overhead when a protocol binds using **muxTkBind()** to an **M_BLK**-oriented (END-style) device. There is no translation overhead, however, when a protocol binds using **muxBind()** to an **M_BLK**-oriented driver, or binds to using **mux2Bind()** to an IPNET-native driver.^a

- a. The **muxTkBind()** binding method had no translation overhead with the no-longer-supported NPT-style drivers.

Let us consider the (somewhat similar) arguments to **muxBind()**, **muxTkBind()**, and **mux2Bind()**:

```
PROTO_COOKIE muxBind
(
    char * pName,
    int    unit,
    BOOL   (*stackRcvRtn) (void*, long, M_BLK_ID, LL_HDR_INFO *, void*),
    STATUS (*stackShutdownRtn) (void*, void*),
    STATUS (*stackTxRestartRtn) (void*, void*),
    void   (*stackErrorRtn) (END_OBJ*, END_ERR*, void*),
    long   type,
    char*   pProtoName,
    void*   pSpare
);

PROTO_COOKIE muxTkBind
(
    char *      pName,
    int         unit,
    FUNCPTTR    stackRcvRtn,
    FUNCPTTR    stackShutdownRtn,
    FUNCPTTR    stackTxRestartRtn,
    VOIDFUNCPTTR stackErrorRtn,
    long        type,
    char *      pProtoName,
    void *      pNetCallbackId,
    void *      pNetSvcInfo,
    void *      pNetDrvInfo
);

void * mux2Bind
(
    char * name,
    int    unit,
    BOOL   (*stackRcvRtn) (void * callbackArg, struct Ipcom_pkt_struct * pkt),
    STATUS (*stackShutdownRtn) (PROTO_COOKIE cookie, void * callbackArg),
    STATUS (*stackTxRestartRtn) (void * callbackArg),
    void   (*stackErrorRtn) (void * callbackArg, END_ERR * err),
    unsigned short type,
    char * pProtoName,
    void * callbackArg
);
```

Each binding routine takes an initial two arguments specifying the driver name and device unit number of the device to bind to. Each routine also takes four function pointers to 'callback' routines that the MUX will call. The purpose of these routines is to:

- deliver packets to the protocol
- command the protocol to detach itself from the device
- notify the protocol that transmission may continue after a send attempt 'stalled'
- notify the protocol of various other events relating to the device it is bound to.

The primary differences between the binding methods is the signatures of the callback functions. The **muxTkBind()** prototype, unfortunately, does not indicate the actual effective calling signature of its callbacks, but they are as follows:

```
BOOL (*stackRcvRtn) (void*, long, M_BLK_ID, void *);  
STATUS (*stackShutdownRtn) (void *);  
STATUS (*stackTxRestartRtn) (void *);  
void (*stackErrorRtn) (void*, END_ERR*);
```

Each routine also takes a *type* argument, which is the network service type. This is either an 'ethertype' value¹ identifying the layer 3 protocol (such as 0x0800 for IPv4, 0x0806 for ARP, and so on), or one of three special service types not associated with a particular layer 3 protocol number. This is discussed more fully below.

Each bind routine is also passed a **string** naming the protocol, which is used in **muxShow()** output, and a **void *** argument (variously called *pSpare*, *pNetCallbackId*, or *callbackArg*) that is passed to each of the callback functions as one of its arguments.

A service may bind to more than one network interface; we call the pairing of a network interface with the service bound to it, a 'binding instance.' Each of the routines returns an opaque value—a protocol binding cookie—that identifies the binding instance to the MUX. The return value may be used in calls to MUX routines that expect a protocol binding cookie argument, such as **muxSend()**, **muxTkSend()**, **muxTkPollReceive()**, **muxIoctl()**, **muxUnbind()**, **muxProtoInfoGet()**, and others.

The **muxTkBind()** API has two additional arguments, *pNetSvcInfo* and *pNetDrvInfo*, that relate to the very rarely used **END_BIND_QUERY** mechanism. This mechanism allows a driver to be notified of protocols binding to its devices, and to potentially exchange information between a protocol that is binding and the device driver.

With regard to the network service type value, a service is called a "normal service" or a "typed protocol" if this value is an ethertype identifying a particular layer 3 protocol. Otherwise, the network service type must be one of the following three special values.

- A **MUX_PROTO_SNARF** type service (or snarf protocol) sees all the received packets that are processed by any device to which it is bound, and that are not consumed by an earlier-bound snarf service.

1. Ethertype values are defined by RFC 1700 and other sources.

- A **MUX_PROTO_PROMISC** type service (promiscuous protocol) sees all received packets that are processed by any device to which it is bound, that are not consumed by any snarf, normal, or earlier-bound promiscuous protocol.
- A **MUX_PROTO_OUTPUT** type service sees outgoing rather than incoming packets.

A normal service (or “typed” protocol) sees only packets received by the devices that it is bound to, that match its type, and that were not consumed by any snarf service. At most one typed protocol of a given network service type may be bound to a network device. For any packet received on a device, any snarf services bound to the device see the packet first, then the bound normal service that matches the packet’s service type (if any), then any promiscuous protocol bound to the device; always with the proviso that if an earlier service consumes the packet, it will not be seen by any later service.



NOTE: Since snarf services process all packets before any typed protocol sees them, the presence of snarf services can decrease the receive performance of all typed protocols bound to the same interface. The WDB agent, when running using the **WDB_COMM_END** communication type, attaches itself to a network device as a snarf service. When measuring network performance, either do not include the WDB agent, or run performance benchmarks over different interfaces than the one to which the WDB agent is attached (unless you really intend to measure the performance impact of the WDB agent). Some snarf protocols may not need to be permanently attached to an interface. It will help performance to detach any such snarf protocol from an interface when it is not needed, only reattaching it later if it is needed again.

A service consumes a packet by returning **TRUE** (or any nonzero value) from its receive routine; it is then responsible for freeing the packet. A service receive routine that does not consume a packet passed to its receive routine returns **FALSE** and should not modify or free the packet.

6.2.3 Using MUX/Service Interface Routines

The subsections that follow provide an overview of how a network service handles the following tasks:

- sending packets
- controlling devices
- shutting down an interface

Sending Packets

A service sends packets over a network device by calling one of the three routines²:

```
int mux2Send (END_OBJ * pEnd, struct Ipcom_pkt_struct * pkt);  
  
STATUS muxSend (PROTO_COOKIE cookie, M_BLK_ID pMblk);  
  
STATUS muxTkSend (PROTO_COOKIE cookie, M_BLK_ID pNBuf, char * destAddr,  
                 USHORT svcType, void * extra);
```

2. Here we consider only normal sends, not polled mode sends.

The first routine represents the packet as an `Ipcom_pkt` structure (or a chain of such structures), while the latter two routines represent a packet as an `M_BLK` tuple (or chain of such tuples).

First Argument

The first argument of each routine identifies the network device over which to send the packet. For `mux2Send()`, this argument is explicitly a pointer to an `END_OBJ`, while the latter two functions expect a protocol binding cookie. In this release, `muxSend()` and `muxTkSend()` will accept an `END_OBJ` pointer in place of the protocol cookie, so, in fact, all of the functions can be called with an `END_OBJ` pointer as the first argument. But `mux2Send()` may not be passed a protocol binding cookie as its first argument.

`muxTkSend()` Additional Arguments

The `muxTkSend()` routine differs in that it has additional arguments. When the `destAddr` argument is not `NULL`, the packet in the `pNBuf` buffer does not have a link header; `destAddr` points at a link-layer destination address (such as a 6-byte ethernet MAC address), and `svcType` contains the network service type for the outgoing packet, already converted to network byte order. The MUX³ is expected to prepend a full link-layer header to the provided packet. When `destAddr` is `NULL`, the packet `pNBuf` already contains a full link-level header. The extra argument is a remnant from the no-longer-supported NPT driver style; it provided a way that a `muxTkSend()` could pass additional data to the NPT driver send routine. However, in this release and later, this argument is unused and should be set to `NULL`.

Return Value

All three “send” routines return `OK` (zero) if the send succeeds; in that case, the driver takes ownership of the packet, and frees it when transmission completes.

But if `mux2Send()` returns `-IP_ERRNO_EWOULDBLOCK`, or if either of the other two routines returns `END_ERR_BLOCK`, this indicates that the device does not have sufficient resources—usually, space in the TX DMA ring—to complete the send at present. This is called a “transmit stall.” The caller maintains ownership of the packet, and the driver arranges to call `muxTxRestart()` later, when the send might succeed if retried. The `muxTxRestart()` call by the driver results in a call to the transmit restart routines of all services bound to the device, as discussed below.

Any other return value is unexpected, and probably indicates a defect. The driver's send routine frees the packet in this case. One condition that could cause such an error in some drivers, and might crash others, is if the packet that the caller is attempting to send is longer than the maximum frame size for the device (such as 1518 bytes for ethernet when VLAN tags are allowed and jumbo frames are not enabled). It is the responsibility of the caller to respect the device's MTU.⁴

3. Or an NPT-style driver, no longer supported.

4. The device's current MTU may be fetched using the `EIOCGIFMTU` `END` ioctl command, or set (within the device limit) using the `EIOCSIFMTU` `END` ioctl. However, coordination between multiple services attached to a device when one of them changes the configured MTU of the device is not automatic, and may be problematic. For such shared devices, it may be best not to change the device from its default MTU, unless some sort of out-of-band coordination between the bound services is arranged.

Constructing Link Level Headers

Both **mux2Send()** and **muxSend()** require full packets with link-level headers included. **muxTkSend()** allows omitting the link level header, but it is more efficient for a service to construct the link header itself rather than to allow **muxTkSend()** to construct the link header. This is primarily because if **muxTkSend()** constructs the link header itself, and the driver send returns **END_ERR_BLOCK**, then **muxTkSend()** would have to restore the packet to its original state without a link header. Since the packet will either be dropped or queued for resending by the service, this is probably wasted effort.

If a service will add the link header to a network-layer datagram, it must first perform any necessary conversion from service addresses to the link-layer addresses appropriate to the device being used. Knowing the destination and source link-layer addresses and the network service type, an **M_BLK** oriented service may call **muxAddressForm()** or **muxLinkHeaderCreate()** to prefix a link header to a packet, making use of the device's **xFormAddress()** routine. A **mux2Bind()** service bound to an IPNET-native device could use the analogous **formLinkHdr** function pointer member of the device's **END_OBJ** to do the same thing, using **Ipcom_pkt** packets. Alternatively, if the service knows the type of the **END** interface being used, and knows the format of the link header for that specific interface type, the service may use its own means to construct a link header; however, the service might not automatically work with other types of device drivers that provide their own special **xFormAddress()** or **formLinkHdr()** routines. The IPNET protocols currently follow the latter approach, and construct link headers themselves.

Shutting Down an Interface

Relatively few applications need to unload a network device from the MUX, but for those that do, the MUX provides the function **muxDevUnload()**. If your application calls **muxDevUnload()**, the MUX calls the **xStackShutdownRtn()** routine registered at bind time for each network service that is still bound to the device (see [*xStackShutdownRtn\(\)*](#), p.114).



WARNING: Do not call **muxDevUnload()** for a device managed by a VxBus driver. VxBus drivers expect to call **muxDevUnload()** themselves in their **{vxbDrvUnlink}()** methods, and instability may result if **muxDevUnload()** is called for a VxBus network device instance by other code. See the *VxWorks Device Driver Developer's Guide* for more information about unloading VxBus network devices.

Within this shutdown routine, the network service must start the process of detaching from the device. This process includes calling **muxUnbind()** to unbind the network service from the device. It is not necessary that the shutdown routine actually complete the detach process (that is, call **muxUnbind()**) within the shutdown routine itself, but the shutdown routine must at least schedule this work. If any of the **xStackShutdownRtn()** calls returns a value other than OK, **muxDevUnload()** immediately returns ERROR. If all of the bound service **xStackShutdownRtn()** calls return OK, then when all services have completed unbinding from the device, and when any additional references to the device that were acquired using **muxDevAcquire()** are released using **muxDevRelease()**, the device is removed from the MUX, the driver's unload routine is called.



WARNING: It is not safe to call **muxDevUnload()** for an interface unless all the protocols that are bound to it support a working **xStackShutdownRtn** callback routine. In particular, since the WDB agent does not currently support unbinding from an END device, do not attempt to unload a device to which the WDB agent is bound; the **muxDevUnload()** call will pend forever.

As it may be difficult for some services to accomplish all the work necessary to close and detach from a network interface entirely in the context of the synchronous **xStackShutdownRtn()** callback, some applications may find it more convenient to start the process of detaching services from a network interface first (using service-specific detach routines), calling **muxDevUnload()** only after all such problematic services have completed unbinding from the interface.

6.2.4 Service Routines Registered when Binding to a Device

When a network service binds to a network driver through the MUX, it must provide references to routines that the MUX can call to handle the following:

- passing a packet into the service
- passing exceptional event notifications to the service
- restarting transmission by the service through the network device
- shutting down the network service

The prototypes of the routines that you specify to handle these cases differ depending on which routine you call to bind the service to a network interface in the MUX (**mux2Bind()**, **muxTkBind()**, or **muxBind()**). This section describes the four MUX interface routines that a service implements and references in a **mux[Tk]Bind()** call.

xStackShutdownRtn()

If your application calls **muxDevUnload()** for a loaded network device, **muxDevUnload()** in turn calls the **xStackShutdownRtn()** of each service that is bound to that device.

Within this routine, the network service must do whatever is necessary to initiate the process of detaching the service from the device. This process includes releasing any internal references the service has to the device and calling **muxUnbind()** to unbind the service from the device. However, the process need not be completed in **xStackShutdownRtn()**. The current implementation uses a reference count mechanism, and completes the unload process when all services have unbound from the device and any other references acquired using **muxDevAcquire()** have been released.

mux2Bind() Version

For a service bound with **mux2Bind()**, the **xStackShutdownRtn()** prototype is:

```
STATUS xStackShutdownRtn
(
    PROTO_COOKIE cookie,    /* return value from mux2Bind() */
    void * netCallbackId    /* identifies the device to the service */
)
```

The *cookie* argument is the binding instance cookie returned by **mux2Bind()**. This is probably not useful to the routine.

The *netCallbackId* parameter is the “callbackArg” that is passed to **mux2Bind()**. This value is opaque to the MUX, but the network service understands it and uses it to identify the particular network interface that is being shut down. It is typically a pointer to the service's data structure that represents that interface.

muxTkBind() Version

For a service bound with **muxTkBind()**, the **xStackShutdownRtn()** prototype is:

```
STATUS xStackShutdownRtn
(
    void * netCallbackId /* the handle/ID installed at bind time */
)
```

The MUX passes this routine a single argument, which is the identifier that the service passed into the MUX during the **muxTkBind()** call. This value is opaque to the MUX, but the network service understands it and uses it to identify the particular network interface. It is typically a pointer to the service's data structure that represents that interface.

muxBind() Version

For a service bound with **muxBind()**, the **xStackShutdownRtn()** prototype is:

```
STATUS xStackShutdownRtn
(
    void * pBindCookie, /* binding cookie returned from muxBind() */
    void * netCallbackId /* the handle/ID installed at bind time */
)
```

The MUX passes this routine two arguments:

- The binding instance cookie that was returned from **muxBind()**.
- The identifier that the service passed into the MUX during the **muxBind()** call. This value is opaque to the MUX, but the network service understands it and uses it to identify the particular END interface. It is typically a pointer to the service's data structure that represents that interface.

xStackRcvRtn()

The MUX delivers the packets it receives from the network device to a service by calling the **xStackRcvRtn()** callback that the service registered when it bound to the device. The **xStackRcvRtn()** is called only in the context of the network job queue used by the network interface.

The *netCallbackId* parameter that the MUX passes into **xStackRcvRtn()** is the callback argument that the service specified at bind time. The *type* parameter specifies the network service type of the packet. The *pPkt* parameter points to an **M_BLK** tuple (or **Ipcom_pkt**) that describes the packet. Currently, drivers must always provide received packets in a single contiguous data segment.

If a network service accepts the packet by returning **TRUE**, it takes ownership of the packet and is responsible for freeing the given **M_BLK** chain when the service is finished with it. If it returns **FALSE**, it should neither free nor modify the packet.

mux2Bind() Version

For a service bound with **mux2Bind()**, the **xStackRcvRtn()** prototype is:

```
BOOL xStackRcvRtn
(
    void * netCallbackId,          /* identifies the device to the service */
    struct Ipcom_pkt_struct * pPkt /* the packet being delivered */
)
```

The *netCallbackId* parameter passed to **mux2Bind()** identifies the device to the service.

The *pPkt* parameter is a pointer to an **Ipcom_pkt** describing the packet being delivered. The network service type of the packet is available in the VxWorks-only **net_svc** member of the **Ipcom_pkt** structure. For more information on how packets are represented by **Ipcom_pkt** structures, see the *VxWorks Device Driver Developer's Guide*.

muxTkBind() Version

For a service bound with **muxTkBind()** the **xStackRcvRtn()** prototype is:

```
BOOL xStackRcvRtn
(
    void *      netCallbackId, /* the handle/ID installed at bind time */
    long        type,          /* network service type of the packet */
    M_BLK_ID    pPkt,          /* the packet as an M_BLK tuple chain */
    void *      pSpareData     /* pointer to optional data from driver */
)
```

For services bound with **muxTkBind()**, the link header is always present in the cluster in the first tuple of the chain describing the packet, but the lead **M_BLK** may be adjusted to skip over the link header:

- For normal typed protocols, **pPkt->mBlkHdr.mData** is advanced by the size of the link header, while **pPkt->mBlkHdr.mLen** and **pPkt->mBlkPktHdr.len** are decreased by the size of the link header.
- For **MUX_PROTO_SNARF** and **MUX_PROTO_PROMISC** services, the lead **M_BLK** is not adjusted, that is **pPkt->mBlkHdr.mData** still points to the start of the link header, and **pPkt->mBlkHdr.mLen** and **pPkt->mBlkPktHdr.len** still include the length of the link header.
- For **MUX_PROTO_OUTPUT** services bound to **ENDs**, **pPkt->mBlkHdr.mData** is advanced by the size of the link header, while **pPkt->mBlkHdr.mLen** and **pPkt->mBlkPktHdr.len** are decreased by the size of the link header. A pointer to the destination MAC address in the header is placed in **pPkt->mBlkPktHdr.rcvif**.
- The size of the link header (the network service offset) is stored in **pPkt->mBlkHdr.offset1**.

muxBind() Versions

For a service bound with **muxBind()** the **xStackRcvRtn()** prototype is:

```
BOOL xStackRcvRtn
(
    void *      pBindCookie, /* returned by muxBind() */
    long        type,        /* the network service type of the packet */
    M_BLK_ID    pPkt,        /* the packet as an M_BLK tuple chain */
    LL_HDR_INFO * pLLHInfo,  /* link-level header info structure */
    void *      pCallbackId  /* the handle/ID installed at bind time */
)
```

In this version of `xStackRcvRtn()`, `pBindCookie` is the binding cookie that is returned by `muxBind()`. As this value is opaque to the service, it is probably less useful than the `netCallbackId` value passed into the other version of `xStackRcvRtn()`.

For all services bound with `muxBind()`, the link-level header is present at the start of the packet, and is described in the `LL_HDR_INFO` structure pointed to by the `pLLHInfo` argument. The MUX calls the END device's `xPacketDataGet()` routine to parse the link header and fill out this structure. The most important information in this structure is the size of the link header, but it also holds the offsets and sizes of the source and destination link-level addresses, as well as (redundantly) the network service type of the packet.

`xStackErrorRtn()`

A device notifies the MUX of various exceptional events it encounters by calling `muxError()`, and the MUX forwards these to the network services that are bound to the device by calling `xStackErrorRtn()`. It is the responsibility of the network service to take any necessary action when it receives the event notification.

The MUX passes this routine a pointer to an `END_ERR` structure and the callback argument `netCallbackId` that was passed at bind time.

`mux2Bind()` Version

For a service bound with `mux2Bind()`, the `xStackErrorRtn()` prototype is:

```
void xStackErrorRtn
(
    void * netCallbackId, /* identifies the device to the service */
    END_ERR * err         /* identifies the event */
)
```

`muxTkBind()` Version

For a service bound with `muxTkBind()` the `xStackErrorRtn()` prototype is:

```
void xStackErrorRtn
(
    void *      netCallbackId, /* the handle/ID installed at bind time */
    END_ERR * pError          /* pointer to structure containing error */
)
```

`muxBind()` Version

For a service bound with `muxBind()` the `xStackErrorRtn()` prototype is:

```
void xStackErrorRtn
(
    void *      pEND,          /* END_OBJ passed to the MUX by the driver */
    END_ERR * pError,          /* pointer to structure containing error */
    void *      netCallbackId /* the handle/ID installed at bind time */
)
```

This version of the `xStackErrorRtn()` routine takes an additional argument, `pEnd`, which describes the END device.

xStackRestartRtn()

The MUX calls the **xStackRestartRtn()** routine to restart transmission over a network device by any network services bound to the device that care to do so.

When an device's **xSend()** routine returns **END_ERR_BLOCK** (for an **M_BLK**-oriented driver) or **-IP_ERRNO_EWOULDBLOCK** (for an IPNET-native driver), it is indicating that it cannot schedule the packet for transmission immediately (usually due to a temporary lack of space in the transmit ring). The sending service may choose to drop the packet, or to hold on to it for later retransmission. Having returned **END_ERR_BLOCK** (or **-IP_ERRNO_EWOULDBLOCK**), the driver guarantees that it will call **muxTxRestart()** when the device is again ready to accept packets for transmission. **muxTxRestart()** calls the **xStackRestartRtn()** routine for each service that is bound to the device and that provided such a routine. The **xStackRestartRtn()** routine may respond by sending any packets that it has queued for the device, until it sends them all or the send routine returns **END_ERR_BLOCK** (or **-IP_ERRNO_EWOULDBLOCK**) once more.

The MUX passes this routine the callback argument value that the service passed to the bind routine.

mux2Bind() Version

For a service bound with **mux2Bind()**, the **xStackRestartRtn()** prototype is

```
STATUS xStackRestartRtn
(
    void * netCallbackId /* identifies the device to the service */
)
```

muxTkBind() Version

For a service bound with **muxTkBind()** the **xStackRestartRtn()** prototype is:

```
STATUS xStackRestartRtn
(
    void * netCallbackId /* the handle/ID installed at bind time */
)
```

muxBind() Version

For a service bound with **muxBind()** the **xStackRestartRtn()** prototype is:

```
STATUS xStackRestartRtn
(
    void * pEND, /* END_OBJ passed to the MUX by the driver */
    void * netCallbackId /* the handle/ID installed at bind time */
)
```

This version of the **xStackRestartRtn()** routine takes an additional argument, **pEnd**, which describes the END device.

6.3 Adding a Socket Interface to Your Service

One way to allow applications to access your network service is to add sockets support to the service. In order to make it easier for you to write a network service

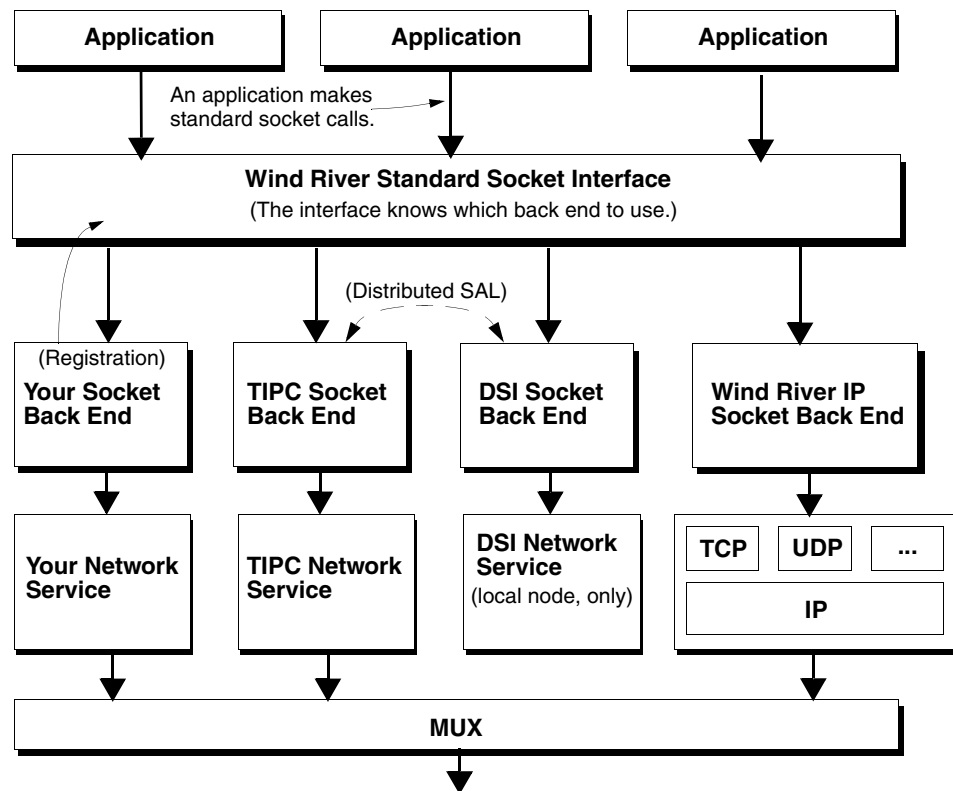
that includes sockets support, the Wind River Network Stack includes a standard sockets interface.

With the standard socket interface, you can add new socket back ends to access your network service, and through it, the network. This allows developers who are already familiar with the standard sockets API to more easily use your service.

With the standard sockets interface, an application can create and use sockets of different address families, which may be managed by different back end service implementations. A layered architecture makes this possible. The Wind River standard sockets interface, implemented by **sockLib**, is a layer above your back end socket layer, as shown in Figure 6-1.

This section shows you how to implement a sockets back end.

Figure 6-1 The Standard Socket Interface



6.3.1 Process Overview

To create a socket, an application calls the standard function **socket()**, and receives a socket descriptor in return. The **socket()** routine looks up the correct back end implementation to use based upon the specified **domain** (address family) argument. If it finds a registered back end that handles that address family, **socket()** calls the back end's socket creation function, obtaining back a data structure that represents the socket. The **socket()** routine then completes the data structure by allocating a descriptor from the I/O system and associating it with the socket data structure, and returns the descriptor to the caller.

The socket data structure contains a pointer to a table of pointers to functions that the back end provides to implement all the various socket operations (see [6.3.3 Socket Functional Interface](#), p.121). You implicitly specify this table when you register the socket back end by calling **sockLibAdd()**, as discussed in [6.3.2 Registering a Socket Back End](#), p.120. When an application makes sockets API calls other than **socket()**, the socket descriptor returned by **socket()** is one of the arguments. The **sockLib** implementation for each such sockets API call converts the socket descriptor to the underlying socket data structure using I/O system descriptor look-up facilities, fetches the pointer to the back end's function table, and calls the appropriate back-end function to complete the call.

The I/O system functions **read()**, **write()**, **ioctl()**, and **close()** may also be used on socket descriptors. In this case, the I/O system converts the descriptor to the underlying socket data structure, calls the registered **sockLib** read, write, ioctl, or close function, which in turn calls the back end's read, write, ioctl, or close handler via the back end function table. **sockLib** registers a single I/O system driver to handle all socket descriptors.



WARNING: In the present release, the socket data structure used by **sockLib** is in fact a **struct socket**, declared in *installDir/components/ip_net2-6.x/vxcoreip/include/net/socketvar.h*. This is primarily for historical reasons. The **struct socket** structure contains many members, only a few of which are needed by **sockLib** itself; and some socket back ends will prefer not to use the other members of **struct socket**. It is very likely that a future release will replace (or modify) **struct socket** with a much smaller structure (which would be embeddable in a back end's private data structure representing a socket). The following members from **struct socket** are likely to remain in the new, smaller structure:

```
struct sockFunc * pSockFuncTbl; /* socket back-end function table */
int so_fd; /* the socket file descriptor */
void * so_bkendaux; /* socket back-end auxiliary data */
```

If your back end uses other members of **struct socket**, then after the change to the smaller structure, you may have to move these members to your back end's own private socket data structure.

Generally, the interface between **sockLib** and socket back ends, as well as the interface for registering socket back ends, although described here as it presently exists, should be considered somewhat fluid. Wind River may modify these interfaces as it sees fit in future releases.

6.3.2 Registering a Socket Back End

You can register a socket back end implementation by calling **sockLibAdd()** some time during system start-up after **sockLib** is itself initialized by a call to **sockLibInit()**.

The **sockLibAdd()** routine has the following prototype:

```
STATUS sockLibAdd
(
    FUNCPTR sockLibInitRtn, /* back end's initialization routine */
    int domainMap, /* unused */
    int domainReal, /* address family */
)
```

The routine returns **OK**, or **ERROR** if the routine could not add the socket back end. The routine takes three parameters:

sockLibInitRtn

A pointer to your **xSockLibInit()** routine that **sockLibAdd()** invokes. **sockLibAdd()** calls this routine as if it had the following prototype:

```
SOCK_FUNC * xSockLibInit (void);
```

That is, it is passed no arguments, and it is expected to return a pointer to an initialized **SOCK_FUNC** structure, which is the table of function pointers for the back end (see [6.3.3 Socket Functional Interface](#), p. 121).

domainMap

sockLibAdd() ignores this parameter.

domainReal

This parameter specifies the address family that this back end implements. A back end may support more than one address family, but in such a case you must call **sockLibAdd()** multiple times, once per address family. Allowed address families are in the range from 1 to **AF_MAX-1**, and the **AF_** constants that identify these address families (**AF_INET**, and so forth) are declared in *installDir/components/ip_net2-6.x/vxcoreip/include/sys/socket.h*.

At most one back end will handle sockets of any given address family. For example, the native Wind River Network Stack normally handles sockets of the **AF_INET**, **AF_INET6**, **AF_ROUTE**, and **AF_PACKET** families. A socket back end of your own implementation may not handle one of these address families without disabling handling of that family by the native stack.

6.3.3 Socket Functional Interface

The socket functional interface is the set of implementations of standard socket routines that a particular socket back end supports. **SOCK_FUNC** is declared in *installDir/components/ip_net2-6.x/vxcoreip/include/sockFunc.h* as follows:

```
typedef struct sockFunc                                /* SOCK_FUNC */
{
    FUNCPTR      libInitRtn;                            /* unused */
    FUNCPTR      acceptRtn;                             /* accept() */
    FUNCPTR      bindRtn;                               /* bind() */
    FUNCPTR      connectRtn;                            /* connect() */
    FUNCPTR      connectWithTimeoutRtn;                 /* connectWithTimeout() */
    FUNCPTR      getpeernameRtn;                        /* getpeername() */
    FUNCPTR      getsocknameRtn;                       /* getsockname() */
    FUNCPTR      listenRtn;                             /* listen() */
    FUNCPTR      recvRtn;                               /* recv() */
    FUNCPTR      recvfromRtn;                           /* recvfrom() */
    FUNCPTR      recvmsgRtn;                            /* recvmsg() */
    FUNCPTR      sendRtn;                               /* send() */
    FUNCPTR      sendtoRtn;                             /* sendto() */
    FUNCPTR      sendmsgRtn;                            /* sendmsg() */
    FUNCPTR      shutdownRtn;                          /* shutdown() */
    FUNCPTR      socketRtn;                             /* socket() */
    FUNCPTR      getsockoptRtn;                         /* getsockopt() */
    FUNCPTR      setsockoptRtn;                         /* setsockopt() */
    FUNCPTR      zbufRtn;                               /* not supported */

    /* The following IO-system handlers are called via wrappers */
    /* in sockLib.c. */

    FUNCPTR      closeRtn;                              /* close() */
    FUNCPTR      readRtn;                               /* read() */
}
```

```
FUNCPTR    writeRtn;                /* write()          */
FUNCPTR    ioctlRtn;               /* ioctl()         */
} SOCK_FUNC;
```

The use of the **FUNCPTR** type is unfortunate, as it provides neither type checking, nor any guide to those who implement back-ends of the arguments passed to the functions, nor the return values expected of them. With few exceptions, you can get the pseudo-prototype for one of these routines, which indicates how it is actually called, by considering the corresponding standard sockets API function prototype in **sockLib.h**, or the corresponding I/O system function prototype in **ioLib.h** (replacing the integer socket descriptor argument with a **struct socket *** argument). For instance, the connect API is prototyped in **sockLib.h** as the following:

```
extern STATUS connect (int s, struct sockaddr * name, int namelen);
```

So, the back end function called through the **connectRtn** function pointer is called as if it had the following prototype:

```
STATUS xConnectRtn (struct socket * so, struct sockaddr * name, int namelen);
```

sockLib's connect() implementation converts the integer socket descriptor passed as its first argument **s**, to the **struct socket** pointer **so**, and calls the back end's **xConnectRtn()** routine, replacing **s** with **so** and passing the **name** and **namelen** arguments unchanged. From the **so** argument, the socket back end can find its own private data for the socket; the **so_bkendaux** member of **struct socket** is intended as a pointer to such private data. (Alternatively, since the back end's **socketRtn()** is responsible for allocating the **struct socket**, the back end may choose to embed the **struct socket** in a larger structure containing also the private data.)

The return value from the back end **xConnectRtn()** is same as the return value from **connect()**.

Consult the **sockLib** reference pages for additional information on the intended behavior of the sockets API functions. External sockets API information is also very useful; for instance *IEEE Std 1003.1* contains the official Posix descriptions of sockets API functions and data structures. (For various historical reasons, the VxWorks sockets API prototypes do not always match exactly those defined by IEEE 1003.1.) For general background on sockets programming, see W. Richard Stevens, *Unix Network Programming - Networking APIs: Sockets and XTI*, Volume 1.

There are exceptions to the above rule-of-thumb. These are **socketRtn**, **acceptRtn**, and **ioctlRtn**:

xSocketRtn()

The **xSocketRtn()** routine has the following prototype:

```
int xSocketRtn
(
    int          domain,    /* socket domain or address family number */
    int          type,      /* socket's nature, e.g. SOCK_DGRAM */
    int          protocol,  /* the protocol variety of the socket */
    struct socket ** ppSo   /* the socket structure */
)
```

The back end's **xSocketRtn()** is responsible for allocating and initializing a **struct socket** and any other structures that the back end needs to represent a socket of the specified **domain**, **type**, and **protocol** (these are passed directly from the corresponding arguments to the **socket()** routine). The allocation may be done using the kernel heap, a **netBufLib** pool, or a back-end specific method. If the **xSocketRtn()** cannot allocate and initialize a socket of the specified kind,

xSocketRtn() must free any partial allocations, set **errno** to the appropriate value (such as **ENOMEM**, **ENOBUFS**, **EAFNOSUPPORT**, **EPROTONOSUPPORT**, **EPROTOTYPE**), and return **ERROR**.

Otherwise, **xSocketRtn()** stores a pointer to the allocated **struct socket** at the address specified by **ppSo**, and returns **OK**. The **socket()** code will then itself store a pointer to the back end's **SOCK_FUNC** table in the returned **struct socket**'s **pSockFuncTbl** member, then attempt to allocate a file descriptor from the I/O system, associating it with the socket. The **socket()** call may still fail if the calling RTP (possibly the kernel) is out of file descriptors. In this case, the **socket()** code will immediately call the back end's **xCloseRtn()** function, to destroy the socket, and return **ERROR**. On the other hand, if a file descriptor is successfully allocated, the **socket()** code stores that file descriptor in the **so_fd** member, and return the file descriptor as its result.

sockLib() itself has no requirements as to how the back end initializes the **struct socket** structure that it allocates; however, the back end will probably want to initialize any members (other than **pSockFuncTbl** and **so_fd**) that it needs, in particular setting **so_bkendaux** to point to any auxiliary private data the back end wishes to maintain for the socket.

xAcceptRtn()

The **xAcceptRtn()** routine has the following prototype:

```
STATUS xAcceptRtn
(
    struct socket ** ppSo,      /* IN: parent socket. OUT: child socket. */
    struct sockaddr * addr,    /* Address of child's peer. */
    int *          addrlen     /* Length of child's peer's address. */
);
```

The back end's **xAcceptRtn()** is called by the **accept()** code in **sockLib**. This **accept()** code does some checking, however, before it calls **xAcceptRtn()**. If **addr** is non-NULL but **addrlen** is NULL, **accept()** returns an error. Otherwise, **accept()** attempts to convert the descriptor passed as its first argument to a **struct socket**. If the descriptor is not a valid socket descriptor, **accept()** again returns an error. If the back end does not provide any accept handler, that is, if the **acceptRtn** member of the back end's **SOCK_FUNC** structure is NULL, **accept()** again returns **ERROR**.

Otherwise, the **accept()** code calls the back end's **xAcceptRtn()** function, passing as the first argument the address of a pointer to the **struct socket**, converted from the socket descriptor passed to **accept()**. The other two arguments to **xAcceptRtn()** are passed directly from the corresponding arguments of **accept()**.

The back end's **xAcceptRtn()** must check that the socket passed in by the first argument is in fact a listening parent socket, capable of providing child socket connections to **accept()**. If not, **xAcceptRtn()** must return **ERROR** and set **errno** appropriately (see *IEEE Std 1003.1*). If the parent can provide child connections, but no completed child socket connection is presently queued for the parent, then **xAcceptRtn()** must do one of the following:

- set **errno** to **EAGAIN** or **EWOULDBLOCK** and return **ERROR**, if the parent socket is non-blocking; or
- if the parent socket is blocking, pend until either a completed child socket connection becomes available (or optionally: until a timeout, signal, or other

implementation defined event occurs, in which case **errno** should be set appropriately, and **ERROR** returned).

If a completed child socket connection becomes available, **xAcceptRtn()** allocates a **struct socket** for it (and any other needed private data structures), and stores a pointer to the child **struct socket** at the address passed in the **ppSo** argument, overwriting the previous pointer to the parent's **struct socket**. If the **addr** argument is non-NULL, **xAcceptRtn()** obtains the child's peer's socket address, and copies it (truncating to the length specified in the **int** pointed to by **addrlen**, if necessary) to the specified address **addr**; and finally storing the actual address length in the **integer** pointed to by **addrlen**. **xAcceptRtn()** then returns **OK**.

If **xAcceptRtn()** does not return **ERROR**, the **accept()** code stores a pointer to the back end's **SOCK_FUNC** table in the child socket's **pSockFuncTbl** member, then goes on to attempt to allocate a socket descriptor from the I/O system for the child socket. If this fails, **accept()** immediately calls the back end's **xCloseRtn()** routine and returns **ERROR**.

Otherwise, **accept()** stores the allocated socket descriptor in the child **struct socket**'s **so_fd** member, and returns that socket descriptor as its result.

xIoctlRtn()

The **xIoctlRtn()** routine has the following prototype:

```
int xIoctlRtn
(
    struct socket * so,      /* the socket */
    u_long         cmd,     /* ioctl command code */
    void *         data     /* ioctl argument */
    void *         mode     /* indicates if the call is from user space */
)
```

The back end's **xIoctlRtn()** routine is called when the I/O system processes an **ioctl()** call made on a socket descriptor. The routine is called as the rule of thumb would suggest, passing the **struct socket** pointer **so** instead of a socket file descriptor and passing the **ioctl** command and data arguments unchanged, except that **xIoctlRtn()** is also passed another argument **mode**. This argument is **NULL** if **ioctl()** was called from the kernel; it is an arbitrary non-NULL value if **ioctl()** was called from a user-space RTP. This indication is intended to help support validation of user-space **ioctl** arguments.

6.3.4 Memory Validation and Socket **ioctls**

Socket APIs are expected to validate their arguments for proper memory access when called from a user-space RTP application. This is accomplished for most socket calls in the **socketScLib** shim library. This contains the system call handlers for sockets API system calls; these handlers execute in the kernel and perform argument memory access validation before calling the appropriate kernel socket APIs in **sockLib**. (Memory validation is done using the **scMemValidate()** routine; see its reference entry for more information.)

Memory validation for the **read()** and **write()** buffer and length arguments is done by the I/O system's system call handler code. However, **ioctl()** calls made on socket descriptors are a special case. The I/O system level does not have knowledge about the form and use of the **ioctl** arguments passed to **ioctl**

operations implemented by the lower-level “driver” code, such as the socket back ends, and so cannot do the memory validation. In VxWorks, the lower-level drivers (including socket back ends) are expected to do memory validation for the `ioctl`s they implement which may be called by RTP applications.

sockScLib provides a pair of functions that can aid a socket back end in doing `ioctl` argument memory validation. These functions should only be called through the following function pointers:

```
int (*pSockIoctlMemVal)
(
    unsigned int cmd,
    void *      data
);

STATUS (*pUnixIoctlMemVal)
(
    unsigned int cmd,
    const void * pData
);
```

The function pointers are only non-NULL when RTP support is included in the VxWorks image. They should only be called when the **mode** argument passed to `xIoctlRtn()` is non-NULL, indicating an `ioctl()` call from user space.

Most (although unfortunately not all) `ioctl` commands on sockets encode the length and usage of the command argument in the `ioctl` command code itself. `Ioctl` codes following the conventions in **target/h/sys/ioctl.h** or **installDir/components/ip_net2-6.x/vxcoreip/include/ipnet/ipioctl.h** encode whether the `ioctl` argument is a pointer to a buffer that is read into the kernel, written to by the kernel, or both; and if so, how large the buffer is. The top-level memory validation for such `ioctl` codes may be performed by calling **pUnixIoctlMemVal()**, passing it the command code and the `ioctl` data argument. This routine returns **OK** if memory validation succeeds, otherwise it sets **errno** appropriately and returns **ERROR**.

There are some limitations of the function referenced by **pUnixIoctlMemVal**:

- It does not check that the `ioctl` code is supported by the back end.
- It assumes without any check that the code follows the conventions encoding the parameter length and usage, as described in **target/h/sys/ioctl.h**.
- It only does top-level checking: if the `ioctl` parameter is a pointer to a buffer holding a data structure that contains additional pointers, these other pointers are not validated. Validating them is the responsibility of the back end.

The **pSockIoctlMemVal()** function pointer behaves similarly to **pUnixIoctlMemVal()**, except that it additionally validates memory for a small number of `ioctl()` codes that do not follow the conventions upon which **pUnixIoctlMemVal()** depends. In the present release, these additional `ioctl` codes are **FIONBIO**, **FIONREAD**, **FIOSELECT**, and **FIOWNSELECT**. It also supports **SIOCMUXPASSTHRU** and **SIOCMUXL2PASSTHRU**, doing second-level validation of the embedded MUX `ioctl` commands in these two `ioctl`s' arguments. For any other `ioctl` code passed to **pSockIoctlMemVal()**, it simply calls **pUnixIoctlMemVal()**.

Working with the 802.1Q VLAN Tag

7.1	Introduction	127
7.2	Adding VLAN Support	128
7.3	About the 802.1Q VLAN Tag Header	128
7.4	MUX Extensions for Layer 2 VLAN Support	129
7.5	Current MUX-L2 Limitations	133
7.6	VLAN Management	134
7.7	Using the MUX-L2 Show Routines	140

7.1 Introduction

This chapter shows you how to work with 802.1Q VLAN tagging in the Wind River Network Stack. It assumes that you are familiar with the principles and operations of IEEE 802.1Q VLAN.



NOTE: 802.1Q VLAN tagging is available in the Wind River Platforms builds of the network stack. The Wind River General Purpose Platform, VxWorks Edition, does not include 802.1Q VLAN tagging.

VLAN tagging is part of the network stack. You can access it by any of the following methods:

- **muxL2...**() routines (if you have built the stack with MUX-L2 support)
- extensions to the socket interface
- a pseudo-interface with which you can manage the VLAN as a subnet



NOTE: Building a bootrom with VLAN is not supported.

7.2 Adding VLAN Support

Include the `INCLUDE_IPNET_USE_VLAN` (**VLAN Pseudo Interface support**) component in your build if you want it to include Layer 2 subnet-based VLANs. If you include this component, this initializes the FreeBSD-style VLAN pseudo-interface for subnet-based VLAN support.

MUX Layer 2 Support

The `INCLUDE_MUX_L2` (**MUX Layer 2 support**) component pulls in the MUX network interface library for layer 2. Including this component initializes the MUX-L2 infrastructure for VLAN support.

To allow the network stack to interoperate with MUX-L2, you must rebuild it with the `IPCOM_VXWORKS_USE_MUX_L2` flag by using one of the following methods:

- Enable the `IPCOM_VXWORKS_USE_MUX_L2` `#define` found in `ipcom/port/vxworks/config/ipcom_pconfig.h`
- Build with the flag `ADDED_CFLAGS+=-DIPCOM_VXWORKS_USE_MUX_L2`

The `INCLUDE_MUX_L2` component requires the following components:

- `INCLUDE_END` (**END interface support**)
- `INCLUDE_ETHERNET` (**Ethernet multicast library support**)

The `INCLUDE_MUX_L2` component contains the following configuration parameters:

`MUX_L2_MAX_VLANS_CFG` (**Maximum number of 802.1Q VLANs supports**)
the maximum number of 802.1Q VLANs supported on the target (default = 16)

`MUX_L2_NUM_PORTS_CFG` (**Number of ports that the device has**)
the maximum number of physical ports available to the target (default = 16)

L2Config

The `INCLUDE_L2CONFIG` (**l2config**) component provides support for the layer 2 configuration utility. If you include this component, this initializes the L2 command-line configuration utility. This component requires the `INCLUDE_MUX_L2` component.

7.3 About the 802.1Q VLAN Tag Header

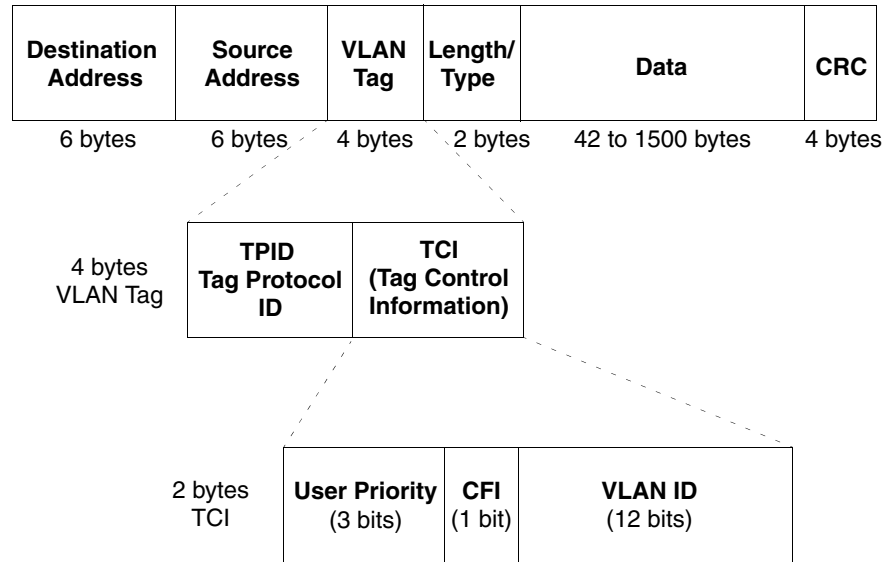
The Wind River VLAN implementation supports the following three frame types:

- **Untagged frames** – frames that do not carry any identification of the VLAN to which they belong
- **Priority-tagged frames** – frames that include a tag header carrying explicit user priority information but not identifying the frames as belonging to a specific VLAN
- **VLAN-tagged frames** – frames that include an explicit identification of the VLAN to which they belong

The VLAN tag header is as shown in Figure 7-1.

The 802.1Q tag is a four-byte field after the six-byte Source Address field and before the two-byte length/type field in the Ethernet header. An 802.1Q VLAN tagging implementation indicates that a frame is tagged by setting its Type field to the VLAN Identifier Protocol (0x8100). This means the next two bytes contain Tag Control Information.

Figure 7-1 VLAN Tag Header Format on Ethernet



The two-byte Tag Control Information consists of a 3-bit priority (0-7) value, a Canonical Format Indicator (CFI) field (0 for Ethernet), and a 12-bit VLAN Identifier (VID). The 12-bit VID field can take any value from 0 to 4095, but two of these values have special meanings according to the 802.1Q specification: The value of all ones (0xFFF) is reserved but currently unused; the value of all zeros (0x000) indicates a that the frame is priority-tagged and that no VID is present in the frame.

7.4 MUX Extensions for Layer 2 VLAN Support

This section describes how to programmatically control MUX-L2 VLAN support if you have built your stack with MUX-L2 interoperability.

Overview of MUX-L2 VLAN Management

The MUX-L2 extensions allow you to manage the VLAN membership for a VxWorks target. These extensions support the following 802.1Q characteristics:

- VLAN classification of untagged, priority-tagged, or VLAN-tagged ingress frames.

- Port-based VLAN classification as the default ingress rule (that is, all untagged and priority-tagged frames that a port receives are classified as belonging to the VLAN whose VID is associated with that port).
- Tagging of egress frames as VLAN-tagged, priority tagged, or untagged frames on a per-port basis.
- Enabling a port to be a member of multiple VLANs.
- Ingress Filter and Ingress Acceptable Frame Type configuration on a per-port basis.
- Ingress Filter configuration on a per-port basis.
- Ingress Acceptable Frame Type configuration on a per-port basis.
- The ability to send untagged frames for some VLANs and VLAN-tagged frames for others on a per-port basis.
- The assignment of all VLAN-enabled ports to the default PVID of 1. The PVID value of a port is configurable.

You can access and control this functionality through the **muxL2...**() routines, through a socket interface, or with a VLAN pseudo-interface.

7.4.1 Enabling VLAN Support for a Port

For an END device loaded to MUX, call the **muxL2PortAttach**() routine to enable VLAN support for the port.



NOTE: The **muxL2PortAttach**() routine assumes an Ethernet device. If you are using a non-Ethernet device, call **muxL2PortAltAttach**() instead.

The **muxL2PortAttach**() routine prepares the port for VLAN support as follows:

- It joins the port to the default VLAN with a VID of 1, according to the 802.1Q requirement. It also configures the port to transmit untagged frames on the default VLAN. You can change the egress tagging state for the default Port VLAN ID (PVID) by calling **muxL2Ioctl**() with the **MUXL2IOCSPORTVLAN** control command.
- It initializes the port-specific attributes with the following defaults:
 - Default Port User priority: 0
 - Ingress Acceptable Frame Filter Type: admits all frame types
 - Ingress Filter: False
- It queries the hardware for its VLAN capabilities.

A driver that supports hardware VLAN tagging must indicate this by setting flags in the **cap_available** member of the **END_CAPABILITIES** structure the driver returns in response to a **EIOCGIFCAP** message to its **xIoctl**() routine (see [5.4.3 xIoctl](#)(), p.72). Those flags are as follows:

- **IPCOM_IF_DRV_CAP_VLAN_MTU** – indicates that the driver can handle slightly larger than normal frames (that is, frames with a VLAN tag). This notifies the MUX-L2 that it can leave the MTU for the port at the normal setting. If the **IPCOM_IF_DRV_CAP_VLAN_MTU** flag is not set and

software VLAN-tagging is required, the MUX-L2 decreases the hardware MTU by 4-bytes.

- **IPCOM_IF_DRV_CAP_VLAN_HWTAGGING_TX** – indicates that the driver can insert the VLAN tag into a frame on egress, by using the information that it reads from the **pkt->link_cookie** field in host-byte order.
- **IPCOM_IF_DRV_CAP_VLAN_HWTAGGING_RX** – indicates that the driver can strip the VLAN tag from a received frame and store that tag information in the **pkt->link_cookie** field in host-byte order.
- It determines which type of Ethernet address format the device can support (Ethernet Type 2 encapsulation or 802.3 style length encapsulation). This information is required when the MUX-L2 assembles an Ethernet header for the egress frame.
- It saves the address of the original END driver's **pFuncTable** function table. The MUX-L2 will restore the original driver's function table during **muxL2PortDetach()**.
- It replaces the driver's registered **xPacketDataGet()** with **muxL2IngressClassify()**, and the **xFormAddress()** function pointer with **muxL2EgressClassify()**. For more information about MUX-L2 ingress and egress frame processing, see [7.4.3 MUX-L2 Ingress Rules](#), p.131, and [7.4.4 MUX-L2 Egress Rules](#), p.132.



NOTE: As an alternative to **muxL2PortAttach()**, you can call **muxL2Ioctl()** using the **MUXL2IOCSPORTATTACH** control command.

7.4.2 Disabling VLAN Support for a Port

To disable VLAN support for a port, call **muxL2PortDetach()**. This routine detaches the port from the MUX-L2, removes the port from all the VLAN memberships it has joined, and restores the original driver's function table. If the port is removed from the MUX, **muxDevUnload()** calls **muxL2PortDetach()** as well.

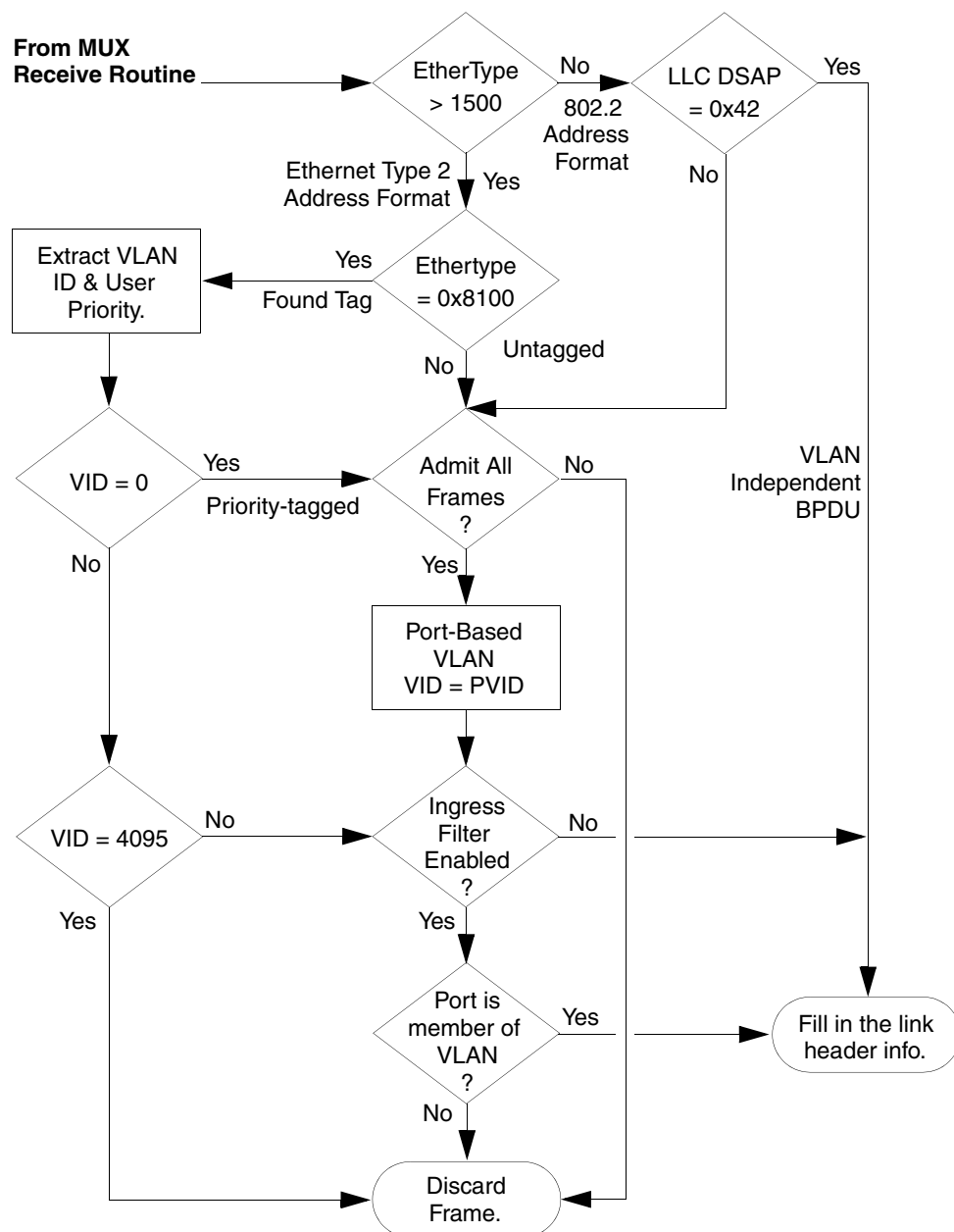


NOTE: As an alternative to **muxL2PortDetach()**, you can call **muxL2Ioctl()** using the **MUXL2IOCSPORTDETACH** control command.

7.4.3 MUX-L2 Ingress Rules

When a frame arrives on a port, the driver's ISR schedules the frame processing work to **tNet0**. The MUX receive routine would normally schedule a call to the driver's **xPacketDataGet()** to separate the address information and data in the frame. However, for a VLAN-enabled port, the MUX receive routine schedules a call to **muxL2IngressClassify()** to filter the received frame according to its VLAN header tag. [Figure 7-2](#) shows how the ingress filter handles an incoming frame.

Figure 7-2 MUX-L2 Ingress Rules



7.4.4 MUX-L2 Egress Rules

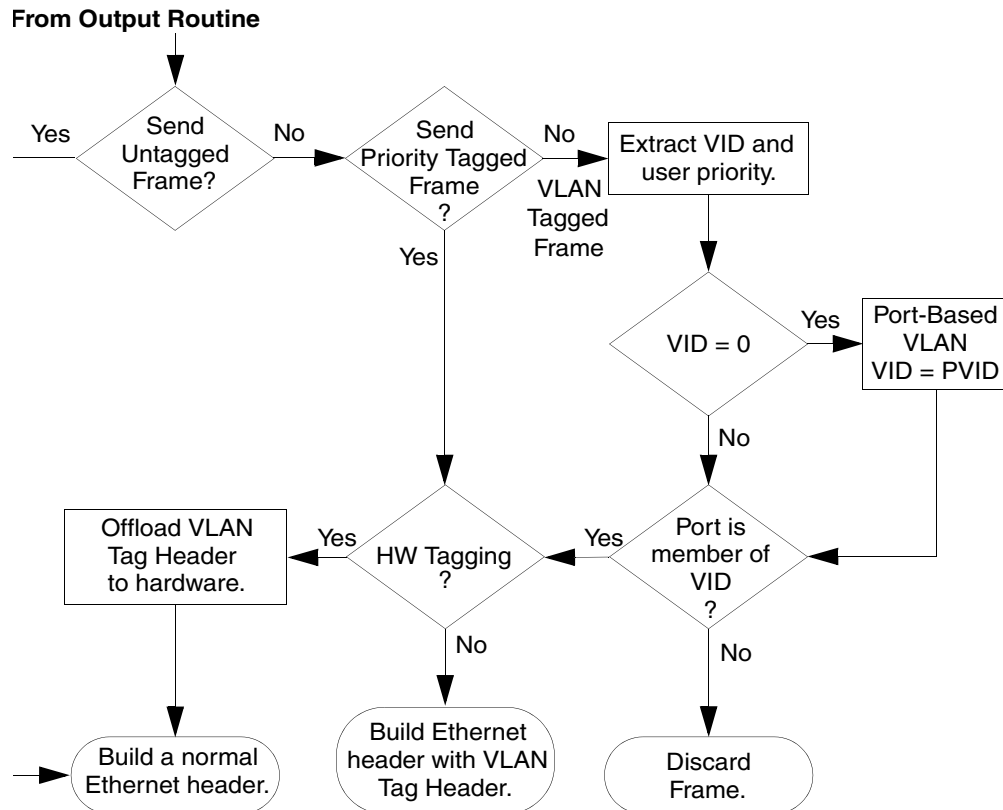
When the MUX needs to transmit a frame, it normally calls the driver's `xFormAddress()` routine to create and prepend a link-layer-appropriate frame header to the `M_BLK` chain containing outgoing data. However, for packets transmitted over a VLAN-enabled port, 802.1Q requires some additional pre-processing.

In addition, 802.1Q requires that a port transmits only VLAN-tagged frames or untagged frames but never transmits using both formats for the same VLAN. To

support the egress tagging decision on a per-port per-VLAN basis, the MUX-L2 keeps track of the per-port egress tagging configuration for each VLAN.

To evaluate an outgoing frame in accordance with this information, the MUX calls **muxL2EgressClassify()** to determine whether the outgoing frame should be tagged or untagged, and then to build the link-layer header based on the tagging decision. Figure 7-3 shows how the egress filter handles an outgoing frame.

Figure 7-3 MUX-L2 Egress Rules



7.4.5 Accessing the MUX L2 Control Routines

Call **muxL2Ioctl()** to access the MUX-L2 control functionality. These control functions let you do such things as retrieve port information and set the ingress frame filter type. For more information, see the **muxL2Ioctl()** reference entry.

7.5 Current MUX-L2 Limitations

The current MUX-L2 implementation has the following known limitations:

- There is no support for the automatic distribution of VLAN configuration using the GARP VLAN Registration Protocol (GVRP). The MUX-L2 support for VLAN is limited to those VLANs that are created statically.

- The MUX-L2 does not configure and operate the address learning process described in the 802.1Q specification. Therefore, it is not capable of broadcasting or multicasting frames to multiple ports belonging to a VLAN.
- Although the MUX-L2 allows the 802.1P User Priority to be specified with an egress VLAN-tagged frame, it does not support user priority to traffic class mapping described in the 802.1Q specification. It also does not provide any mechanism to perform the ingress user priority regeneration as described in the 802.1Q specification.
- Although the MUX-L2 implements selected RFC 2674 static VLAN objects, the VLAN configuration methodology is not compatible with the RFC 2674 MIB. RFC 2674 VLAN management is VLAN-centric and requires a port list bitmap specifying the ports belonging to a VLAN. The VLAN management for the MUX-L2 is port-centric and achieves VLAN configuration on a per-port basis.
- The support for 802.1Q VLAN tagging is currently implemented for END drivers only.
- The MUX-L2 implementation is provided in the context of Ethernet as the underlying data link technology. Because the fundamental VLAN operation and behavior are independent of the underlying data link, the implementation can be easily modified to adapt to a non-Ethernet environment.
- Wind River Learning Bridge is not compatible with the MUX-L2.

7.6 VLAN Management

The following subsections describe two mechanisms with which you can manage a VLAN:

- [7.6.3 Socket-Based VLAN Management](#), p.138
- [7.6.2 Subnet-Based VLAN Management](#), p.135

7.6.1 MUX-L2 VLAN Management

If you enable MUX-L2 functionality when you build the network stack, you can use the **l2config** utility to access the Layer 2 set routines supported by **muxL2Ioctl()**. The **l2config** utility does not give you access to the **muxL2Ioctl()** “get” functionality, which, because of its use of structures, is more suited to programmatic use. However, you can access much of the same information from the command line using **muxL2Show()** or **muxL2VlanShow()**, which are described in their respective reference entries and in [7.7 Using the MUX-L2 Show Routines](#), p.140.

For port-oriented configuration needs, **l2config** allows you to do the following:

- Attach/detach a port to/from MUX-L2.
- Set the default port VID (PVID).
- Set the default user priority.

- Set the acceptable ingress frame type.
- Set the ingress frame filter.

For configuration needs involving both the port and a VLAN, **l2config** allows you to do the following:

- Join a port to a VLAN.
- Set the egress frame type for the VLAN.
- Leave a VLAN that a port joined previously.

For more information, see the examples below and the reference entry for **l2config**.

Sample MUX-L2 Configuration

The following example attaches the **fei1** port to MUX-L2:

```
-> l2config "vlandev fei1 attach"
value = 0 = 0x0
```

The following example enables the ingress frame filter for port **fei1**. It also sets the **fei1** ingress acceptable frame filter type to **ADMIT_TAGGED_ONLY_FRAMES**:

```
-> l2config "vlandev fei1 infilter on ingress admittag"
value = 0 = 0x0
```

The following example joins the **fei1** port to the VLAN with VID 20 and configures the egress frame type for the VLAN to transmit VLAN-tagged frames only:

```
-> l2config "vlandev fei1 join vid 20 egress tagged"
value = 0 = 0x0
```

The following example removes the **fei1** port from VLAN 20, the VLAN to which it was joined previously:

```
-> l2config "vlandev fei1 leave vid 20"
value = 0 = 0x0
```

The following example shows how you can issue multiple command options at the same time, using **l2config**. The attach and join commands above can be combined into a single command:

```
-> l2config "vlandev fei1 attach join vid 20 egress tagged"
value = 0 = 0x0
```

7.6.2 Subnet-Based VLAN Management

To support VLAN routing, FreeBSD uses the VLAN pseudo-interface to demultiplex VLAN-tagged frames into logical VLAN network interfaces. The Wind River Network Stack adapts this technique to support subnet-based VLAN configuration.

Each VLAN pseudo-interface can be created at run time by using the **ifconfig()** **create** command. For each pseudo-interface, call **ifconfig** to assign a VLAN, a parent interface, and a numeric VID. The parent interface must be a physical interface that is attached to the IP layer at the time you create the VLAN pseudo-interface.

A single parent interface can support multiple VLAN pseudo-interfaces provided that the pseudo-interfaces have different VIDs. The parent interface must be a member of the VLAN for the VID assigned to the VLAN pseudo-interface.

To configure the VLAN pseudo-interface, use the following three **ifconfig** options:

vlanInterfaceName **create**

Create the specified VLAN pseudo-interface named by *vlanInterfaceName*. *vlanInterfaceName* must start with the letters "vlan"; for example: **vlan**, **vlan10**, or **vlanPrivate**.

vlanInterfaceName **destroy**

Delete the specified VLAN pseudo-interface from the network stack.

vlanInterfaceName **vlan** *vlanID* **vlanif** *interfaceName* **vlanpri** *priority*

Set the VLAN ID to *vlanID*, associate the physical interface *interfaceName* with *vlanInterfaceName*, and assign the Class Of Service value *priority* to the VLAN tag for this VLAN pseudo-interface. *vlanID* is a 16-bit number between 1-4094 that is used to create an 802.1Q VLAN header for packets the stack sends from the VLAN pseudo-interface. *priority* is a three-bit value.

Packets transmitted through the VLAN interface will be diverted to the physical interface *interfaceName* with 802.1Q VLAN encapsulation. Packets with 802.1Q encapsulation received by the physical interface with the correct VLAN ID will be diverted to the associated VLAN pseudo-interface. The VLAN interface is assigned a copy of the physical interface's flags and Ethernet address. If the VLAN interface already has a physical interface associated with it, this command fails. To change the association to another physical interface, you must first clear the existing association.

Note that you must set **vlan** and **vlanif** at the same time.

Consequences of Changing the VID

The Wind River Network Stack allows you to change the parent interface and the VID only when the VLAN interface is down. For example, to change the parent interface to **fei1** and the VID for the VLAN pseudo-interface to **1234** on a created, configured, and up VLAN interface, type the following:

```
-> ifconfig vlanLab down vlanif fei1 vlan 1234 up
```

Be aware that changing the VID for the VLAN pseudo-interface does not automatically remove the parent interface from the VLAN membership associated with the old VID. It also does not automatically add the parent interface to the member set specified by the new VID. Therefore, the parent interface must be a member of the VLAN specified by the new VID before the new VID can be assigned. The parent interface remains a member of the VLAN specified by the old VID unless the membership is explicitly removed.

Example of Subnet-Based VLAN Management

The following examples show how to create VLAN pseudo-interfaces. The first examples rely on the compact interface naming style. The examples after that rely on the restrictive interface naming style.

If you have built the network stack to support MUX-L2, when you create the VLAN pseudo-interface the network stack will implicitly attach the physical parent interface to MUX-L2 and will join the parent interface to the VLAN that you have configured for the VLAN pseudo-interface. Once the stack joins the port to MUX-L2, MUX-L2 enforces strict ingress and egress VLAN rules (as described in sections [7.4.4 MUX-L2 Egress Rules](#), p.132 and [7.4.5 Accessing the MUX L2 Control](#)

[Routines](#), p.133). When you attach the parent physical interface (port) to MUX-L2, you can manage the port by calling `l2config()` (for instance, to set the port ingress or egress properties).

Examples Using the Compact Creation API

The following example uses the compact interface naming style to create a VLAN pseudo interface `fei1.50` with IP address 190.0.2.234/24. It also specifies the parent interface `fei1` and VID 50 for the VLAN pseudo-interface. You must have already attached `fei1` to the network stack.

```
-> ifconfig "fei1.50 create inet 190.0.2.234/24"
value = 0 = 0x0
-> ifconfig "fei1.50"
fei1.50: flags=48043<UP,BROADCAST,RUNNING,MULTICAST,INET_UP> mtu 1496
      inet 190.0.2.234 netmask 0xffffffff broadcast 190.0.2.255
      ether 00:08:c7:c9:24:76
      vlan: 50 user priority: 0 parent interface: fei1
value = 0 = 0x0
```

The following example destroys the `fei1.50` VLAN pseudo interface previously created.

```
-> ifconfig "fei1.50 destroy"
value = 0 = 0x0
```

The following example uses the compact interface naming style to create a VLAN pseudo interface, `fei0.20`, with IP address 190.0.4.234/24. It also specifies the parent interface `fei0`, VID 20, and user priority 5 for the VLAN pseudo-interface. You must have already attached `fei0` to the network stack.

```
-> ifconfig "fei0.20 create"
value = 0 = 0x0

-> ifconfig "fei0.20 190.0.4.234/24"
value = 0 = 0x0

-> ifconfig "fei0.20 vlanpri 5"
value = 0 = 0x0

-> ifconfig "fei0.20"
fei0.20: flags=48043<UP,BROADCAST,RUNNING,MULTICAST,INET_UP> mtu 1496
      inet 190.0.4.234 netmask 0xffffffff broadcast 190.0.4.255
      ether 00:03:47:b0:d7:17
      vlan: 20 user priority: 5 parent interface: fei0
value = 0 = 0x0
```

Examples Using the Name-Restrictive Creation API

The following example creates a VLAN pseudo-interface, `vlan0`, with IP address 190.0.2.123/24, assigns the pseudo-interface with VID 20, and associates it with the parent interface `fei1`. You must have already attached `fei1` to the network stack.

```
-> ifconfig "vlan0 create"
value = 0 = 0x0

-> ifconfig "vlan0"
vlan0 Link type:Layer 2 virtual LAN HWaddr
00:01:02:03:04:10 Queue:none
      vlan: 20 parent: fei0
      inet 190.0.2.123 mask 255.255.255.0 broadcast 190.0.2.255
      inet 224.0.0.1 mask 240.0.0.0
      inet6 unicast FE80::201:2FF:FE03:410%vlan0
prefixlen 64 automatic
      inet6 unicast FE80::%vlan0 prefixlen 64 anycast
      inet6 multicast FF02::1%vlan0 prefixlen 16 automatic
      inet6 multicast FF02::1:FF03:410%vlan0 prefixlen 16
      inet6 multicast FF02::1:FF00:0%vlan0 prefixlen 16
```

```
UP RUNNING SIMPLEX BROADCAST MULTICAST
MTU:1496 metric:0 VR:0
RX packets:0 mcast:0 errors:0 dropped:0
TX packets:13 mcast:15 errors:0
collisions:0 unsupported proto:0
RX bytes:0 TX bytes:1138
-> ifconfig "vlan0 vlan 20 vlanif fe1"
value = 0 = 0x0

-> ifconfig "vlan0 190.0.2.123/24"
value = 0 = 0x0

-> ifconfig "vlan0"
vlan0: flags=48043<UP,BROADCAST,RUNNING,MULTICAST,INET_UP> mtu 1496
      inet 190.0.2.123 netmask 0xffffffff broadcast 190.0.2.255
      ether 00:08:c7:c9:24:76
      vlan: 20 user priority: 0 parent interface: fe1
value = 0 = 0x0
```

The following example destroys the **vlan1** VLAN pseudo-interface previously created.

```
-> ifconfig "vlan0 destroy"
value = 0 = 0x0
```

7.6.3 Socket-Based VLAN Management

Wind River extends the socket API to include a new socket option, **SO_VLAN**, and a new structure, **sovlan**. These provide a mechanism that can carry all the information relevant to VLAN configuration. If you treat the VID and the user priority as socket-level configuration options, you can use these extensions by calling **getsockopt()** and **setsockopt()** and thus get or set VLAN membership information.

SO_VLAN can only be used for send operations.

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

```
struct sovlan
{
    /*
     * If so_onff is set, the vlan id and/or user priority will be copied
     * to the socket structure and SO_VLAN so_option will be set. If so_onff
     * is not set, the SO_VLAN so_option for the socket will be cleared.
     */
    int          vlan_onoff; /* on/off option */

    /*
     * The priority_tagged boolean must be set to true if application using
     * socket-based vlan requires to egress 802.1P priority-tagged frame
     * (i.e. the value of vid is zero). Defaults to false. If set to true,
     * the value specified by the vid will be ignored.
     */
    BOOL         priority_tagged;

    unsigned short vid;          /* VLAN ID, valid vid: 1..4094 */
    unsigned short upriority;    /* User Priority, valid priority: 0..7 */
};
```

After an application creates a socket, it can call **setsockopt()** to configure the VID and/or user priority for the socket. In order to transmit a VLAN-tagged or priority-tagged frame, the port/interface that the socket is bound to must already be attached to the MUX-L2, as described previously. If a port transmits a VLAN-tagged frame, the port must also be a member of the VLAN for which the socket-based VLAN is configured.

Setting User Priority for Transmitted Priority-Tagged Frames

The following code fragment is an example of how to call `setsockopt()` in order to configure the user priority for a socket and configure that socket to transmit priority-tagged frames.

```
struct sovlan vl;

/* set up the vlan_onoff to indicate that the VID and or User Priority
 * are valid */

vl.vlan_onoff = 1;
/*
 * Informs lower-layers (such as subnet-based VLAN) that the
 * information provided is for Priority-tagged frame and that lower-layers
 * must not alter the VLAN control information for VID configuration
 */

vl.priority_tagged = TRUE;

/* VID is not applicable for Priority-tagged frame */

vl.vid = 0;

/* Configure the Priority-tagged frame for User Priority with value 7 */

vl.upriority = 7;
if (setsockopt (s, SOL_SOCKET, SO_VLAN, (char *) &vl, sizeof (struct sovlan))
    < 0)
    printf( "setsockopt SO_VLAN for socket %d failed\n", s);
```

Setting User Priority for Transmitted VLAN-tagged Frames

The following code fragment is an example of how to call `setsockopt()` in order to configure the user priority for the specified socket and to configure that socket to transmit VLAN-tagged frames.

```
struct sovlan vl;

/* setup vlan_onoff to indicate that VID and/or User Priority * are valid */

vl.vlan_onoff = 1;

/*
 * Informs lower-layers (such as Subnet-based VLAN) that the
 * information provided is for VLAN-tagged frame and that lower-layers should
 * alter the VLAN control information for VID if the VID is not specified
 */

vl.priority_tagged = FALSE;

/*
 * Specifies VID with value of 0 to allow lower-layers (such as Subnet-
 * based VLAN) to insert the appropriate VID to the VLAN
 * control information for the outgoing VLAN-tagged frame.
 */

vl.vid = 0;

/* Configure the VLAN-tagged frame for User Priority with value 3 */
vl.upriority = 3;

if (setsockopt (s, SOL_SOCKET, SO_VLAN, (char *)&vl, sizeof (struct sovlan))
    < 0)
    printf ("setsockopt SO_VLAN for socket %d failed\n", s);
```

Clearing the VLAN Configuration for the Socket

Consider the following code fragment calls **setsockopt()** to clear the VLAN configuration for a socket:

```
struct sovlan vl;

bzero ((char *) &vl, sizeof (struct sovlan));

/* reset all the VLAN control information for the socket */

vl.vlan_onoff = 0;

if (setsockopt (s, SOL_SOCKET, SO_VLAN, (char *) &vl, sizeof (struct sovlan))
    < 0)
    printf ("setsockopt SO_VLAN for socket %d failed\n", s);
```

Getting Configuration Information

The following code fragment is an example of how to call **getsockopt()** to retrieve the VLAN configuration for a socket.

```
struct sovlan vl;
int vsize = sizeof (struct sovlan);

bzero ((char *) &vl, sizeof (struct sovlan));
if (getsockopt (s, SOL_SOCKET, SO_VLAN, (char *) &vl, &vsize) < 0)
    printf ("getsockopt SO_VLAN for socket %d failed\n", s);
if (0 == vl.vlan_onoff)
    printf ("No VLAN control info for socket %d\n", s);
else
{
    if (vl.priority_tagged)
        printf ("Socket %d Priority-Tagged User Priority %d\n", s,
            vl.upriority);
    else
        printf ("Socket %d VLAN-Tagged VID %d User Priority %d\n", s, vl.vid,
            vl.upriority);
}
```



NOTE: For the VLAN-tagged frame, if **getsockopt()** returns a VID value of 0, this implies that the application uses the socket-based VLAN, chooses not to configure the VID, and relies on the lower-layers (such as subnet-based VLAN) to insert the appropriate VID to the VLAN control information for the outgoing VLAN-tagged frame.

7.7 Using the MUX-L2 Show Routines

For debugging and diagnostic purposes, the MUX-L2 provides **muxL2Show()**, **muxL2StatShow()**, **muxL2VlanShow()**, and **muxL2VlanStatShow()**.

Example 7-1 **muxL2Show()**

Call **muxL2Show()** to display the configuration of ports registered with the MUX-L2.

```
-> muxL2Show
max number of physical ports: 16
max number of vlans device supports: 16
number of vlans configured in device: 3
```

```

number of ports attached to MUX-L2: 2

Device: <fei> Unit: <1> L2 Port Object: 0x10cc040 endObjId: 1
Port VID (PVID): 1 Port User Priority: 0
Port ingress filter: TRUE
Port ingress acceptable frame filter type: Admit vlan-tagged frames only
Hardware supports vlan tagging: FALSE
Hardware supports vlan mtu: FALSE
Number of vid configured for the port: 2
Port VLAN membership:
    VID 1          Egress: untagged-tagged
    VID 20         Egress: vlan-tagged

Device: <fei> Unit: <0> L2 Port Object: 0x10e40a0 endObjId: 2
Port VID (PVID): 1 Port User Priority: 0
Port ingress filter: TRUE
Port ingress acceptable frame filter type: Admit All Frames
Hardware supports vlan tagging: FALSE
Hardware supports vlan mtu: FALSE
Number of vid configured for the port: 2
Port VLAN membership:
    VID 1          Egress: untagged-tagged
    VID 100        Egress: vlan-tagged
value = 1 = 0x1

```

Example 7-2 muxL2VlanShow()

Calls **muxL2VlanShow()** to display the VLAN configurations maintained by the MUX-L2 on a per-VLAN basis.

```

-> muxL2VlanShow

VLAN 1: Number of Members: 2 Egress Untagged: 2
VLAN 20: Number of Members: 1 Egress Untagged: 0
VLAN 100: Number of Members: 1 Egress Untagged: 0

Port to device name mapping:
    Port 1      -> fei1
    Port 2      -> fei0

VLAN Membership information:
(Legend: 'M' = Member '-' = Unspecified)
    VLAN 1      : MM-----
    VLAN 20     : M-----
    VLAN 100    : -M-----

VLAN Egress Frame Type:
(Legend: 'T' = Vlan-Tagged 'U' = Untagged '-' = Unspecified)
    VLAN 1      : UU-----
    VLAN 20     : T-----
    VLAN 100    : -T-----

value = 1 = 0x1

```

Example 7-3 muxL2StatShow() and muxL2VlanStatShow()

The MUX-L2 also maintains various VLAN statistics on a per-port per-VLAN basis. These statistics are disabled by default for performance reasons. To include these statistics, build MUX-L2 with **MUX_L2_VLAN_STATS** defined. If VLAN statistics are included, **muxL2VlanStatShow()** and **muxL2StatShow()** can be used to monitor the traffic on a per-port per-VLAN basis.

```

-> muxL2StatShow
fei1 Port Statistics:
    Number of received frames discarded due to non-vlan
    reasons (i.e. Discard Ingress Filtering): 9
    Number of egress frames discarded due to Egress
    Rules violation: 0

```

```
VLAN 1 staticstics:
Number of frames received: 0
Number of frames transmitted: 0
Number of received frames discarded: 0

VLAN 20 staticstics:
Number of frames received: 166
Number of frames transmitted: 226
Number of received frames discarded: 0

fei0 Port Statistics:
Number of received frames discarded due to non-vlan
reasons (i.e. Discard Ingress Filtering): 0
Number of egress frames discarded due to Egress
Rules violation: 0

VLAN 1 staticstics:
Number of frames received: 0
Number of frames transmitted: 0
Number of received frames discarded: 0

VLAN 100 staticstics:
Number of frames received: 393
Number of frames transmitted: 486
Number of received frames discarded: 0

value = 0 = 0x0

-> muxL2VlanStatShow
fei1 VLAN 1 staticstics
    Number of frames received: 0
    Number of frames transmitted: 0
    Number of received frames discarded: 0

fei0 VLAN 1 staticstics
    Number of frames received:
    Number of frames transmitted: 0
    Number of received frames discarded: 0

fei1 VLAN 20 staticstics
    Number of frames received: 166
    Number of frames transmitted: 226
    Number of received frames discarded: 0

fei0 VLAN 100 staticstics
    Number of frames received: 393
    Number of frames transmitted: 486
    Number of received frames discarded: 0

value = 0 = 0x0
```


Quality of Service

- 8.1 Introduction 143
- 8.2 Differentiated Services 144
- 8.3 Network Interface Output Queues 152

8.1 Introduction

Quality of Service (QoS) refers to the capability of a network to treat some traffic flows differently than others. The most common usage is to give a specific traffic flow a better service than the normal best-effort service.



NOTE: The QoS feature is available only in the Wind River Platforms builds of the network stack. The Wind River General Purpose Platform, VxWorks Edition, does not support QoS.

Different types of flow have different requirements; for example, some flows have restrictions in latency, while others have restrictions in minimum bandwidth, and so on. For example:

- Interactive traffic, like telnet and SSH, performs better with low latency so that the user does not experience delay when typing.
- FTP traffic performs better when it can use as much bandwidth as possible; it does not matter if the latency is high or if the data arrives in bursts.

Telnet will have a high latency if routers treat all traffic equally, which is the default “best effort” behavior used by most routers. But routers could improve latency if each router identifies the two varieties of flow and lets the telnet packets move ahead of FTP packets that the router has already queued to send on the outgoing interface. Doing so will ordinarily have little effect on the FTP application, since telnet normally uses little bandwidth.

8.2 Differentiated Services

The Differentiated Services (DiffServ) architecture is based on a model in which an edge router classifies traffic entering a network, possibly conditions it (for instance to reduce jitter or latency), and assigns it to different behavior aggregates. It assigns packets to behavior aggregates by setting a single differentiated-services (DS) *codepoint* in the value of the *DS field* of the IP datagram (the TOS field for IPv4 or the traffic-class field for IPv6). The core routers of the network then may forward these packets according to the per-hop behavior they associate with each DS codepoint.

To control, create, and delete interface output queues, use the API that is defined in the following file:

```
installDir/components/ip_net2-6.x/ipnet2/include/ipnet_qos.h
```

8.2.1 Including DiffServ in a Build

To include DiffServ in a VxWorks build, include the DiffServ build components listed below. You can do this through either Workbench or the **vxprj** command-line utility.

The following six build components enable DiffServ (there are no build parameters for any of the components):

Differentiated Services (INCLUDE_IPNET_DIFFSERV)

The main component for differentiated services.

Classifier (INCLUDE_IPNET_CLASSIFIER)

Classifier component.

Simple Marker (INCLUDE_IPNET_DS_SM)

Component for the simple marker (see [SimpleMarker](#), p.151).

Single Rate Three Color Marker (INCLUDE_IPNET_DS_SRTCM)

Component for the single-rate, three-color marker (see [Single-Rate Three-Color Marker](#), p.151).

IPCOM QoS commands (INCLUDE_IPQOS_CMD)

Enables the use of shell commands for configuring DiffServ.

IPCOM output queue commands (INCLUDE_IPQUEUE_CONFIG_CMD)

Enables the use of shell commands for configuring output queues.

8.2.2 Using DiffServ

You can configure a DiffServ traffic classifier to run either in behavior aggregate mode or in multifield mode:

- In behavior aggregate mode, the classifier looks only at the DS field (the TOS field in IPv4, or the traffic class for IPv6).
- In multifield mode, the classifier may look at any field supported by the IPNET classifier—this mode is slower but more flexible.

To run in behavior aggregate mode, define the macro **IPNET_DIFFSERV_CLASSIFIER_MODE_BA** in the file

installDir/components/ip_net2-6.x/ipnet2/config/ipnet_config.h; undefine this macro to run in multifield mode.

Adding a Filter Rule for a Meter/Marker Entity

To add a filter rule for a meter/marker entity when running DiffServ in multifield mode, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCXADSFILTER, &filter);
```

Where **filter** is a **ds_filter** object that describes the filter you are adding (see [ds_filter Class](#), p.147). The network stack chooses an ID for the filter and stores it in the **id** member of this object. Use this ID number if you need to refer to this specific filter, for instance if you attach it to a DiffServ meter/marker entity.

You can also use a **qc** command, which has the following format:

```
qc filter add dev device parent queueID handle filterID [filterArgs] flowid queueID
```

The arguments to this command are as follows:

dev device

The device to which you are attaching the filter, for instance **eth0**.

parent queueID

The identifying number of the container queue to which you are adding the filter.

handle filterID [*filterArgs*]

The *filterID* is the identifying number of the filter. You may use the following arguments in the *filterArgs* argument to describe the filter:

proto number

tclass number

srcport range

dstport range

srcaddr address[/*prefix*]

dstaddr address[/*prefix*]

flowid queueID

The identifying number of the destination queue for packets that match the filter.

For example:

To add a filter identified by the number five to the container queue identified by the number one, so that all TCP packets (protocol number six) are filtered into the queue identified by the number 31, use the following **qc** command:

```
> qc filter dev eth0 parent 1 handle 5 proto 6 flowid 31
```

To add a second filter (identified by the number three) to the same container queue that filters all UDP packets (protocol number 17) that are sent to 2001::/16 into the same queue, use the following **qc** command:

```
> qc filter dev eth0 parent 1 handle 3 proto 17 srcaddr 2001::/16 flowid 31
```

Deleting a Filter Rule from a Meter/Marker Entity

To delete a filter rule from a meter/marker entity when running DiffServ in multifield mode, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCXDDSFILTER, &filter);
```

In this call, **filter** is a **ds_filter** object that describes the filter you are deleting (see [ds_filter Class](#), p.147). You only need to set the **id** member of this object in order to specify which filter you want to delete.

You can also use a **qc** command, which has the following format:

```
# qc filter del filterID
```

Creating a Meter/Marker Entity

To create a meter/marker entity, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCXDSCREATE, &entity);
```

In this call, **entity** is an object of the **ds** superclass (see [ds Class](#), p.148). The network stack will fill in the **id** field of this object. Use this identifying number to identify this entity in future calls.

See [8.2.4 Creating New Meter/Marker Entity Varieties](#), p.149, for descriptions of the meter/marker entities that are part of the Wind River Network Stack and for instructions on how to add new entities.

Deleting a Meter/Marker Entity

To delete a meter/marker entity, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCXDSEDESTROY, &entity);
```

In this call, **entity** is an object of the **ds** superclass (see [ds Class](#), p.148), but you need only fill in the **id** member of this object in order to sufficiently identify the entity you want to delete.

Mapping a Filter to a Meter/Marker Entity

To map a filter (or DS codepoint if you are operating in multifield mode) to a meter/marker entity, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCXADSMAP, &mapping);
```

In this call, **mapping** is an object of the **ds_map** class that describes the mapping you are establishing (see [Mapping from a Filter Rule to a Meter Marker Entity](#), p.149).

Removing a Filter-to-Meter/Marker Entity Mapping

To remove a mapping between a filter (or DS codepoint if you are operating in multifield mode) and a meter/marker entity, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOXDDSMAP, &mapping);
```

In this call, **mapping** is an object of the **ds_map** class that describes the mapping that you are removing (see [Mapping from a Filter Rule to a Meter Marker Entity](#), p.149).

8.2.3 Classes

The following sections describe classes of objects associated with DiffServ:

- **ds_filter** – see [ds_filter Class](#), p.147
- **ds** – see [ds Class](#), p.148
- **ds_map** – see [Mapping from a Filter Rule to a Meter Marker Entity](#), p.149

ds_filter Class

A DiffServ filter rule is instantiated as an object of the **ds_filter** class, which contains an int, read-only, Id for this filter and a **classifier_rule** structure.

The members of the **classifier_rule** class are defined as follows:

mask

A mask that indicates which fields must match in order to trigger the filter rule. Construct this mask by ANDing together one or more of the **CLS_RULE_x** constants:

- **CLS_RULE_DS** – DS field
- **CLS_RULE_PROTO** – protocol field
- **CLS_RULE_SADDR** – source address
- **CLS_RULE_DADDR** – destination address
- **CLS_RULE_SPORT** – source port
- **CLS_RULE_DPORT** – destination port

ds

The value that a packet must have in its DS field in order to trigger the filter rule (assuming the **CLS_RULE_DS** flag is set in **mask**). The DS field is the traffic class field for IPv6 and the TOS field for IPv4.

proto

The value that a packet must have in its IP header's protocol field in order to trigger the filter rule (assuming the **CLS_RULE_PROTO** flag is set in **mask**).

sport_low

The lowest source port a UDP or TCP packet can come from and still trigger the filter rule (assuming the **CLS_RULE_SPORT** flag is set in **mask**).

sport_high

The highest source port a UDP or TCP packet can come from and still trigger the filter rule (assuming the **CLS_RULE_SPORT** flag is set in **mask**).

dport_low

The lowest destination port a UDP or TCP packet can be destined for and still trigger the filter rule (assuming the **CLS_RULE_DPORT** flag is set in **mask**).

dport_high

The highest destination port a UDP or TCP packet can be destined for and still trigger the filter rule (assuming the **CLS_RULE_DPORT** flag is set in **mask**).

af

The address family the packet must belong to in order to trigger the filter rule, either **AF_INET** or **AF_INET6**.

saddr_prefixlen

The prefix length (mask) that the filter rule uses when it checks whether the source address matches (assuming the **CLS_RULE_SADDR** flag is set in **mask**).

daddr_prefixlen

The prefix length (mask) that the filter rule uses when it checks whether the destination address matches (assuming the **CLS_RULE_DADDR** flag is set in **mask**).

saddr

The source address (or network) that packets must match in order to trigger the filter rule (assuming the **CLS_RULE_SADDR** flag is set in **mask**).

daddr

The destination address (or network) that packets must match in order to trigger the filter rule (assuming the **CLS_RULE_DADDR** flag is set in **mask**).

ds Class

Meter/marker entities are objects of a variety of the **ds** class. The members of this class are defined as follows:

id

The ID of this meter/marker entity. The network stack sets this during the **create** operation.

name

The name of the meter/marker entity. The names of the two entity varieties that come with the Wind River Network Stack are:

"srTCM" – single-rate, three-color marker

"SimpleMarker" – simple marker

d

An object of the specific entity class. All meter/marker entities created by Wind River have an data type that starts with **ds_**, for instance **ds_sm** (see [SimpleMarker](#), p.151) or **ds_srtcm** (see [Single-Rate Three-Color Marker](#), p.151). The **ds_data** pseudoclass is a union of all of these classes.

Mapping from a Filter Rule to a Meter Marker Entity

When an edge router in multifield mode creates a new classifier rule it gets an ID number in return. When it creates a meter /marker entity, it also gets an ID number in return. The router can search a database that maps between these two varieties of ID value, so that when a packet matches a rule with a particular rule ID value the router can determine the corresponding entity ID value. The database of mappings is a set of **ds_map** objects. The members of this class are as follows:

filter_id

The ID of the filter rule.

ds_id

The ID of the meter /marker entity that applies to packets that match the filter rule.

For instructions on how to establish a mapping of this sort, see [Mapping a Filter to a Meter/Marker Entity](#), p.146.

8.2.4 Creating New Meter/Marker Entity Varieties

To create a new meter /marker entity variety (other than the simple marker and single-rate three-color marker, which already exist), do the following:

1. Implement all of the routines pointed to by function pointers in the **Ipnet_diffserv_handlers** structure (see [Implement the Function Pointers in the Ipnet_diffserv_handlers Structure](#), p.149).
2. Define and register a factory function for meter /marker entities (see [Define and Register a Factory Function for Meter/Marker Entities](#), p.150).

Step 1: Implement the Function Pointers in the Ipnet_diffserv_handlers Structure

A meter /marker entity must implement all of the routines pointed to by function pointers in the **Ipnet_diffserv_handlers** structure, which is defined in *installDir/components/ip_net2-6.x/ipnet2/src/ipnet_diffserv.h*. The members of this structure are as follows:

meter_input

You can set this pointer to **IP_NULL** if this entity will do no metering on the flow passing through it. You should otherwise set it to point to a routine that measures some property of the flow and keeps track of the result in some private data area. The prototype of this routine is as follows:

```
void myMeterInput (Ipnet_diffserv_handlers * handlers,
                  Ipcom_pkt * packet)
```

marker_input

You can set this pointer to **IP_NULL** if this entity will do no (re)marking of packets. Otherwise set this to point to a routine that marks the packet based on the configuration of the meter /marker or the property measured by the meter function. Your routine may write directly into the DS field of the **Ipcom_pkt** that it receives as the **pkt** argument. **pkt->ipstart** is the offset into the **pkt->data** area at which the IP header is stored. The prototype of this routine is as follows:

```
void myMarkerInput (Ipnet_diffserv_handlers * handlers,
                   Ipcom_pkt * packet)
```

destroy

Set this pointer to point to a routine that frees all resources held by the meter/marker. Do not set this pointer to `IP_NULL`. The prototype of this routine is as follows:

```
void myDestroy (Ipnet_diffserv_handlers * handlers);
```

Step 2: Define and Register a Factory Function for Meter/Marker Entities

Define a factory function for the new meter/marker entity and register it with the network stack. You must register the factory function as an **Ipnet_diffserv_handlers_template**.

```
typedef struct Ipnet_diffserv_handlers_template_struct
{
    const char *      name;
    Ipnet_diffserv_ctor create;
} Ipnet_diffserv_handlers_template;
```

The fields in this structure are defined as follows:

name

The name of the meter/marker entity. The network stack associates this name with the **create** routine below, so that the stack calls this function when meter/marker entity structures (**ds** objects) with this name are passed to the `SIOCXDSCREATE` ioctl operation (see [Creating a Meter/Marker Entity](#), p.146).

create

The **create** routine the network stack calls for meter/marker entities whose **name** members match **name**, above. The stack passes data (the **d** member of the entity's **ds** object) as the first argument to this routine. This routine has the following prototype:

```
int myCreate (void * arg, Ipnet_diffserv_handlers ** phandler);
```

For the network stack to be able to use the new meter/marker entity type, you must register the factory function by passing it in to **ipnet_diffserv_register_ctor()** from the **ipnet_diffserv_init()** routine, which is defined in:

```
installDir/components/ip_net2-6.x/ipnet2/src/ipnet_diffserv.c
```

For example, the single-rate three-color marker declares a routine called **ipnet_diffserv_srtcm_template()**, which returns a static variable of type **Ipnet_diffserv_handlers_template** that it initializes with a name and a pointer to a constructor function, so its factory registration call looks like this:

```
ipnet_diffserv_register_ctor (ipnet_diffserv_srtcm_template ())
```

8.2.5 Using Existing Meter/Marker Entity Varieties

The following two meter/markers are already implemented and part of the stack:

- a simple marker (**SimpleMarker**, see [SimpleMarker](#), p.151)
- a single-rate, three-color marker (**srTCM**, see [Single-Rate Three-Color Marker](#), p.151)

SimpleMarker

The simple marker, **SimpleMarker**, is defined in:

installDir/components/ip_net2-6.x/ipnet2/src/ipnet_ds_sm.c.

The simple marker copies a specific DS value into each IP header DS field on all packets that match a filter and can also set the drop precedence on every packet that matches.

The members of the **ds_sm** class are defined as follows:

mask

This member determines if the marker should set the DS field and/or set the drop precedence. Construct the value of the mask by **ANDING** the following two **DS_SM_x** constants:

- **DS_SM_DS_VAL** – the DS value
- **DS_SM_DROP_P** – the drop precedence

ds_value

The DS value that the marker sets on each packet that matches (if **mask** has the **DS_SM_DS_VAL** bit set).

drop_precedence

The drop precedence (one of the **IPCOM_PKT_DROP_PRECEDENCE_x** constants) that the marker sets on each packet that matches (if **mask** has the **DS_SM_DROP_P** bit set).

Single-Rate Three-Color Marker

The single-rate, three-color marker, **srTCM**, is defined in:

installDir/components/ip_net2-6.x/ipnet2/src/ipnet_ds_srtcm.c

The single-rate, three-color marker meters the byte rate of a flow and marks packets as green, yellow, or red. The shaper prioritizes green packets over yellow packets and yellow packet over red packets. See RFC 2697 for a complete description of the **srTCM** meter/marker.

The members of the **ds_srtcm** class are as follows:

mode

The mode in which the **srTCM** marker is operating. This can be either **DS_SRTCM_MODE_COLOR_BLIND** or **DS_SRTCM_MODE_COLOR_AWARE**.

CIR

The Committed Information Rate (bytes/second): the maximum, long term, data rate the flow can have and still have the marker mark it as green.

CBS

The Committed Burst Rate (bytes): the maximum size of the token bucket for green packets.

EBS

The Excess Burst Rate (bytes): the maximum size of the token bucket for yellow packets.

ds_green

The DS value that the marker gives to green packets.

ds_yellow

The DS value that the marker gives to yellow packets.

ds_red

The DS value that the marker gives to red packets.



NOTE: Set at least one of **CBS** or **EBS** to be greater than zero and at least as big as the largest possible packet in the flow.

8.3 Network Interface Output Queues

You can attach an *interface output queue* to every network interface in the network stack. All packets that the network stack sends through such an interface pass through a queue, and the queue can meter and enforce a maximum throughput on the packet flow.

There are two varieties of interface output queues in the network stack:

- *Leaf queues*, in which the packets are stored. These cannot have child queues. (See [8.3.2 Leaf Queues](#), p.156.)
- *Container queues*, which have one or more child queues (which can be either leaf queues or container queues) and a set of rules that determine which child queue to queue each packet in (see [8.3.3 Container Queues](#), p.159).

The API that you use to create, control, and delete interface output queues is defined in `installDir/components/ip_net2-6.x/ipnet2/include/ipnet_qos.h`.

ifqueue_qos class

To establish a queue, set the members of an object of the **ifqueue_qos** class and then attach this queue object to an interface using one of the techniques described in [Adding an Interface Output Queue](#), p.154. The **ifqueue_qos** class.

The members of the **ifqueue_qos** class are as follows:

ifq_name

The name of the network interface that this queue attaches to, for instance: "eth0".

ifq_type

The type of queue, for instance:

"fifo" – see: [FIFO](#), p.156

A first-in/first-out queue with a queue limit.

"dpaf" – see: [Drop Precedence-Aware FIFO](#), p.157

A first-in/first-out queue with three drop precedence levels (low, medium, and high) and with a queue limit; packets marked "high" are dropped before those marked "medium" or "low".

"null"

A queue in which all packets are dropped; some DiffServ shapers need queues of this sort.

"none"

Indicates that the interface does not have an interface output queue.

If your system automatically attaches queues to all interfaces, you can effectively remove a queue from a particular interface (for instance, a pseudo-interface that you do not want to filter) by setting its queue type to **none**.

"netemu" – see: *Network Emulator*, p.157

Can add latency, and can reorder, drop, and corrupt packets; you can use this to test various network conditions.

"mbc" – see: *Multiband Container (MBC)*, p.160

Holds an array of queues, arranged in order of priority (the lower the index in the array, the higher the priority); packets dequeue from the container in order of priority.

"htbc" – see: *Hierarchy Token Bucket Container (HTBC)*, p.160

Holds a set of queues that are not prioritized relative to each other; packets dequeue from these queues in a round-robin fashion.

ifq_id

The identifying number of the queue. In **GET** operations, if you set this to **IFQ_ID_NONE**, the operation returns the root queue, otherwise it returns the queue with this ID.

In **SET** operations, you can set this to a specific ID if you want to operate on a specific queue, or you can set this to **IFQ_ID_NONE** if you want the stack to select a unique ID. If the specified queue ID already exists, a **SET** operation replaces the queue. If you replace a container queue, this removes all of its children.

All queues attached to a particular interface have unique identifying numbers, but the same number may be used to refer to different queues that are attached to different interfaces.

ifq_parent_id

The ID of the parent of this queue if this is a child queue, or **IFQ_ID_NONE**, if this is the root queue.

ifq_count

(Read-only.) The number of packets in this queue, if it is a leaf queue, or the sum of packets in all of its child queues, if it is a container queue.

ifq_data

An object that defines the characteristics of the particular type of queue. This may be an object of a queue class of your own invention, or one of the following:

- **ifqueue_fifo** – see: *FIFO*, p.156
- **ifqueue_dpaf** – see: *Drop Precedence-Aware FIFO*, p.157
- **ifqueue_netemu** – see: *Network Emulator*, p.157
- **ifqueue_mbc** – see: *Multiband Container (MBC)*, p.160
- **ifqueue_htbc** – see: *Hierarchy Token Bucket Container (HTBC)*, p.160

8.3.1 Operations

This section describes the various things you can do with output queues:

- [Adding an Interface Output Queue](#), p.154
- [Getting an Object that Describes an Interface Output Queue](#), p.154
- [Adding a Filter Rule to a Container Queue](#), p.155
- [Deleting a Filter Rule from a Container Queue](#), p.156

Adding an Interface Output Queue

To add an interface output queue to a network interface, or to replace one that was previously added, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCSIFQUEUE, &queue);
```

Where **queue** is a **ifqueue_qos** object that describes the leaf or container queue you are adding (see [ifqueue_qos class](#), p.152).

If the **ifq_id** member of the **ifqueue_qos** object matches that of a queue that is already attached to the interface, this operation will replace that queue with the one described by **queue**.

You can also use the following **qc** command:

```
# qc [-v virtualRouter] queue add [parameters]
```

For instance, if you want to test how your system would work under conditions when there was 100 millisecond latency on interface **lo0**, you could create a network emulator queue on that device that simulates such latency, with the following **qc** command:

```
# qc queue add dev lo0 netemu limit 10 min_latency 100 max_latency 100
```

If you want to rate-limit all traffic on interface **eth0** to two Mbits per second, you could do this by establishing a rate-limited HTBC queue with the following command:

```
# qc queue add dev eth0 htbc rate 2000kbps
```

Getting an Object that Describes an Interface Output Queue

To get an object that describes an interface output queue on a network interface, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCGIFQUEUE, &queue);
```

Where **queue** is a **ifqueue_qos** object (see [ifqueue_qos class](#), p.152).

If you call this with the **ifq_id** field of **queue** set to **IFQ_ID_NONE**, this routine will fill **queue** so that it describes the root queue on this interface, otherwise it will fill **queue** so that it describes the queue matching **ifq_id**.

You can also use a **qc** command, which has the following format:

```
# qc [-v virtualRouter] queue show [parameters]
```

Adding a Filter Rule to a Container Queue

To add a filter rule to a container queue on a network interface, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCXAIFQFILTER, &filter);
```

In this call **filter** is an **ifqueue_filter** object that describes the filter you are adding (see [Filter Rules](#), p.159).

You can also use a **qc** command, which has the following format:

```
# qc filter add dev device parent queueID handle filterID [filterArgs] flowid queueID
```

The arguments to this command are as follows:

dev device

The device to which you are attaching the filter, for instance **eth0**.

parent queueID

The identifying number of the container queue to which you are adding the filter.

handle filterID [filterArgs]

The *filterID* is the identifying number of the filter. You may use the following arguments in the *filterArgs* argument to describe the filter:

proto number

tclass number

srcport range

dstport range

srcaddr address[/prefix]

dstaddr address[/prefix]

flowid queueID

The identifying number of the destination queue for packets that match the filter.

For example:

To add a filter identified by the number 5 to the container queue identified by the number 1, so that all TCP packets (protocol number 6) are filtered into the queue identified by the number 31, use the following **qc** command:

```
# qc filter dev eth0 parent 1 handle 5 proto 6 flowid 31
```

To add a second filter (identified by the number three) to the same container queue that filters all UDP packets (protocol number 17) that are sent to 2001::/16 into the same queue, use the following **qc** command:

```
# qc filter dev eth0 parent 1 handle 3 proto 17 srcaddr 2001::/16 flowid 31
```

Deleting a Filter Rule from a Container Queue

To delete a filter rule from a container queue on a network interface, you can either call a routine from within a program or use a **qc** command interactively.

To invoke this operation from within a program, use the following call:

```
ioctl (sock_fd, SIOCXDIFQFILTER, filter);
```

In this call **filter** is an **ifqueue_filter** object that describes the filter you are removing (see [Filter Rules](#), p.159).

You can also use a **qc** command, which has the following format:

```
# qc [-v virtualRouter] filter del [parameters]
```

8.3.2 Leaf Queues

Leaf queues cannot have children and you cannot assign filter rules to them.

The Wind River Network Stack includes the following kinds of leaf queues:

- [None](#), p.156
- [FIFO](#), p.156
- [Drop Precedence-Aware FIFO](#), p.157
- [Network Emulator](#), p.157

None

- Queue name: **none**

If you specify **none** as the queue type, the interface does not have an interface output queue. If your system automatically attaches queues to all interfaces, you can effectively remove a queue from a particular interface by setting its queue type to **none**.

FIFO

- Queue name: **fifo**
- File: *installDir/components/ip_net2-6.x/ipnet2/src/ipnet_pkt_queue_fifo.c*
- **qc** command:

```
# qc queue add dev device fifo limit number
```

This is the default queue for all interfaces in the network stack.

You can use FIFOs as buffers to handle temporary peaks in traffic. You can also use them as leaf queues in more complex queue hierarchies.

Packets dequeue from a FIFO queue, in the same order as they arrived on the queue, without regard to the packets' individual properties.

The cost of queuing or dequeuing a packet on a FIFO queue is always O(1).

Create a FIFO queue object in this way:

1. Create an object of class **ifqueue_qos** and fill its members appropriately (see [ifqueue_qos class](#), p.152).
2. Set the **ifq_type** member of that object to "fifo".

3. Create an object of class **ifqueue_fifo** and set its **fifo_limit** member to the maximum number of packets that can be stored in the queue.
4. Set the **ifq_data** member of the **ifqueue_qos** object to the **ifqueue_fifo** object.

Drop Precedence-Aware FIFO

- Queue name: **dpaf**
- File: *installDir/components/ip_net2-6.x/ipnet2/src/ipnet_pkt_queue_dpaf.c*
- **qc** command:

```
# qc queue add dev device dpaf limit number
```

Drop Precedence-Aware FIFO queues (DPAFs) work as normal FIFOs until the maximum number of packets in the queue is exceeded. If the limit is exceeded, a DPAF checks the drop precedence of the new packet. The following outcomes are possible:

- Reject this packet if it has high drop precedence.
- Drop a high-drop-precedence packet from the queue and replace it with the new packet if this packet has medium or low drop precedence.
- Drop a medium-drop-precedence packet from the queue and replace it with the new packet if this packet has low drop precedence.
- Reject this packet if there are no packets in the queue with higher drop precedence that the DPAF can drop.

The extra overhead compared to FIFO makes this queue slower. The cost of queuing/dequeuing a packet is $O(\log(n))$, where n is the number of packets in the queue.

Create a FIFO queue object in this way:

1. Create an object of class **ifqueue_qos** and fill its members appropriately (see [ifqueue_qos class](#), p.152).
2. Set the **ifq_type** member of that object to "dpaf".
3. Create an object of class **ifqueue_dpaf** and set its **dpaf_limit** member to the maximum number of packets that can be stored in the queue.
4. Set the **ifq_data** member of the **ifqueue_qos** object to the **ifqueue_dpaf** object.

Network Emulator

- Queue name: **netemu**
- File:
installDir/components/ip_net2-6.x/ipnet2/src/ipnet_pkt_queue_netemu.c
- **qc** command:

```
# qc queue add dev device netemu limit number [min_latency msec] \
[max_latency milliseconds] \
[drop probability [random] | pattern {0|1},{0|1}[,...,{0|1}]] \
[corrupt probability [random]]
```

The network emulator is a testing and debugging tool. You can use it to introduce jitter and latency into a stream, which may result in packet reordering, or to drop and corrupt packets.

A **netemu** queue with zero latency, zero drops, and zero corruption is equivalent to a FIFO queue.

Create a network emulator queue object in this way:

1. Create an object of class **ifqueue_qos** and fill its members appropriately (see [ifqueue_qos class](#), p.152).
2. Set the **ifq_type** member of that object to "**netemu**".
3. Create an object of class **ifqueue_netemu** and set its members appropriately (see below).
4. Set the **ifq_data** member of the **ifqueue_qos** object to the **ifqueue_netemu** object.

The members of the **ifqueue_netemu** class are as follows:

netemu_limit

The maximum number of packets that can be stored in this queue.

netemu_min_latency

netemu_max_latency

The minimum and maximum latency that the emulator adds to each packet, in milliseconds. The emulator will evenly distribute the latency on individual packets between [**netemu_min_latency**..**netemu_max_latency**].

netemu_random_drop

netemu_drop_probability

netemu_drop_pattern

netemu_drop_pattern_len

Set **netemu_random_drop** to TRUE to drop individual packets with the probability of 1/**netemu_drop_probability**. Set **netemu_random_drop** to FALSE to drop a packet every **netemu_drop_probability** packets. For example, if you set **netemu_drop_probability** to 4, setting **netemu_random_drop** to FALSE means the emulator drops every fourth packet, while setting it to TRUE means that for each packet there is a one-in-four probability that the emulator will drop it.

You can also set **netemu_drop_pattern** to a bitmask that represents a regular pattern of **netemu_drop_pattern_len** bits, with bits marked "1" representing dropped packets, so that, for instance, the pattern 00000011

(**netemu_drop_pattern** = 0x00000003, **netemu_drop_pattern_len** = 8) will drop the last two of every eight packets.

netemu_random_corrupt

netemu_corrupt_probability

Set **netemu_random_corrupt** to TRUE to corrupt individual packets with the probability of 1/**netemu_corrupt_probability**. Set **netemu_random_corrupt** to FALSE to corrupt a packet every **netemu_corrupt_probability** packets. For example, if you set **netemu_corrupt_probability** to 4, setting **netemu_random_corrupt** to FALSE means the emulator corrupts every fourth packet, while setting it to TRUE means that for each packet there is a one-in-four probability that the emulator will corrupt it.

8.3.3 Container Queues

A *container queue* contains one or more child queues (child queues may be container queues or leaf queues or a combination of both). *Filter rules* determine which child queue the container queue stores a particular packet in.

Container Superclass

All container classes are subclasses of the **ifqueue_container** class (which is to say that the first member of the structure that defines a container class is an **ifqueue_container** structure). The members of this class are as follows:

child_count

the number of child queues this container has (a child queue that is a container queue only counts as a single child even if it in turn has multiple children)

child_ids

an array that contains the queue IDs of each of the child queues

Filter Rules

You may attach filter rules to container queues (see [Adding a Filter Rule to a Container Queue](#), p.155). These rules control in which child queue a container queue should place packets. If a packet does not match any rule, the container queue queues the packet on its default child queue.

The **ifqueue_filter** class describes a filter rule. The members of this class are defined as follows:

filter_ifname

The name of the network interface this filter operates on.

filter_id

The ID of this rule. The stack sets this field when the filter is added; if you are deleting a filter, set this field to indicate which filter you want to delete.

filter_queue_id

The ID of the (container) queue this filter applies to.

filter_child_queue_id

The ID of the child queue into which packets matching this rule will be queued by the container queue.

filter_rule

The actual rule, in the form of an **classifier_rule** object (see [8.2 Differentiated Services](#), p.144).

Available Container Queues

The Wind River Network Stack includes the following container queues:

- [Multiband Container \(MBC\)](#), p.160
- [Hierarchy Token Bucket Container \(HTBC\)](#), p.160

Multiband Container (MBC)

- Queue name: **MBC**
- File: *installDir/components/ip_net2-6.x/ipnet2/src/ipnet_pkt_queue_mbc.c*
- **qc** command:

```
# qc queue add dev device mbc bands number [default_band number]
```

An MBC container queue keeps an array of child queues (bands) in decreasing priority. An MBC queue always dequeues packets from the first non-empty queue in its array of queues.

You can use this variety of container queue when certain kinds of traffic, such as signalling, must transmit as quickly as possible, while other traffic, like e-mail, can wait for low-traffic conditions.

When you create an MBC queue, it initially has an array of FIFO child queues. You may replace these with queues of another variety by issuing an **SIOCSIFQUEUE** ioctl call (see [Adding an Interface Output Queue](#), p.154).

Define an MBC queue by setting the **ifq_type** member of the **ifqueue_qos** structure to "**mbc**" and the **ifq_data** member of the **ifqueue_qos** structure to an object of the **ifqueue_mbc** class. The members of this class are defined as follows:

mbc_container

The base class for container queues.

mbc_bands

The number of bands (child queues) that this container queue manages. You may only set this member prior to the time you create the MBC queue.

mbc_default_band

Which of the bands (child queues) the MBC container queue puts a packet into if that packet does not match any of the filter rules. This is an index value in the range [0..**mbc_bands**).

Hierarchy Token Bucket Container (HTBC)

- Queue name: **HTBC**
- File: *installDir/components/ip_net2-6.x/ipnet2/src/ipnet_pkt_queue_htbc.c*
- **qc** command:

```
# qc queue add dev device htbc rate rate [burst number]
```

Use the HTBC queue when you want to control the bandwidth usage on an interface. The queue calculates the data rate by taking the sum of bandwidth used by all children; it calculates this when it dequeues packets. It considers all children to have the same priority, so it dequeues packets from the child queues in a round-robin fashion.

You can add or delete child queues from an HTBC container queue throughout the lifetime of the HTBC. When you create an HTBC, it comes with a single child FIFO queue, the default queue, attached to it. You can add more queues, or replace this default queue with one of another variety, by using the **SIOCSIFQUEUE** ioctl call (see [Adding an Interface Output Queue](#), p.154).

Define an HTBC queue by setting the **ifq_type** member of the **ifqueue_qos** structure to "**htbc**" and the **ifq_data** member of the **ifqueue_qos** structure **ifqueue_htbc** class. The members of this class are defined as follows:

htbc_container

The base class for containers.

htbc_byterate

The maximum bandwidth, in bytes per second, at which this queue may send.

htbc_token_limit

The maximum number of tokens this queue can have. For each byte that an HTBC container queue dequeues from one of its children and sends, it consumes one token. You must set this value to be greater than the MTU (maximum transmission unit) of the interface you attach the queue to. A reasonable value might be in the range **htbc_byterate**/100 to **htbc_byterate**. A larger value results in HTBC dropping fewer packets during temporary bursts in the flow, so the actual value depends on the acceptable level of “burstiness.”

htbc_default_id

The ID number of the default queue, into which HTBC places all packets that do not match any filter rule.

8.3.4 Adding a New Queue Type

Create a new leaf queue variety by adding its structure to the **ifqueue_data** union (see *ifqueue_qos class*, p.152), implementing all routines in the **Ipnet_pkt_queue** structure, and registering an instance of that type with the **ipnet_pkt_queue_register()** routine.



NOTE: The Wind River Network Stack registers its queues, like the FIFO and HTBC queues, in the **ipnet_pkt_queue_init()** routine, which is located in *installDir/components/ip_net2-6.x/ipnet2/src/ipnet_pkt_queue.c*. You may want to register any new queues you add at the same time, which you can do by changing this routine to make additional registration calls.

The members of this structure are defined as follows:

type

The name of the queue type.

impl_size

The size of the structure used by this queue, which may be an **Ipnet_pkt_queue** structure or a subclass structure that derives from it.

c_ops

If this queue is a container queue, it must implement this interface. Write these routines to return 0 (zero) on success, or an **IPNET_ERRNO_x** error code on failure (except for **q_get**, which returns a queue structure, or **IP_NULL** if it finds no queue that matches the ID).

q_get – retrieves a queue by ID number

```
Ipnet_pkt_queue_struct * my_q_get
(Ipnet_pkt_queue_struct * containerQueue, int queueID)
```

q_insert – adds a queue to the container

```
int my_q_insert (Ipnet_pkt_queue_struct * containerQueue,
Ipnet_pkt_queue_struct * addThisQueue);
```

q_remove – removes a queue from the container

```
int my_q_remove (Ipnet_pkt_queue_struct * containerQueue,
Ipnet_pkt_queue_struct * removeThisQueue);
```

f_insert – adds a filter to a queue

```
int my_f_insert (Ipnet_pkt_queue_struct * containerQueue,  
                int filterID, classifier_rule * rule, int childQueueID);
```

f_remove – removes a filter from a queue

```
int my_f_remove (Ipnet_pkt_queue_struct * containerQueue,  
                int filterID)
```

enqueue or enqueue_locked

Enqueues a packet on this queue. This routine has the following prototype:

```
int myEnqueue (struct Ipnet_pkt_queue_struct * queue, Ipcom_pkt * packet)
```

Write this routine to return 0 (zero) on success, or an **IPNET_ERRNO_x** error code on failure.

dequeue or dequeue_locked

Dequeues a packet from this queue. This routine has the following prototype:

```
Ipcom_pkt * myDequeue (Ipnet_pkt_queue_struct * queue)
```

Write this routine to return the next packet (according to the rules of the queue), or **IP_NULL** if the queue is empty.

requeue or requeue_locked

Puts a packet back on the queue that was removed from the queue with the **dequeue** function. This routine has the following prototype:

```
void myRequeue (Ipnet_pkt_queue_struct * queue, Ipcom_pkt * packet);
```

count or count_locked

Returns the number of packets in this queue, or the sum of packets in all child queues if this is a container queue. This routine has the following prototype:

```
int myCount (Ipnet_pkt_queue_struct * queue);
```

reset

Removes all packets from the queue and resets the internal state of the queue. This routine has the following prototype:

```
void myReset (Ipnet_pkt_queue_struct * queue);
```

dump

Fills an **ifqueue_x** structure with the current configuration of this queue. This routine has the following prototype:

```
void myDump (Ipnet_pkt_queue_struct * queue,  
            union ifqueue_data * data);
```

Write this routine so that it fills the **data** structure with the configuration information specific to this queue.

configure

Configures the queue based on the **ifqueue_x** structure for this queue, found in the **data** parameter. This routine has the following prototype:

```
int myConfigure (Ipnet_pkt_queue_struct * queue,  
                union ifqueue_data * data);
```

Write this routine so that it returns the number of elements that may be placed in the queue.

init

The routine that initializes the queue after memory has been allocated for it. This routine has the following prototype:

```
int myInit (Ipnet_pkt_queue_struct * queue);
```

Write this routines to return 0 (zero) on success, or an `IPNET_ERRNO_x` error code on failure.

destroy

The routine that frees all resources allocated by this queue. This routine has the following prototype:

```
void myDestroy (Ipnet_pkt_queue_struct * queue);
```



NOTE: Do not set the remaining fields of this structure, such as `id`, `parent_id`, `netif`, `prev`, and `next`.

8.3.5 Example—Reserving Bandwidth for an Application

This example demonstrates how to use an HTBC interface queue to reserve bandwidth for an application. To specify the application, we identify its specific UDP or TCP port—in this example, port 5001.

Step 1: Create an MBC.

Using the `qc` command, create an MBC with two bands—one for traffic with unlimited bandwidth and one for the application for which you want to reserve bandwidth. Use the first band as the default (used if there is no matching rule).

1. Issue the following command to create the MBC:

```
# qc queue add dev eth0 mbc bands 2 default_band 0
```

2. Issue a `qc show` command to verify that the MBC has been created:

```
# qc queue show dev eth0
mbc/1[0] queue at eth0
  bands: 2, default_band: 0
  fifo/1000[1] queue at eth0
    limit: 16
  fifo/1001[1] queue at eth0
    limit: 16
```

Step 2: Attach an HTBC to the second band of the MBC.

The child queue in that band has ID 1001. Replace it with one with an HTBC with a limit of 1 Mbit per sec.

1. Issue the following command to create the child queue:

```
# qc queue add dev eth0 parent 1 handle 1001 htbc rate 1Mbit burst 100kb
```

The specified **burst** parameter must be low enough to improve the accuracy of the rate limiter, yet high enough to create a bucket container of adequate size for the specified rate, which is set in relation to the clock speed of the operating system. For further information on specifying this parameter, see [Specifying the burst Parameter](#), p.164.

2. Issue another `qc show` command to verify the creation of this queue.

```
# qc queue show dev eth0
mbc/1[0] queue at eth0
  bands: 2, default_band: 0
  fifo/1000[1] queue at eth0
    limit: 16
  htbc/1001[1] queue at eth0
    rate: 125kbps, burst: 100k, default id: 1002
    fifo/1002[1001] queue at eth0
      limit: 16
```



NOTE: The speed is always shown as kilobytes per second, so 1 Mbit/sec is shown as 125 KB/sec.

Step 3: Create a filter rule for the application.

Create a filter rule that puts all traffic for the application into the band with the HTBC connected to it and all other traffic into the other band.

1. Issue the following command to create the rule:

```
# qc filter add dev eth0 parent 1 dstport 5001 flowid 1001
```

This command adds a filter rule to a queue that is attached to device **eth0**.

The network stack will apply this rule when something is queued at the MBC queue (parent 1, which is the ID of the MBC). If the rule matches, it will use the child queue with ID 1001 (flowid 1001). The rule specifies a destination port of 5001. (Because no protocol is defined, the rule applies to both UDP and TCP port 5001.)

2. Issue a **ttcp** command to verify the result. The **ttcp** command must use TCP port 5001 by default.

```
# ttcp -n 1000 192.168.130.4
ttcp-t: fd=28, buflen=8192, nbuf=1000, align=16/0, port=5001,
sockbufsize=32767 TCP -> 192.168.130.4
ttcp-t: socket
ttcp-t: setsockopt(IP_SO_REUSEADDR)
ttcp-t: bind IPv4 0
ttcp-t: setsockopt(sndbuf)
ttcp-t: connect
ttcp-t: 8192000 bytes in 68581 milliseconds = 116 KB/sec, 119 B/msec
+++
ttcp-t: 1000 I/O calls, msec/call = 68, calls/sec = 14
```

The throughput of 116 KB/sec is a bit slower than the configured value of 125 KB/sec, but that is expected because the limit includes all headers, while **ttcp** measures the throughput of the TCP payload.

3. Change the port to 5002, which is not bandwidth-limited:

```
# ttcp -n 1000 -p 5002 192.168.130.4
ttcp-t: fd=29, buflen=8192, nbuf=1000, align=16/0, port=5002,
sockbufsize=32767 TCP -> 192.168.130.4
ttcp-t: socket
ttcp-t: setsockopt(IP_SO_REUSEADDR)
ttcp-t: bind IPv4 0
ttcp-t: setsockopt(sndbuf)
ttcp-t: connect
ttcp-t: 8192000 bytes in 447 milliseconds = 17897 KB/sec, 18326 B/msec
+++
ttcp-t: 1000 I/O calls, msec/call = 0, calls/sec = 2237
```

The throughput changes to 18 MB/sec. The sample output shown above was prepared using a debug build of the network stack running in the simulator. The throughput would be even higher on a network stack prepared for deployment on an actual target.

Specifying the burst Parameter

The **burst** parameter specifies the size of the “bucket” that defines the capacity of an HTBC. Packets received on a queue are added to this bucket, and packets

transmitted on a queue are subtracted from this bucket, both at the rate specified by the **rate** parameter.

If incoming packets exceed the **burst** parameter, the excess packets are dropped. If outgoing packets are transmitted at a rate that drains the bucket, transmission stops until incoming packets have filled the bucket in an amount equal to the amount of outgoing packets in the queue. The bucket must therefore be sized so that is neither completely drained nor completely filled by the traffic anticipated on the queue, given the specified **rate** parameter.

To set the **burst** parameter, you must take the tick rate of the VxWorks operating system into account, which is about 60 Hz. Note further that the network stack might begin transmitting packets on the queue just after a tick, so the bucket must be large enough to allow transmission for two full ticks.

You can calculate the value for this parameter as follows:

$$R/60 * 2 = B$$

where

R = rate

B = burst

For example, if you set the **rate** parameter to 1 Mbit, your calculation for the **burst** parameter is as follows:

$$1 * 1024^2 / 60 * 2 = 35 \text{ KB}$$

You can set the **burst** parameter to a higher value to allow temporary bursts of traffic, but the sustained rate can never exceed the value specified by the **rate** parameter, regardless of the value of the **burst** parameter. If you specify too low a value for the **burst** parameter, the shaped bandwidth will be lower than the value configured by the **rate** parameter.

Ingress Traffic Prioritization

- 9.1 Introduction 167
- 9.2 Factors to Consider Before Using Ingress Filtering 168
- 9.3 Building VxWorks to Include Ingress Traffic Prioritization 169
- 9.4 Implementing an Ingress Filter Routine 170

9.1 Introduction

By default, the network stack treats all packets that arrive at an interface equally and processes them in the order of their arrival. Ingress traffic prioritization is a quality of service (QoS) feature that allows you to assign priorities to the packets that arrive at an individual interface and have the stack process higher-priority packets before lower-priority packets.



NOTE: The QoS feature is available only in the Wind River Platforms builds of the network stack. The Wind River General Purpose Platform, VxWorks Edition, does not support QoS.

The Wind River Network Stack does not support ingress filtering in SMP builds.

To use Wind River ingress traffic prioritization, you must do the following:

1. Statically configure a job queue to hold packets that a filter routine prioritizes (see [9.3 Building VxWorks to Include Ingress Traffic Prioritization](#), p.169).
2. Implement one or more ingress filter routines that classify packets and assign priorities.

Wind River provides a function prototype for this routine and a set of sample implementations (see [9.4 Implementing an Ingress Filter Routine](#), p.170).

3. Register your filter routine to filter incoming traffic on a specific interface (see [9.4.1 Registering an Ingress Filter Routine](#), p.171).

The filter routine you implement must assign each incoming packet to one of the following categories:

- Packets for the stack to process immediately (`QOS_DELIVER_PKT`)

- Packets to queue for the stack to process later (**QOS_DEFER_PKT**)
- Packets for the stack to drop (**QOS_IGNORE_PKT**). Typically, these are packets in which the filter routine detects an error.

If the filter routine assigns a packet for the stack to process later (deferred processing), the filter routine must also assign the packet a priority.

9.2 Factors to Consider Before Using Ingress Filtering

This section applies to ingress traffic prioritization that uses the standard network-stack queue for incoming traffic; circumstances are different if you create custom job queues using **jobQueueLib** (see the table entry for **Ingress QoS Job Queue** in [Table 9-1](#) under [9.3 Building VxWorks to Include Ingress Traffic Prioritization](#), p.169).

There are two things you should consider before you decide to use ingress traffic prioritization and develop an ingress filter routine:

- On a system with multiple interfaces for incoming messages, you may need to associate a filter routine with each interface. You can associate a single routine with multiple interfaces.
- The current implementation of ingress traffic prioritization does not provide for congestion handling or “fairness.”

Systems with Multiple Interfaces for Incoming Traffic

When you register an ingress filter routine, you associate the routine with a specific interface. Typically, the filter routine designates some packets for immediate delivery and other packets for the stack to process later (deferred processing). The routine assigns those packets that it designates for deferred processing a priority that determines the order in which the stack will process them—the stack queues higher priority packets ahead of lower priority packets.

If you do not associate an ingress filter routine with an interface, the stack treats all packets that the interface receives equally and processes them as if a filter routine had prioritized all of them for immediate delivery. As a result, the stack processes packets that arrive at an interface without ingress filtering before those packets that an ingress filter designated for deferred processing, even if those deferred packets have a high priority.

To ensure that the stack does not wait to process deferred packets until after it has processed all other packets, you must assign an ingress filter routine to each interface that receives incoming traffic. You may assign a single filter routine to multiple interfaces.

Traffic Congestion and Fairness

The stack queues, by priority, those packets that the ingress filter designates for deferred processing. The stack processes all higher-priority packets before any

lower-priority packets. This means that during heavy incoming traffic, lower-priority packets can take up an increasing amount of buffer space without the stack processing them. The current implementation does not limit the number of packets that the stack can queue for deferred delivery. As a result, it is possible for deferred packets to exhaust the pool of available network-interface receive buffers.

Driver Variety

Currently, ingress filtering in the Wind River Network Stack will only work with END drivers.

9.3 Building VxWorks to Include Ingress Traffic Prioritization

To include ingress traffic prioritization in an image, include the **Ingress Traffic Prioritization** (**INCLUDE_QOS_INGRESS**) build component in your VxWorks Image Project. In addition, you may need to change the default values of the ingress-traffic-prioritization build parameters. [Table 9-1](#) describes the parameters.

Table 9-1 Ingress Traffic Prioritization Configuration Parameters

Workbench Description and Parameter Name	Default Value and Type
Ingress QoS Job Queue QOS_JOBQ <p>The JOB_QUEUE_ID of the queue into which the ingress filter places prioritized and deferred packets (QOS_DEFER_PKT).</p> <p>If you do not change the default value, the ingress filter places deferred packets into the network stack's standard queue for incoming packets.</p> <p>You can create a job queue specifically to handle deferred packets. For information, see the reference page for jobQLib.</p>	netJobQueueId long
Ingress Traffic Prioritization Job Queue Priority QOS_JOBQ_PRI <p>The jobQLib task priority for the job that handles deferred packets.</p> <p>Do not change the default priority unless you create one or more separate job queues based on jobQLib.</p>	NET_TASK_QJOB_PRI - 1 long
Ingress default deferred Job Queue Priority QOS_DEFAULT_PRI <p>The default priority that the ingress filter assigns to incoming packets. This can be a value from 0 to 31, with higher values having a higher priority.</p>	0 long

9.4 Implementing an Ingress Filter Routine

To use ingress filtering, first implement a filter routine based on the following function prototype:

```
int ingressFilterRoutine
(
    END_OBJ * pEnd,
    M_BLK_ID * ppMblk,
    int * pPri
)
```

The parameters to this routine are as follows:

pEnd

A pointer to an object that describes the END device over which the packet arrived.

ppMblk

A pointer to an **M_BLK_ID** that points to an incoming packet.

pPri

A pointer to an integer that can hold the incoming packet's priority value. This priority can be a value from 0 to 31, with higher values having a higher priority.

Return values from the routine apply to the packet referenced by **ppMblk**. Valid return values are:

QOS_DELIVER_PKT

Deliver the packet without delay to the upper layer protocol for processing.

QOS_DEFER_PKT

Queue the packet for delivery according to its priority, as given in **pPri**. If your routine returns this value, it must also set ***pPri** accordingly.

QOS_IGNORE_PKT

Ignore (drop) the packet. Typically, this value indicates that the filter routine detected an error in the packet. When the routine returns this value, it assumes responsibility for calling **m_freem()** to free the packet's memory space at ***ppMblk**.

You can find sample implementations of the **ingressFilterRoutine()** prototype in the following file:

installDir/components/ip_net2-6.x/vxcoreip/src/dlink/qosIngressHooks.c

You can find the following sample routines in this file:

etherQosHook()

Assigns priorities to Ethernet packets based on the protocol specified in the Ethernet header's **type** field.

vlanQosHook()

Assigns priorities based on the priority field in the VLAN header.

ipProtoQosHook()

Assigns priorities based on the transport protocol and port-number fields in the packet's header.

dscpQosHook()

Assigns priorities based on the packet's differentiated-services code-point (DSCP) field.

9.4.1 Registering an Ingress Filter Routine

To register an ingress filter routine for an interface, call the **qosIngressHookSet()** routine.

The syntax for **qosIngressHookSet()** is:

```
STATUS qosIngressHookSet
(
    int                unit,
    char *             ifname,
    QOS_ING_HOOK       hookRtn
)
```

The parameters to this routine are as follows:

unit

The unit number of the interface, for example, 0.

ifname

The name of the interface, for example, "fei".

hookRtn

A pointer to the ingress-filter routine that you are registering.

Deactivating Ingress Traffic Prioritization on an Interface

To deregister an ingress filter routine from an interface, which deactivates ingress traffic prioritization on an interface, call **qosIngressHookSet()** with **hookRtn** set to **NULL**.

A

Networking Shell Commands

[A.1 Introduction 173](#)

[A.2 Networking Shell Commands 173](#)

A.1 Introduction

This appendix provides reference information for the some of the shell commands used to configure the Wind River Network Stack and related software modules at run time.

A.2 Networking Shell Commands

ifconfig

Name

ifconfig - set or get configuration values for a network interface

Synopsis

The shell command **ifconfig** can be used to assign information to a network interface, retrieve information on a network interface, and generate reports. This command is accessed from the kernel shell, which can be used in either of the following modes:

- command interpreter mode
- C interpreter mode

When the shell is set to C interpreter mode, all command arguments must be enclosed in double quotation marks ("). In this reference entry, all syntax diagrams

and command examples are enclosed in double quotation marks. If you are using the shell in command interpreter mode, omit these quotation marks.

The syntax for the command varies by usage.

The valid syntax for assigning information to a network interface is as follows:

```
ifconfig "interface [protocol] [parameter parameterValue]"
```

The valid syntax for retrieving information on a network interface is as follows:

```
ifconfig "[flags] [interface] [protocol]"
```

The following combinations of flags are valid when using **ifconfig** to generate reports:

```
ifconfig "[-L] interface [protocol ]"  
ifconfig "-a [-L] [-d] [-u] [protocol ]"  
ifconfig "-l [-d] [-u] [protocol ]"  
ifconfig "[-L] [-d] [-u]"
```

Description

Use **ifconfig** to configure a network interface or to retrieve network interface configuration values. The **ifconfig()** API can be used programmatically.

As a convenience, failing to specify a value for *parameter* before the address is interpreted as indicating the default, **add**. Similarly, because of a prejudice in favor of IPv4, failing to specify a value for *protocol* is interpreted as indicating the default of **inet**, which specifies IPv4.

The correct usage of the elements of an **ifconfig()** call is as follows:

flags

The *flags* values apply when the **ifconfig** command (or **ifconfig()** call) is used to generate a report.

Use **-a** instead of specifying an *interface* value if you want a report containing information on all interfaces.

Use **-d** to limit the report to down interfaces.

Use **-l** to limit the report to a simple list of all interfaces without additional information.

Use **-L** to request lifetime information on an IPv6 address presented as a time offset string.

Use **-u** to limit the report to up interfaces.

interface

Use this parameter to specify the name of the network interface to which the command applies. The generic format of a network interface name is *UnitnameUnitnumber*, which results in names such as **fei0**.

protocol

This element lets you specify the network protocol associated with the call. The default *protocol* value is **inet**, which indicates the IPv4 protocol. The other valid value for *protocol* is **inet6**.

address

If the *protocol* is **inet**, the *address* is either a host name already present in the host name database (see **hosts()**), or an IPv4 Internet address expressed in Internet standard dot notation. You can also use the CIDR notation (also known as the slash notation) to specify netmask with the address. For example, using CIDR

(slash) notation, 192.168.0.1/16 is a possible **inet** *address* string. However, if the protocol is **inet6**, the slash in an *address* string is interpreted as an indicator of the prefix length. For example, ::1/128. If you do not use slash notation to specify a prefix length in an **inet6** *address*, **ifconfig** assumes a default value of 64. To override this default without using the slash notation, you can use the **prefixlen** parameter. See the entry under the *parameters* description for more information.

destAddr

This element of an **ifconfig()** call appears only when the call applies to the local end of a point-to-point link. In this case, the *destAddr* value indicates the address of the correspondent on the other end the link. See the **add** entry under the *parameters* description below.

parameters

These elements of an **ifconfig()** call let you set the value of the configuration parameters associated with a network interface. The following is a list of valid values for *parameters*.

add

Add another network address to this interface. You may want to use this parameter when changing network numbers. It leaves the old address in place, which allows the interface to still accept packets destined to the old address. If the new address is on the same subnet as the first network address for this interface, you must specify a netmask of 0xffffffff. If this **add** applies to the local end of a point-to-point connection, you need to include the other end's destination address. The syntax is:

```
ifconfig "protocol add address destAddr"
```

Note that the equivalent command **alias** is currently not supported.

delete

Remove the network address specified.

anycast

Specify that the IPv6 address configured is an anycast address. Based on the current specification, only routers may configure anycast addresses. Anycast address will not be used as source address of any of outgoing IPv6 packets.

Note that the equivalent commands **-alias** and **-remove** are not currently not supported.

cga

The address is cryptographically generated (inet6 only). Available only when **IPNET_USE_RFC3971** has been defined.

down

Mark an interface as down. If an interface is marked as down, IP does not transmit through that interface. If possible, **ifconfig "interface down"** also disables packet reception on the specified interface. Although marking the interface as down prevents IP from using routes associated with that interface, it does not remove the routes from the routing table.

If the interface is later marked as up, IP will be able to use the old routes associated with the interface (provided some agent, such as a routing protocol or a user, has not explicitly deleted the old routes).

create

Create the specified network pseudodevice. If the interface is given without a unit number, try to create a new device with an arbitrary unit number. If creation of an arbitrary device is successful, the new device name is printed to standard output.

destroy

Destroy the specified network pseudodevice.

mtu *n*

Set the maximum transmission unit (MTU) of the interface to *n*. The default MTU value is interface specific and may not be configurable.

preferred

Set the preferred lifetime for the specified IPv6 address.

prefixlen *length*

This *parameters* value is valid only when *protocol* is **inet6**. The *length* value specifies the number of address bits used for dividing networks into subnets. The *length* must be an integer value between 0 and 128. It is usually 64 under the current IPv6 assignment rule.

If the **prefixlen** is omitted, a default value of 64 is assumed. You can also use the "slash after the address" notation to specify the prefix length.

tentative

Set the tentative bit in the specified IPv6 address.

-tentative

Clear the tentative bit in the specified IPv6 address.

up

Mark an interface as up. Use this *parameters* value to undo a previous **ifconfig "interface down"** call. This status change is a software event only. Simply marking an interface as up cannot reinitialize the hardware associated with that interface.

valid

Set the valid lifetime for the specified IPv6 address.

dstaddr

Set remote address to 'a' (**inet** and PPP only).

lladdr *addr*

Set interface link address to *addr*.

vr *vr*

Set virtual router to *vr*.

dhcp

Enable DHCP autoconfiguration (**inet** only).

-dhcp

Disable DHCP autoconfiguration (**inet** only).

link

Enable special processing at the link level.

-link

Disable special processing at the link level.

promisc

Enable promiscuous mode at the interface.

-promisc

Disable promiscuous mode at the interface.

Wrapper Changes

The **ifconfig** backwards compatibility wrapper routine does not support (or supports differently) the following parameters:

Command Name	Alternative	Description
alias	add	Add inet6 or inet addresses
-alias	delete	Remove inet6 or inet addresses
remove	delete	Remove inet6 or inet addresses.
arp	None	Enable the Address Resolution Protocol on the specified network interface.
-arp	None	Disable the Address Resolution Protocol on the specified network interface.
autoconf	None	Enable IPv6 auto configuration for the specified interface.
-autoconf	None	Disable IPv6 auto configuration for the specified interface.
broadcast	None	Specify the address used to broadcast to the network (inet only).
debug	None	Enable driver dependent debugging code.
-debug	None	Disable driver dependent debugging code.
deprecated	None	Set the IPv6 deprecated address bit.
-deprecated	None	Clear the IPv6 deprecated address bit.
media type	None	Set the media type of the interface.
mediaopt opts	None	Set media options of the interface.
vlan vlan_tag	None	Set the VLAN ID of the interface (1-4094).
vlanpri value	None	Set the VLAN User Priority (0-7).
vlandev iface	None	Associate a physical interface to a VLAN pseudo interface.
-vlandev iface	None	Disassociate physical interface from a VLAN pseudo interface.
metric	None	Set the routing metric for the interface.
pltime	preferred	Preferred lifetime for the specified IPv6 address.
-tentative	None	Clear the tentative bit in the specified IPv6 address.
vltime	valid	Set the valid lifetime for the specified IPv6 address.
dstaddr	different	Set PPP remote address (inet only). Part of the add parameter.

Unsupported Wrappers

The following flags (used for report-generating **ifconfig()** calls) are not supported:

Flag	Alternative	Description
-d	None	Limit report to down interfaces.
-L	None	Request lifetime information on an IPv6 address presented as a time offset string.
-l	None	Limit the report to a simple list of all interfaces without additional information.
-u	None	Limit the report to up interfaces.

New Wrappers

The following are new commands:

Command	Alternative	Description
lladdr <i>addr</i>	New	Set interface link address to <i>addr</i> .
vr <i>vr</i>	New	Set virtual router to <i>vr</i> .
dhcp	New	Enable DHCP auto configuration (inet only).
-dhcp	New	Disable DHCP auto configuration (inet only).
link[0-2]	New	Enable special processing at the link level.
-link[0-2]	New	Disable special processing at the link level.

qc

Name

qc - configure network interface output queue(s)

Synopsis

```
qc [-v vr] queue [ show | add | del ] dev if [ root | parent q-id ] [ handle q-id ] type
[type_specific_opt]
qc [-v vr] filter [ show | add | del ] dev if parent q-id handle f-id [filter_args] flowid
q-id
```

Description

qc is used to control which queues are added to network interfaces and is used to control the filter rules on container queues.

-V *vr*

Apply the operation on the specified virtual router.

queue

Create a queue, using the specified parameters.

filter

Create a filter, using the specified parameters.

show

Show the configuration for a queue or filter.

add

Add/replace a queue or filter.

del

Delete a queue or filter.

dev *if*

Specifies which network interface (or network device) the queue or filter applies to.

root

Specifies that this queue has no parent queue and is rooted directly at the network interface.

parent

Specifies which queue this queue is a child to. The parent queue must be one with a name ending with a "c" (mbc, htbc). The "c" stands for container and means that the queue is a container for one or more child queues.

handle

Specifies the ID (or handle) of this queue. It is only needed for replace operations.

Adding an MBC with two bands will automatically add two FIFO queues as children to the MBC. Those children will automatically be given unique IDs (or handles). Any of the children may be replaced by some other queue by using **handle** *id-of-child* with an **add** command. (There is no special replace operation; adding a queue with an ID that already exists means that the queue will be replaced.)

flowid *q-id*

Specifies the child queue to which a packet is queued when the packet meets the criteria specified by a filter.

A filter definition consist of a number of fields that must match a packet, like source address equal to *src_addr*, TOS equal to *type_of_service*, etc.

A queue container such as an MBC holds one or more children. Let's assume it has two children, with the IDs I1 and I2.

The flowid of a filter attached to the MBC specifies the children to which a matching packet is queued. So *q-id* would be I1 or I2 in this example.

Every queue container has an implicit filter that matches any packet. Use **default_band** *id* to specify the *id* value of that child when you create an MBC.

q-id

ID of an output queue. Must be unique per interface, unless it should replace an existing queue.

f-id

ID of a filter, must be unique per queue unless it should replace an existing filter.

type

Queue type, e.g., FIFO, HTBC.

range

Numeric range, format is **a**, **a-**, **-b** or **a-b**.

addr

IPv4 or IPv6 address. Note that source and destination address must be specified for the same domain per rule.

prefix

Number of bites used for the network identifier.

The filter parser accepts the following fields for *filter_args*:

proto *number*

tclass *number* (**tclass** can be substituted for **tos**)

srcport *range*

dstport *range*

srcaddr *addr[/prefix]*

dstaddr *addr[/prefix]*

Examples

1. Add an HTBC as root to eth0, max bandwidth is 2 Mbit with a burst of 100 KB:

```
qc queue add dev eth0 root handle 3 htbc rate 2Mbit burst 100kb
```
2. Add a FIFO that can queue up to 50 packets to the HTBC added in Example 1:

```
qc queue add dev eth0 parent 3 handle 31 fifo limit 50
```
3. Add a filter with ID 5 to queue (1) so that all TCP packets end up in Example 2:

```
qc filter dev eth0 parent 1 handle 5 proto 6 flowid 31
```
4. Add another filter to queue (1) so that all UDP packets sent to 2001::/16 end up in Example 2:

```
qc filter dev eth0 parent 1 handle 3 proto 17 srcaddr 2001::/16 flowid 31
```

qos

Name

qos - controls the QoS features in the Wind River Network Stack

Synopsis

qos [*option*]... *op optype* [*args*]

Description

The **qos** command is used to configure and view the Quality of Service features within the network stack, except interface queues, which use **qc**.

The following options exist:

-4

Use IPv4 as address domain (default).

-6

Use IPv6 as address domain.

-V *tab*

Operate on virtual router with index *tab*.

The following operations (*op*) exist:

add

Add something.

delete

Delete something.

get

Lookup and display something.

list

Lookup and display something.

The following operation types (*optype*) exist:

policy

Policy routing.

Policy routing arguments:

Add operation:

from *addr*

Apply this rule if the source address is *addr*.

to *addr*

Apply this rule if the destination address is *addr*.

proto *p*

Apply this rule if the transport proto is *p*.

tos *n*

Apply this rule if the TOS is *n*.

tclass *n*

Apply this rule if the traffic class is *n*.

flow *f*

Apply this rule if the traffic flow is *f*.

ifindex *if*

Apply this rule if the **ifindex** of the incoming interface is *if*.

flags *f*

Apply this rule if the packet has the appropriate flags set or unset. The flags are comma-separated, and to require a flag to be not set, it should be prepended with an explanation point (!). Existing flags are **forwarded**, **multicast**, **tunneled**, **broadcast**, **fragment**, **mf ipv4**, **ipv6**.

scope *s*

Apply this rule if the scope of the address is *s*.

reverse

If this rule is applied, do the lookup based on *src* instead of *dst*.

table *t*

Use the routing table with policy ID *t* if the rule matches.

last

Do not look at any more rules, even if there is no match on this table.

prio *p*

The priority of the rule to add [-32768..32767].

Delete/get operation:

id *i*

The ID of the rule to delete or show.

route

Name

route - a utility to manually manipulate network routing tables

Synopsis

```
route [-v routetab] [-n] command [[modifiers] args]
```

Description

The **route** utility supports a limited number of general options, but a rich command language enables the user to specify any arbitrary request that could be delivered via the programmatic interface.

The **route** *command* options are as follows.

add

Add a route.

delete

Delete a route.

change

Change aspects of a route (such as its gateway).

get

Look up and display the route for a destination.

show

Print out the route table (similar to **netstat -r**).

monitor

Continuously report any changes to the routing information base, routing lookup misses, or suspected network partitionings.

vr

Add or delete a virtual route table.

The **monitor** command has the syntax:

```
route [-n] monitor
```

The **vr** command has the syntax:

```
route [vr] {add | delete} vr_index
```

vr_index 0 (the default route table) cannot be added or deleted. The new route table is completely empty.

Route Utility Commands

The route utility commands have the following syntax:

```
route [-n] command [-net | -host] [-option ...] dest_ip [gate_ip]
```

Options for these **route** commands include the following:

[net | host]

Forces the destination to be interpreted as a network or a host, respectively.

dest_ip

The IPv4/IPv6 destination host or network address.

gate_ip

The IPv4/IPv6 host serving as the gateway.

Route Utility Command Options

Additional options for **route** utility commands include:

-V *vr*

Specifies the virtual router index. If no VR index is specified, 0 is used. Must be the first switch if present.

-n

Disables DNS address lookup.

-nollinfo

Do not show routes which have the **IPNET_RTM_LLINFO** flag set.

-T *table*

Specifies the route table ID. If no route table ID is specified, the default (defined by **IPCOM_ROUTE_TABLE_DEFAULT**) is used.

-inet

IPv4 route (default)

-inet6

IPv6 route

-netmask *a.b.c.d*

Specifies an IPv4 destination netmask for **-net** routes. The netmask is stated in dotted decimal notation (*a.b.c.d*), where each portion of the mask is an integer between 0-255, representing the value of the bytes in that position in the mask.

-prefixlen *val*

Prefix length for **-net** routes.

-dev *ifname*

Device name. This option must be used if *gate_ip* is not specified. It may also be used if *gate_ip* is specified and *gate_ip* can be reached by more than one link; in such a case, use **-dev** *ifname* to specify which link should be used.

-mpls *nhlfe_key*

Specify MPLS shortcut route.

If the destination is directly reachable via an interface requiring no intermediary system to act as a gateway, specify the **-iface** route flag; the gateway given is the address of this host on the common network, indicating the interface to be used for transmission. For further information on **-iface**, see [Route Flags](#), p.184.

The optional **-netmask** qualifier is used to manually add subnet routes with netmasks different from that of the implied network interface. Specify an additional address parameter (to be interpreted as a network mask). You can override the implicit network mask generated in the **AF_INET** case by using this option after the destination parameter. You can also use the modifier **-prefixlen** for similar purposes in the IPv6 case.

Route Flags

Routes have associated flags, which influence operation of the protocols when the router transmits packets to destinations matched by those routes. To set (or sometimes clear) these flags, indicate the corresponding modifiers as follows:

-cloning

IPNET_RTF_CLONING - generates a new route on use.

-xresolve

IPNET_RTF_XRESOLVE - emit a message on use (for external lookup).

-iface

~IPNET_RTF_GATEWAY - destination is directly reachable.

-static

IPNET_RTF_STATIC - manually added route.

-nostatic

~IPNET_RTF_STATIC - pretend route added by kernel or daemon.

-reject

IPNET_RTF_REJECT - emit an ICMP unreachable when matched.

-blackhole

IPNET_RTF_BLACKHOLE - silently discard packets (during updates).

-llinfo

IPNET_RTF_LLINFO - validly translate proto address to link address.

-proto1

IPNET_RTF_PROTO1 - set protocol specific routing flag #1.

-proto2

IPNET_RTF_PROTO2 - set protocol specific routing flag #2.

-pref

IPNET_RTF_PREF - always prefer this route.

-srcaddr

IPNET_RTF_SRCADDR - gateway specifies the default source address.

Optional Modifiers

The optional modifiers **-rtt**, **-rttvar**, **-mtu**, **-hopcount**, and **-expire** provide initial values to quantities maintained in the routing entry by transport level protocols, such as TCP. In a **change** or **add** command where the destination and gateway are not sufficient to specify the route (as in the ISO case, where several interfaces may have the same address), the router can use the **-ifp** or **-ifa** modifiers to determine the interface or interface address.

Examples

Add the default IPv4 gateway to 10.1.1.1:

```
# route add default 10.1.1.1
```

Add the IPv4 network route 14.1/16 on the specified interface:

```
# route add -dev eth0 -net -netmask 255.255.0.0 14.1.0.0
```

Add an IPv4 route for host 15.1.6.7 to gateway 10.1.1.22:

```
# route add -host 15.1.6.7 10.1.1.22
```

Add the default IPv6 gateway to FEC0::1:0:0:0:6:

```
# route add -inet6 default FEC0::1:0:0:0:6
```

Add a new virtual router:

```
# route vr -add 1
```

slab

The **slab** command shows a snapshot of how many buffers are allocated from the available slab caches. It also shows how much memory all slabs use from the operating system.

Name

slab - show how many buffers are allocated from the available slab caches

Synopsis

```
slab [-g]
```

Description

-g

Forces a garbage collection of the slabs, which means that all slab caches that do not have any allocated buffers at the moment are destroyed and the memory they are using is returned to the operating system.

Output

Full

A slab is **Full** if all objects are allocated.

Partial

A slab is **Partial** if there is at least one free object and at least one allocated object. New allocations are always taken from **Partial** slabs, whenever possible.

Empty

A slab is **Empty** if all objects are free. Empty slabs are freed if another allocation cannot be satisfied without exceeding the maximum allowed amount of memory.

Example A-1

Example Using slab

This example shows the layout of the **slab** cache memory handler.

```
[vxWorks *]# slab
Memory pool: IPNET memory pool has 229636 bytes allocated, alloc high is 229636,
limit is 20971520
  Slab: IPNET bound socket, object size 16, alignment 4
    Full:    0
    Partial: 1
Objects - total: 198, allocated 3
  Empty:    0
  Slab: IPNET timer, object size 20, alignment 16
    Full:    0
    Partial: 0
    Empty:   1
Objects - total: 123, allocated 0
  Slab: IPNET kioevent softirq, object size 24, alignment 4
    Full:    0
```

```
Partial: 0
Empty: 0
Slab: IPNET poll, object size 76, alignment 4
Full: 0
Partial: 0
Empty: 1
Objects - total: 49, allocated 0
Slab: IPNET packet header, object size 608, alignment 16
Full: 0
Partial: 1
Objects - total: 100, allocated 28
Empty: 0
Slab: IPNET 1500 bytes packet buffer, object size 1666, alignment 16
Full: 0
Partial: 1
Objects - total: 30, allocated 28
Empty: 0
Slab: IPNET 3000 bytes packet buffer, object size 3166, alignment 16
Full: 0
Partial: 0
Empty: 0
Slab: IPNET 10000 bytes packet buffer, object size 10166, alignment 16
Full: 0
Partial: 0
Empty: 0
Slab: IPNET 65536 bytes packet buffer, object size 65702, alignment 16
Full: 0
Partial: 0
Empty: 1
Objects - total: 1, allocated 0
Slab: IPNET IPv4 address, object size 112, alignment 4
Full: 0
Partial: 1
Objects - total: 34, allocated 5
Empty: 0
Slab: IPNET IPv6 address, object size 140, alignment 4
Full: 0
Partial: 1
Objects - total: 27, allocated 10
Empty: 0
Slab: IPNET socket, object size 540, alignment 16
Full: 0
Partial: 1
Objects - total: 30, allocated 3
Empty: 0
Slab: TCP segment, object size 32, alignment 4
Full: 0
Partial: 0
Empty: 0
Slab: IPCOM network job, object size 16, alignment 16
Full: 0
Partial: 0
Empty: 1
Objects - total: 123, allocated 0
[vxWorks *]#
```

Example A-2 Deciphering slab Output

Below is a description of slab output.

```
Slab: IPNET timer, object size 20, alignment 16
Full: 0
Partial: 1
Objects - total: 123, allocated 8
Empty: 0
...
Slab: IPNET 1500 bytes packet buffer, object size 1666, alignment 16
Full: 0
Partial: 2
Objects - total: 30, allocated 12
Objects - total: 30, allocated 28
```

```
Empty: 1
Objects - total: 30, allocated 0
```

The entries above describe two "memory cache" instances.

The first instance above is named "IPNET timer". It contains "objects" (which are memory areas) of size 20 bytes. That is the *only* size that can be allocated from this memory cache.

Each object is aligned to a 16 byte boundary (in this case, it happens to be the cache line size on this board).

The cache has one "slab" assigned to it. That slab is in **Partial** state, which means it has both allocated and free objects. Eight objects are currently allocated out of a total of 123.

The second instance is named "IPNET 1500 bytes packet buffer" (this is a replacement of the previous packet pool). Each object is 1666 bytes and aligned to nearest 16 bytes.

It has currently 3 slabs assigned to it, 2 are **Partial** and one is **Empty**, which means all its objects free.

The slabs in **Empty** state would be free if the stack requested explicit garbage collection of all memory. It would also be free if the target hits its allocation limit and another memory cache, say cache "C", fails fail to return an object unless one or more of **Empty** slabs on other memory caches are garbage collected and the memory is remade into objects of type C.

sysctl

The **sysctl** command is used to get or set system parameters.

Name

sysctl - Get or set **sysctl** values

Synopsis

```
sysctl -w variable=value
sysctl -a
sysctl variable
```

Description

-a

List all **sysctl** parameters.

-w *variable=value*

Change the value of *variable* to the specified value.

value

Value of a system variable.

sysvar

The **sysvar** command lists and modifies global variables, as follows.

System variables are similar to UNIX environment variables, except that they are available throughout the system to any process. For example, if you are running an IKE process, you can issue a **sysvar** command to alter a NAT variable.

The **sysvar** command uses a treelike data structure for all the network components and services. For example, the system variable **iptcp.ConnectionTimeout** defines the number of seconds the network stack tries to create connection before giving up.

The system variable **ipssh.service.port_fwd** controls whether port forwarding can be used.

System variables are similar to the components you configure at build time using the Workbench **Kernel Configuration Editor**. Unlike kernel components, however, the **sysvar** command modifies parameters at run time.



NOTE: Not all global variables can be changed at run time. Some parameters can only be reset at build time, using Workbench or **vxprj**.

To include this command in your project, include **INCLUDE_IPCOM_SYSVAR_CMD**.

Name

sysvar – lists, gets, and defines system variables

Synopsis

```
sysvar list [name[*]]  
sysvar get name  
sysvar unset name[*]  
sysvar set [-c | -o | -r] name value
```

Description

Command options are as follows:

name
Name of a system variable.

-c
OK to create.

-o
OK to overwrite.

-r
Flag read-only.

value
Value of a system variable.

Numerics

802.1Q VLAN, *see* VLAN

A

acceptRtn, *see* xAcceptRtn()
address learning (802.1Q) 134
addresses, *see also* Internet addresses
addrGet(), *see* xAddrGet()
AF_INET 121
AF_INET6 121
AF_PACKET 121
AF_ROUTE 121
altq_hdr 12
anchor, shared-memory 45
attaching a stack to a network interface
 overview of 28
ATTR_AC_ISR 15
ATTR_AC_SH_ISR 14
ATTR_AI_ISR 15
ATTR_AI_SH_ISR 14
aux, M_BLK field 12

B

backplane processor numbers 43
backplane, shared memory 43
backplanes
 processor numbers 43
 shared-memory networks, using with 43
bcastFlag 87
bindRtn 121
boot line parameters
 shared-memory network example 54
buffer pool
 netBufLib 10
BUS_INT 53

C

cap_available
 VLAN tagging 130
.cdf files
 network services and 108
CL_BLK 10, 11, 12
CL_DESC 17
CL_POOL 18
Classifier 144
clBlk, *see* CL_BLK
clBlkNum 17
clDescTblNumEnt 15, 17
close()
 socket descriptors 120
closeRtn 121
CLS_RULE_x 147
clusters 10
 definition of 10
 determining address of 11
COMM_END 96
common Enhanced Network Device support 60
configNet.h 94
configure() 162
connectRtn 121
connectWithTimeoutRtn 121
container queues 159
 filter rule addition 155
 filter rule deletion 156
 Hierarchy Token Bucket Container (HTBC) 160
 multiband container (MBC) 160
 type of interface output 152
count() and count_locked() 162

D

DEFAULT_CPUS_MAX 54
dequeue() and dequeue_locked() 162
destroy() 150, 163
device instances
 loading and unloading 64

- device link status 32
- devInstanceInit2() 70, 71
 - xLoad() and 65
- devname 28
- Differentiated Services (build component) 144
- differentiated services, *see* DiffServ
- DiffServ 144
 - behavior aggregate mode 144
 - configuration components 144
 - configuring 144
 - meter/marker entities 149
 - multifield mode 144
- domainMap 121
- domainReal 121
- drop presence-aware FIFO (DPAF) queues 157
- DRV_CTRL
 - created by xLoad() 94
- ds class 148
- DS codepoint 144
- DS field 144
- ds_filter 147
- ds_map 149
- dscpQosHook() 171
- dump() 162

E

- EIOCGADDR 73, 77
- EIOCGFLAGS 77
- EIOCGHDRLEN 78
- EIOCGIFCAP 77
 - VLAN tagging 130
- EIOCGIFMEDIA 77
- EIOCGMEDIALIST 77
- EIOCGMIB2 78, 103
- EIOCGMIB2233 78, 103
- EIOCGMTU 78
- EIOCGPOLLCONF 78, 104
- EIOCGPOLLSTATS 78, 104
- EIOCGRCVJOBQ 78
- EIOCMULTIADD 73, 77
- EIOCMULTIDEL 73, 77
- EIOCMULTIGET 77
- EIOCPOLLSTART 77, 84
- EIOCPOLLSTOP 78
- EIOCQUERY 78, 79, 89, 93
- EIOCSADDR 73, 77
- EIOCSFLAGS 77, 78
- EIOCSIFCAP 77
- EIOCSIFMTU 78
- END (Enhanced Network Driver) 91
 - entry points
 - list exported to MUX 68
 - ioctl support 72
- END driver polled statistics support 61
- END drivers
 - ioctl support 72

- END interface support 60, 128
- END_BIND_QUERY 79, 89
- END_CAPABILITIES 77
 - VLAN tagging 130
- END_ERR_BLOCK 79, 80
- END_IFCOUNTERS 78, 103
- END_IFDRVCONF 78, 103
- END_MEDIA 77
- END_MEDIALIST 77
- END_OBJ 96
 - establishing 66
 - populated how 96
 - releasing 67
- END_OBJ_INIT() 66
- END_QUERY 78, 93
- END_QUERY structure
 - xIoctl() and 79
- END_RCV_RTN_CALL() 91
- END_RCVJOBQ_INFO 78
- END_TX_SEM_GIVE() 71
- END_TX_SEM_TAKE() 70
- END2_LINKBUFPOOL_CLSIZE 22
- END2_LINKBUFPOOL_NTUPLES 22
- END2-style interface support 60
- end8023AddressForm() 87, 91
- endBind (NET_FUNCS member) 89
- endCommon 60
- endDevTbl[] 94
- endEtherAddressForm() 87, 91
- endEtherPacketAddrGet() 88, 91
- endEtherPacketDataGet() 88, 91
- endLib 60, 66, 91
- endM2Free() 103
- endM2Init() 66
 - example 103
- endM2Packet()
 - examples 104
- endPoolCreate() 66
- endPoolDestroy() 68
- endPoolJumboCreate() 66
- enqueue() and enqueue_locked() 162
- esm (boot device) 44
- etherMultiAdd() 81
- etherMultiDel() 82
- etherMultiGet() 83
- Ethernet multicast library support 128
- etherQosHook() 170

F

- f_insert() 162
- f_remove() 162
- fast path
 - optional modifiers 184
- FIFO queues 156
- filter rules 159
- FIONBIO 125

FIONREAD 125
 FIOSELECT 125
 FIOUNSELECT 125
 flow control 80
 formAddress(), *see* xFormAddress()

G

GARP VLAN Registration Protocol 133
 gateway 28
 gateway inet boot parameter 54
 gateway6 29
 gateways
 example 52
 General Purpose Platform
 QoS and 167
 getpeernameRtn 121
 getsocknameRtn 121
 getsockopt() 138
 getsockoptRtn 121
 GVRP 133

H

header (M_PKT_HDR field) 12
 heartbeat, shared-memory 45
 host name
 assigning to an address 31
 hostAdd()
 assigning a host name to an address 31
 hosts 50
 example 52
 hosts.equiv 50
 example 52
 hwconf.c 94

I

ifconfig shell command 173
 ifconfig()
 accessing with ipnet_cmd_ifconfig() 93
 configuring a network interface with 29
 network interface starting and enabling 32
 reconfiguring a network interface 31
 retrieving interface information with 29
 using 29
 VLAN 135
 examples 137
 IFCONFIG_n 28
 ifEndObj 103
 IFF_ALLMULTI 82
 IFF_BROADCAST 78
 IFF_MULTICAST 78, 82
 IFF_RUNNING 32
 IFF_SIMPLEX 78
 IFF_UP 32, 78
 ifname 28
 ifPollInterval 103
 IFQ_ID_NONE 153
 ifq_type 152
 ifqueue_container 159
 ifqueue_filter 155, 159
 ifqueue_qos 152, 154
 ifValidCounters 103
 INCLUDE_SM_COMMON 46
 INCLUDE_END 60, 128
 INCLUDE_END_COMMON 60
 INCLUDE_END_ETHER_HDR 61
 INCLUDE_END_POLLED_STATS 61
 INCLUDE_END2 60
 INCLUDE_END2_LINKBUFPOOL 21
 INCLUDE_ETHERNET 128
 INCLUDE_IFCONFIG 29
 INCLUDE_IPCOM_SYSVAR_CMD 188
 INCLUDE_IPNET_CLASSIFIER 144
 INCLUDE_IPNET_DIFFSERV 144
 INCLUDE_IPNET_DS_SM 144
 INCLUDE_IPNET_DS_SRTCM 144
 INCLUDE_IPNET_IFCONFIG_n 28, 93
 INCLUDE_IPNET_USE_TUNNEL 34
 INCLUDE_IPNET_USE_VLAN 128
 INCLUDE_IPQOS_CMD 144
 INCLUDE_IPQUEUE_CONFIG_CMD 144
 INCLUDE_IPWRAP_IFCONFIG 27
 INCLUDE_IPWRAP_IPPROTO 27
 INCLUDE_L2CONFIG 128
 INCLUDE_LINKBUFPOOL 11
 INCLUDE_MUX 60
 INCLUDE_MUX_COMMON 60
 INCLUDE_MUX_L2 128
 INCLUDE_MUX_OVER_END2 61
 INCLUDE_MUX2 60
 INCLUDE_MUX2_OVER_END 61
 INCLUDE_MUXTK 60
 INCLUDE_MUXTK_OVER_END 62
 INCLUDE_MUXTK_OVER_END2 62
 INCLUDE_NET_POOL 20
 INCLUDE_NETBUFADVLIB 11, 13
 INCLUDE_NETBUFLIB 11, 16
 INCLUDE_NETBUFPOOL 10
 INCLUDE_NETPOOLSHOW 11
 INCLUDE_QOS_INGRESS 169
 INCLUDE_SHELL_INTERP_CMD 27
 INCLUDE_SM_COMMON 52
 INCLUDE_SM_NET 52
 INCLUDE_SM_NET_SHOW 52
 INCLUDE_SM_SEQ_ADDR 50
 INCLUDE_VXMUX_MBLK 62
 inet 28
 inet on backplane boot parameter 49, 54
 inet6 29
 Ingress default deferred Job Queue Priority 169
 ingress filter routine 170

- ingress filtering 167
 - configuration components 169
 - jobQueueLib 168
 - multiple interfaces 168
- Ingress QoS Job Queue 168, 169
- Ingress Traffic Prioritization 169
- Ingress Traffic Prioritization Job Queue Priority 169
- init() 162
- input queue
 - processors, shared memory 47
- interface output queues 152
 - adding 154
 - retrieving 154
- Internet addresses
 - correcting interface assignment errors 31
- interrupt service routines
 - registering 70
- ioctl()
 - see also xioctl()
 - socket descriptors 120
- ioctlRtn see xioctlRtn()
- ioctlsocket
 - socket
 - memory validation 124
- IP addresses, example 3
- ipAttach() 93, 108
 - example 108
- IPCOM output queue commands 144
- IPCOM QoS commands 144
- ipcom_drv_eth_init() 93, 108
 - example 108
- IPCOM_IF_DRV_CAP_VLAN_HWTAGGING_RX 131
- IPCOM_IF_DRV_CAP_VLAN_HWTAGGING_TX 131
- IPCOM_IF_DRV_CAP_VLAN_MTU 130
- Ipcom_pkt
 - MUX protocols 59
- IPCOM_VXWORKS_USE_MUX_L2 128
- ipnet_cmd_ifconfig() 93
- IPNET_DIFFSERV_CLASSIFIER_MODE_BA 144
- Ipnet_diffserv_handlers 149
- Ipnet_diffserv_handlers_template 150
- ipnet_diffserv_init() 150
- ipnet_diffserv_register_ctor() 150
- ipnet_diffserv_srcm_template() 150
- Ipnet_ifqueue structure 152
- Ipnet_ifqueue_container structure 159
- Ipnet_pkt_queue 161
 - defined 161
- ipnet_pkt_queue_init() 161
- ipnet_pkt_queue_register() 161
- ipProtoQosHook() 170
- IPv6
 - hostname, assigning an address to 31

J

jobQLib 169

jobQueueLib 100, 168
jobQueuePost()

- network job queues 100

L

l2config 128

- examples 135

- leaf queues 152, 156
- drop presence-aware FIFO (DPAF) 157
- FIFO 156
- Ipnet_pkt_queue 161
- network emulator (netemu) 157
- Learning Bridge
- MUX-L2 and 134
- Legacy linkBufPool for END2 devices 21
- libInitRtn see xSockLibInit()
- link status 32
- link_cookie 131
- linkBufPool 10, 14, 66
- alignment requirements of 17
- cluster size limitations 15
- cluster size restrictions of 18
- netPoolInit() and 17
- listenRtn 121
- LL_HDR_INFO 88, 117
- location monitors 47

M

M_BLK 10, 11

- hidden NET_POOL pointer in 17
- members of 12

- M_BLK ethernet/802.3 61
- m_freem() 170
- M_LINK 10, 15, 16
- M_PKT_HDR 12
- M_PREPEND() 20
- m_prepend() 20
- M2_ID 78
- M2_INTERFACETBL 78
- mailbox interrupts 47
- marker_input() 149
- max # of cpus for shared network 44
- Maximum number of 802.1Q VLANs supports 128
- mBlk, see M_BLK
- mbuf 11
- mCastAddrAdd(), see xMCastAddrAdd()
- mCastAddrDel(), see xMCastAddrDel()
- mCastAddrGet(), see xMCastAddrGet()
- memArea 17, 18
- member_to_object() 101
- memory requirements routines 18
- netBufLib 19
- memSize 17, 18

- meter/marker entities
 - creating 146, 149
 - deleting 146
 - filter mapping 146
 - filter rule addition 145
 - filter rule deletion 146
 - using 150
- meter_input() 149
- MTU 161
- MULTI_TABLE 83
- multi-band container (MBC) 160
- multiList 82
- MUX
 - components 60
 - driver interface 96
 - driver styles 58
 - OSI model, and 24
 - overview 23
 - programming interface 63
 - protocol styles 59
 - service/MUX interface 92
 - wrappers 61
- MUX common support 60
- MUX Layer 2 support 128
- MUX mux2Bind() service 60
- MUX muxBind() service 60
- MUX muxTkBind() service 60
- MUX support 60
- MUX_L2_MAX_VLANS_CFG 128
- MUX_L2_NUM_PORTS_CFG 128
- MUX_L2_VLAN_STATS 141
- MUX_PROTO_OUTPUT 116
- MUX_PROTO_PROMISC 116
- MUX_PROTO_SNARF 116
- muxAddressForm() 20
- muxBind() 93
 - service functions of 114
- muxCommon 60
- muxDevConnect() 70, 94
 - when called 97
- muxDevLoad() 70, 94
 - called by tUsrcRoot 92
 - when called 97
- muxDevStart() 70, 94
- muxDevStop() 95
- muxDevUnload() 67, 95, 113
 - VLAN 131
- muxError() 117
- muxFormAddress() 87
- MUX-L2
 - egress rules 132
 - ingress rules 132
- muxL2EgressClassify() 131
- muxL2IngressClassify() 131
- MUXL2IOCSPORTATTACH 131
- MUXL2IOCSPORTDETACH 131
- MUXL2IOCSPORTVLAN 130
- muxL2Ioctl() 130, 131, 133
 - get functionality 134

- muxL2PortAttach() 130
 - alternative to 131
- muxL2PortDetach() 131
 - alternative to 131
- muxL2Show() 134, 140
 - example 140
- muxL2StatShow() 140
 - example 141
- muxL2VlanShow() 134, 140
 - example 141
- muxL2VlanStatShow() 140
 - example 141
- muxLinkHeaderCreate() 87
- muxMCastAddrAdd() 81
- muxPacketAddrGet() 88
- muxPollSend()
 - when called 96
- muxSend() 20, 80
- muxTkBind() 93
 - service functions of 114
- muxTkCookieGet() 93
- muxTkPollReceive() 85
- muxTkPollSend() 83
- muxTkSend() 20, 80
- muxTxRestart() 80
- muxUnbind() 95

N

- NB_BUFTYPE_CL_BLK 19
- NB_BUFTYPE_CLUSTER 19
- NB_BUFTYPE_M_BLK 19
- neighbor discovery 34
- NET_FUNCS 68
- NET_POOL 14, 17
 - hidden pointer in M_BLK 17
- NET_POOL_NAME_SZ 14
- NET_TASK_QJOB_PRI 100, 169
- NETBUF_ALIGN 16, 17, 18
- NETBUF_ALIGNED 16
- NETBUF_CFG 14
 - attributes 14
 - bMemExtraSize 14
 - bMemPartId 14
 - clDescTblNumEnt 15
 - ctrlNumber 14
 - ctrlPartId 14
 - pCIDescTbl 15
 - pDomain 14
 - pName 14
- NETBUF_CL_DESC 15
- NETBUF_LEADING_CLSPACE_DRV 16
- netBufAdvLib 11
- netBufLib 8, 12
 - back ends 10
 - linkBufPool 10
 - netBufPool 10

- buffer pool 10
- display routines 11
- netBufLib pools
 - attributes of 13
 - creating 12
 - finding 12
 - freeing 12
 - parents and children 13
- netBufLib.c 12
- netBufPool 10, 13
 - alignment requirements of 17
 - cluster size limitations 15
 - cluster size restrictions of 18
 - netPoolInit() and 17
- netCIBlkGet() 12
- netCIBlkJoin() 12
- netClusterGet() 12
- netLib.h 100
- netMblkChainDup() 12
- netMblkCICChainFree() 12
- netMblkCIFree() 12
- netMblkCIJoin() 12
- netMblkDup() 88
- netMblkGet() 12
- netMblkToBufCopy() 12, 84
- _netMemReqDefault() 18
- netPoolAttach() 13
- netPoolBind() 13
- netPoolCreate() 12, 13, 66
 - example 16
- netPoolDetach() 13
- netPoolIdGet() 12
- netPoolInit() 12
 - clDescTblNumEnt 18
 - limitations of 13
 - pClDescTbl 17
 - pFuncTbl 18
 - pMclBlkConfig 17
 - pNetPool 17
 - releasing pools created with 68
- netPoolNameGet() 12
- netPoolRelease() 12, 13, 68
- netPoolUnbind() 13
- netTupleGet() 12
- network devices
 - flags 78
 - loading and starting 94
 - stopping and unloading 95
- network drivers
 - data structures shared with MUX 96
 - delay mechanisms in 102
 - globally visible routines of 97
 - polled mode 96
- network emulator (netEmu) queues 157
- network heartbeat 45
- network interface drivers
 - see network drivers
- network interfaces
 - assigning an address, netmask, or prefix 29

- attaching to stack 28
- configuration
 - ifconfig() 29
- configuring 29
- fixing IP address assignment errors 31
- pausing 32
- reconfiguring 31
- retrieving configuration information 29
- retrieving information about 29
- starting and enabling 32
- starting at run time 25
- network services
 - binding to an interface 93
 - socket interface, adding a 118
 - subroutines for 114
 - unbinding from an interface 93
 - writing a sublayer 107
- NUM_DAT_CLBLKS 21
- NUM_DAT_MBLKS 21
- NUM_DAT_n 21
- NUM_SYS_CLBLKS 20
- NUM_SYS_MBLKS 20
- NUM_SYS_n 21, 21
- Number of ports that the device has 128

P

- packet statistics 102
- packetDataGet(), see xPacketDataGet()
- packets
 - consuming 111
- pausing, network interfaces 32
- pClDescTbl 15
- pDst.mBlkHdr.reserved field 87
- pFuncTable
 - muxL2PortAttach() and 131
- pFuncTbl 13
- _pLinkPoolFuncTbl 14, 18
- PMA_DAT_n 21
- PMA_DATPOOL 21
- PMA_SYS_n 21
- PMA_SYSPPOOL 21
- pMemReq 19
- pMemReqRtn 17
- PMS_DAT_n 21
- PMS_DATPOOL 21
- PMS_SYS_n 21
- PMS_SYSPPOOL 21
- pNetBufCfg 14
- _pNetPoolFuncTbl 13, 18
- polled mode 96
- pollRcv(), see xPollRcv()
- pollSend(), see xPollSend()
- POOL_FUNC 18, 19
- pools
 - attributes of (netBufLib) 13
 - creating (netBufLib) 12

- finding (netBufLib) 12
- freeing (netBufLib) 12
- parent and child (netBufLib) 13
- protocol state functionality, enabling for an interface 32
- pSockFuncTbl 123
- pSockIoctlMemVal 125
- pSockIoctlMemVal() 125
- pUnixIoctlMemVal 125
- pUnixIoctlMemVal() 125

Q

- q_get() 161
- q_insert() 161
- q_remove() 161
- qc shell command 178
- QJOB 100
 - allocation 101
 - example 101
 - reposting 102
- QoS 143
 - availability in Wind River Platforms 3
 - General Purpose Platform and 143
 - ingress filtering 167
 - ingress traffic prioritization 167
- qos shell command 180
- QOS_DEFAULT_PRI 169
- QOS_DEFER_PKT 168, 169, 170
- QOS_DELIVER_PKT 167, 170
- QOS_IGNORE_PKT 168, 170
- QOS_JOBQ 169
- QOS_JOBQ_PRI 169
- qosIngressHookSet() 171
- Quality of Service, *see* QoS
- queues
 - container queues 152, 159
 - Drop Precedence-Aware FIFO (dpaf) queues 157
 - fifo queues 156
 - Hierarchy Token Bucket Container (HTBC)
 - queues 160
 - interface output queues 152, 154
 - leaf queues 152, 156
 - Multiband Container (MBC) queues 160
 - network emulator (netemu) queues 157
 - registering 161

R

- RARP 28
 - server 28
- rcvif 12
- read()
 - socket descriptors 120
- read-modify-write cycle
 - shared memory and 46

- readRtn 121
- receive descriptors
 - allocating 66
- receiveRtn 91
 - when it is valid 95
- recvfromRtn 121
- recvmsgRtn 121
- recvRtn 121
- requeue() and requeue_locked() 162
- reset() 162
- restarting, network interfaces 32
- RFCs
 - RFC 1213
 - MIB structure, getting for network driver 78
 - packet statistics 103
 - RFC 1853
 - GIF tunnels 34
 - RFC 2002
 - GRE tunnels 34
 - minimal encapsulation 35
 - RFC 2233
 - MIB structure, getting for network driver 78
 - packet statistics 103
 - RFC 2473
 - GIF tunnels 34
 - RFC 2529
 - 6over4 tunnels 34
 - RFC 2674
 - VLAN, static objects 134
 - RFC 2697
 - single-rate, three-color marker 151
 - RFC 2784
 - GRE tunnels 34
 - RFC 2849
 - test and documentation addresses 3
 - RFC 2893
 - gif devices, style of configured tunnels 37
 - RFC 3056
 - 6to4 tunnels 34
- .rhosts 50
 - example 52
- route flags 184
- route shell command 182
- routed 52

S

- scMemValidate() 124
- send(), *see* xSend()
- sendmsgRtn 121
- sendRtn 121
- sendtoRtn 121
- setsockopt() 138
 - VLAN examples 139
- setsockoptRtn 121
- shared memory
 - location of 46

- size of 46
 - shared memory master CPU number 44
 - shared-memory anchor 45
 - initializing 44
 - shared-memory heartbeat 45
 - maintaining 44
 - shared-memory network 43
 - anchor 45
 - specifying in the boot line 45
 - configuring 50, 52
 - example configuration 51
 - heartbeat 45
 - interrupts, interprocessor 47
 - types 48
 - multiple on one backplane 44
 - object area 46
 - sequential addressing 48
 - size 46
 - TAS operation size 47
 - test-and-set instruction 46
 - test-and-set type 47
 - troubleshooting 54
 - VxMP and 44
 - shared-memory network master 44
 - shutdownRtn 121
 - Simple Marker 144
 - simple marker 151
 - SimpleMarker 148, 151
 - Single Rate Three Color Marker 144
 - single-rate, three-color marker 151
 - SIOCGIFFLAGS
 - example 32
 - SIOCGIFQUEUE 154
 - SIOCMUXL2PASSTHRU 125
 - SIOCMUXPASSTHRU 125
 - SIOCSIFQUEUE 154, 160
 - SIOCXADSFILTER 145
 - SIOCXADSMAP 146
 - SIOCXAIFQFILTER 155
 - SIOCXDDSFILTER 146
 - SIOCXDSCREATE 146
 - SIOCXDSDestroy 146
 - SIOXDDSMAP 147
 - SIZ_SYS_n 21
 - slab shell command 185
 - SM_ANCHOR_ADRS 53
 - SM_ANCHOR_OFFSET 52
 - SM_ANCHORS_ADRS 45
 - SM_CPUS_MAX 44, 54
 - SM_INT_* interrupt types 48
 - SM_INT_ARGn 48, 53
 - SM_INT_BUS 53
 - SM_INT_TYPE 48, 53
 - SM_MASTER 44, 53
 - SM_MAX_WAIT 53
 - SM_MEM_ADRS 46, 53
 - SM_MEM_SIZE 46, 53
 - SM_OBJ_MEM_SIZE 46, 53
 - SM_OFF_BOARD 52
 - SM_PKTS_SIZE 53
 - SM_TAS_HARD 47, 54
 - SM_TAS_SOFT 47, 54
 - SM_TAS_TYPE 47, 54
 - smEnd 43
 - smNetShow() 55
 - example 49
 - sample output 49
 - shared-memory network starting addresses, finding 49
 - SMP
 - ingress filtering and 167
 - snarf services 111
 - so_bkendaux 122, 123
 - SO_VLAN 138
 - SOCK_FUNC 121, 123
 - socket ioctls
 - memory validation 124
 - socket structure 120
 - socket() 119
 - socketRtn, *see* xSocketRtn()
 - sockets interface for a new network service
 - adding 118
 - functions 121
 - sockLib
 - standard socket interface 119
 - sockLibAdd() 120
 - sockLibInit() 120
 - sockLibInitRtn 121
 - sovlan structure 138
 - srTCM 148, 151
 - srTCM (single-rate, three-color marker) 151
 - start(), *see* xStart()
 - statistics, packet 102
 - stop(), *see* xStop()
 - sysBusTas() 47
 - sysctl shell command 187
 - sysIntConnect() 70
 - sysLib 47
 - sysSmAnchorAdrs 53
 - sysSmLevel 53
 - sysvar shell command 188
- ## T
- TAS operation size 47
 - tentative 29
 - test-and-set
 - shared memory use 46
 - test-and-set type 47
 - transmit descriptors
 - allocating 66
 - troubleshooting
 - shared-memory networks 54
 - Tunnel Interface support 34
 - tunneling 33
 - availability in Wind River Platforms 2

- configured 34
 - IPv6 packets using a gif 37
 - IPv6 packets using an stf 36
 - multicast 34
 - setting endpoints 34, 35
- tunnels
 - automatic 34
 - configured 34
 - creating 33
 - gif
 - example 38, 39
 - varieties of 34
- tuples 11
 - copying chains 12
 - creating manually 12
 - freeing chains 12
 - required by the stack 95
- tUsrRoot
 - role in network stack initialization 92

U

- unload(), *see* xUnload()
- user priority (802.1P) 134

V

- VID
 - changing 136
- VLAN 127
 - adding support for 128
 - disabling support for 131
 - management of 134
 - MUX-L2 extensions for 129
 - pseudo-interfaces, creating 136
 - setsockopt() examples 139
 - show routines for 140
 - subnet-based 135
 - tag header 128
 - tagging, availability in Wind River Platforms 2
 - VID, changing 136
- VLAN frames
 - priority-tagged 128
 - untagged 128
 - VLAN-tagged 128
- VLAN Pseudo Interface support 128
- VLAN tag header 129
- vlanQosHook() 170
- VXB_DEVICE_ID 97
- vxbDmaBufLib 66
- vxbDrvUnlink() 67
- vxmux_null_buf 11

W

- WDB agent 96
 - snarf service 111
- WDB_COMM_END
 - snarf service 111
 - xPollRcv() 85
 - xPollSend() 83
- Wind River documentation 4
- Wind River Platforms, features unique to
 - 802.1Q VLAN tagging 2, 127
 - QoS 3, 143, 167
 - tunneling 2, 33
- write()
 - socket descriptors 120
- writeRtn 122

X

- xAcceptRtn() 123
- xAddrGet() 88
- xAttach() 93, 108
- xDetach() 93, 108
- xEndBind() 89, 93
- xFormAddress() 20, 87, 131
 - VLAN 132
- xIoctl() 72
 - template 73
 - xEndBind() and 89
- xIoctlRtn() 124
- xLoad() 94, 97
 - called by muxDevLoad() 92
 - devInstanceInit2() and 65
 - implementing 97
 - two-pass algorithm of 97
- xMCastAddrAdd() 81
 - template 81
- xMCastAddrDel() 82
 - template 82
- xMCastAddrGet() 83
 - template 83
- xPacketDataGet() 117, 131
 - VLAN 131
- xPollRcv() 85, 86
- xPollSend() 83
- xSend() 80
 - END implementation 80
- xShutdownRtn() 95
- xSocketRtn() 122
- xSockLibInit() 121
- xStackErrorRtn() 117
- xStackRcvRtn() 115
- xStackRestartRtn() 80, 118
- xStackShutdownRtn() 113
- xStart() 70, 94
 - template 70
- xStop() 71, 95

template [71](#)
xUnload() [67, 95](#)

Z

zbufRtn [121](#)