

# **Netaccess Series Release 7 QNX Neutrino Driver Programmer's Manual**

Document Number 800-1015-001 AC

Printed May 2001



## General Notices

Brooktrout Technology reserves the right to make changes or improvements to the product described in this manual at any time and without notice. Every attempt has been made to insure that the information contained in this document is accurate and complete. However, Brooktrout Technology will not be responsible for any inaccuracies or omissions in this or any of its other technical publications.

The software described in this manual is furnished under a license and may not be used or copied only in accordance with such license.

Brooktrout Technology will not be responsible for any loss of data or information resulting from the use of this product. In no event will Brooktrout Technology be liable for any incidental, consequential, or indirect damages (including but not limited to loss of business profits, business interruption, or loss of information) arising out of the use or inability to use this product. This includes any claim by any other party.

Copyright © 2001 by Brooktrout Technology, Inc. All rights reserved. Neither this publication nor any part of this publication may be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior written permission from Brooktrout Technology.

Printed In U.S.A.

## Trademarks

Instant ISDN Software, NS300, PRI-PCI, PRI-CPCI, PRI-ISA-24, PRI-ISA48, BRI-ISA8, and SMI are trademarks of Brooktrout Technology, Inc.

QNX is a registered trademark of QNX Software Systems, Ltd.

Other trademarks used in this publication are property of their respective companies.

## International Notice

Due to differing national regulations and approval requirements, certain Brooktrout products are designed for use only in specific countries, and may not function properly in a country other than the country of designated use. As a user of these products, you are responsible for ensuring that the products are used only in the countries for which they were intended. For information on specific products, contact Brooktrout Technology.

18 Keewaydin Drive  
Salem, NH 03079  
603-898-1800  
[www.brooktrout.com](http://www.brooktrout.com)

## Brooktrout Technical Support

For Brooktrout Technical Support, see *Appendix A*.

# Brooktrout Technology, Inc.

## Software License Agreement

### Proprietary Rights

The Software is subject to the protection of the copyright laws of the U.S. and foreign jurisdictions, which prohibit unauthorized copying and distribution of copyrighted works. The Software incorporates proprietary and confidential algorithms and techniques which are subject to legal protection as trade secrets. Brooktrout is the sole owner of all proprietary rights in the Software, except for certain portions which are proprietary to third party licensors of Brooktrout. The User is granted only those rights expressly conferred by this License Agreement.

### License

Brooktrout licenses the User to use the Software subject to and in accordance with the following provisions. The software is distributed with network interface boards or boards with network interface functionality that are manufactured and sold by Brooktrout ("Brooktrout Hardware"), and is licensed solely for use in connection with the Brooktrout Hardware. The Software, and modified versions thereof, may be operated only on the central processing unit of any computer served by one or more items of Brooktrout Hardware and may, where appropriate in connection with such use, be downloaded onto memory located on Brooktrout Hardware, and may be modified (if modification is otherwise permitted pursuant to the following provisions), reproduced and distributed only for purposes of such use. Any other use, modification, reproduction or distribution is expressly prohibited.

Licensing provisions applicable to particular Software products are as follows:

1. API, Application and Driver software and Downloadable Firmware distributed in the form of object code:
  - a. The User may incorporate the Software into his own work providing functional and value enhancements and may duplicate and distribute the resulting work as he chooses provided that the resulting work is designed solely for use in connection with Brooktrout Hardware and may be distributed only together with items of Brooktrout Hardware solely for use in connection with such items.
  - b. The User may not modify the Software nor decompile, reverse engineer, disassemble, or otherwise reduce the Software to a human perceivable form.
  - c. When the User incorporates the Software into his product, Brooktrout's copyright notice must be included in the new work.
2. API, Application or Driver software distributed in the form of source code:
  - a. The User may modify the Software and must incorporate it into his own work providing functional and value enhancements. He may duplicate and distribute the resulting work in object code form only provided that the resulting work is designed solely for use in connection with Brooktrout Hardware and may be distributed only together with items of Brooktrout Hardware solely for use in connection with such items. He may not distribute the Software in source form.
  - b. The Software is confidential and proprietary to Brooktrout and the User must protect it in a manner similar to the protection he affords his own confidential and proprietary information.
  - c. When the User incorporates the Software into his product, Brooktrout's copyright notice must be included in the new work.

The reproduction, distribution and modification rights provided above applies to all Software distributed by Brooktrout unless a specific license agreement stating otherwise is attached or part of a contract under which such Software is being provided. In those cases, the specific license agreement will apply.

### Distribution

Any distribution of the Software (including modified versions thereof) which is authorized hereby shall be made (a) in object form only; (b) only to purchasers of units of Brooktrout Hardware, or of products including Brooktrout Hardware, and (c) only pursuant to license agreements containing provisions substantially equivalent to those included herein with respect to the Software distribution. Except as expressly permitted hereby, the user may not distribute the Software, or any copy by transfer, lease, loan or any other means.

### Termination

The User's license to use the Software may be terminated by Brooktrout in the event of any failure to comply with the above restrictions or any other terms of this License Agreement. In the event of termination of the license, the User must destroy or return to Brooktrout all copies of the Software in his possession.

## **Limited Warranty**

Brooktrout warrants for a period of 90 days following delivery that the disk on which the Software is recorded and which is provided by Brooktrout is free from defects in materials and workmanship. Brooktrout does not warrant that operation of the Software will be uninterrupted or error-free, or that it will satisfy the User's requirements. BROOKTROUT DISCLAIMS ALL OTHER WARRANTIES EXPRESS OR IMPLIED, INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## **Limitation of Liability**

Brooktrout's entire liability and the User's exclusive remedy in connection with the Software will be the replacement of any disk not meeting the above limited warranty upon return of the disk to Brooktrout. In no event will Brooktrout be liable for damages, including any lost profits or other incidental or consequential damages, arising out of or related to the Software and its use, even if Brooktrout has been advised of the possibility of such damages.



# Table of Contents

## Chapter 1 – Overview

Introduction .....	1-1
QNX Driver Architecture and Layout .....	1-1
General Operation .....	1-2
Management Stream .....	1-2
Driver Library .....	1-3
Notes .....	1-3

## Chapter 2 – Installation and Operation

Introduction .....	2-1
Preinstallation Requirements .....	2-1
Installation Overview .....	2-1
Installation and Setup of the QNX Driver for Neutrino .....	2-1
Starting the Driver .....	2-2
Uninstalling the Driver .....	2-2
Running the Driver Test Programs .....	2-2
Tunable Parameters .....	2-4
Examples .....	2-5

## Chapter 3 – Driver Services

Introduction .....	3-1
IOCTL_NAI_RESET .....	3-3
IOCTL_NAI_SEND_SRECORD .....	3-4
IOCTL_NAI_SET_VTTY .....	3-5
IOCTL_NAI_CLEAR_VTTY .....	3-6
IOCTL_NAI_VTTY_READ .....	3-7
IOCTL_NAI_VTTY_WRITE .....	3-8
IOCTL_NAI_CTL_READ .....	3-9
IOCTL_NAI_CTL_WRITE .....	3-10
IOCTL_NAI_READ_CRASH_DATA .....	3-11
IOCTL_NAI_DEBUG .....	3-12
IOCTL_NAI_GET_ADAP_INFO .....	3-13
IOCTL_NAI_GET_STRM_INFO .....	3-14
IOCTL_NAI_SET_DCHAN .....	3-15
IOCTL_NAI_RUN_TEST .....	3-16
IOCTL_NAI_TX_HALT .....	3-17
IOCTL_NAI_TX_RESUME .....	3-18
IOCTL_NAI_RX_HALT .....	3-19

IOCTL_NAI_RX_RESUME .....	3-20
IOCTL_NAI_SET_LOW_WATER .....	3-21
IOCTL_NAI_SET_MGMT .....	3-22
IOCTL_NAI_SET_PULSE .....	3-23
IOCTL_NAI_HS_GET_STATUS .....	3-24
IOCTL_NAI_HS_STOP .....	3-25
IOCTL_NAI_HS_GET_DEVICE_STATUS .....	3-26
IOCTL_NAI_HS_QUIESCE .....	3-27
IOCTL_NAI_HS_USER .....	3-28
IOCTL_NAI_HS_AUTO .....	3-29
IOCTL_NAI_HS_DISABLE .....	3-30
IOCTL_NAI_HS_INSERT .....	3-31
IOCTL_NAI_HS_REMOVE .....	3-32

## Chapter 4 – Hot Swap

Introduction .....	4-1
Implementation .....	4-1
Utility .....	4-6

## Chapter 5 – Library Services

Introduction .....	5-1
Functional Library .....	5-4
Control Path .....	5-4
Control Messaging .....	5-4
NaiControlRead .....	5-5
NaiControlWrite .....	5-6
VTTY Handling .....	5-7
NaiVttyRead .....	5-7
NaiVttyWrite .....	5-8
NaiSetVTTY .....	5-9
NaiClearVTTY .....	5-10
Data Messaging .....	5-11
NaiDataRead .....	5-11
NaiDataWrite .....	5-12
NaiTxHalt .....	5-13
NaiTxResume .....	5-14
NaiRxHalt .....	5-15
NaiRxResume .....	5-16
Device Management .....	5-17
NaiOpenAdapter .....	5-17
NaiRunDiags .....	5-18
NaiCloseAdapter .....	5-19
NaiResetAdapter .....	5-20
NaiDownloadAdapter .....	5-21
NaiReadCrashData .....	5-22
NaiOpenChannel .....	5-23

NaiCloseChannel .....	5-24
NaiSetMgmt .....	5-25
NaiSetPulse .....	5-26

## Appendix A – Brooktrout Customer Support

Customer First .....	A-1
Before You Call .....	A-1
Information About Your System .....	A-1
Contacting Brooktrout Customer Support .....	A-2
Additional Brooktrout Support Services .....	A-3

## Appendix B – Using the Virtual TTY Diagnostic Port Program

Starting the Virtual TTY .....	B-1
Displaying Memory .....	B-1
Viewing the Diagnostics Menu .....	B-4
Version String .....	B-4
Give/Take Indexes .....	B-4
Control Message Queues .....	B-5
Interrupt Status Block .....	B-5
Tracing Link Activity .....	B-6
Running a Level 1 Trace .....	B-6
Running a Level 2 Trace .....	B-8
Interpreting the I Frame Header .....	B-9
Interpreting the Message Headed .....	B-10
Interpreting Information Element .....	B-11



# List of Figures

Figure 1-1 Driver and Library Relationship .....	1-3
Figure B-1 Memory Map for PCI Boards .....	B-2
Figure B-2 Memory Map for cPCI Boards .....	B-3
Figure B-3 Level 1 Trace Example .....	B-7
Figure B-4 Level 2 Trace Example .....	B-9
Figure B-5 I Frame Formats .....	B-10
Figure B-6 Message Structures .....	B-11
Figure B-7 IE Formats .....	B-12



# List of Tables

Table 2-1 Driver Test Programs .....	2-2
Table 3-1 Driver Services .....	3-1
Table 5-1 Control Messaging .....	5-1
Table 5-2 VTTY Handling .....	5-1
Table 5-3 Data Messaging .....	5-2
Table 5-4 Device Management .....	5-2
Table 5-5 Hot Swap Management .....	5-2
Table B-1 Trace Values & Meanings .....	B-8
Table B-2 Q.931 Message Types .....	B-11
Table B-3 Q.931 Information Element Identifiers .....	B-13





# Overview

## Introduction

This Programmer's Manual describes the installation, operation and driver utilities provided by Release 7 of the WAN PCI/cPCI OEM Driver for Neutrino; this Driver release supports the Instant ISDN Software™, Release 7 family.

This document has been designed for Neutrino system developers who are using WAN PCI or cPCI boards in their application. It assumes experience with the Neutrino environment and Neutrino system tools, the C programming language, and using intelligent controller boards in a CompactPCI or PCI -bus environment.

Have the following publications on hand when using the QNX Driver:

- *Instant ISDN Software SMI Reference* defines the Simple Message Interface (SMI™) control messages used to implement advanced telecommunications and data networking services using Brooktrout boards.
- Appropriate *Technical Description* for the board(s) used.
- Appropriate information in the Neutrino documentation set.

## QNX Driver Architecture and Layout

The QNX Driver allows the application programmer complete access to the functionality of the Netaccess Series 7 family of PCI and CompactPCI ISDN controllers in a Neutrino environment.

The QNX Driver can support any number of Netaccess Series 7 controllers in a single chassis. Several parameters of the driver can be tuned for optimal use in a given configuration. For information about these parameters, see [Chapter 2](#).

The driver, as provided, contains two components: the driver itself (devt-nai) and an interface library (nailib.a). Communication to the driver is done through standard file I/O calls: open()/close(), read()/write() for data messaging and devctl() for all other function including transmit and receipt of Instant ISDN SMI control messages.

## General Operation

It is important to understand that operations under Neutrino are synchronous. Operations on a file descriptor (**open**, **close**, **read**, **write**, **devctl**) will pend until completed and can not overlap. That is, a read from one process which is pending may cause operations from any other process to pend as well until the original read completes. The QNX driver is designed to give a developer a way to avoid deadlock situations like this and to try and maximize the throughput of the driver.

For open and close processing, the driver tries to do as little work as possible and avoids any operations that might cause this thread to pend. On close this is not entirely avoidable. Close processing will attempt to disable a channel if the user application has “forgotten” to do so. In doing this, the close operation may need to pend until there is room on the L4L3 SMI queue. Unless an application has opened many channels and subsequently forgotten to issue disables before shutting down, the likelihood of having to wait for room on the L4L3 SMI message queue is very small.

For write operations, the driver makes a copy of the data to be sent. This allows the driver to complete the write operation immediately therefore allowing an application to issue multiple write operations and fill the channels transmit queue. It also allows an application developer freedom over where buffers are allocated from as there is no implicit ownership of the buffer by the driver for (potentially) long periods of time.

For read operations (control or data), the driver makes use of the Neutrino “pulse” mechanism. This is a service that allows the driver to send an asynchronous message to an application with a minimum of data (one long word). The driver uses this to indicate to an application when (and what form of) data has arrived. One pulse will be sent for every read operation that can be immediately completed, so an application can be simply coded to wait for pulses and issue the appropriate read command for each pulse type that can be received. The actual value of the pulses needs to be configured and can be done so per file descriptor (see [Chapter 3](#) and [Chapter 5](#) for more information). Neutrino itself places restrictions on how pulses can be used, please refer to the Neutrino documentation for details on this.

## Management Stream

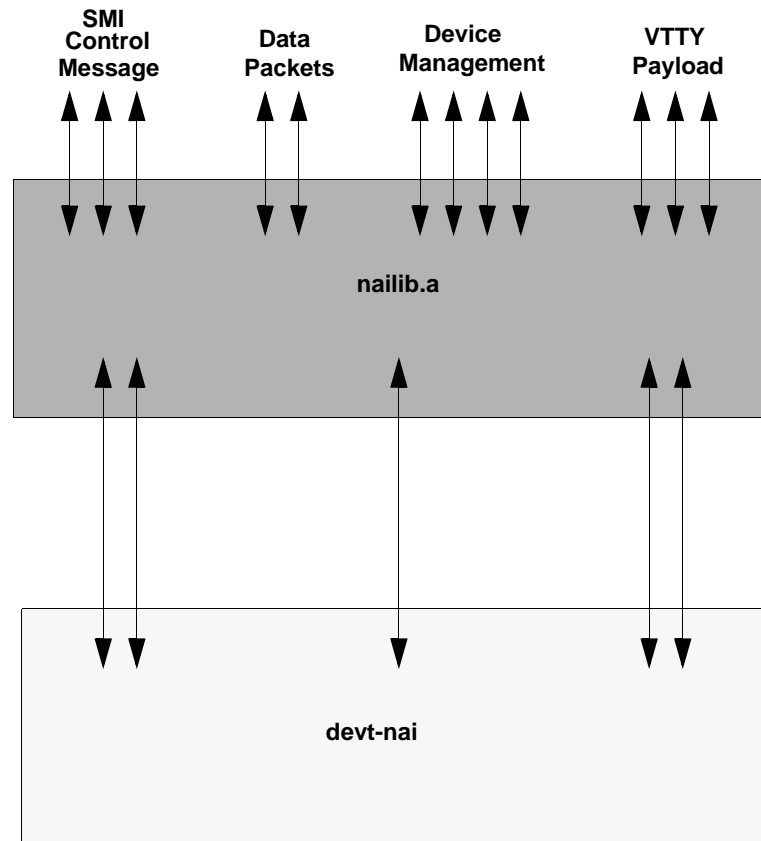
The Neutrino driver implements a multiplexing/demultiplexing of SMI control messages. This means that responses to commands issued on a particular file descriptor will only be received on that file descriptor. However, some L3L4 SMI control messages are globally applicable. In other cases, it may be possible for the driver to receive messages for which a destination file descriptor can not be found (say a message arrives late or loses a race condition with close processing). To cover for these cases the Neutrino driver implements the concept of a management channel. Essentially a management channel is a channel designated to be the receptacle for all messages that are otherwise undeliverable.

One management channel is allowed for each controller in the chassis. Any attempt to enable a second management channel will be erred (EBUSY). The management function is cleared when the management stream is closed. If a management stream has not been declared, all messages that would have been delivered are silently discarded.

# Driver Library

The library provides a simplified interface based on functional boundaries: Control, Data, Management and a diagnostics connection referred to as the Virtual TTY (VTTY) Port.

*Figure 1-1* illustrates the relationship of the driver and the library.



**Figure 1-1. Driver and Library Relationship**

It is anticipated that the interface library in nailib.a will provide an early prototype platform and stepping off point for more complex programming scenarios. The library is not designed as the “final” programming API.

## Notes

The driver, as delivered, is not in an “end customer” usable form. It is anticipated that an application programmer will integrate some or all of the binaries provided in their own system integration.

This OEM driver kit includes all source code to allow free modification and distribution of the driver kit.





# Installation and Operation

## Introduction

This chapter describes how to install the driver for QNX Neutrino.

## Preinstallation Requirements

The preinstallation requirements for the QNX driver are that a Brooktrout PCI card or CompactPCI card be installed into your system chassis; see the appropriate Brooktrout Technical Description for further details on the correct installation of the controllers. The system requirements are for 8MB of free disk space.

## Installation Overview

Installation of the QNX driver is done via the QNX “install” utility.

## Installation and Setup of the QNX Driver for Neutrino

To install the QNX Driver:

1. Login to QNX as the 'root' user and 'cd' to the directory where you want the driver installed.
2. Iso9660fsys & ' to load CD driver assume it mounts cd as '/cd0'  

```
'install -u /cd0/naidrv_tar.gz'
```

**Note:** Alternatively, if you have obtained the release from the Brooktrout FTP site (naidrv\_tar.gz), you can type 'install -u naidrv\_tar.gz'.

3. When the install script asks if you want to continue, enter 'Y'.
4. When Setup is complete, you will need to build the driver and sample sources to complete the installation. To do this, enter 'make' at the command prompt.

## Starting the Driver

The driver can be started as a background process by typing: `./devt-nai &` at the command prompt. An optional file name parameter can be passed to the driver on startup. This file must contain assignments of the tunable parameters (discussed further in [Chapter 2](#)).

## Uninstalling the Driver

Before installing a new version of the driver, Brooktrout recommends uninstalling the previous driver first.

To uninstall the Driver:

1. Change directories to where you installed the driver.
2. Issue a `'rm -R *'`.

## Running the Driver Test Programs

After driver installation, two directories contain sample test applications which demonstrate usage of the QNX driver API:

- `/test` contains general unit test programs ([Table 2-1](#))
- `/atp` contains a series of test programs designed by the driver review team for acceptance testing.

**Table 2-1. Driver Test Programs**

Test Name(s)	Purpose/Summary
test/download	A sample of how to download the adapter.
test/test2 Test/test	Downloads the controller and allows a channel to be started and data to be sent and received.
test/vtty	A simple virtual terminal for the IISDN diagnostic cards. Allows access to the diagnostic port on the Brooktrout controller without requiring a physical cable be plugged into the diag port.
atp/t1_192	Configure and enable all 192 64Kb T1 channels; pass data. Requires loopback plugs in all spans of the transition module. Supports CPCI cards only.
atp/e1_248	Configure all 8 spans as E1 and enable all channels; passes data in loopback mode. Requires loopback plugs in all spans of the transition module. Supports CPCI cards only.
atp/h110_256	Map all channels to H.110 bus; Configure as H110 Master; pass data. Requires an H.110 loopback plug ( <i>cPCI only</i> )

Table 2-1. Driver Test Programs (Continued)

Test Name(s)	Purpose/Summary
atp/t1_hy8	Enables all T1 interfaces with 3 superchannels (8x64kps); passes data in loopback mode. Requires loopback plugs in all spans of the transition module. Supports CPCI cards only.
atp/sr16h110_256	Maps 256 16Kb substrate channels to the H.110 bus; configures the Brooktrout board as H.110 Master; pass data. Requires an H.110 loopback plug ( <i>cPCI only</i> )
atp/sr16t1_256	Maps 256 16Kb channels to/from T1; pass data in loopback. Requires loopback plugs in all spans of the transition module. Supports CPCI cards only.
test/down2	A second example of downloading the adapter using the provided library
test/smi.c	Reads an SMI message and does a raw dump of the data
atp/2boards.c	Verifies the operation of 2 boards
atp/atp.c	A library of common functions to the ATP programs
atp/r256.c	Verifies relay operation on 256 channels
atp/rcomplex.c	Verifies relay operation on a complex set of rules
atp/rcopy.c	Verifies the copy operation of relay
atp/rdefer.c	Verifies the deferred operation of relay
atp/rlatmeas.c	These two applications are used to measure the latency of the relay module
atp/rlatwait.c	These two applications are used to measure the latency of the relay module
atp/tx_rxsmi.c	Verifies sending/receiving SMI messages
atp/udpecho.c	Verifies operation of UDP layer
atp/udprelay.c	Verifies operation of UDP in conjunction with the relay module.
test/crash.c	Dumps the IISDN logout block from a board crash.
atp/hy22.c	Tests hyperchannel.
atp/288.c	Opens 288 data channels and passes data. This tests greater than 256 lapdids.
/atp/384.c	Opens 384 data channels and passes data. This tests greater than 256 lapdids.
/atp/geo.c	This test geographic addressing.
/atp/pktest.c	This is a packet test.
/atp/rxflood.c	Floods the receive queue.
/atp/txflood.c	Floods the transmit queue.
/hs/autohs.c	Set auto mode for hot swap.
/hs/boot.c	Boots the adapter.
/hs/devstat.c	Displays the device status. Various information.
/hs/hs.c	A library of common functions to the hot swap programs.

**Table 2-1. Driver Test Programs (Continued)**

Test Name(s)	Purpose/Summary
/hs/insert.c	Call hot swap insert IOCTL in user mode.
/hs/quiesce.c	Call hot swap quiesce IOCTL in user mode.
/hs/remove.c	Call hot swap remove IOCTL in user mode.
/hs/stat.c	Displays the hot swap status and various parameter information.
/hs/userhs.c	Set user mode for hotswap.

## Tunable Parameters

You may need to “tune” some Brooktrout board parameters to optimize your application's performance or to eliminate conflict with other devices. You can modify tunable QNX Driver parameters through the use of a configuration file. If you modify parameter values, you must restart the QNX Driver, since the driver reads this information only when it is started.

The QNX Driver has the following controllable parameters. These parameters will read from the config file on the driver startup only. A sample configuration file is provided in `./driver/config`.

The recognized parameters are:

- `NaiDbgFlags` (default of 0; range 0 to 0xffff)
- `NaiDchanDescrAddr` (default of 0xfe3f9000)
- `NaiDchanEventQaddr` (default of 0xfe3fe000)
- `NaiNumDchanDescr` (default of 256 + 32)
- `NaiNumL34DchanEvent` (default of 1024)
- `NaiNumL43DchanEvent` (default of 0)
- `NaiNumTXBufs` (default of 8; range 3 to 255)
- `NaiNumRXBufs` (default of 8; range 3 to 255)
- `NaiRXBufSize` (default of 1520; range 2 to 16384)
- `NaiTXBufSize` (default of 1520; range 2 to 16384)
- `NaiLowWaterMarkDefault` (default of 2; range 0 to 255)
- `NaiFilterStatus` (default of 0; range 0 to 1)
- `NaiPendTxFlow` (default of 1; range 0 to 1).
- `NaiBufPoolSize` (default of 4 meg) This is for IISDN 6.7.59 and above. This parameter configures the buffer pool size in megabytes. Ie 4 = 4meg.
- `NaiHotSwap` (default is 0 = off 1 = on) This parameter turns hot swap on or off.
  - ◆ `NaiHotSwap = 0` The driver will not support hot swap
  - ◆ `NaiHotSwap = 1` The driver `hs_type` is in `HS_AUTO` mode
  - ◆ `NaiHotSwap = 2` The driver `hs_type` is in `HS_USER` mode.
- This parameter turns hot swap on or off.

Each line of the configuration file can contain:

- A comment (any line starting with '#')
- Blank space

- An assignment

Assignment of a tunable can be either of the following formats:

- “variable = value” or
- “variable value”

Variables are not case sensitive. Values can be in hex, decimal or octal.

## Examples

- `NaiRXBufSize = 240`
- `NaiTXBufSize = 512`
- `NaiFilterStatus 1`
- `NaiNumTXBufs 0xF`



# Driver Services

## Introduction

This chapter describes the Driver services available to the application through Neutrino system calls. [Table 3-1](#) lists each service and describes its purpose; the services are listed in alphabetical order in the subsections that follow.

**Table 3-1. Driver Services**

Service	Purpose	
Devctl	IOCTL_NAI_RESET	Halts an adapter. No input.
	IOCTL_NAI_SEND_SRECORD	Downloads ACSII S-records from the IISDN bootfile to the WAN controller.
	IOCTL_NAI_SET_VTTY	Identifies this handle as a VTTY recipient.
	IOCTL_NAI_CLEAR_VTTY	Removes this handle as the VTTY recipient.
	IOCTL_NAI_VTTY_READ	Reads data from the VTTY data buffer area.
	IOCTL_NAI_VTTY_WRITE	Writes data to the VTTY; data buffer area.
	IOCTL_NAI_CTL_READ	Reads an SMI control queue entry; data buffer is input.
	IOCTL_NAI_CTL_WRITE	Writes an SMI control queue; data buffer is input.
	IOCTL_NAI_READ_CRASH_DATA	Read the logout block, interrupt status block and other pertinent crash information of an adapter; used for IISDN debugging.
	IOCTL_NAI_DEBUG	Sets the driver debug bit mask, which controls the level of debug messages printed by the driver at run time.
	IOCTL_NAI_GET_ADAP_INFO	Returns information about the selected adapter.
IOCTL_NAI_GET_STRM_INFO	Returns information about this handle; data buffer is input.	

Table 3-1. Driver Services

Service	Purpose	
Devctl	IOCTL_NAI_SET_DCHAN	Allocates a particular data channel descriptor number to this handle.
	IOCTL_NAI_RUN_TEST	Initiates a diagnostic self test on the adapter.
	IOCTL_NAI_TX_HALT	Halts the transmit data pump on this handle.
	IOCTL_NAI_TX_RESUME	Re-initiates the transmit data pump on this channel.
	IOCTL_NAI_RX_HALT	Stops the receive data pump on this handle
	IOCTL_NAI_RX_RESUME	Resumes the receive data pump on this handle.
	IOCTL_NAI_SET_LOW_WATER	Sets the Low water mark for this data channel.
	IOCTL_NAI_SET_PULSE	Indicates to the driver which Channel ID should receive pulses for events on this handle. Also defines what the pulse values should be for all pulse types used by the driver.
	IOCTL_NAI_SET_MGMT	Identifies this handle as being the one that should receive all global or otherwise undeliverable L3L4 SMI messages.
	IOCTL_NAI_HS_GET_STATUS	Displays board level and hot swap information.
	IOCTL_NAI_HS_STOP	Resets the adapter.
	IOCTL_NAI_HS_GET_DEVICE_STATUS	Displays adapter device information.
	IOCTL_NAI_HS_QUIESCE	Used in Hot Swap User mode. Allows the user to quiesce the board for hot swap removal.
	IOCTL_NAI_HS_USER	Sets hot swap mode to USER mode.
	IOCTL_NAI_HS_AUTO	Sets hot swap mode to AUTO mode
	IOCTL_NAI_HS_DISABLE	Sets hs_type to HS_NONE
	IOCTL_NAI_HS_INSERT	Used in Hot Swap User Mode. Allows the user to insert a card and initialize the software that supports the adapter.
IOCTL_NAI_HS_REMOVE	Used in Hot Swap User mode. Allows the user to remove a card from the chassis.	
IOCTL_NAI_HS_GROW	Not supported at this time.	

## IOCTL\_NAI\_RESET

**Structure**

```
#include "nailib.h "  
int  devctl(fd,IOCTL_NAI_RESET, NULL, 0, NULL);  
int  fd;
```

**Usage** This ioctl puts the controller into a halted and reset state. After this ioctl, the controller will only respond to an IOCTL\_NAI\_SEND\_SRECORD.

**Example**

```
int  err;  
err = devctl(fd, IOCTL_NAI_RESET, NULL, 0, NULL);
```

**Returns** **EOK** - The card has been reset.

## IOCTL\_NAI\_SEND\_SRECORD

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_SEND_SRECORD, buffer, length, NULL);
int  fd;
char *buffer;
int  length;
```

**Usage** This ioctl takes a buffer containing an S-record, converts it to binary and downloads it to the Brooktrout adapter. When it sees an S-record of S9 or S7, the ioctl will start the controller.

'buffer' must contain a single, complete and valid S-Record.

'length' is the byte length of the 'buffer' string.

**Example**

```
int  err, len;
err = devctl(fd, IOCTL_NAI_RESET, NULL, 0, NULL);
if (err) exit(-1);
if ((fp = fopen("naid.0", "r")) == NULL) {
    printf("Can't open naid.0\n");
    return;
}
while (!feof(fp)) {
    fgets(buffer, 256, fp);
    len = strlen(buffer);
    err = devctl(fd, IOCTL_NAI_SEND_SRECORD, buffer, len,
                NULL);

    if (err) {
        printf("Can't write to controller.\n");
        return;
    }
}
```

**Returns**

- EOK** - A good s-record was received and processed
- EINVAL** - The S-record given was somehow not properly formed
- EIO** - The adapter is already running or failed to start.

## IOCTL\_NAI\_SET\_VTTY

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_SET_VTTY,NULL, 0,NULL);
int  fd;
```

**Usage** This ioctl identifies a particular file descriptor (fd) as the recipient of any and all VTTY data from the controller. It also enables VTTY processing on the controller. Note that only one fd can be marked as the VTTY recipient for a given controller.

**Example**

```
int  err;
err = devctl(fd, IOCTL_NAI_SET_VTTY, NULL, 0, NULL);
```

**Returns** **EOK** - The VTTY has been assigned to this fd.  
**EBUSY** - Another fd has already claimed the VTTY.

## IOCTL\_NAI\_CLEAR\_VTTY

**Structure**

Structure

```
#include "nailib.h"
int devctl(fd, IOCTL_NAI_CLEAR_VTTY, NULL, 0, NULL);
int fd;
```

**Usage**

This ioctl disassociates a particular file descriptor as being the recipient of VTTY data and disables any VTTY processing on the controller.

**Example**

```
int err;
err = devctl(fd, IOCTL_NAI_CLEAR_VTTY, NULL, 0, NULL);
```

**Returns**

**EOK** - The VTTY has been released.

**EACCES** - This fd does not currently own the VTTY and therefore can't release it.

---

## IOCTL\_NAI\_VTTY\_READ

- Structure**            `#include "nailib.h"`  
`int    devctl(fd, IOCTL_NAI_VTTY_READ, buffer, length, NULL);`  
`int    fd;`  
`char   *buffer`  
`int    length`
- Usage**                This ioctl reads VTTY data from the controller. For this ioctl to succeed, a `IOCTL_NAI_SET_VTTY` must have been issued on this same fd. The ioctl will return `ENOENT` if data is not immediately available.
- 'buffer' is where the data read will be placed.
- 'length' is the maximum amount of data that 'buffer' can contain. This ioctl will return the number of bytes actually read.
- Example**              `char   buffer[256];`  
`int    length = sizeof(buffer);`  
`int    err;`  
`err = devctl(fd, IOCTL_NAI_VTTY_READ, buffer, length, NULL);`  
`if (err < 0)   exit(1);`  
`for (i = 0; i < bytes_returned; i++) {`  
`printf("%c", buffer[i]);`  
`}`
- Returns**              **EOK** - VTTY data has been read.  
**EACCES** - This fd does not own the VTTY.  
**ENOENT** - There is no VTTY data to read.  
**EACCES** - length of the data received.

## IOCTL\_NAI\_VTTY\_WRITE

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_VTTY_WRITE,buffer, length, NULL);
int  fd;
char *buffer
int  length
```

**Usage** This ioctl writes VTTY data to the controller. For this ioctl to succeed a IOCTL\_NAI\_SET\_VTTY must have been issued on the same fd. The ioctl will pend until there is room available in the VTTY queue.

'fd' is the file descriptor on which a previous IOCTL\_NAI\_SET\_VTTY was issued.

'buffer' contains the data to be written.

'buffer\_length' is the number of bytes to write.

**Example**

```
char *buffer = "d fe0f0000 40\n";
int  length = strlen(buffer);
int  err;
err = devctl(fd, IOCTL_NAI_VTTY_WRITE, buffer, length, NULL);
```

**Returns**

- EOK** - Data was written to the VTTY.
- EACCES** - This fd does not own the VTTY.
- ENOSPC** - There is no space available to write the VTTY message.

## IOCTL\_NAI\_CTL\_READ

### Structure

```
#include "nailib.h"

int devctl(fd, IOCTL_NAI_CTL_READ, L34msg_buffer,
           L34msg_buffer_length, NULL);

int fd;
char *L34msg_buffer;
int L34msg_buffer_length = sizeof(L3_to_L4_struct);
int info;
```

### Usage

This ioctl reads a SMI Control L3L4 message from the L3L4 SMI message queue.

'fd' is the file descriptor on which the control read event is to occur.

'L34msg\_buffer' is the buffer to receive the message into. 'L34msg\_buffer\_length' is the length of the buffer and should always be at least 512 bytes.

### Example

```
L3_to_L4_struct buffer;
int info, err;
err = devctl(fd, IOCTL_NAI_CTL_READ, (char *) buffer,
            sizeof(L3_to_L4_struct), &info);

if (err) {
    ..do error handling..
}

switch (buffer.msgtype) {
    case L3L4mALERTING:
        ....
}
}
```

### Returns

**EOK** - An L3L4 SMI message has been read.

**ENOENT** - There are no L3L4 SMI messages waiting to be read.

**EINVAL** - Buffer is not large enough to hold an L3L4 message.

## IOCTL\_NAI\_CTL\_WRITE

### Structure

```
#include "nailib.h"
int devctl(fd, IOCTL_NAI_CTL_WRITE, L43msg_buffer,
           L43msg_buffer_length, NULL);

int fd;
char *L43msg_buffer;
int L43msg_buffer_length = sizeof(L4_to_L3_struct);
int info;
```

### Usage

This ioctl writes an L4L3 SMI control message to the L4L3 SMI message queue. 'l43msg\_buffer' is the L4L3 message buffer to write. 'l43msg\_buffer\_length' is the length of the buffer and should always be 512 bytes. 'info' does not contain any meaningful data upon completion.

### Example

```
L4_to_L3_struct buffer;
int info, err;
buffer.msgtype = L4L3mREQ_LINE_STATUS;
err = devctl(fd, IOCTL_NAI_CTL_WRITE, (char *) buffer,
             sizeof(L4_to_L3_struct), &info);
```

### Returns

**EOK** - A L4L3 SMI message has been written.

**ENOMEM** - There was not enough memory to allocate the receive and transmit buffers for this particular L4L3mENABLE\_PROTOCOL message.

**ENOSPC** - There was no space available on the L4L3 SMI queue for this message.

**EINVAL** - Buffer is not large enough to hold an L4L3 message.

## IOCTL\_NAI\_READ\_CRASH\_DATA

**Structure**

```
#include "nailib.h"
int devctl(fd, IOCTL_NAI_READ_CRASH_DATA, buffer, length,
           NULL);

int fd;
NaiCrashData_t*buffer;
int length = sizeof(NaiCrashDump_t);
```

**Usage** This ioctl is used to retrieve information about a controller crash dump. It is primarily used for IISDN software debugging.

'buffer' is where to put the crash dump information. 'length' should be at least size of (NaiCrashDat\_t).

**Example**

```
NaiCrashData_tbuffer;
int err;
err = devctl(fd, IOCTL_NAI_READ_CRASH_DATA, &buffer,
             sizeof(NaiCrashData_t), NULL);
```

**Returns** **EOK** - The crash dump area has been read.

## IOCTL\_NAI\_DEBUG

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_DEBUG, debug_value, length, NULL);
int  fd;
short *debug_value;
int  length;
```

**Usage** This ioctl is used to set the debug bit mask which controls the level of driver debug output printed at run time. This ioctl is useful primarily for driver debugging purposes only.

Debug bitmask: values can be found in NAILIB.H.

'debug\_value' points to a short containing the bitmask requested.

'length' should be 2.

**Example**

```
short debug_flag = 0x0123;
int  err;
err = devctl(fd, IOCTL_NAI_DEBUG, &debug_flag, sizeof(short), NULL);
```

**Returns** **EOK** - The debug flag has been set.

## IOCTL\_NAI\_GET\_ADAP\_INFO

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_GET_ADAP_INFO, buffer, length, NULL);
int  fd;
char *buffer;
int  length;
```

**Usage** This ioctl returns the adapter structure associated with this fd. It is useful primarily for debugging purposes only.

'fd' is the file descriptor on which the adapter information is to occur.

'buffer' is where to place the adapter structure read.

'length' must be at least 'sizeof(NaiAdapter\_t)'.

**Example**

```
NaiAdapter_tbuffer;
int          err;
err = devctl(fd, IOCTL_NAI_GET_ADAP_INFO, &buffer,
            sizeof(NaiAdapter_t), NULL);
```

**Returns** **EOK** - The adapter structure has been read.

## IOCTL\_NAI\_GET\_STRM\_INFO

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_GET_STRM_INFO, buffer, length, NULL);
int  fd;
char *buffer;
int  length;
```

**Usage** This ioctl returns the stream structure associated with this fd. It is useful primarily for debugging purposes only.

'fd' is the file descriptor on which the stream information event is to occur.

'buffer' is where to place the stream structure read.

'length' must be at least 'sizeof(NaiStream\_t)'.

**Example**

```
NaiStream_t buffer;
int         err;
err = devctl(fd, IOCTL_NAI_GET_STRM_INFO, &buffer,
            sizeof(NaiStream_t), NULL);
```

**Returns** **EOK** - The stream structure has been read.

---

## IOCTL\_NAI\_SET\_DCHAN

### Structure

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_SET_DCHAN, &channel_number,
           sizeof(channel_number), NULL);
int  fd;
short channel_number;
```

### Usage

This ioctl sets (or can be used to reassign) the data channel descriptor number associated with this fd. This value is used to determine which d channel in the SMI the fd will read and write data messages to and from.

'channel\_number' points to a short containing the actual data channel number (between 0 and 255).

**Note:** The data channel number can also be set when issuing an L4L3mENABLE\_PROTOCOL message.

**Note:** This mechanism allows for two fds to claim the same d channel. This is not a suggested mode of operation, but the driver does not disallow it.

### Example

```
short channel_number = 5;
int  err;
err = devctl(fd, IOCTL_NAI_SET_DCHAN, &channel_number,
           sizeof(short), NULL);
```

### Returns

**EOK** - The dchannel number has been set.

## IOCTL\_NAI\_RUN\_TEST

<b>Structure</b>	<pre>#include"nailib.h" int  devctl(fd, IOCTL_NAI_RUN_TEST, NULL, 0, NULL); int  fd;</pre>
<b>Usage</b>	<p>This ioctl is used to start the diagnostic POST (Power On Self Test).</p>
<b>Example</b>	<pre>int  err; err = devctl(fd, IOCTL_NAI_RUN_TEST, NULL, 0, NULL);</pre>
<b>Returns</b>	<p><b>EOK</b> - The POST test was started. <b>EIO</b> - The POST failed.</p>

## IOCTL\_NAI\_TX\_HALT

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_TX_HALT,NULL, 0,NULL);
int  fd;
```

**Usage** This ioctl is used to stop the transmission of data on a given fd. Instant ISDN will stop transmitting data as soon as the current buffer transmission is complete. The ioctl will not complete until IISDN has acknowledged that transmission has been halted.

**Example**

```
int  err;
err = devctl(fd, IOCTL_NAI_TX_HALT, NULL, 0, NULL);
```

**Returns** **EOK** - The transmit stream has been halted.

## IOCTL\_NAI\_TX\_RESUME

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_TX_RESUME, NULL, 0, NULL);
int  fd;
```

**Usage** This ioctl is used to restart the transmission of data on a given fd after it has been stopped by an IOCTL\_NAI\_TX\_HALT ioctl. Instant ISDN will start transmitting data immediately upon receiving this call. The ioctl will not complete until IISDN has acknowledged that transmission has been restarted.

**Example**

```
int  err;
err = devctl(fd, IOCTL_NAI_TX_RESUME, NULL, 0, NULL);
```

**Returns** **EOK** - The transmit stream has been restarted.

## IOCTL\_NAI\_RX\_HALT

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_RX_HALT, NULL, 0,NULL);
int  fd;
```

**Usage** This ioctl is used to stop the reception of data on a given fd. Instant ISDN will stop receiving data as soon as the current data buffer has been received. The ioctl will not complete until Instant ISDN has acknowledged that reception has been halted.

**Example**

```
int  err;
err = devctl(fd, IOCTL_NAI_RX_HALT, NULL, 0, NULL);
```

**Returns** **EOK** - The receive stream has been halted.

## IOCTL\_NAI\_RX\_RESUME

**Structure**

```
#include "nailib.h"
int devctl(fd, IOCTL_NAI_RX_RESUME, NULL, 0, NULL);
int fd;
```

**Usage**

This ioctl is used to restart the reception of data on a given fd after it has been stopped by an IOCTL\_NAI\_RX\_HALT ioctl. Instant ISDN will start receiving data immediately upon receiving this call. The ioctl will not complete until IISDN has acknowledged that reception has been restarted.

**Example**

```
int err;
err = devctl(fd, IOCTL_NAI_RX_RESUME, NULL, 0, NULL);
```

**Returns**

**EOK** - The receive stream has been restarted.

---

## IOCTL\_NAI\_SET\_LOW\_WATER

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_SET_LOW_WATER_MARK, &water_mark,
           sizeof(water_mark), NULL);
short water_mark=2;
int  err;
```

**Usage** This ioctl resets the IISDN data channel descriptor low water mark for controlling the number of transmit complete interrupts (for more information, refer to the SMI Programmer's Guide). A default value is set when the fd is opened (value depends on the current driver tunable parameter "LowWaterMark").

'water\_mark' points to a short value containing the requested level.

'length' should be 2.

**Example**

```
short water_mark=2;
int  bytes;
err=devctl(fd, IOCTL_NAI_SET_LOW_WATER_MARK, &water_mark,
          sizeof(water_mark), NULL) {
```

**Returns** **EOK** - The low water mark has been reset.

## IOCTL\_NAI\_SET\_MGMT

**Structure**

```
#include"nailib.h"
int  devctl(fd, IOCTL_NAI_SET_MGMT, NULL, 0, NULL);
int  fd;
```

**Usage**

This ioctl is used to identify a particular file descriptor as being the one which should receive all SMI messages that are global in nature are otherwise not deliverable.

**Example**

```
int  err;
err = devctl(fd, IOCTL_NAI_SET_MGMT, NULL, 0, NULL);
```

**Returns**

**EOK** - The management channel stream has been marked.

**EBUSY** - Another stream has already claimed the management role.

## IOCTL\_NAI\_SET\_PULSE

**Structure**

```
#include "nailib.h"
int    devctl(fd, IOCTL_NAI_SET_PULSE, &pulse, sizeof(pulse),
          NULL);

NaiPulse_t pulse;
```

**Usage** This ioctl associates a given Channel ID with a file descriptor and defines the pulse code values that will be used to indicate the arrival of SMI Control data, VTTY data or Received data on a data channel descriptor. The driver will use the given Channel ID to create a connection to the application on which it will send these pulses to indicate the arrival of new work.

**Example**

```
#define CONTROL_PULSE_CODE      0x11 // arbitrary
#define VTTY_PULSE_CODE        0x22 //  choices
#define DATA_PULSE_CODE      0x33

int    err;
NaiPulse_t  pulse;

pulse.chid = ChannelCreate(_NTO_CHF_UNBLOCK | _NTO_CHF_DISCONNECT);
if (pulse.chid == -1)  exit(1);
pulse.ctl_pulse = CONTROL_PULSE_CODE;
pulse.vtty_pulse = VTTY_PULSE_CODE;
pulse.data_pulse = DATA_PULSE_CODE;
err = devctl(fd, IOCTL_NAI_SET_PULSE, &pulse,
          sizeof(pulse), NULL);
```

**Returns** **EOK** - The id has been accepted.

## IOCTL\_NAI\_HS\_GET\_STATUS

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_HS_GET_STATUS, NULL, 0, NULL);
int  fd;
```

**Usage**

This ioctl displays information about the adapter including: board type, board state, bus and slot, shared memory base, shared memory length, maximum shared memory length, i/o memory base, i/o memory length, interrupt number, interrupt id, hot swap state, hot swap type, hot swap control and status register contents.

**Example**

```
int  err;
err  =  devctl(fd, IOCTL_NAI_HS_GET_STATUS, NULL, 0, NULL);
```

**Returns**

EOK

## IOCTL\_NAI\_HS\_STOP

**Structure**            `#include "nailib.h"`  
                         `int    devctl(fd, IOCTL_NAI_HS_STOP, NULL, 0, NULL);`  
                         `int    fd;`

**Usage**                **This ioctl calls NaiReset(adapter) and reset the adapter.**

**Example**              `int    err;`  
                         `err = devctl(fd, IOCTL_NAI_HS_STOP , NULL, 0, NULL);`

**Returns**              **EOK**

## IOCTL\_NAI\_HS\_GET\_DEVICE\_STATUS

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_HS_GET_DEVICE_STATUS, NULL, 0, NULL);
int  fd;
```

**Usage** This ioctl displays various adapter device information such as: vendor id, device id, class codes, revision id, bus number, device number, status register, command register, header type, BIST, latency timer, cache line size, sub vendor id, subsystem id, max latency, min gnt, pci interrupt pin, interrupt line, capabilities pointer and BAR 0-3.

**Example**

```
int  err;
err  = devctl(fd, IOCTL_NAI_HS_GET_DEVICE_STATUS, NULL, 0, NULL);
```

**Returns** EOK

## IOCTL\_NAI\_HS\_QUIESCE

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_HS_QUIESCE, NULL, 0, NULL);
int  fd;
```

**Usage**

This ioctl is used for hot swap and it will quiesce the board and prepare the software for a removal. This can be used in hot swap USER mode only.

**Example**

```
int  err;
err = devctl(fd, IOCTL_NAI_HS_QUIESCE, NULL, 0, NULL);
```

**Returns**

EOK

## IOCTL\_NAI\_HS\_USER

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_HS_USER, NULL, 0, NULL);
int  fd;
```

**Usage** This ioctl sets the `hs_type` to `USER` mode. Which allows the user to utilize ioctls to control hot swap activity.

**Example**

```
int  err;
err  =  devctl(fd, IOCTL_NAI_HS_USER, NULL, 0, NULL);
```

**Returns** EOK - switched to user mode successfully.

## IOCTL\_NAI\_HS\_AUTO

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_HS_AUTO, NULL, 0, NULL);
int  fd;
```

**Usage** This ioctl sets the `hs_type` to AUTO mode which allows the software to automatically control hot swap activity.

**Example**

```
int  err;
err = devctl(fd, IOCTL_NAI_HS_AUTO, NULL, 0, NULL);
```

**Returns** EOK

## IOCTL\_NAI\_HS\_DISABLE

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_HS_DISABLE, NULL, 0, NULL);
int  fd;
```

**Usage** This ioctl turns the `hs_type` to `HS_NONE` and turns off hot swap functionality.

**Example**

```
int  err;
err  = devctl(fd, IOCTL_NAI_HS_DISABLE, NULL, 0, NULL);
```

**Returns** EOK

## IOCTL\_NAI\_HS\_INSERT

**Structure**

```
#include "nailib.h"
int  devctl(fd, IOCTL_NAI_HS_INSERT, NULL, 0, NULL);
int  fd;
```

**Usage** This ioctl is used for hot swap and it should be called after the user has inserted the adapter into the slot and closed the latches. This notifies the software that a card is in the slot and ready for activity. This can be used in hot swap USER mode only.

**Example**

```
int  err;
err = devctl(fd, IOCTL_NAI_HS_INSERT, NULL, 0, NULL);
```

**Returns** EOK

## IOCTL\_NAI\_HS\_REMOVE

**Structure**

```
#include "nailib.h"
int  devctl(fd IOCTL_NAI_HS_REMOVE, NULL, 0, NULL);
int  fd;
```

**Usage**

This ioctl is used for hot swap and it should be called after the user has quiesced the. This notifies the software that a card is in the slot and ready for removal. This can be used in hot swap USER mode only.

**Example**

```
int  err;
err  =  devctl(fd, IOCTL_NAI_HS_REMOVE , NULL, 0, NULL);
```

**Returns**

EOK



# Hot Swap

## Introduction

This section describes the Netaccess Series 7 QNX/Neutrino Driver Hot Swap (HS) implementations of the driver and host application usage. The Hot Swap feature is supported on the NS300 CompactPCI cards only.

## Implementation

To activate this feature, the user must first set **NaiHotSwap = 1** in the config file when the driver is first started. This notifies the driver to support hotswap.

There are two ways that hot swap can be implemented. The first is the default setting which has an **AUTO** mode. This implementation is activated at boot up time and it sets the **hs\_type** to **HS\_AUTO**. All hot swap activity is now to be controlled automatically through the driver. The driver polls the hot swap control and status register every 1.5 seconds. If there is a change in state such as **INS\_STAT** or **REM\_STAT** goes high, the driver will take the appropriate action.

On insertion **INS\_STAT = 1**, the driver will re-read the pci-bios information and re-allocate the software resources for the adapter in which time the software will reset. The **INS\_STAT** bit will then clear and which will also clear the Blue LED on the adapter signaling the user that it is now time to boot the adapter.

In the case of **REM\_STAT = 1**, the driver will automatically do a quiesce and removal. This will de-allocate the software resources and mask interrupts for this adapter. The driver will then clear the **REM\_STAT** bit and cause the LED to be illuminated, signaling the user that it is OK to physically remove the adapter from the chassis.

The second implementation of hot swap is **USER** mode. This allows the user to control Hot Swap activity through library calls. The **NaiHsUser()** library call will notify the driver to set **hs\_type = HS\_USER**. See the sample application `userhs.c` in `/hs` for an example.

## Sample Code from userh.c

```

/* userhs.c 4/4/01 edl
   Switch hs_type to USER mode
   parameter: bus and slot

   example:
   ./userhs 1 10
*/

#include <sys/neutrino.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include "nailib.h"

main(int argc, char **argv)
{
    int          err,fd, bus, slot;

    bus = atoi(argv[1]);
    slot = atoi(argv[2]);

    fd = NaiOpenAdapter(bus,slot);

    if (fd < 0){
        printf("Unable to open adapter\n");
        return -1;
    }
    err = NaiHsUser(fd);
    if (err)
        printf("NaiHsUser err %d\n",err);
    return;
}

```

When this mode is selected, the **ENUM** bit in the Hot Swap control and status register is turned on, thereby disabling the use of the **INS\_STAT** and **REM\_STAT** bits. When the user wants to do a removal, they must first call **NaiHsQuiesce()**; see example `quiesce.c` in **/hs**. This will mask interrupts and de-allocate resources.

## Sample Code from `quiesce.c`

```

/* quiesce.c 4/4/01 edl
   Hot Swap Queisce
   parameters: bus and slot

   example:
   ./quiesce 1 10
*/

#include <sys/neutrino.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include "nailib.h"

main(int argc, char **argv)
{
    int          err,fd, bus, slot;

    bus = atoi(argv[1]);
    slot = atoi(argv[2]);

    fd = NaiOpenAdapter(bus,slot);

    if (fd < 0){
        printf("Unable to open adapter\n");
        return -1;
    }
    err = NaiHsQuiesce(fd);
    if (err)
        printf("NaiHsQuiesce err %d\n",err);
    return;
}

```

Secondly, the user must call **NaiHsRemove()**; see the example in `remove.c` in **/hs**. This will remove the adapter and illuminate the Blue LED notifying the user that it is now time to physically remove the adapter from the chassis. When the user wants to insert the adapter, this is accomplished by physically inserting the adapter and calling **NaiHsInsert()**; see `insert.c` in **/hs** for an example. With this call, the driver will unmask interrupts, read pci-bios information, re-allocate software resources, clear the Blue LED and reset the adapter making it ready for boot up.

**Sample Code from remove.c**

```
/* remove.c 4/4/01 edl
   Hot Swap Remove
   parameter: bus and slot

   example:
   ./remove 1 10
*/

#include <sys/neutrino.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include "nailib.h"

main(int argc, char **argv)
{
    int          err,fd, bus, slot;

    bus = atoi(argv[1]);
    slot = atoi(argv[2]);

    fd = NaiOpenAdapter(bus,slot);

    if (fd < 0){
        printf("Unable to open adapter\n");
        return -1;
    }
    err = NaiHsRemove(fd);
    if (err)
        printf("NaiHsRemove err %d\n",err);

    return;
}
```

## Sample Code from insert.c

```

/* insert.c 4/4/01 edl
   Hot Swap Insert

   parameters: bus and slot

   example:
   ./insert 1 10
*/

#include <sys/neutrino.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include "nailib.h"

main(int argc, char **argv)
{
    int          err, fd, bus, slot, chid;

    bus = atoi(argv[1]);
    slot = atoi(argv[2]);

    fd = NaiOpenAdapter(bus,slot);

    if (fd < 0){
        printf("Unable to open adapter\n");
        return -1;
    }
    err = NaiHsInsert(fd);
    if (err)
        printf("NaiHsInsert err %d\n",err);

    return;
}

```

The user can reboot the card (see `boot.c` in `/hs` for an example). To switch back to auto mode, use the library call `NaiHsAuto()`; see `autohs.c` in `/hs` for an example. This clears the `ENUM`, `INS_STAT` and `REM_STAT` bits in the hot swap control and status register, and sets `hs_type` to `HS_AUTO`.

## Sample Code from autohs.c

```
/* autohs.c 4/4/01 edl
   Switch hs_type to AUTO mode
   parameter: bus and slot

   example:
   ./autohs 1 10

*/

#include <sys/neutrino.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include "nailib.h"

main(int argc, char **argv)
{
    int          err,fd, bus, slot;

    bus = atoi(argv[1]);
    slot = atoi(argv[2]);

    fd = NaiOpenAdapter(bus,slot);

    if (fd < 0){
        printf("Unable to open adapter\n");
        return -1;
    }
    err = NaiHsAuto(fd);
    if (err)
        printf("NaiHsUser err %d\n",err);
    return;
}
```

Turning off Hot Swap use with the **NaiHsDisable()** library call will change `hs_type` to **HS\_NONE** and no hot swap activity will be recognized by the driver.

## Utility

There are two utilities in the `/hs` directory: `devstat.c` and `stat.c`. Both applications display various information about the device and hot swap states.

To execute the stat utility. Change directory to the **/hs** directory and type:

```
./stat 1 10
```

Where **1** is the bus and **10** is the slot number. The following output will be displayed by the driver:

```
*****  
Board Info For -- IISDN_CPCI_RESPIN  
State -- AS_RUNNING Bus 1 Slot 10  
Global Control Register xc9  
Shared memory base xfc000000 length x 2000000  
Maximum Shared memory length x 2000000  
I/O memory base xfe6fef00 len x      100  
Interrupt Number 9 ID 6  
  
Hot Swap State BRKT_S3  Type HS_AUTO  
Hot Swap Control & Status register info:  
ENUM_MASK is x0          HS_LOO is x0  
HS_REM_STAT is x0       HS_INS_STAT is x0  
*****
```

To execute the devstat utility, change directory to **/hs** and type the following command:

```
./devstat 1 10
```

## Sample Code from stat.c

```
/* stat.c 4/4/01 edl
   Hot Swap Status
   parameter: bus and slot

   example:
   ./stat 1 10
*/

#include <sys/neutrino.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include "nailib.h"

main(int argc, char **argv)
{
    int          err,fd, bus, slot;

    bus = atoi(argv[1]);
    slot = atoi(argv[2]);

    fd = NaiOpenAdapter(bus,slot);

    if (fd < 0){
        printf("Unable to open adapter\n");
        return -1;
    }
    err = NaiHsStatus(fd);
    if (err)
        printf("NaiHsStatus err %d\n",err);
    return;
}
```

---

Where 1 is the bus and 10 is the slot number. The following output will be displayed by the driver:

```
Vendor ID           = 1011h
Device ID           = 46h
Class_Codes         = 6800h
Revision ID         = 1h
Bus Number          = 1
Device number       = 10
Status Reg          = 290h
Command Reg         = 117h
Header type         = 0h
BIST                = 0h
Latency Timer       = 40h
Cache Line Size     = 8h
Sub Vendor ID       = 11ceh
Subsystem ID        = 102h
Max Lat             = 20ns
Min Gnt             = 2ns
PCI Int Pin         = INT 1
Interrupt line      = 9
Capabilities Pointer = dch
BAR[0]              = fe6ff000h
BAR[1]              = dc01h
BAR[2]              = fe6fef00h
BAR[3]              = fc000000h
```



# Library Services

## Introduction

This chapter describes the QNX Driver Library available for use in application development. [Table 5-1](#) through [Table 5-4](#) lists each library function and describes its purpose; the functions appear in alphabetical order throughout this chapter.

**Table 5-1. Control Messaging**

Control Message Handling	Purpose
NaiControlRead	Read an L3L4 SMI control message from the adapter.
NaiControlWrite	Write an L4L3 SMI control message to the adapter.

**Table 5-2. VTTY Handling**

VTTY Handling	Purpose
NaiVttyRead	Read data from the VTTY port; useful for diagnostic checking and application debugging.
NaiVttyWrite	Write diagnostic commands to the VTTY port.
NaiSetVTTY	Selects this handle as being the VTTY recipient for the adapter.
NaiClearVTTY	Releases this particular handle as being the VTTY recipient for the adapter.

**Table 5-3. Data Messaging**

<b>Data Handling</b>	<b>Purpose</b>
NaiDataRead	Data is read from the data channel descriptor, up to the length of bytes requested.
NaiDataWrite	Data is written to the data channel descriptor, up to the length of bytes requested.
NaiTxHalt	Stops an outgoing data stream on the data channel descriptor.
NaiTxResume	Restarts an outgoing data stream on the data channel descriptor.
NaiRxHalt	Stops an incoming data stream on the data channel descriptor.
NaiRxResume	Restarts an incoming data stream on the data channel descriptor.

**Table 5-4. Device Management**

<b>Device Management</b>	<b>Purpose</b>
NaiOpenAdapter	Opens the adapter and associates the file descriptor with a particular adapter card.
NaiCloseAdapter	Closes and releases a previously open handle.
NaiResetAdapter	Places adapter in a reset mode.
NaiDownloadAdapter	Downloads Instant ISDN software to the adapter.
NaiReadCrashData	Reads the crash dump area on the adapter.
NaiOpenChannel	Opens an adapter and associates a data channel descriptor to the selected file descriptor.
NaiCloseChannel	Closes and release a previously open handle.
NaiRunDiags	Initializes diagnostic self test.
NaiSetMgmt	Associates the file descriptor as the recipient of all SMI management messages or otherwise lost L3L4 SMI control messages.
NaiSetPulse	Sets the pulse value.

**Table 5-5. Hot Swap Management**

<b>Hot Swap Management</b>	<b>Purpose</b>
NaiHsAuto(int board)	Sets hot swap to HS_AUTO mode
NaiHsDisable(int board)	Sets hot swap to HS_NONE.
NaiHsUser(int board)	Sets hot swap to HS_USER

---

<b>Hot Swap Management</b>	<b>Purpose</b>
NaiHsQuiesce(int board)	Removes software resources in preparation for a hot swap removal.
NaiHsRemove(int board)	De-allocates software resources for hot swap removal.
NaiHsInsert(int board)	Allocates software resources for hot swap insertion.
NaiHsStatus(int board)	Displays various board status and information.
NaiHsDevStatus(int board)	Displays various device level status and information.

## Functional Library

The following section describes the functional interface that is provided with the driver in a very cursory manner.

## Control Path

The following subsections illustrate the function calls which are used for passing SMI messages to and from the driver.

## Control Messaging

These routines are used to pass SMI L4L3 and L3L4 messages to and from the adapter.

---

# NaiControlRead

**Structure**

```
#include          "nailib.h"
int              NaiControlRead(fd, buffer);
HANDLE          fd;
L3_to_L4_struct  buffer
```

**Usage** This call reads an L3L4 SMI control message from the adapter specified by the given handle. The call will pend until a message is available.

**Example**

```
L3_to_L4_struct  msg;

memset(&msg, 0, sizeof(msg));
if (NaiControlRead(fd, &msg) {
    error_out("Can't read control message.");
}
switch (msg.msgtype) {
    case L3L4mALERTING:
        ...
}
```

**Returns**

- = 0 Success.
- > 0 Return code holds error.

## NaiControlWrite

**Structure**

```
#include          "nailib.h"
int              NaiControlWrite(fd,buffer)
HANDLE          fd
L4_to_L3_struct  *buffer;
```

**Usage** This call writes an L3L4 SMI control message to the adapter specified by the given handle. The call will return ENOSPC if there isn't any room to write a new control message.

**Example**

```
L4_to_L3_struct  msg;

memset(&msg, 0, sizeof(msg));
msg.msgtype = L4L3mREQ_LINE_STATUS;
if (NaiControlWrite(fd, &msg)) {
    error_out("Can't write control message.");
}
```

**Returns**

```
= 0    Success.
> 0    Return code holds error.
```

---

# VTTY Handling

These messages are used for reading, writing and managing the IISDN VTTY port. These messages are not supported on WAN adapters.

## NaiVttyRead

**Structure**

```
#include    "nailib.h"
int        NaiVttyRead(fd, buffer, length);
HANDLE    fd;
char      *buffer;
int       *length;
```

**Usage** This call reads up to 'length' bytes of data from the VTTY port on the adapter specified by the given 'fd'. VTTY data is stored in 'buffer'. Upon completion, 'length' contains the actual number of bytes read. This call will return ENOENT if there isn't any data immediately available.

**Example Output**

```
char  buffer[256];
int   length;

length = sizeof(buffer);
if (NaiVttyRead(fd, buffer, &length)) {
    error_out("Can't read data.");
}
for (i = 0; i < length; i++) {
    printf("%c", buffer[i]);
}
```

**Returns**

```
= 0    Success.
> 0    Failed. Return code holds error.
```

# NaiVttyWrite

**Structure**

```
#include    "nailib.h"
int        NaiVttyWrite(fd, buffer, length);
HANDLE     fd;
char       *buffer;
int        *length;
```

**Usage**            This call writes up to 'length' bytes of data to the VTTY port on the adapter specified by the given 'fd'. Data is stored in 'buffer'. Upon completion, 'length' contains the actual number of bytes written. This call will pend until the requested data can be written to the VTTY port.

**Example Output**

```
char  ch;
int   length;

length = 1;
ch = getchar();
if (NaiVttyWrite(fd, &ch, &length)) {
    error_out("Can't write data.");
}
printf("Wrote %d byte(s).\n", length);

Returns            = 0    Success.
                     > 0    Failed. Return code holds error.
```

# NaiSetVTTY

**Structure**

```
#include    "nailib.h"
int         NaiSetVTTY(fd);
HANDLE     fd;
```

**Usage**            This call selects this fd as being the VTTY recipient for the adapter.

**Returns**

```
    = 0    Success.
    > 0    Failed. Return code holds error.
```

## NaiClearVTTY

**Structure**

```
#include    "nailib.h"
int        NaiClearVTTY(fd);
HANDLE     fd;
```

**Usage**            This call releases this particular fd as being the VTTY recipient for the adapter specified by the fd.

**Returns**

```
    = 0    Success.
    > 0    Failed. Return code holds error.
```

# Data Messaging

These messages are used to send and receive data on a particular Dchannel descriptor. The messages also allow some level of management in being able to halt and resume the transmit and receive side independently.

## NaiDataRead

**Structure**

```
#include    "nailib.h"
int        NaiDataRead(fd,buffer, length);
HANDLE    fd
char      *buffer;
int       *length
```

**Usage** This call reads up to 'length' bytes of data from the data channel descriptor specified by the given 'fd'. Data is stored in 'buffer'. Upon completion, 'length' contains the actual number of bytes read. This call will return ENOENT if there is no data immediately available.

**Example Output**

```
char  buffer[256];
int   length;

length = sizeof(buffer);
if (NaiDataRead(fd, buffer, &length)) {
    error_out("Can't read data.");
}
printf("Read %d bytes.\n", length);
```

**Returns**

```
= 0    Success.
> 0    Failed. Return code holds error.
```

## NaiDataWrite

**Structure**

```
#include    "nailib.h"
int        NaiDataWrite(fd, buffer, length);
HANDLE     fd
char       *buffer
int        *length
```

**Usage** This call writes up to 'length' bytes of data to the data channel descriptor specified by the given 'fd'. Data is stored in 'buffer'. Upon completion, 'length' contains the actual number of bytes written. If the 'NaiPendTxFlow' tunable is set to 0, this routine will return ENOSPC if there isn't any room on the transmit queue. If the tunable is set to 1, then the call will pend until room is available.

**Example Output**

```
char  buffer[256];
int   length;

length = sizeof(buffer);
memset(buffer, 0x55, length);
if (NaiDataWrite(fd, buffer, &length)) {
    error_out("Can't write data.");
}
printf("Wrote %d bytes.\n", length);
```

**Returns**

```
= 0    Success.
> 0    Failed. Return code holds error.
```

# NaiTxHalt

**Structure**

```
#include    "nailib.h"
int        NaiTxHalt(int stream);
int        stream
```

**Usage**            This call stops the outgoing data stream on the data channel descriptor specified by the given 'fd'. The call will pend until Instant ISDN acknowledges that the transmit stream has been stopped.

**Returns**

```
    = 0    Success.
    > 0    Failed. Return code holds error.
```

## Notes

## NaiTxResume

**Structure**

```
#include    "nailib.h"
int        NaiTxResume(stream);
int        stream;
```

**Usage**            This call restarts the outgoing data stream on the data channel descriptor specified by the given 'fd'. The call will pend until Instant ISDN acknowledges that the transmit stream has been restarted.

**Returns**

```
    = 0    Success.
    > 0    Failed. Return code holds error.
```

# NaiRxHalt

**Structure**

```
#include    "nailib.h"
int        NaiRxHalt(int stream);
int        stream;
```

**Usage**            This call stops the incoming data stream on the data channel descriptor specified by the given 'fd'. This call will pend until Instant ISDN acknowledges that the receive stream has been halted.

**Returns**

```
    = 0    Success.
    > 0    Failed. Return code holds error.
```

## NaiRxResume

**Structure**

```
#include    "nailib.h"
int        NaiRxResume(stream);
int        stream;
```

**Usage**            This call restarts the incoming data stream on the data channel descriptor specified by the given 'fd'. The call will pend until Instant ISDN acknowledges the receive stream has been restarted.

**Returns**

```
    = 0    Success.
    > 0    Failed. Return code holds error.
```

---

# Device Management

The routines are used to open close the driver as well as generally manage the device

## NaiOpenAdapter

**Structure**

```
#include    "nailib.h"
int         NaiOpenAdapter(bus_number, slot_number);
int         bus_number;
int         slot_number;
```

**Usage** This call opens the driver and associates it with a particular adapter card. The `bus_number` and `slot_number` are used to form the device name to be opened ("`/dev/naidrv%02x,%02x`", `bus_number`, `slot_number`).

**Returns**

```
>= 0    Successful open, return value is the fd.
<= 0    Open failed.return code holds error
```

# NaiRunDiags

**Structure**

```
#include    "nailib.h"
int        NaiRunDiags(fd);
fd         fd;
```

**Usage**            This call starts a diagnostic sequence on PCI cards.

**Returns**

```
=0        Success
>0        Run failed. Return code holds error.
```

# NaiCloseAdapter

**Structure**            `#include     "nailib.h"`  
                      `int           NaiCloseAdapter(fd);`  
                      `fd            fd;`

**Usage**                **This call closes and releases a previously open fd.**

**Returns**              `Always returns 0.`

## NaiResetAdapter

**Structure**

```
#include    "nailib.h"
int        NaiResetAdapter(fd);
fd         fd;
```

**Usage**            This call will place the adapter associated with 'fd' into a reset state.

**Returns**

```
    = 0    Success.
    > 0    Failed. Return code holds error.
```

# NaiDownloadAdapter

**Structure**

```
#include    "nailib.h"
int        NaiDownloadAdapter(fd, file_name);
int        fd;
char       *file_name;
```

**Usage** This routine downloads the adapter. The 'file\_name' is an S-Record file

**Note:** This call will pend until the controller image starts running.

## Example Output

```
fd = NaiOpenAdapter(bus_number, slot_number);
if (fd == -1){
    error_out("Can't open device.", fd, bus_number, slot_number);
}
if ((ret = NaiDownloadAdapter(fd, "nail.0")) {
    error_out("Can't download adapter.", ret);
}
```

**Returns**

- = 0 Download successful.
- > 0 Download failed. Return code holds error.  
errno holds the error.

## NaiReadCrashData

**Structure**

```
#include          "nailib.h"
int              NaiReadCrashData(fd, buffer);
int             fd;
NaiCrashData_t  *buffer;
```

**Usage**            This call reads the crash dump area on the adapter specified by the given fd. This information can be useful for Brooktrout support personnel in pin-pointing an exact IISDN problem.

**Returns**

```
    = 0    Success.
    > 0    Failed. Return code holds error.
```

# NaiOpenChannel

**Structure**

```
#include "nailib.h"
int NaiOpenChannel(bus_number, slot_number,
                  channel_number);

int bus_number;
int slot_number;
int channel_number;
```

**Usage** This call opens a particular adapter and associates a Data Channel descriptor with it.

**Returns**

- >= 0 Successful open, return value is the fd.
- < 0 Open failed. Return code holds error.

## NaiCloseChannel

**Structure**

```
#include    "nailib.h"
int        NaiCloseChannel (fd);
int        fd;
```

**Usage**            This call closes and releases a previously open fd.

**Returns**            Always returns 0.

# NaiSetMgmt

**Structure**

```
#include    "nailib.h"
int        NaiSetMgmt (fd);
int        fd;
```

**Usage**                Sets this file descriptor as the “management stream”. The management stream is a stream designated to receive all SMI messages that are 'global' in nature (that is, the may affect the status of multiple streams), or those that don't have any other delivery point.

**Returns**

```
>= 0    Successful.
< 0    Failed.    Return code holds error.
```

# NaiSetPulse

**Structure**

```
#include "nailib.h"

int NaiSetPulse(fd chid, ctl, vtty, data);
int fd;
int chid;
int ctl, vtty, data,
```

**Usage** Sets the channel id that should receive pulses for this fd. Sets the channel id and pulse codes that should receive pulses for this fd. Once set, pulses will be generated for all SMI messages that affect the status of the file descriptor, for all data arriving that is destined to this file descriptor and, if set, the arrival of all VTTY data. Pulse values are kept on a per file descriptor basis, so multiple file descriptors can have differing pulse values.

**Example**

```
#define CONTROL_PULSE_CODE 0x11 // arbitrary
#define VTTY_PULSE_CODE 0x22 // choices
#define DATA_PULSE_CODE 0x33

int err;
int chid;

chid = ChannelCreate(_NTO_CHF_UNBLOCK | _NTO_CHF_DISCONNECT);
if (chid < 0) exit(1);

err = NaiSetPulse(fd, chid, CONTROL_PULSE_CODE,
                 VTTY_PULSE_CODE,
                 DATA_PULSE_CODE);
```

**Returns**

```
>= 0 Successful.
< 0 Failed. Return code holds error.
```



# Appendix A

## Brooktrout Customer Support

### Customer First

Brooktrout is committed to delivering complete Customer Satisfaction. We endeavor to make all of our products reliable, easy to install and easy to use. If you need additional technical assistance after reading the *Technical Description* and *Programmer's Manual*, Brooktrout provides access to a wide range of service and support offerings.

### Before You Call

When contacting Brooktrout Customer Support, please call from a location where you can operate your system, including the ability to remove or look at the hardware components of the Brooktrout card. Also, please fill out the form below prior to calling, so your Customer Support Engineer can have a clear picture of your system and any potential conflicts between devices.

#### *Information About Your System*

Hardware Serial No.:
Board Model:

Operating System:	Release/Version:
IISDN Software Release/Version:	

---

Hardware Configuration (all devices in system, including Brooktrout Controller, video cards, ethernet cards, etc):

DEVICE	IRQ	I/O BASE	SHARED MEMORY BASE
(1)			
(2)			
(3)			
(4)			

Describe any application specific information (including scenario, protocol configuration, and connected equipment):

Concise summary of problem:

## Contacting Brooktrout Customer Support

Purchase of a Brooktrout Developer's Kit entitles you to 12-months of FREE, unlimited phone support during standard business hours. Our Technical Assistance Center also offers a wide variety of additional fee-based support services, including Extended Support Plans (24-Hour Support, Per-Call Support), Training Lab, Application Developer's Lab, On-Site Support, and more.

Brooktrout Customer Support can be reached via phone, fax, email and postal mail. Our support center is staffed Monday through Friday, 9:00am to 6:00pm, EST.

Please fill out and return the Warranty Registration Card supplied with your Brooktrout product. Your information will be entered into our Support Database for product tracking and will facilitate better customer support to your business center.

**Mailing Address**            Brooktrout Technology, Inc.  
   18 Keewaydin Drive  
   Salem, NH 03079

**Support Phone**            (800) 435-7926 (Toll Free)      9am - 6pm EST, Mon - Fri  
   (603) 890-7298 (Direct)        9am - 6pm EST, Mon - Fri  
   (603) 894-4545 (Fax)            24 hours

**eMail**                            support@brooktrout.com

## **Additional Brooktrout Support Services**

### **Web Page**

Using an Internet connection, you can access our web page on the World Wide Web:

**<http://www.brooktrout.com>**

From our web page, you can access the latest technical and product information and the latest versions of our device drivers, Instant ISDN software, FAQ files – you can even open a trouble ticket with a Brooktrout Support Engineer.





# Appendix B

## Using the Virtual TTY Diagnostic Port Program

This appendix describes the operations and activities of the Virtual TTY diagnostic monitor test program.

### Starting the Virtual TTY

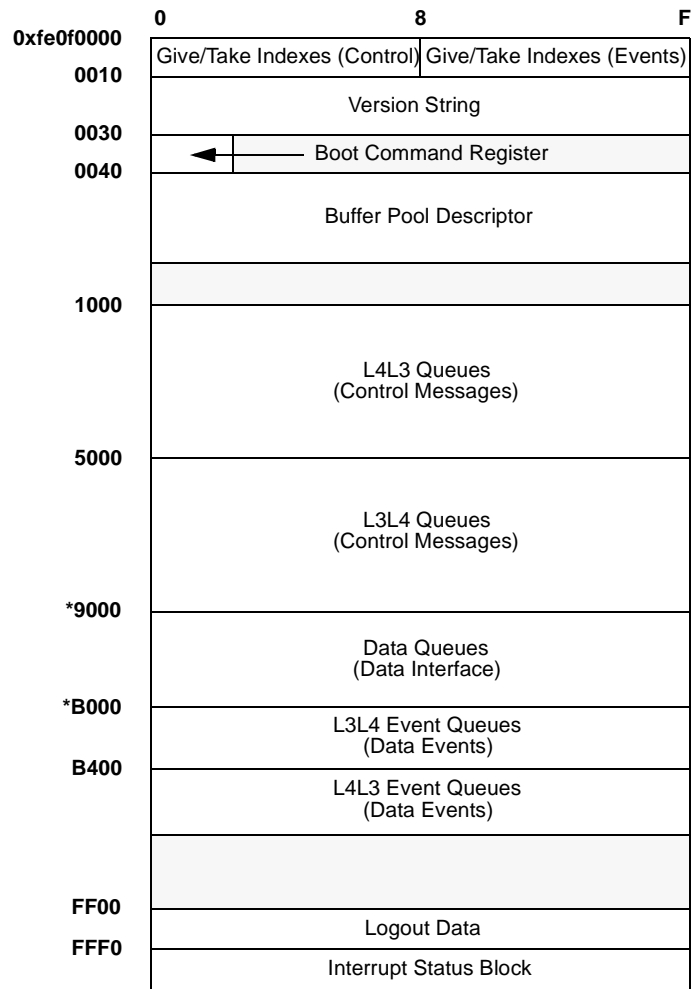
After the software and Brooktrout card have been installed on the machine, the Virtual TTY may be run. If you have not installed the driver, please refer to *Chapter 2* for instructions on conducting that operation.

To run the Virtual TTY, type './test/vtty<bus><slot>' at the command prompt.

### Displaying Memory

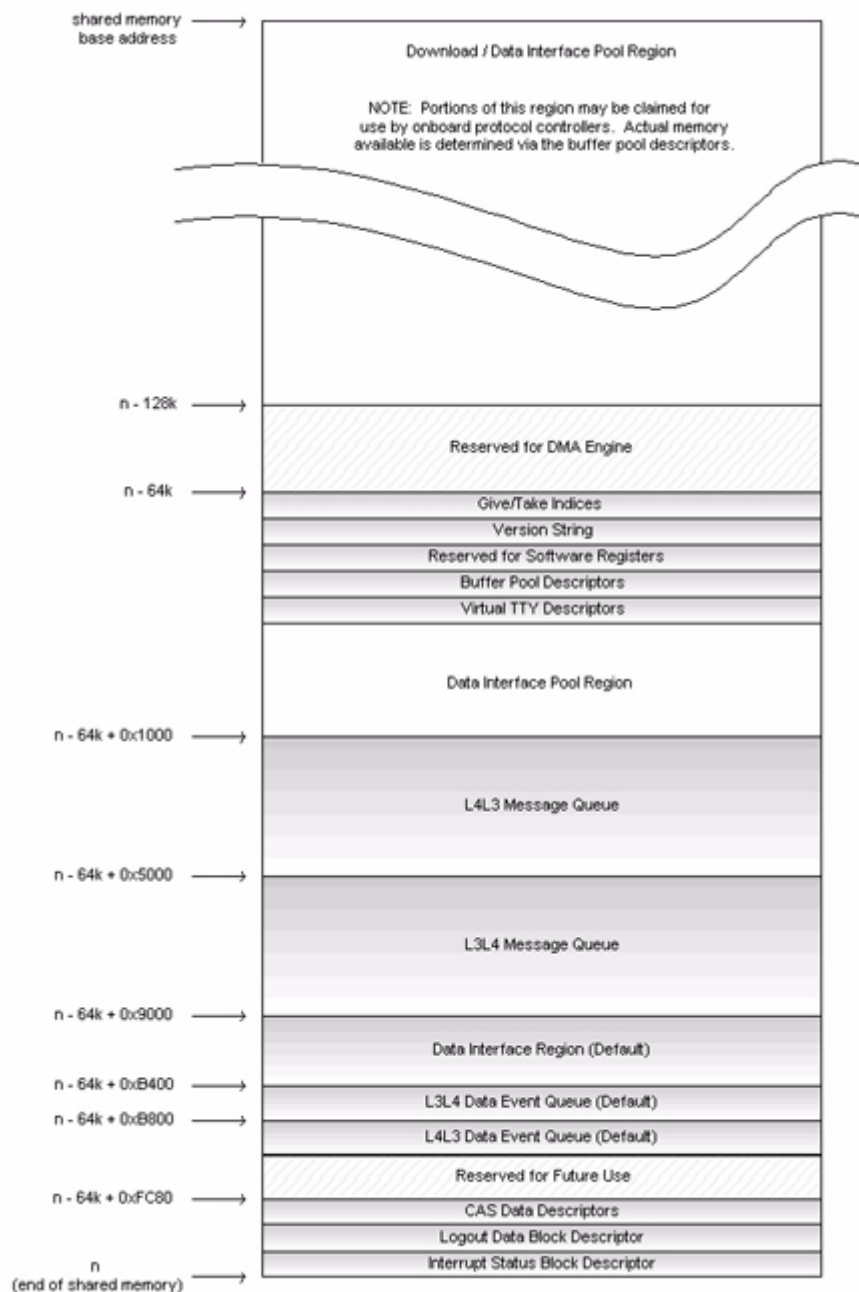
The PRI-PCI board initialization process downloads Instant ISDN Software and configures the board's memory map as shown in Figure B-1 and Figure B-2. The shared memory area starts at the address given by the command `sh inv`, which will display the shared memory base. The shared memory is then determined by the shared memory base plus `f0000`.

**Note:** PRI-cPCI and PRI-PCI adapter boards employ different base addresses. The base address for PRI-cPCI adapter boards is `0xfe3f0000`; the base address for PRI-PCI adapter boards is `0xfe0f0000`.



**Note:** Memory Map addresses for Data Queues and L3L4 Event Queues are configurable (*IISDN 6.8* and above). The addresses provided in this figure represent their default values.

**Figure B-1. Memory Map for PCI Boards**



**Figure B-2. Memory Map for cPCI Boards**

To display a 16-byte memory area, type **d** followed by a space, enter the address (**addr**) to display and press **Return**. You have the option of viewing a larger area of memory by entering a hexadecimal range (**len**). For example, to display a 0x100 area starting at 0xf000000, type

```
d f000000 100
```

and press **Return**.

Common memory areas you may want to display include:

- Version string
- Give/Take Indexes
- Control message queues

- **Interrupt Status Block**

Instructions for interpreting the displays for these areas are provided in the subsections that follow.

## Viewing the Diagnostics Menu

To view the diagnostics menu, press `?` or `sh ?` at any `>` prompt.

The diagnostic menu offers two functions:

- *Display buffer* can be used to display an area of the board's memory
- *Set link trace level* can be used to trace Layer 2 and Layer 3 ISDN message activity on active PRI spans



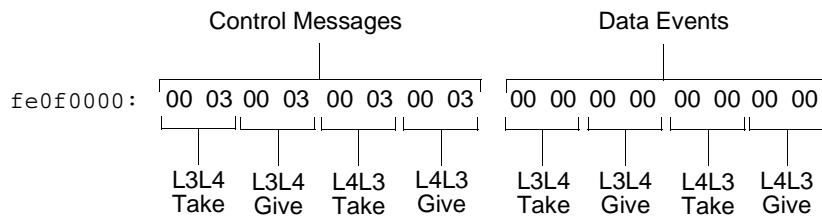
Do not use the read and write options under this menu (`rb addr`, `rw addr`, `rl addr`, `wb addr val`, `ww addr val`, and `wl addr val`). These options are intended for Brooktrout development use only and can adversely affect PCI board operation if not used properly.

## Version String

The board's version string occupies 0010 and 0020. Each hexadecimal byte in the string represents an ASCII character code.

## Give/Take Indexes

The Give/Take Indexes are located at off set 0000. The first 8 bytes in the address apply to SMI control messages passed over the control interface; the second 8 bytes apply to data events passed over the data interface. Two sets of Give/Take indexes, one for L4L3 (host to PCI board) messages and one for L3L4 (PCI board to host) messages, exist for each interface. A sample Give/Take index display for a PCI board that has successfully received and transmitted five control messages is shown below:



## Control Message Queues

Control message queues are located within offsets 1000 to 5000 (L4L3 messages) and 5000 to 9000 (L3L4 messages). Each control message queue contains 32 buffers. Because each buffer is 512 bytes long, the buffers start at 0x200 intervals. For example, following a board reset, the first L4L3 message is written to 1000, the second to offset 1200, the third to offset 1400, and so forth.

L4L3 messages contain an 8-byte common header; L3L4 messages contain a 12-byte common header. A sample L4L3mSET\_TSI message with the common header elements identified is displayed below:

L4 Reference

	Message ID	Call Reference	
	LAP-D ID		Logical Link ID

```

fe0f1000: 00 b5 00 00 00 00 00 00 20 14 01 04 00 1e 00 00
fe0f1010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fe0f1020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fe0f1030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

**Note:** L3L4 message headers contain additional information on the interfaces and B-channels used for the call; refer to the Instant ISDN Software SMI Reference for a complete description of the L3L4 common message header.

## Interrupt Status Block

The Interrupt Status Block (ISB) is located at FFF0. The PCI board writes to this memory area when it generates interrupts. During normal processing, the board writes an interrupt reason value of 0x04 (L3L4irsnNORMAL) to the first byte in this area. The PCI board also uses this area to signal a board failure and to supply additional error information.

A sample Interrupt Status Block that indicates a fatal processing error (0x03 L3L4irsnFATAL\_ERROR) due to an unknown interrupt (0xFB PRIerrorUNKNOWN\_INTERRUPT) on a PCI board is shown below:

	Error Code		Error Pointer	Error Argument	Timestamp		IRQ Level

```

fe0ffff0: 00 00 00 00 00 00 00 00 0e 00 ed 00 00 00 00 00

```

Refer to the C header file pri.h for Instant ISDN Software for a complete listing of interrupt reason codes and error codes. In the case of a board failure, these codes, together with the error pointer and error argument, provide useful information for Brooktrout Customer Support to diagnose and correct the problem.

## Tracing Link Activity

The diagnostic trace function allows you to trace Layer 2 and Layer 3 messages entering and leaving the HDLC controller on the PRI board. The trace function displays link layer protocol messages only, such as ISDN Q.931 and X.25 LAP-B; T1 robbed bit and “raw” HDLC modes are not supported. The trace display resembles a simple protocol analyzer, with the message type decoded and its direction shown.

Two types of traces are supported:

- A *Level 1* trace shows Layer 2 and Layer 3 messages being passed over the links, and provides some protocol and routing information.
- A *Level 2* trace resembles a Level 1 trace but also displays the received/transmitted message Information frame in hexadecimal format. This hexadecimal string contains Layer 2 and Layer 3 ISDN frame headers, as well as the complete contents of the Layer 3 frame.

**Note:** The transmit buffer for the serial diagnostic port is limited to 4 K in size. If you start a link trace during a period of high message traffic, the messages may wrap the buffer and some messages may not be displayed.

## Running a Level 1 Trace

To start a Level 1 trace, type **L2** and press **Return**. To stop the trace, type **L0** and press **Return**. Values that appear during the trace are defined in Table B-1. Figure B-3 shows an example trace of an ISDN Q.931 call, from link establishment to call release. The trace was run on a PRI-PCI48D with the T1 spans connected by a cross-over cable to show activity at both ends of the T1 connection.

Ch#	Time	Direct	SAPI	TEI	C/R	Type	N(s)	N(r)	P/F	Size
00	1B76	Xmit	00	00	0	SABME			1	0003
01	1B76	Rcvd	00	00	0	SABME			1	0003
01	1B76	Xmit	00	00	0	UA			1	0003
00	1B77	Rcvd	00	00	0	UA			1	0003
00	1B7A	Xmit	00	00	0	Setup	00	00	0	0027
01	1B7C	Rcvd	00	00	0	Setup	00	00	0	0027
01	1B7C	Xmit	00	00	1	Prcdng	00	01	0	000E
00	1B7D	Rcvd	00	00	1	Prcdng	00	01	0	000E
00	1B7D	Xmit	00	00	1	RR		01	0	0004
01	1B7D	Xmit	00	00	1	Alrtng	01	01	0	0009
01	1B7E	Rcvd	00	00	1	RR		01	0	0004
00	1B7E	Rcvd	00	00	1	Alrtng	01	01	0	0009
00	1B7E	Xmit	00	00	1	RR		02	0	0004
01	1B7E	Xmit	00	00	1	Connct	02	01	0	0009
01	1B7F	Rcvd	00	00	1	RR		02	0	0004
00	1B7F	Rcvd	00	00	1	Connct	02	01	0	0009
00	1B7F	Xmit	00	00	0	ConAck	01	03	0	0009
01	1B81	Rcvd	00	00	0	ConAck	01	03	0	0009
01	1B81	Xmit	00	00	0	RR		02	0	0004
00	1B82	Rcvd	00	00	0	RR		02	0	0004
00	2C13	Xmit	00	00	0	Discct	02	03	0	000D
01	2C15	Rcvd	00	00	0	Discct	02	03	0	000D
01	2C15	Xmit	00	00	0	RR		03	0	0004
00	2C16	Rcvd	00	00	0	RR		03	0	0004
00	2F3D	Xmit	00	00	0	Release	03	03	0	0010
01	2F3E	Rcvd	00	00	0	Release	03	03	0	0010
01	2F3E	Xmit	00	00	1	RelCom	03	04	0	0009
00	2F40	Rcvd	00	00	1	RelCom	03	04	0	0009

Figure B-3. Level 1 Trace Example

**Table B-1. Trace Values & Meanings**

Value	Meaning
Ch#	HDLC channel used for formatting; possible values are 0 – 64 <sup>a</sup>
Time	Hexadecimal timestamp incremented at 5 ms intervals
Direct	Direction of frame; possible values are Xmit (transmitted by PRI-PCI board) and Rcvd (received by PRI-PCI board)
SAPI	Service Access Point Identifier which identifies the type of D-channel signaling performed; typical values are 00 (ISDN call control), 16 (X.25 packet communication) or 63 (management procedures)
TEI	Terminal Endpoint Identifier which identifies a particular endpoint device
C/R	Command/Response bit that identifies the frame as either a command (C) or response (R); possible values vary depending whether the PRI-PCI board is performing user side or network side signaling. For user side (Symmetric, V.120 or Q.922 LAP-F), 0 indicates a command and 1 indicates a response. For network side, 0 indicates a response and 1 indicates a command.
Type	Q.921/Q.931 message frame or UNKNI for unknown Information frames
N(s)	Sequence number assigned to the frame sent by the transmitting device
N(r)	Expected sequence number of the next frame to be received from the transmitting device
P/F <sup>b</sup>	Poll/final bit which indicates the device is polling for a response from the other end, sending a final frame in response to a command, or neither. Possible values are 1 (polling for response or responding to command) or 0 (not polling or unsolicited response).
Size	Number of bytes in frame (shown in hexadecimal)

- a. The trace function monitors messages received and transmitted by the processor side of the HDLC controller. Because these messages pass through the MVIP switching matrix between the HDLC controller and the network, you must map individual HDLC channels to T1 spans using L4L3mSET\_TSI commands. Depending on your mappings, the HDLC channel numbers may not match the channel number of the span's D-channel.
- b. If the message frame is a command (based on the C/R bit), this is a Poll bit; if the message frame is a response, this is a Final bit.

## Running a Level 2 Trace

To start a Level 2 trace, type **L2** and press **Return**. To stop the trace, type **L 0** and press **Return**. Figure B-4 shows a Level 2 trace example for the initial stages of an ISDN call establishment.

Ch#	Time	Direct	SAPI	TEI	C/R	Type	N(s)	N(r)	P/F	Size
00	092C	Xmit	00	00	0	SABME			1	0003
01	092E	Xmit	00	00	1	SABME			1	0003
00	092F	Rcvd	00	00	1	SABME			1	0003
00	092F	Xmit	00	00	1	UA			1	0003
01	0930	Rcvd	00	00	1	UA			1	0003
00	09EB	Xmit	00	00	0	SABME			1	0003
01	09EB	Rcvd	00	00	0	SABME			1	0003
01	09EB	Xmit	00	00	0	UA			1	0003
00	09EC	Rcvd	00	00	0	UA			1	0003
00	09F0	Xmit	00	00	0	Setup	00	00	0	0027
000100000802000105040288901803A983816C090081313233343536377008										
80										
31323334353637										
01	09F2	Rcvd	00	00	0	Setup	00	00	0	0027
000100000802000105040288901803A983816C090081313233343536377008										
80										
31323334353637										
01	09F2	Xmit	00	00	1	Prcdng	00	01	0	000E
0201000208028001021803A98381										

Figure B-4. Level 2 Trace Example

The remainder of this section explains how to interpret a hexadecimal string displayed in the trace. The hexadecimal string consists of the following components:

- Information (I) Frame header

**Note:** A Level 2 trace displays hexadecimal strings for I Frame messages only. Supervisory (S Frame) messages, such as Receiver Ready (RR), and Unnumbered (U Frame) messages, such as SABME and UA, are not displayed in hexadecimal format.

- Message header
- Information Elements (IEs)

## Interpreting the I Frame Header

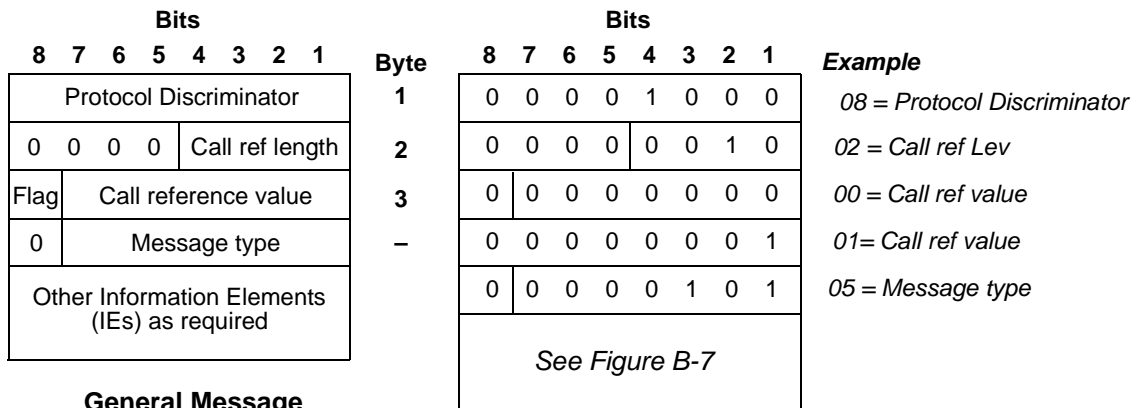
The I Frame header contains Layer 2 routing and packet transaction information. The first four bytes of the hexadecimal string comprise the I Frame header.

```

00 09F0    Xmit    00    00    0    Setup    00    00    0    0027
000100000802000105040288901803A983816C090081313233343536377008
└──────────┘
    I Frame
    Header

```



**General Message****Example SETUP Message****Figure B-6. Message Structures****Table B-2. Q.931 Message Types**

Message Type Bits	Hex	Message
0 0 0 0 0 0 0 1	01	Alerting
0 0 0 0 0 0 1 0	02	Call Proceeding
0 0 0 0 0 1 1 1	07	Connect
0 0 0 0 1 1 1 1	0F	Connect Acknowledge
0 0 0 0 0 0 1 1	03	Progress
0 0 0 0 0 1 0 1	05	Setup
0 0 0 0 1 1 0 1	0D	Setup Acknowledge
0 1 0 0 0 1 0 1	45	Disconnect
0 1 0 0 1 1 0 1	4D	Release
0 1 0 1 1 0 1 0	5A	Release Complete
0 1 0 0 0 1 1 0	46	Restart
0 1 0 0 1 1 1 0	4E	Restart Acknowledge
0 1 1 1 1 0 1 1	7B	Information
0 1 1 0 1 1 1 0	6E	Notify
0 1 1 1 1 1 0 1	7D	Status
0 1 1 1 0 1 0 1	75	Status Enquiry

## Interpreting Information Element

For Q.931 call control messages, the first Information Element (IE) starts at byte offset 10 in the hexadecimal string. Each message may contain several IEs of either fixed (single byte) or variable length.

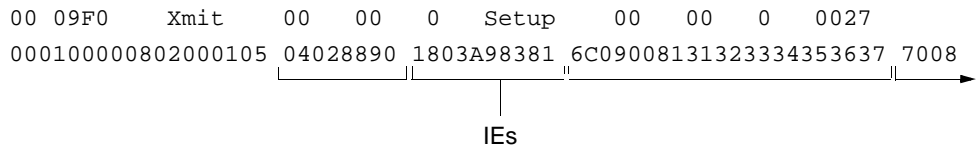
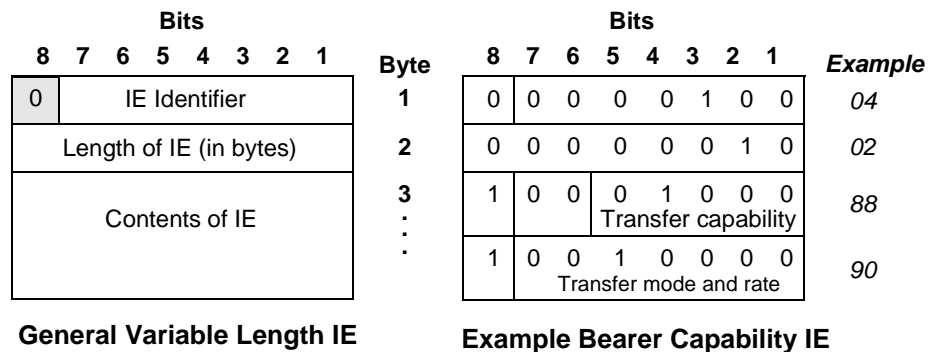


Figure B-7 compares the general IE format against the first IE contained in the example SETUP message, and illustrates the following points:

- A value of 0 in the shaded bit position indicates a variable-length IE; a value of 1 in that position indicates a single byte IE.
- Note:** Single byte IEs are commonly used for locking codeset shifts. Locking shift IEs appear only after all variable-length IEs within the message. Refer to the Bellcore Technical Reference TR-TSY-000268 for more information on the structure and use of single byte IEs and codeset shifts.
- The IE identifier value 0x04 indicates a Bearer Capability IE; refer to Table B-3 for possible IE identifier values. IEs appear in messages in ascending order according to their identifier number.
  - The 2-byte length of the IE value indicates that it contains only the required structures for a Bearer Capability IE.
  - The IE contents indicate an information transfer capability of unrestricted digital information (0x88) and a transfer rate/mode equal to 64 kbps/circuit mode (0x90).



General Variable Length IE

Example Bearer Capability IE

Figure B-7. IE Formats

**Table B-3. Q.931 Information Element Identifiers**

IE Identifier Bits	Hex	Information Element
0 0 0 0 0 1 0 0	04	Bearer capability
0 0 0 0 1 0 0 0	08	Cause
0 0 0 1 0 1 0 0	14	Call state
0 0 0 1 1 0 0 0	18	Channel identification
0 0 0 1 1 1 1 0	1E	Progress indicator
0 0 1 0 1 1 0 0	2C	Keypad
0 0 1 1 0 1 0 0	34	Signal
0 1 0 0 0 0 0 0	40	Information rate
0 1 0 0 0 0 1 0	42	End-to-end transit delay
0 1 0 0 0 0 1 1	43	Transit delay selection & indication
0 1 0 0 0 1 0 0	44	Packet-layer binary parameters
0 1 0 0 0 1 0 1	45	Packet-layer window size
0 1 0 0 0 1 1 0	46	Packet size
0 1 1 0 1 1 0 0	6C	Calling party number
0 1 1 0 1 1 0 1	6D	Calling party subaddress
0 1 1 1 0 0 0 0	70	Called party number
0 1 1 1 0 0 0 1	71	Called party subaddress
0 1 1 1 1 0 0 0	78	Transit network selection
0 1 1 1 1 0 0 1	79	Restart indicator
0 1 1 1 1 1 0 0	7C	Low-layer compatibility
0 1 1 1 1 1 0 1	7D	High-layer compatibility

For additional information on Layer 2 and Layer 3 ISDN message headers and processing, refer to the following documents:

- CCITT Recommendation I.441/Q.921/Q.931
- Bellcore Technical References TR-TSY-000268 and TR-TSY-000793



## B

Bellcore standards B-13  
Brooktrout Customer Support A-1

## C

Command/Response (C/R) bit B-8  
control interface B-4  
control message queues B-3, B-5  
control messages  
    SMI 1-1  
Control Messaging 5-1, 5-4  
Control Path 5-4  
Customer Support A-1

## D

data interface B-4  
Data Messaging 5-2, 5-11  
D-Channel descriptor 5-23  
Device Management 5-2, 5-17  
DeviceIoControl 3-1  
diagnostic utilities  
    tracing messages B-6

## F

Frame Check Sequence (FCS) B-10  
Functional Library 5-4

## G

Give/Take indexes B-3, B-4

## H

Hot Swap Management 5-2

## I

I Frame header B-9

Information Elements (IEs) B-9, B-11, B-12  
Installation and Setup of the QNX Driver for  
    Neutrino 2-1

Installation Overview 2-1

Interrupt Status Block (ISB) B-4, B-5

IOCTL\_NAI\_CAS\_READ 3-2

IOCTL\_NAI\_CHECK\_READ 3-2

IOCTL\_NAI\_CLEAR\_VTTY 3-1, 3-6

IOCTL\_NAI\_CTL\_READ 3-1, 3-9

IOCTL\_NAI\_CTL\_WRITE 3-1, 3-10

IOCTL\_NAI\_DEBUG 3-1, 3-12

IOCTL\_NAI\_DOWNLOAD 3-4

IOCTL\_NAI\_GET\_ADAP\_INFO 3-1, 3-13

IOCTL\_NAI\_GET\_STRM\_INFO 3-1, 3-14

IOCTL\_NAI\_READ\_ISB 3-1, 3-11

IOCTL\_NAI\_RESET 3-1, 3-3

IOCTL\_NAI\_RUN\_TEST 3-2, 3-16

IOCTL\_NAI\_RX\_HALT 3-2, 3-19

IOCTL\_NAI\_RX\_RESUME 3-2, 3-20

IOCTL\_NAI\_SEND\_SRECORD 3-1, 3-4

IOCTL\_NAI\_SET\_DCHAN 3-2, 3-15

IOCTL\_NAI\_SET\_LOW\_WATER 3-2, 3-21

IOCTL\_NAI\_SET\_VTTY 3-1, 3-5

IOCTL\_NAI\_TX\_HALT 3-2, 3-17

IOCTL\_NAI\_TX\_RESUME 3-2, 3-18

IOCTL\_NAI\_VTTY\_READ 3-1, 3-7

IOCTL\_NAI\_VTTY\_WRITE 3-1, 3-8

IRP Driver Library 1-3

ISDN PRI signaling B-6

## L

L4L3mENABLE\_PROTOCOL 3-15

LAP-B message tracing B-6

## M

message queues B-3

Multi-Vendor Integration Protocol (MVIP) bus  
    bus mappings B-8

## N

NaiAcquireVTTY 5-9

NaiBufPoolSize 2-4

NaiClearVTTY 5-1  
 NaiCloseAdapter 5-2, 5-19  
 NaiCloseChannel 5-2, 5-24, 5-25, 5-26  
 NaiControlRead 5-1, 5-5  
 NaiControlWrite 5-1, 5-6  
 NaiDataRead 5-2, 5-11  
 NaiDataWrite 5-2, 5-12  
 NaiDbgFlags 2-4  
 NaiDchanDescrAddr 2-4  
 NaiDchanEventQaddr 2-4  
 NaiDownloadAdapter 5-2, 5-21  
 NaiFilterStatus 2-4  
 NaiHotSwap 2-4  
 NaiHsAuto 5-2  
 NaiHsDevStatus 5-3  
 NaiHsDisable 5-2  
 NaiHsInsert 5-3  
 NaiHsQuiesce 5-3  
 NaiHsRemove 5-3  
 NaiHsStatus 5-3  
 NaiHsUser 5-2  
 NaiLowWaterMarkDefault 2-4  
 NaiNumDchanDescr 2-4  
 NaiNumL34DchanEvent 2-4  
 NaiNumL43DchanEvent 2-4  
 NaiNumRXBufs 2-4  
 NaiNumTXBufs 2-4  
 NaiOpenAdapter 5-2, 5-17  
 NaiOpenChannel 5-2, 5-23  
 NaiPendTxFlow 2-4  
 NaiReadCrashData 5-2  
 NaiReadLogoutBlock 5-22  
 NaiReleaseVTTY 5-10  
 NaiResetAdapter 5-2, 5-20  
 NaiRunDiags 5-2  
 NaiRXBufSize 2-4  
 NaiRxHalt 5-2, 5-15  
 NaiRxResume 5-2, 5-16  
 NaiSetMgmt 5-2  
 NaiSetPulse 5-2  
 NaiSetVTTY 5-1  
 NaiTXBufSize 2-4  
 NaiTxHalt 5-2, 5-13  
 NaiTxResume 5-2, 5-14  
 NaiVttyRead 5-1, 5-7  
 NaiVttyWrite 5-1, 5-8

## P

Poll/Final (P/F) bit B-8  
 Preinstallation Requirements 2-1

## Q

Q.931 message tracing B-6, B-10

## R

related publications 1-1

## S

Service Access Point Identifier (SAPI) B-8, B-10  
 SMI 1-1  
 SMI messages B-5  
 starting Driver 2-2

## T

Terminal Endpoint Identifier (TEI) B-8, B-10  
 test programs 2-2  
 tracing
 

- general B-6
- running a Level 1 trace B-6
- running a Level 2 trace B-8

 tunable parameters 2-4

## U

Uninstalling the QNX Driver 2-2

## V

version string B-4  
 VTTY Handling 5-1, 5-7