

**Netaccess Series Release 7.5
Windows 2000 I/O Request
Packet Driver
Programmer's Manual**

Driver Release 7.5

Document Number 937-110-20

Printed July 2002



General Notices

Copyright© 2000-2002, Brooktrout Technology, a Brooktrout Company.

All rights reserved.

This product may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from Brooktrout Technology.

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

Printed in the United States of America.

Trademarks

Brooktrout Inc., Brooktrout Technology, Netaccess, Instant RAS, Instant ISDN, and TruFax are registered trademarks of Brooktrout, Inc.

TR Series, TR114, TRNIC, Universal Port, TR2000 Series, TR2001, TR1000, RDSP Series, RTNI Series, Ensemble Series, Vantage DSPM, Vantage PCI Series, Vantage Series, AnyCall, AccuCall, AccuSwitch, AccuSpan, AccuLock, AccuTalk, AccuDigit, AccuRate, AccuPitch, AccuTone, AccuPulse, RDSPTest, RFX, RTNI, Prelude, RealCT, CTMedic, Prompt Studio, VEdit, BTGateway, SpeechPac, and BTStack323 are trademarks of Brooktrout, Inc.

Windows, Windows 2000, Windows NT, Windows 95, Windows 98, Microsoft SQL Server, Excel, FoxPro, FoxBase, and Visual C++ are registered trademarks of Microsoft Corporation.

UNIX is a registered trademark licensed exclusively through X/Open Company, Ltd.

MVIP is a trademark of Go-MVIP, Inc.

Pentium and Intel are registered trademarks of Intel Corporation.

Adobe and Acrobat are registered trademarks of Adobe Systems Incorporated.

Other company or product names mentioned herein may be trademarks or registered trademarks 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

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
Software Development Environment	1-1
Windows 2000 IRP Driver Architecture and Layout	1-1
IRP Driver Library	1-2
Features of the Windows 2000 IRP Driver	1-3
Notes	1-4

Chapter 2 – Installation and Operation

Introduction	2-1
Preinstallation Requirements	2-1
Installation Overview	2-1
Installation and Setup of the 2000 IRP Kit	2-1
Installing the Windows 2000 IRP Driver	2-9
Verifying the Installation	2-10
Uninstalling the Windows 2000 IRP Kit	2-10
Running Windows 2000 IRP Driver Test Programs	2-11
Tunable Parameters	2-18
Device Manager	2-18

Chapter 3 – Driver Services

Introduction	3-1
--------------------	-----

Chapter 4 – Library Services

Introduction	4-1
Functional Library	4-2
Control Path	4-3
NaiControlRead	4-4
NaiControlWrite	4-5
NaiVttyRead	4-6
NaiVttyWrite	4-7
NaiSetVTTY	4-8
NaiClearVTTY	4-9
NaiDataRead	4-10
NaiDataWrite	4-11
NaiTxHalt	4-12
NaiReadWillPend	4-13

NaiTxResume	4-14
CallBack Routine	4-15
NaiDataWriteNoPend	4-16
NaiDataReadNoPend	4-17
NaiOpenAdapter	4-18
NaiRunDiags	4-19
NaiCloseAdapter	4-20
NaiResetAdapter	4-21
NaiDownloadAdapter	4-22
NaiReadCrashData	4-24
NaiOpenChannel	4-25
NaiCloseChannel	4-26
NaiGetDeviceType	4-27
NaiGetVersionIISDN	4-28
NaiLedControl	4-29

Appendix A – Brooktrout Customer Support

Customer First	A-1
Before You Call	A-1
Contacting Brooktrout Customer Support	A-1
Additional Brooktrout Support Services	A-2

Appendix B – Running the Virtual TTY

Starting the Virtual TTY	B-1
Displaying Memory	B-3
Viewing the Diagnostics Menu	B-5
Version String	B-6
Give/Take Indexes	B-7
Control Message Queues	B-7
Interrupt Status Block	B-8
Tracing Link Activity	B-8
Running a Level 1 Trace	B-8
Running a Level 2 Trace	B-10

Appendix C – Understanding the Sample Applications

Sample Application: Test	C-1
Initialization	C-1
Synchronous (or pended) mode	C-5
Asynchronous mode	C-6
Sample Application: VTTY	C-6
Overall	C-7
Classes	C-7
Sample Application: Samp	C-7
Overall	C-7
Classes	C-8

Appendix D – Integrating with the Installation

Files	D-1
Registry Parameters	D-2



Overview

Introduction

This *Programmer's Manual* describes the installation, operation and driver utilities provided by Release 7.5 of the Windows 2000 IRP Kit ; this release of the Windows 2000 IRP Kit supports the Instant ISDN Software™, Release 7.5 (for PCI and CompactPCI controllers).

This document has been designed for Windows 2000 system developers who are using Brooktrout PCI or CompactPCI boards in their application. It assumes experience with the Windows 2000 environment and Windows 2000 system tools, the I/O Request Packet mechanism for Windows 2000, Win32 programming, the C programming language, and using intelligent controller boards in a CompactPCI or PCI bus environment.

Note: All references to “Windows 2000” within this document implies both “Windows 2000” and “Windows XP”.

Software Development Environment

Microsoft's Visual Studio 6.0 (VC++) (with Visual Studio 6.0 Service Pack 5 applied), the platform Software Development Kit (SDK), and the Win2K Driver Development Kit (DDK) are required to build the Netaccess IRP driver, libraries, and applications. These are available via subscription from Microsoft Developer Network (MSDN). If you already have the compiler and SDK, you can obtain the Win2K DDK from the web at www.microsoft.com/ddk.

Have the following publications on hand when using the Windows 2000 IRP 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 Technology adapters.
- Appropriate *Technical Description* for the Brooktrout Technology adapter(s) used.
- Appropriate information in the Windows 2000 documentation set, *Windows 2000 on-line help*, and the *Windows 2000 Device Driver Kit (DDK)*, *Windows 2000 Software Developers' Kit (SDK)*.

Some of these documents are available online <http://www.brooktrout.com>.

Windows 2000 IRP Driver Architecture and Layout

The Windows 2000 I/O Request Packet (IRP) Driver allows the application programmer complete access to the functionality provided by the Brooktrout family of PCI and CompactPCI ISDN adapters in a Windows 2000/XP system environment.

The Windows 2000 I/O Request Packet (IRP) Driver supports any number of Brooktrout controllers in a single chassis. When the Windows 2000 I/O Request Packet (IRP) Driver is installed, a series of parameters (e.g., receive buffer size, number of receives, etc.) are written to the Windows 2000/XP registry. Although not required, you can use the Registry Editor to modify (“tune”) these parameter values. For information about these parameters, see *Section 2*.

The IRP driver, as provided, contains two components: the driver itself (NAIDRV.SYS) and an interface library (NAILIB.DLL). Communication to the driver is done through Win32 file I/O calls: CreateFile()/CloseHandle() for open/close, ReadFile()/WriteFile() for data messaging and DeviceIoControl() for all other function including SMI control messaging.

The diagram below displays the normal data transfer activity within the driver.

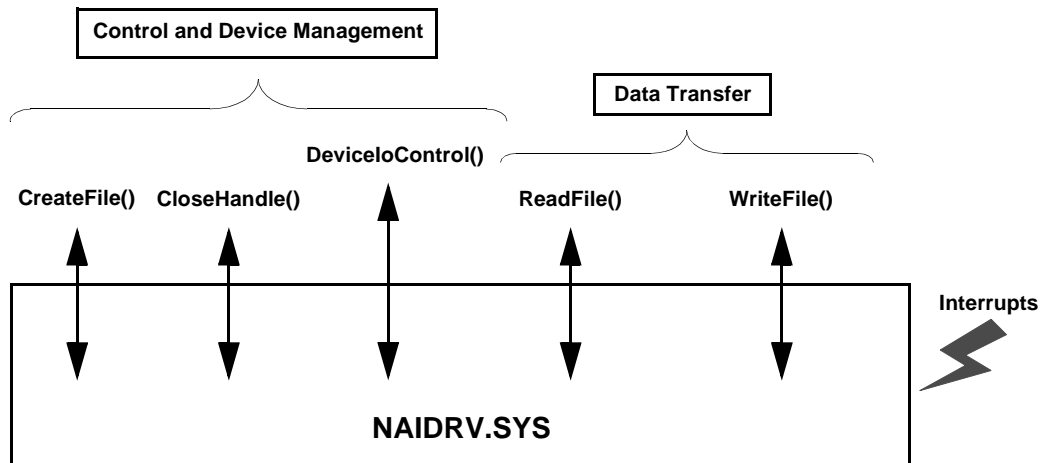


Figure 1-1. Brooktrout I/O Request Packet (IRP) Driver Layout

IRP 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. The library also handles Win32 overlapped I/O so that an application programmer can call the library from within a multithreaded environment and not have to worry about I/O collisions and deadlock.

Figure 1-2 illustrates the relationship of the driver and the library.

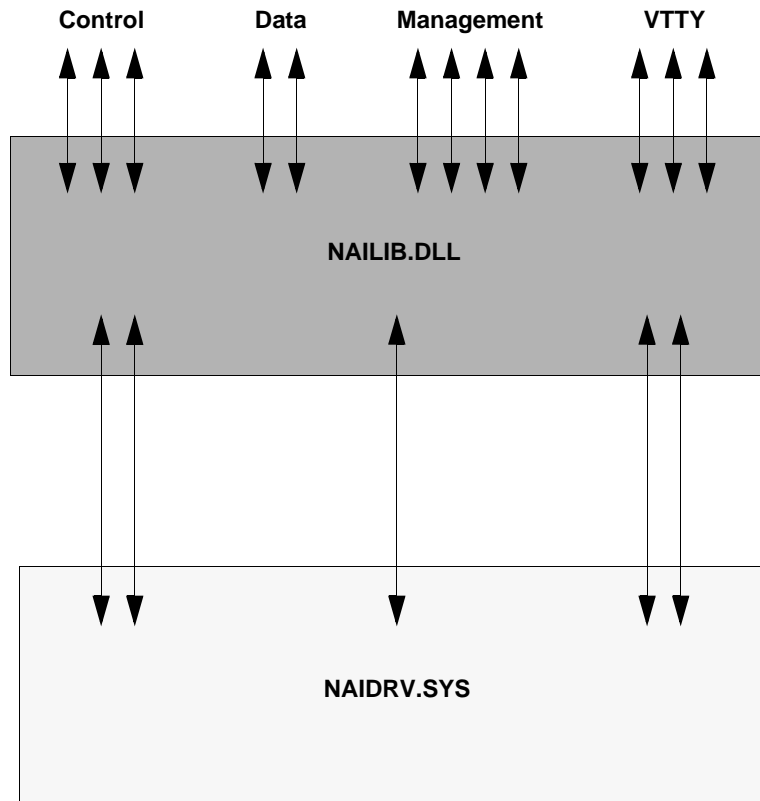


Figure 1-2. Driver and Library Relationship

The library is useful as an early prototype platform and stepping off point for more complex programming scenarios. The library is not intended as the "final" programming API.

Features of the Windows 2000 IRP Driver

The Windows 2000 I/O Request Packet (IRP) Driver was developed and designed to be a high performance, flexible, easily integrated piece of software.

As a high performance module, the IRP driver uses host based buffers, which are supported by the Brooktrout PCI and CompactPCI adapters. The driver implements a DIRECT_IO model, in which the applications buffers are used directly for data transmission. The driver itself does very little processing of actual Simple Message Interface (SMI) messages. Messages are passed through to IISDN and responses are passed back up. The IRP is multiprocessor safe. For maximized concurrency, the driver locks resources associated with each file handle rather than on critical path.

The Windows 2000 I/O Request Packet (IRP) Driver is flexible. With an interface that is mostly "flavor-less", it provides a "foundation layer" in which the user can customize the driver by adding an appropriate interface layer of software. The IRP driver was designed with the intention of supporting multiple concurrent interfaces, as well as supporting as many adapters as can be installed into the chassis.

The Windows 2000 Plug and Play manager auto-detects and configures all PCI and CompactPCI adapters. A library is included, which provides a simplified access and handles Win32 overlapped I/O. The driver and the library are written entirely in C-language using Microsoft Visual C++. The Win2KDDK is required to re-build, modify

and extend the driver, but is not required for building applications. Applications only depend on Win32 and the provided headers, all of which are available without a DDK or SDK.

Notes

The IRP 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 product installation. Information on integrating the installation can be found in *Appendix D*.



Installation and Operation

Introduction

This section describes how to install and to use the Windows 2000/XP IRP Kit.

Preinstallation Requirements

The preinstallation requirements for the Windows 2000/XP IRP driver are that a Brooktrout PCI card or a Brooktrout CompactPCI card be installed into your system chassis; see the appropriate Brooktrout *Technical Description* for further details on the correction installation of the controllers. The system requirements are for 13 MB of free disk space.

Installation Overview

There is one install to perform. The **Windows 2000 IRP Kit** (available on CD-ROM), contains all the source files necessary to rebuild the IRP driver, library and utilities; this install is documented on *page 2-1*. As part of the installation, a ready-to-run copy of the driver and default registry parameters are installed on the system.

Installation and Setup of the 2000 IRP Kit

The **Windows 2000 IRP** kit contains the source code and binary executables for the Windows 2000 IRP driver and utilities. The Brooktrout **Windows 2000 IRP Kit** is available online and on a CD-ROM.

Note: If you are upgrading your Windows 2000 IRP Kit from a previous version, Brooktrout recommends uninstalling the old driver, rebooting the NT server and then installing this new driver fresh. This process guarantees a clean upgrade of the related files. To uninstall the Windows 2000 IRP Kit, refer to *page 2-10*.

To install the Windows 2000 IRP Kit from CD-ROM:

1. If you have not already, login to the system with a user with Administrator privileges.
2. Insert the **Windows 2000 IRP Kit** into your CD-ROM drive.
3. If Autorun is enabled, the installation program will automatically start by launching your default web browser. If not, double click the **Win2kIRP.htm** file in the main directory of the CD.



Figure 2-1. Windows 2000/XP IRP Kit Introduction screen

4. On the left hand side of the web page, click Install IRP Kit. The web page will jump to the installation section where Install Win2K IRP Kit for Intel-based systems will appear. Click that hyperlink to run the setup.exe program.
5. The installation program will ask you whether you would like to save the setup file to disk or run from its current location. Brooktrout recommends simply running the setup from the CD-ROM.



Figure 2-2. File-Run Location Screen

6. A security warning will appear. Click **Yes**.

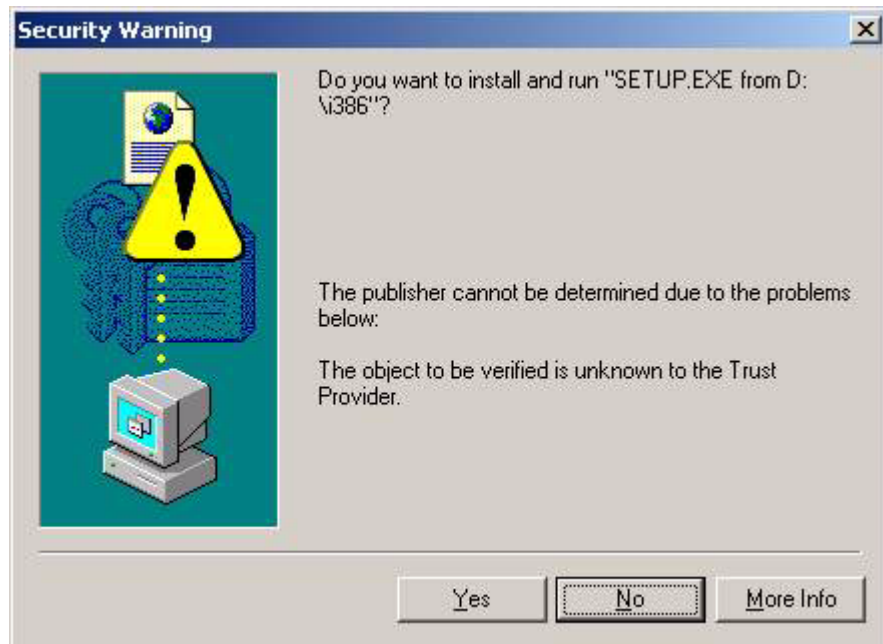


Figure 2-3. System Security Warning Screen

7. Setup will now prepare your system for the installation process.



Figure 2-4. Setup Preparation Screen

8. The installation program welcomes you to the Windows 2000 IRP Kit Setup. Click **Next**.



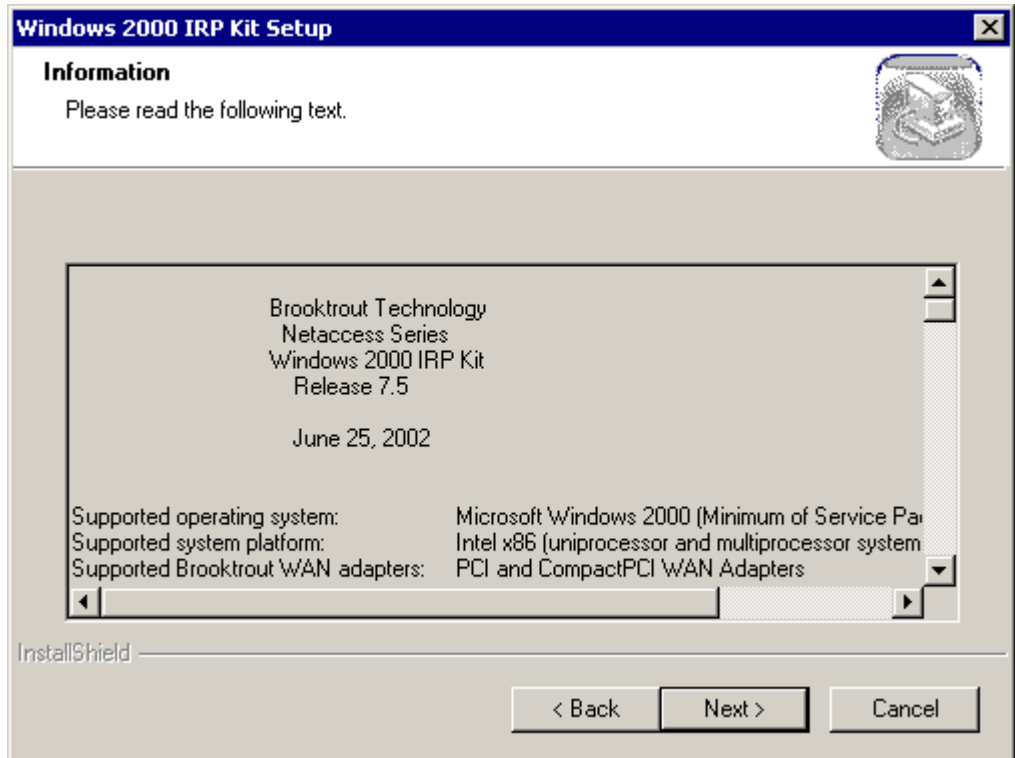
Figure 2-5. Windows 2000 IRP Driver Welcome Screen

9. The license agreement for software developed by and for Brooktrout Technology products in regards to this Windows 2000 IRP Driver installation will appear. Click **Yes** after you have read the agreement.



Figure 2-6. Software License Agreement Screen

10. The Information screen for the **Windows 2000 IRP Kit** will come up, listing the contents of the CD-ROM. You may click to Next> proceed.



11. The installation will use the default directory of:

C:\Program Files\Brooktrout Technology\Windows 2000 IRP Kit
to install the IRP Kit files. You may change that destination by clicking the **Browse...** button and selecting an alternative destination.

Click **Next** when you are ready to proceed.



Figure 2-7. Setup Destination Screen

12. You may select the type of setup you wish to perform: Typical, Compact or Custom. The default and recommendation from Brooktrout is to install a Typical installation of the Windows 2000 IRP driver.

Within the InstallShield project for the Windows 2000 IRP Kit there exist 2 components:

- ◆ Source Files (e.g., C:\Program Files\Brooktrout Technology\Windows 2000 IRP Kit*).
- ◆ DLL Files (i.e., Dynamic Link Libraries) (e.g., C:\winnt\system32*.dll).

The Brooktrout Windows 2000 IRP Kit setup defines 3 Setup Types:

- ◆ Typical
- ◆ Compact
- ◆ Custom.

If you select **Typical** or **Compact**, both components are installed.

If you select **Custom**, you may select any combination of the 2 components for installation.

There is an assumption that you are doing a clean install, and not installing over an old kit, or partially uninstalled kit.



Figure 2-8. Setup Type screen

13. The Windows 2000 IRP installation will now copy files to your system.

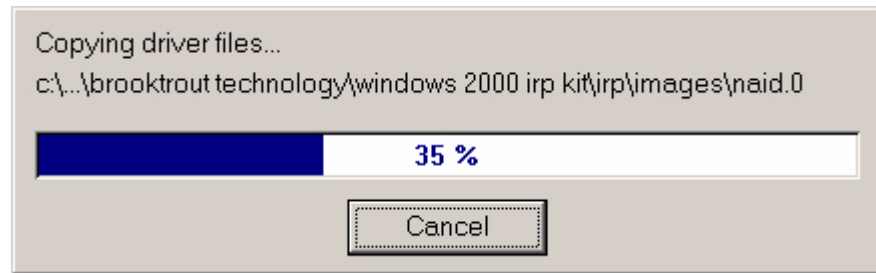


Figure 2-9. Files are now copying to your system

14. Congratulations! You have successfully installed the Windows 2000 IRP driver on to your system. It is not necessary to reboot your system.



Figure 2-10. Setup completion screen

Installing the Windows 2000 IRP Driver

The Windows 2000 IRP driver, and the property page provider, are installed through the “Add New Hardware Wizard”, when it detects that a Brooktrout Netaccess adapter has been installed into the system chassis.

Follow the instructions provided by the Add New Hardware Wizard to install the hardware, device driver, and property page provider.

Once the adapter, driver, and property page provider have been installed, the Device Manger, part of Computer Management, provides the following adapter controls:

- disable / enable adapter
- uninstall / install adapter.

Verifying the Installation

Once the driver has been started, you can verify that the installation is correct by:

1. Change directory to where the verification program resides.

```
C:\Program Files\Brooktrout Technology\Windows 2000 IRP  
Kit\irp\verify\release
```

2. Run the **iiverify** program. It will list the boards that the driver has found. If this list does not match what you believe is installed, you will need to physically verify that each card is installed.
3. Enter a 't' at the program prompt to download and run the card diagnostics.

Uninstalling the Windows 2000 IRP Kit

You will need to remove (uninstall) a previous release of the Windows 2000 IRP Kit from your system in order to install a new release of the kit.

Note: Remember to exit all other programs before uninstalling the Windows 2000 IRP Kit.

To uninstall the Windows 2000 IRP Kit:

1. Click on the **Start** button on the taskbar and select **Settings**, then **Control Panel** icon.
2. Double click on the **Add/Remove Programs** icon from the **Control Panel**.
3. Select **Windows 2000 IRP Kit** from the list of programs in the window screen.
4. Close the **Control Panel** browsing shell.
5. Click the **Add/Remove** button to uninstall.

- The Windows 2000 IRP Kit will be removed from the system.



Figure 2-11. Remove Programs From Your Computer Screen

- Be sure to restart your Windows 2000 system to ensure the uninstallation has been properly completed.

Running Windows 2000 IRP Driver Test Programs

The following test programs are designed to work with the Windows 2000 I/O Request Packet driver.

Table 2-1. Windows 2000 IRP Driver Test Programs

Test Name(s)	Purpose/Summary
ldd	Lists the bus ID and slot ID for every card installed in the system. This application does not support any command line arguments.

Table 2-1. Windows 2000 IRP Driver Test Programs (Continued)

Test Name(s)	Purpose/Summary
test	<p>Configure "n" number of HDLC Channels for the raw HDLC protocol in local loopback mode. The application creates "n" threads, and passes data across each HDLC channel.</p> <p>Command line arguments supported by this application are: -b bus = PCI bus number -s slot = PCI slot number -f firmware = firmware to be downloaded into the adapter -n channels = number of channels</p> <p>Note: This application requires the use of at least 1 adapter.</p>
samp	<p>Sample GUI application showing TSI mapping, Dchannel establishment, Download, placing a call, etc.</p> <p>Note: This application requires the use of at least 1 adapter.</p>
v tty	<p>Virtual Terminal GUI application for PCI and CompactPCI cards; allows user access to the diagnostic port without using the serial diagnostic cable.</p> <p>Note: This application requires the use of at least 1 adapter.</p>
level1/step1	<p>This application provides an example on how to locate a card, reset the card, and download firmware into a card.</p> <p>This application does not support any command line arguments.</p> <p>The application will display the query "Found card at bus #, slot #. download?". If you answer "y" (yes) the card is reset and firmware is downloaded into the card. The application terminates when the download completes. If you answer "n" (no), a query to download the next card is displayed.</p> <p>Note: This application requires the use of at least 1 adapter.</p>
level1/step2	<p>This application provides an example on how to locate a card, reset the card, and download firmware into a card. Same as step1, but automatically picks a card if one is not specified. The application terminates when the download completes.</p> <p>The command line arguments supported by this application are: -b bus = PCI bus number -s slot = PCI slot number The application selects a card, if -b bus or -s slot options were not specified. -f firmware = firmware file to be downloaded into the adapter The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified.</p>

Table 2-1. Windows 2000 IRP Driver Test Programs (Continued)

Test Name(s)	Purpose/Summary
level2/step1	<p>This application provides an example on how to locate a card, reset the card, and download firmware into a card, and use the IISDN SET_HARDWARE message.</p> <p>The command line arguments supported by this application are:</p> <ul style="list-style-type: none"> -b bus = PCI bus number -s slot = PCI slot number <p>The application selects a card, if -b bus or -s slot options were not specified.</p> <ul style="list-style-type: none"> -f firmware = firmware file to be downloaded into the adapter <p>The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified.</p> <p>Menu choices for PCI cards:</p> <ul style="list-style-type: none"> b - Set PCI BRI parameters p - Set PCI PRI parameters m - Set PCI MVIP Master r - Set PCI MVIP Reference s - Set PCI MVIP Slave h/? - this message q - quit <p>Menu choices for CompactPCI cards:</p> <ul style="list-style-type: none"> t - Set cPCI span as T1 e - Set cPCI span as E1 m - Set cPCI H1x0 Master 2 - Set cPCI H1x0 Second s - Set cPCI H1x0 Slave h/? - this message q - quit <p>Note: All level 1 applications require the use of at least 1 adapter.</p>

Table 2-1. Windows 2000 IRP Driver Test Programs (Continued)

Test Name(s)	Purpose/Summary
level2/step2	<p>This application provides an example on how to locate a card, reset the card, and download firmware into the card, and use the IISDN SET_HARDWARE and SET_TSI messages.</p> <p>The command line arguments supported by this application are:</p> <ul style="list-style-type: none"> -b bus = PCI bus number -s slot = PCI slot number <ul style="list-style-type: none"> The application selects a card, if -b bus or -s slot options were not specified. -f firmware = firmware file to be downloaded into the adapter <ul style="list-style-type: none"> The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified. <p>Menu choices for PCI cards:</p> <ul style="list-style-type: none"> m - Map HDLC to MVIP b - Map HDLC to a BRI Span p - Map HDLC to PRI Span B h/? - this message q - quit <p>Menu choices for CompactPCI cards:</p> <ul style="list-style-type: none"> H - Map HDLC to H1X0 r - Map HDLC to H1X0 Subrated s - Map HDLC to PRI Span B h/? - this message q - quit

Table 2-1. Windows 2000 IRP Driver Test Programs (Continued)

Test Name(s)	Purpose/Summary
level2/step3	<p>This application provides an example on how to locate a card, reset the card, download firmware into the card, and use the IISDN SET_HARDWARE, SET_TSI, and ENABLE_PROTOCOL messages.</p> <p>The command line arguments supported by this application are:</p> <ul style="list-style-type: none"> -b bus = PCI bus number -s slot = PCI slot number <p style="padding-left: 40px;">The application selects a card, if -b bus or -s slot options were not specified.</p> <ul style="list-style-type: none"> -f firmware = firmware file to be downloaded into the adapter <p style="padding-left: 40px;">The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified.</p> <p>Menu choices:</p> <ul style="list-style-type: none"> e - enable a channel (submenu) d - disable a channel t - transmit a packet r - receive a packet h/? - this message q - quit <p>Sub-menu choices:</p> <ul style="list-style-type: none"> h - Enable HDLC on a DS0 b - Enable LAP-B on a DS0 d - Enable LAP-D on a DS0 f - Enable LAP-F on a DS0 m - Enable a modem channel s - Enable a HDLC on a super channel r - Enable RAW (unframed) data i - Enable UDP/IP (cPCI only!) ? - this message q - quit <p>Note: This application can be used with a single card, using loop back connectors installed in span A and span B within the transition module.</p> <p>Note: This application can be used with 2 adapters, with cross over cables installed connecting the adapters from span A to span A, and span B to span B. Run step3 specifying the 1st card, and run step3 again specifying the 2nd card.</p>

Table 2-1. Windows 2000 IRP Driver Test Programs (Continued)

Test Name(s)	Purpose/Summary
level2/step4	<p>This application provides an example on how to locate a card, reset the card, download firmware into the card, and use the IISDN SET_HARDWARE, SET_TSI, and ENABLE_PROTOCOL messages, and then perform data passing.</p> <p>The command line arguments supported by this application are:</p> <ul style="list-style-type: none"> -b bus = PCI bus number -s slot = PCI slot number <ul style="list-style-type: none"> The application selects a card, if -b bus or -s slot options were not specified. -f firmware = firmware file to be downloaded into the adapter <ul style="list-style-type: none"> The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified. <p>Menu choices are:</p> <ul style="list-style-type: none"> e - enable HDLC on a channel. d - disable channel. t - test data passing. h/? - this message q - quit <p>Note: This application can be used with a single card, using loop back connectors installed in span A and span B within the transition module.</p> <p>Note: This application can be used with 2 adapters, with cross over cables installed connecting the adapters from span A to span A, and span B to span B. Run step3 specifying the 1st card, and run step3 again specifying the 2nd card.</p>
level3/step1	<p>This application provides the framework for level3, steps 2 through 4.</p> <p>The command line arguments supported by this application are:</p> <ul style="list-style-type: none"> -b bus = PCI bus number -s slot = PCI slot number -f firmware = firmware file to be downloaded into the adapter <ul style="list-style-type: none"> The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified. -n = network side <ul style="list-style-type: none"> (user side, if -n was not specified) <p>The menu choices are:</p> <ul style="list-style-type: none"> h/? - this message q - quit <p>Note: All level 3 applications require the use of 2 adapters, with cross over cables installed connecting the adapters from span A to span A, and span B to span B.</p>

Table 2-1. Windows 2000 IRP Driver Test Programs (Continued)

Test Name(s)	Purpose/Summary
level3/step2	<p>This sample application performs Q.931 call control.</p> <p>The command line arguments supported by this application are:</p> <ul style="list-style-type: none"> -b bus = PCI bus number -s slot = PCI slot number -f firmware = firmware file to be downloaded into the adapter The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified. -n = network side (user side, if -n was not specified) <p>The menu choices are:</p> <ul style="list-style-type: none"> a - toggle flag to accept or reject next call e - enable q931 d - disable q931 m - make a call H - hangup a call h/? - this message q - quit
level3/step3	<p>This sample application performs SW56 (Robbed bit) call control.</p> <p>The command line arguments supported by this application are:</p> <ul style="list-style-type: none"> -b bus = PCI bus number -s slot = PCI slot number -f firmware = firmware file to be downloaded into the adapter The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified. -n = network side (user side, if -n was not specified) <p>The menu choices are:</p> <ul style="list-style-type: none"> a - toggle flag to accept or reject next call e - enable q931 d - disable q931 m - make a call H - hangup a call h/? - this message q - quit

Table 2-1. Windows 2000 IRP Driver Test Programs (Continued)

Test Name(s)	Purpose/Summary
level3/step4	<p>This sample application performs X.25 call control.</p> <p>The command line arguments supported by this application are:</p> <ul style="list-style-type: none"> -b bus = PCI bus number -s slot = PCI slot number -f firmware = firmware file to be downloaded into the adapter <ul style="list-style-type: none"> The application selects a firmware file (based on adapter type: PCI / cPCI), if the -f firmware option was not specified. -n = network side <ul style="list-style-type: none"> (user side, if -n was not specified) <p>The menu choices are:</p> <ul style="list-style-type: none"> a - toggle flag to accept or reject next call e - enable q931 d - disable q931 m - make a call H - hangup a call h/? - this message q - quit

Tunable Parameters

Device Manager

The Device Manager for Brooktrout adapters provides new management mechanism in Windows 2000.

To run the Device Manager, you must go to the Windows 2000 Computer Management utility by clicking **Start->Programs->Administrative Tools->Computer Management**. Select **Device Management** from under **System Tools**.

The Device Manager replaces the Windows 2000 Netaccess IRP Control Panel applet with the library file `naiclass.dll`.

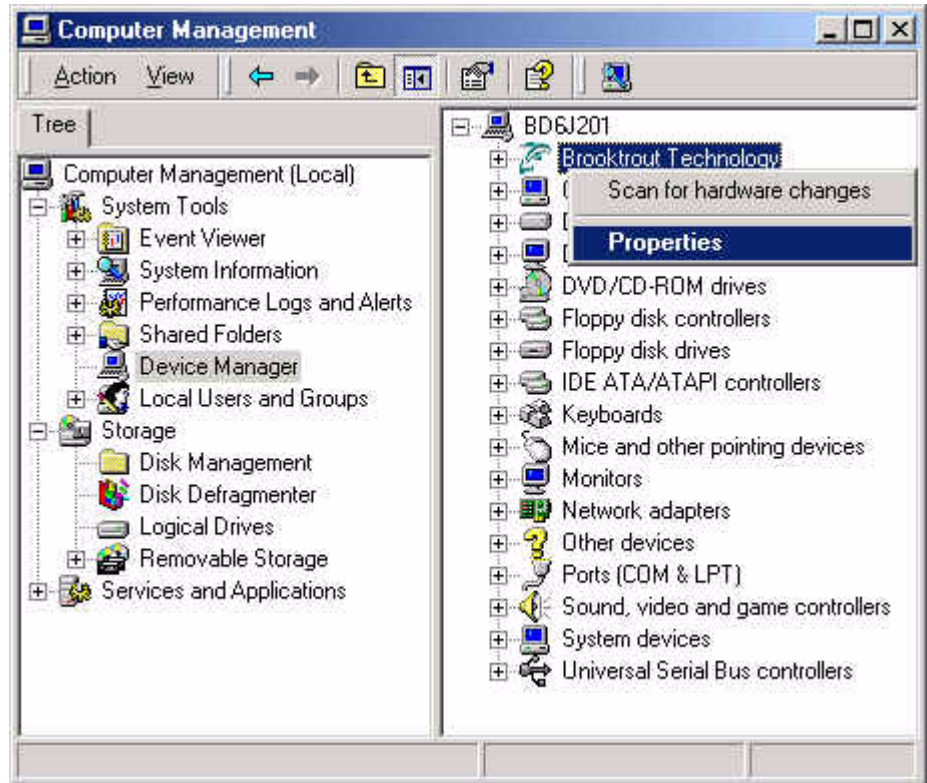


Figure 2-12. Windows 2000 Computer Management

Under the properties for Brooktrout Technology, you are able to Tune and Debug your Brooktrout adapter. Brooktrout Technology Properties Provides global information that applies to all Brooktrout adapters.

With the Tune function, you can adjust the settings at any time, which are acquired by the IRP driver when it is loaded. These settings go into effect after the driver has been unloaded and reloaded.

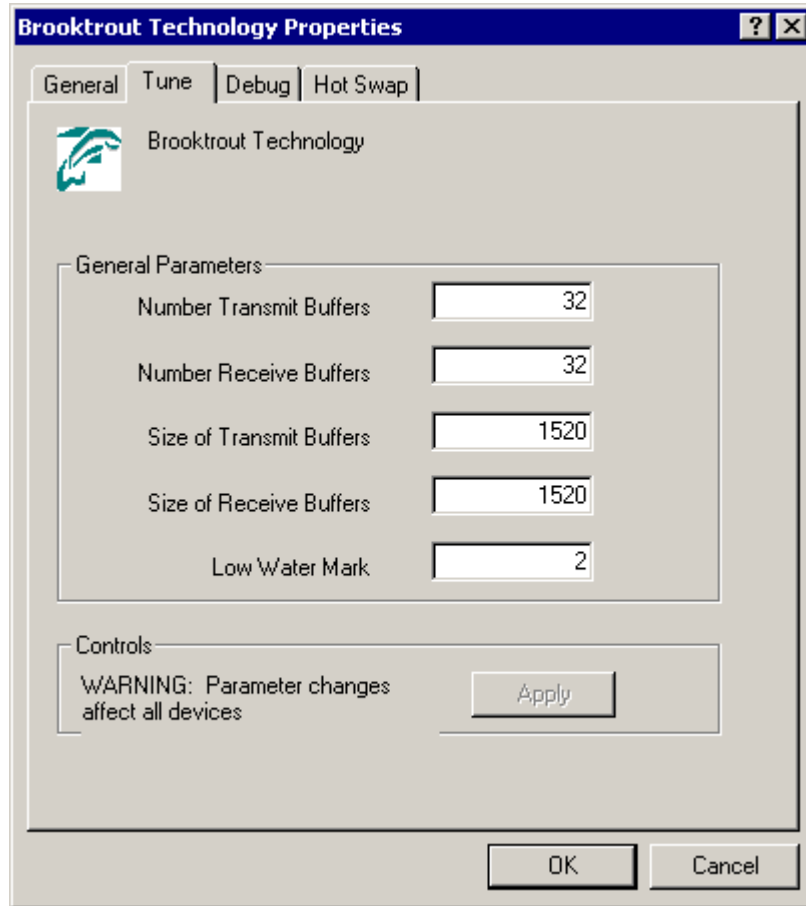


Figure 2-13. Debug Window in the Adapter Properties

The Debug window is used to enable/disable debug print messages within the checked build version of the driver.

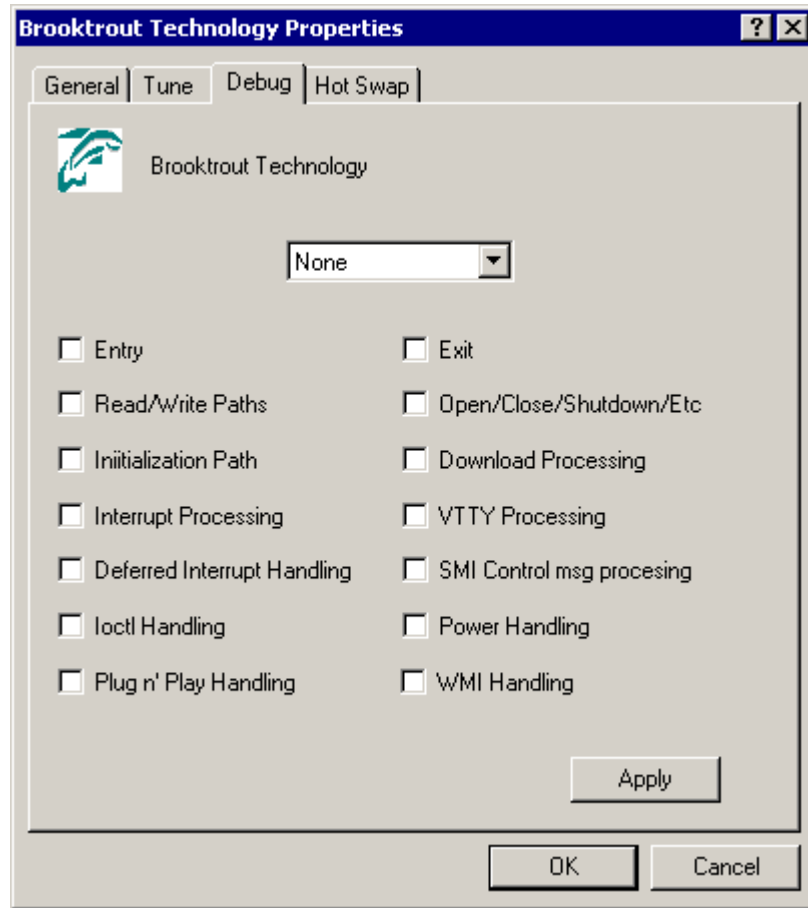


Figure 2-14. Debug Window in the Adapter Properties

The Hot Swap window is used to display the Hot Swap status of the NS300/301 adapters, and provide a mechanism to control extraction/insertion of the adapter, and its LEDs.

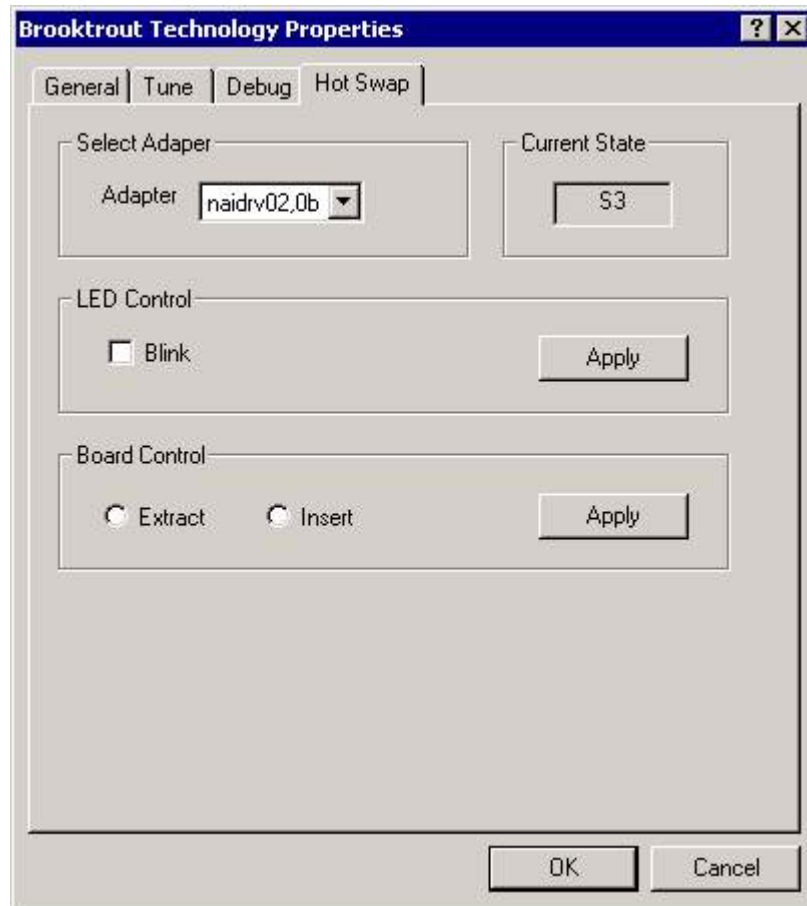


Figure 2-15. Hot Swap Window in the Adapter Properties

Driver Services

Introduction

This section describes the Windows 2000 Driver services available for application processes through Windows NT system calls. *Table 3-1* lists each service and describes its purpose; the services are listed in the subsections that follow.

Table 3-1. Windows 2000 Driver Services

Service	Purpose	
DeviceIoControl	IOCTL_NAI_RESET	Halts an adapter.
	IOCTL_NAI_DOWNLOAD	Downloads an image to an adapter; file name is passed in.
	IOCTL_NAI_SEND_SRECORD	An alternative method of downloading, in which each ASCII S-record is sent to the driver.
	IOCTL_NAI_SET_VTTY	Identifies selected handle as a VTTY recipient.
	IOCTL_NAI_CLEAR_VTTY	Removes selected handle as the VTTY recipient.
	IOCTL_NAI_VTTY_READ	Reads characters from the VTTY.
	IOCTL_NAI_VTTY_WRITE	Writes characters to the VTTY.
	IOCTL_NAI_CTL_READ	Reads a control message from the SMI FIFO queue.
	IOCTL_NAI_CTL_WRITE	Writes an SMI control message to the board.
	IOCTL_NAI_READ_CRASH_DATA	Read the logout block, interrupt service block and other pertinent crash information of an adapter.
	IOCTL_NAI_DEBUG	Sets the debug flag for the driver; useful for only driver debug purposes.
	IOCTL_NAI_GET_VERSION_IISDN	Request the IRP driver to return the version of IISDN it was built with, and the version of IISDN currently running on the adapter.

Table 3-1. Windows 2000 Driver Services

Service	Purpose	
DeviceIoControl	IOCTL_NAI_LED_CONTROL	Request the IRP driver to blink the NS300/NS301 LEDs, or return them to normal operation.
	IOCTL_NAI_SET_DCHAN	Allocates a particular d-channel descriptor number to a handle.
	IOCTL_NAI_RUN_TEST	Initiates a diagnostic test.
	IOCTL_NAI_TX_HALT	Halts transmit data pump on this channel.
	IOCTL_NAI_TX_RESUME	Re-initiates the transmit data pump on this channel.
	IOCTL_NAI_SET_LOW_WATER	Sets the Low water mark for this channel.
	IOCTL_NAI_CHECK_READ	Returns true if next Readfile() is likely to pend.
	IOCTL_NAI_CAS_READ	Reads CAS bits from the SMI after an L4L3mCAS_TO_MEMORY has been issued.
	IOCTL_NAI_CAS_WRITE	Writes CAS bits from the SMI after an L4L3mCAS_TO_MEMORY has been issued.
	IOCTL_NAI_DEVICE_TYPE	Returns the device type of the open handle (PCI or cPCI).
	IOCTL_NAI_SERVICE_EVENT	The IRP device drivers queues these requests on a per file object basis to each adapter. A single request is completed when a Hot Swap Event or Adapter Failure occurs.
	IOCTL_HOT_SWAP_STATE	The IRP device driver returns the cPCI adapter's current Hot Swap State.
	IOCTL_EXTRACTION_AUTHORIZED	Informs the IRP driver that the Win32 user mode application is prepared for the removal of the cPCI adapter.
	IOCTL_HOT_SWAP_EXTRACT_ADAPTER	Request the IRP driver perform an orderly extraction of the cPCI adapter.
IOCTL_HOT_SWAP_INSERT_ADAPTER	Request the IRP driver perform an orderly insertion of the cPCI adapter.	

IOCTL_NAI_RESET

Structure	<pre>#include "nailib.h" BOOL DeviceIoControl (handle, IOCTL_NAI_RESET, NULL, 0, NULL, 0, &bytes_returned, NULL); HANDLE NaiCtrlHandle; int bytes_returned</pre>
Usage	<p>This ioctl puts the controller into a halted and reset state. After this ioctl, the controller will only respond to an IOCTL_NAI_DOWNLOAD or IOCTL_NAI_SEND_SRECORD.</p> <p>'bytes_returned' will not contain any meaningful data on completion.</p>
Example	<pre>int bytes; if (! DeviceIoControl(NaiCtrlHandle, IOCTL_NAI_RESET, NULL, 0 NULL, 0 &bytes, NULL)) return GetLastError();</pre>
Returns	STATUS_SUCCESS - The card has been reset.

IOCTL_NAI_DOWNLOAD

Structure

```
#include    "nailib.h"
int        DeviceIoControl(NaiCtrlrHandle, IOCTL_NAI_DOWNLOAD,
                          fileSpecification, fsLength, NULL, 0 ,
                          &bytesReturned);

HANDLE     NaiCtrlrHandle;
char       *fileSpecification;
int        fsLength;
int        *bytesReturned;
```

Usage This Device I/O Control opens, reads, and downloads the binary image file into the PCI or cPCI adapter, and starts the adapter. downloads the adapter.

“fileSpecification” is a to a binary image file.

“fsLength” is the length of the fully qualified path name (byte string).

“bytesReturned” does not contain any meaningful data on completion.

Notes Downloading of binary files is supported on PCI and cPCI adapters only!

Brooktrout Technology supplies S-Record files. An S-Record file can be converted to a binary image file by using the SREC2BIN utility. The source code for the SREC2BIN utility is available on the FTP site <ftp.netacc.com/support/utills/srec2bin>.

Example Download binary microcode file for PCI & cPCI adapters only.

```

HANDLE ctrlHandle;
charfileSpecification[] = "C:\\temp\\pri.bin";
OVERLAPPEDoverlap;
intbytesReturned;
intlast_err;

ctrlHandle = NaiOpenAdapter(busNumber, slotNumber);
if (ctrlHandle == INVALID_HANDLE_VALUE) {
    printf("NaiOpenAdapter failed with %d\n", GetLastError());
    exit(-1);
}

// Initialize OVERLAPPED I/O structure, & include Manual-Reset Event.
overlap.Offset = 0;
overlap.OffsetHigh = 0;
overlap.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
if (!overlap.hEvent) {
    printf("CreateEvent failed with %d\n", GetLastError());
    exit(-1);
}

// Download the binary image file (e.g., created with the SREC2BIN utility).
if (!DeviceIoControl(ctrlHandle, IOCTL_NAI_DOWNLOAD, fileSpecification,
    (strlen(fileSpecification) + 1), NULL, 0, &bytesReturned, &overlap)) {

    // The IOCTL did not complete successfully, determine why..
    last_err = GetLastError();
    if (last_err == ERROR_IO_PENDING) {
        // The IOCTL is still pending completion, wait for it to complete...
        if (!GetOverlappedResult(ctrlHandle, &overlap, &bytesReturned, TRUE)) {
            CloseHandle(overlap.hEvent);
            printf("GetOverlappedResult failed with %d\n", GetLastError());
            exit(-1);
        }
    }
    else {
        // The IOCTL completed with an error, close handle and return error status.
        CloseHandle(overlap.hEvent);
        printf("IOCTL_NAI_DOWNLOAD failed with %d\n", last_err);
        exit(-1);
    }
}

```

Returns **STATUS_SUCCESS** - Adapter has been successfully downloaded.

STATUS_NO_MEMORY - Failed to download the adapter. The return status describes the specific error that occurred.

This call will also pass through other file system errors such as "file does not exist", "no privileges", etc.

IOCTL_NAI_SEND_SRECORD

Structure

```
#include"nailib.h"
BOOL DeviceIoControl(NaiCtrlHandle, IOCTL_NAI_SEND_SRECORD,
                    buffer, length,NULL, 0, &bytes_returned, NULL);
HANDLE NaiCtrlHandle;
char *buffer;
int length;
int *bytes_returned;
```

Usage This ioctl takes a buffer containing a Motorola ASCII S-record, converts it and downloads it to the adapter card. When it sees a record of S9 (or S7), the ioctl will cause the controller to reset and begin running the downloaded code.

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

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

'bytes_returned' will not contain any meaningful data on completion.

Example

```
int bytes;
If (!DeviceIoControl(NaiCtrlHandle, IOCTL_NAI_RESET, NULL, 0, NULL,
0,
&bytes,NULL)) return GetLastError();
if ((fp = fopen("naid.0", "r")) == NULL) {
printf("Can't open naid.0\n");
return;
}
while (!feof(fp)) {
fgets(buffer, 256, fp);
if (!DeviceIoControl, NaiCtrlHandle, IOCTL_NAI_SEND_SRECORD,
buffer, length, NULL, 0,&bytes_returned, NULL) {
printf("Can't write to controller.\n");
return;
}
}
}
```

Returns

- STATUS_SUCCESS** - A good s-record was received and processed
- STATUS_NAI_BAD_S_RECORD** - The S-record given was somehow not properly formed
- STATUS_NAI_BAD_CHECKSUM** - This S-record contained an improper checksum

IOCTL_NAI_SET_VTTY

Structure

```
#include "nailib.h"
BOOL DeviceIoControl(NaiVttyHandle, IOCTL_NAI_SET_VTTY, NULL, 0,
                    NULL, 0, &bytes_returned, NULL);
HANDLE handle;
int *bytes_returned;
```

Usage

This ioctl identifies a particular end point as the recipient of any and all VTTY data from the controller. It also enables VTTY processing on the controller. Note that only one handle can be marked as the VTTY recipient for a given controller.

'bytes_returned' does not contain any meaningful data on completion.

Example

```
int bytes;
if (!DeviceIoControl(NaiVttyHandle, IOCTL_NAI_SET_VTTY, NULL, 0, NULL,
                    0, &bytes,
                    NULL))
    return GetLastError();
```

Returns

STATUS_SUCCESS - The VTTY has been assigned to this handle.

STATUS_DEVICE_BUSY - Another handle has already claimed the VTTY.

IOCTL_NAI_CLEAR_VTTY

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(NaiVttyHandle, IOCTL_NAI_CLEAR_VTTY, NULL,
                    0, NULL,
                    0, &bytes_returned, NULL);

HANDLE handle;

int *bytes_returned;
```

Usage This ioctl removes a particular end point as being the recipient of VTTY data and disables any VTTY processing on the controller.

'bytes_returned' does not contain any meaningful data on completion for this call.

Example

```
int bytes;

if (!DeviceIoControl(NaiVttyHandle, IOCTL_NAI_CLEAR_VTTY, NULL, 0,
                    NULL, 0, &bytes,
                    NULL))
    return GetLastError();
```

Returns **STATUS_SUCCESS** - The VTTY has been released.

STATUS_INVALID_OWNER - This handle does not currently own the VTTY and therefore can't release it.

IOCTL_NAI_VTTY_READ

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(NaiVttyHandle,
                    IOCTL_NAI_VTTY_READ, NULL, 0, buffer, buffer_length,
                    &bytes_returned, overlapped);

HANDLE NaiVttyHandle;

char *buffer

int buffer_length

int *bytes_returned;

LPOVERLAPPED overlapped
```

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 handle. The ioctl will pend if data is not immediately available.

'buffer' is where the data read will be placed.

'buffer_length' is the maximum amount of data that 'buffer' can contain.

'bytes_returned' will contain the number of bytes actually read upon completion.

Since this ioctl may pend, the 'overlapped' entry may be used to issue multiple ioctl requests.

Example

```
char buffer[256];
int length = sizeof(buffer);
int bytes;
if (!DeviceIoControl(NaiVttyHandle, IOCTL_NAI_VTTY_READ, NULL, 0,
                    buffer, length, &bytes_returned, NULL))
    return GetLastError();
for (i = 0; i < bytes_returned; i++) {
    printf("%c", buffer[i]);
}
```

Returns

STATUS_SUCCESS - VTTY data has been read.

STATUS_INVALID_OWNER - This handle does not own the VTTY.

IOCTL_NAI_VTTY_WRITE

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(NaiVttyHandle, IOCTL_NAI_VTTY_WRITE,
                    buffer, buffer_length, NULL, 0, &bytes_returned,
                    overlapped);

HANDLE NaiVttyHandle;
char *buffer
int buffer_length
int *bytes_returned;
LPOVERLAPPED overlapped
```

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 handle. The ioctl will pend until there is room available in the VTTY queue.

'buffer' contains the data to be written.

'buffer_length' is the number of bytes to write.

'bytes_returned' contains the number of bytes actually written. This ioctl may pend, so the 'overlapped' entry may be used to issue multiple ioctl requests.

Example

```
char*buffer = "d fe0f0000 40\n";
int length = strlen(buffer);
int bytes;
if (!DeviceIoControl(NaiVttyHandle, IOCTL_NAI_VTTY_WRITE, buffer,
                    length, NULL, 0, &bytes_returned, &overlap))
    return GetLastError();
```

Returns

STATUS_SUCCESS - Data was written to the VTTY.

STATUS_INVALID_OWNER - This handle does not own the VTTY.

IOCTL_NAI_CTL_READ

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(NaiVttyHandle, IOCTL_NAI_CTL_READ, NULL,
0,
L34msg_buffer, L34msg_buffer_length,
&bytes_returned, &overlapped);

HANDLE NaiVttyHandle;
char *L34msg_buffer
int L34msg_buffer_length
int *bytes_returned;
OVERLAPPED overlapped
```

Usage This ioctl reads a L3L4 message from the L3L4 SMI message control queue. The ioctl will pend if there are no messages available.

'L34msg_buffer' is the buffer to receive the message into.

'L34msg_buffer_length' is the length of said buffer and should always be at least 512 bytes.

'bytes_returned' does not contain any meaningful data upon completion. Since this ioctl may pend, the 'overlapped' entry may be used to issue multiple ioctl requests.

Note: L3L4 messages are defined in **IISDN.H**.

Example

```
L3_to_L4_struct buffer;
int bytes;
if (!DeviceIoControl(board, IOCTL_NAI_CTL_READ, NULL, 0, (char *)
&buffer, sizeof(L3_to_L4_struct), &bytes_returned, NULL))
return GetLastError();
switch (buffer.msgtype) {
case L3L4mALERTING:
...
}
```

Returns **STATUS_SUCCESS** - An L3L4 SMI message has been read.

IOCTL_NAI_CTL_WRITE

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(NaiVttyHandle, IOCTL_NAI_CTL_WRITE,
                    NULL, 0, L43msg_buffer, L43msg_buffer_length,
                    &bytes_returned, &overlapped);

HANDLE NaiVttyHandle;
char *L43msg_buffer
int L43msg_buffer_length
int *bytes_returned;
OVERLAPPED overlapped
```

Usage

This ioctl writes an L4L3 message to the L4L3 SMI message queue. The ioctl will pend if there is no space available on the queue.

'L43msg_buffer' is the L4L3 message buffer to write.

'L43msg_buffer_length' is the length of said buffer and should always be 512 bytes.

'bytes_returned' does not contain any meaningful data upon completion. Since this ioctl may pend, the 'overlapped' entry may be used to issue multiple ioctl requests.

Example

```
L4_to_L3_struct buffer;
int bytes;
buffer.msgtype = L4L3mREQ_LINE_STATUS;
if (!DeviceIoControl(NaiVttyHandle, IOCTL_NAI_CTL_WRITE, (char *)
                    &buffer, sizeof(L4_to_L3_struct), NULL, 0, &bytes_returned,
                    NULL))
return GetLastError();
```

Returns

STATUS_SUCCESS - A L4L3 SMI message has been written.

STATUS_NO_MEMORY - There was not enough memory to allocate the receive and transmit buffers for this particular L4L3mENABLE_PROTOCOL message.

IOCTL_NAI_READ_CRASH_DUMP

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(NaiVttyHandle, IOCTL_NAI_READ_ISB,
                    buffer, length, NULL, 0, &bytes_returned, NULL);

HANDLE NaiVttyHandle;

NaiCrashData_t*buffer;

int length;

int *bytes_returned;
```

Usage This ioctl is used to retrieve information about a controller crash dump. It is primarily used for debugging and support.

'buffer' is where to put the crash dump information.

'length' should be at least size of (NaiCrashData_t).

'bytes_returned' contains the actual number of bytes written to the crash dump buffer.

Example

```
NaiCrashData_tbuffer;

int bytes;

if (!DeviceIoControl(NaiVttyHandle, IOCTL_NAI_READ_ISB, NULL, 0,
                    buffer, size of (NaiCrashData_t, &bytes_returned,
                    NULL))
    return GetLastError());
```

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

IOCTL_NAI_DEBUG

- Structure**
- ```
#include "nailib.h"
BOOL DeviceIoControl(NaiVttyHandle, IOCTL_NAI_DEBUG, debug_value,
 length, NULL, 0, &bytes_returned, NULL);
HANDLE NaiVttyHandle;
short *debug_value;
int length;
int *bytes_returned;
```
- Usage**
- This ioctl is used to set the debug bit mask which controls the level of debug output printed at run time. This ioctl is useful primarily for 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.
- 'bytes\_returned' does not contain any meaningful data upon completion.
- Example**
- ```
short debug_flag = 0x0123;
int bytes;
if (!DeviceIoControl(NaiVttyHandle, IOCTL_NAI_DEBUG, &debug_flag, 2,
                    NULL, 0, &bytes_returned, NULL))
    last_err = GetLastError();
```
- Returns**
- STATUS_SUCCESS** - The debug flag has been set.

IOCTL_NAI_GET_VERSION_IISDN

Structure

```
#include "nailib.h"

BOOL DeviceIoControl( handle,
IOCTL_NAI_GET_VERSION_IISDN,
NULL,
0,
pVersionInfo,
VERSION_INFO_SIZE,
&bytesReturned,
NULL );

HANDLE handle;
DWORD bytesReturned;
PVERSION_INFO pVersionInfo;
```

Usage Win32 applications can request that the Windows 2000 IRP device driver return the version of IISDN it was built with, and the version of IISDN currently running on the adapter.

Returns **STATUS_SUCCESS** - The IOCTL_NAI_GET_VERSION_IISDN request was successful. (i.e., the VERSION_INFO structure contains IISDN version information).
STATUS_INVALID_PARAMETER - Invalid pointer to, or size of, the VERSION_INFO structure.

Example

```
void GetVersion( HANDLE fd )
{
    BOOL value;
    DWORD bytesReturned;
    VERSION_INFO VersionInfo;

    value = DeviceIoControl( fd,
                            IOCTL_NAI_GET_VERSION_IISDN,
                            NULL,
                            0,
                            &VersionInfo,
                            VERSION_INFO_SIZE,
                            &bytesReturned,
                            NULL);

    if ( value ) {
        printf("IOCTL_NAI_GET_VERSION_IISDN successful\n");
        printf("  IRP driver:  %s\n", VersionInfo.driver);
        printf("  Firmware:    %s\n", VersionInfo.firmware);
    }
    else {
        printf("IOCTL_NAI_GET_VERSION_IISDN failed with 0x%x\n",
              GetLastError());
    }
}
```

IOCTL_NAI_LED_CONTROL

Structure

```
#include "nailib.h"

BOOL DeviceIoControl( handle,
IOCTL_NAI_LED_CONTROL,
pLed,
LED_SIZE,
NULL,
0,
&bytesReturned,
NULL );

HANDLE handle;
DWORD bytesReturned;
PLED pLed;
```

Usage Win32 applications can request that the NT 4.0 IRP device driver blink the NS300 / NS301 adapters LEDs, or return them to normal operation.

The adapter must be running the flash code in order to use this feature.

Returns **STATUS_SUCCESS** - The IOCTL_NAI_LED_CONTROL request was successful. (i.e., the NS300/NS301 LED is configured as requested).

STATUS_INVALID_PARAMETER - Invalid pointer to, or size of, the LED structure.

STATUS_NAI_NS30X_ONLY - Request supported on NS300 & NS301 adapters ONLY.

STATUS_NAI_INVALID_STATE - Request not supported in adapters current state.

Notes This feature is supported on NS300 and NS301 adapters only. In order to use this feature, the adapter must be running the flash code (not the firmware).

Example

```
void SetLed( HANDLE fd, ULONG blink)
{
    BOOL value;
    DWORD bytesReturned;
    LED led;

    led.operation = blink; // 1=Blink LEDs (any non-zero value)
                          // 0=Return to normal operation.

    value = DeviceIoControl( fd,
        IOCTL_NAI_LED_CONTROL,
        &led,
        LED_SIZE,
        NULL,
        0,
        &bytesReturned,
        NULL);

    if ( value ) {
        printf("IOCTL_NAI_LED_CONTROL successful\n");
    }
    else {
        printf("IOCTL_NAI_LED_CONTROL failed with 0x%x\n",
            GetLastError());
    }
}
```

IOCTL_NAI_SET_DCHAN

Structure

```
#include "nailib.h"
BOOL DeviceIoControl(NaiDataHandle, IOCTL_NAI_SET_DCHAN, dchannel,
                    dchannel_length, NULL, 0, &bytes_returned, NULL);
HANDLE NaiVttyHandle;
short *dchannel;
int dchannel_length;
int *bytes_returned;
```

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

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

'dchannel_length' should be 2.

'bytes_returned' does not contain any meaningful data upon completion.

Note: The data channel number can also be set when issuing an L4L3mENABLE_PROTOCOL message.

Note: This mechanism allows for two handles 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 bytes;
if (!DeviceIoControl(NaiDataHandle, IOCTL_NAI_SET_DCHAN,
                    &channel_number, 2, NULL, 0, &bytes_returned, NULL)) {
    last_err = GetLastError();
}
```

Returns **STATUS_SUCCESS** - The data channel number has been set.

IOCTL_NAI_RUN_TEST

Structure

```
#include"nailib.h"
BOOLDeviceIoControl(NaiDiagHandle, IOCTL_NAI_RUN_TEST, NULL, 0,
    NULL, 0, &bytes_returned, NULL);
HANDLENaiDiagHandle;
int *bytes_returned;
```

Usage This ioctl is used to start the diagnostic POST test.
'bytes_returned' contains no meaningful data on completion.

Example

```
intbytes;
if (!DeviceIoControl(NaiDiagHandle, IOCTL_NAI_RUN_TEST, NULL, 0,
    NULL,
    0, &bytes, NULL))
    return GetLastError();
```

Returns **STATUS_SUCCESS** - The POST test was started.

IOCTL_NAI_TX_HALT

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(NaiDataHandle, IOCTL_NAI_TX_HALT, NULL,
    0, NULL, 0, &bytes_returned, NULL);

HANDLE NaiDataHandle;

int *bytes_returned;
```

Usage This ioctl is used to stop the transmission of frames on a given handle. Instant ISDN will stop transmitting frames as soon as possible. The ioctl will not complete until IISDN has acknowledged that transmission has been halted.

'bytes_returned' contains no meaningful data on completion.

Example

```
int bytes;

if (!DeviceIoControl(NaiDataHandle, IOCTL_NAI_TX_HALT, NULL, 0,
    NULL,
    0, &bytes, NULL))
    return GetLastError();
```

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

IOCTL_NAI_TX_RESUME

- Structure**
- ```
#include"nailib.h"
BOOLDeviceIoControl(NaiDataHandle, IOCTL_NAI_TX_RESUME, NULL, 0,
 NULL, 0, &bytes_returned, NULL);
HANDLENaiDataHandle;
int *bytes_returned;
```
- Usage**
- This ioctl is used to restart the transmission of frames on a given handle after they have been stopped by an IOCTL\_NAI\_TX\_HALT ioctl. Instant ISDN will start transmitting frames immediately upon receiving this call. The ioctl will not complete until IISDN has acknowledged that transmission has been restarted.
- 'bytes\_returned' contains no meaningful data on completion.
- Example**
- ```
intbytes;
if (!DeviceIoControl(NaiDataHandle, IOCTL_NAI_TX_RESUME, NULL, 0,
    NULL,
    0, &bytes, NULL))
    return GetLastError();
```
- Returns**
- STATUS_SUCCESS** - The transmit stream has been restarted.

IOCTL_NAI_SET_LOW_WATER

Structure

```
#include "nailib.h"
BOOL DeviceIoControl(NaiDataHandle, IOCTL_NAI_SET_LOW_WATER,
                    water_mark, length, NULL, 0, &bytes_returned, NULL);
HANDLE NaiDataHandle;
short *water_mark;
int length;
int *bytes_returned;
```

Usage

This ioctl resets the IISDN Dchannel descriptor low water mark for controlling the number of transmit complete interrupts (for more information, see the SMI Programmer's Guide). A default value is set when the handle is opened (value depends on the tunable parameter LowWaterMark).

'water_mark' points to a short value containing the requested level.

'length' should be 2.

'bytes_returned' contains no meaningful data upon completion.

Example

```
short water_mark = 2;
int bytes;
if (!DeviceIoControl(NaiDataHandle, IOCTL_NAI_SET_LOW_WATER, (char *)
                    &water_mark, 2, NULL, 0, &bytes_returned, NULL)) {
    last_err = GetLastError();
}
```

Returns

STATUS_SUCCESS - The low water mark has been reset.

IOCTL_NAI_DEVICE_TYPE

Structure

```
DeviceIoControl(NaiCtrlHandle, IOCTL_NAI_DEVICE_TYPE, NULL, 0,
                &device_type, sizeof(int), &bytes_returned, NULL);
HANDLE NaiCtrlHandle;
int device_type;
int bytes_returned;
```

Usage This ioctl can be used to determine the card type associated with this particular handle. It will return one of NAI_PCI_CARD or NAI_CPCI_CARD.

Example

```
int device_type;
int bytes_returned;
HANDLE handle;

if (!DeviceIoControl(NaiCtrlHandle, IOCTL_NAI_DEVICE_TYPE, NULL, 0,
                    &device_type, sizeof(int), &bytes_returned,
                    NULL)) { return GetLastError();
}
printf("device type is %d\n", device_type);
```

Returns **STATUS_SUCCESS** - The device type was returned.

IOCTL_NAI_SERVICE_EVENT

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(          handle,
                          IOCTL_NAI_SERVICE_EVENT,
                          NULL,
                          0,
                          serviceEvent,
                          sizeof(SERVICE_EVENT),
                          &bytesReturned,
                          NULL );

HANDLE          handle;
PSERVICE_EVENT serviceEvent;
int             bytesReturned;
```

Usage

This IOCTL enables a Win32 user mode thread to receive a Service Event in response to either an Adapter Fault or Hot Swap event.

Service Event	Event Type	Description of Service Event
SURPRISE_EXTRACTION	Hot Swap	A surprise extraction of a CompactPCI adapter has occurred.
EXTRACTION_REQUESTED	Hot Swap	A request to extract the CompactPCI adapter has occurred.
ADAPTER_INSERTION	Hot Swap	The CompactPCI adapter has been installed into the system.
FATAL_ERROR	Adapter Fault	Instant ISDN issued an L3L4irsnFATAL_ERROR interrupt.
TIMEOUT_ERROR	Adapter Fault	Instant ISDN has not updated the adapter's timestamp in the past 2 seconds.

Returns

STATUS_SUCCESS - The IOCTL_NAI_SERVICE_EVENT device I/O control completed successfully, and a SERVICE_EVENT was returned.

STATUS_PENDING - The IOCTL_NAI_SERVICE_EVENT device I/O control is queue to the EVENT_LIST_ITEM for this file object and is pending completion.

STATUS_BUFFER_TOO_SMALL - The buffer is too small to contain a SERVICE_EVENT. No information has been written to the buffer.

STATUS_NO_MEMORY - Not enough memory available to allocate an EVENT_LIST_ITEM.

Notes

1. Hot Swap aware Win32 applications are required to acknowledge the receipt of an EXTRACTION_REQUESTED Hot Swap Event by either:
 - a. Calling DeviceIoControl(fd, NIA_EXTRACTION_AUTHORIZED, ...);

- b. Closing the handle on which the EXTRACTION_REQUESTED Service Event was received
2. Hot Swap aware Win32 applications are required to coordinate among themselves to determine which application will download the adapter upon receipt of an ADAPTER_INSERTION Hot Swap Event.

SERVICE_EVENT The following definition is found in the include file naidrv.h.

```
typedef enum {  
SURPRISE_EXTRACTION = 0, // Hot Swap: Surprise extraction occurred.  
EXTRACTION_REQUESTED = 1, // Hot Swap: A request to extract the adapter was made // (EXT  
is active in HS_CSR).  
ADAPTER_INSERTION = 2, // Hot Swap: Adapter has been re-inserted into the system chassis,  
// and the IRP device driver has re-established it's I/O resources.  
FATAL_ERROR = 3, // Adapter Fault: L3L4irsnFATAL_ERROR interrupt received.  
TIMEOUT_ERROR = 4, // Adapter Fault: IISDN failed to update onboard timestamp.  
} SERVICE_EVENT, *PSERVICE_EVENT;
```

Example

```
void ServiceEvent (HANDLE fd )
{
    SERVICE_EVENT  serviceEvent;
    DWORD  bytesReturned;

    value = DeviceIoControl( fd,
                            IOCTL_NAI_SERVICE_EVENT,
                            NULL,
                            0,
                            &serviceEvent,
                            sizeof(SERVICE_EVENT),
                            &bytesReturned,
                            NULL );

    if ( value ) {
=       if ( bytesReturned == sizeof( SERVICE_EVENT ) ) {

        switch ( serviceEvent ) {
        case SURPRISE_EXTRACTION:
            printf("SURPRISE_EXTRACTION\n");
            break;

        case EXTRACTION_REQUESTED:
            printf("EXTRACTION_REQUESTED\n");
            break;

        case ADAPTER_INSERTION:
            printf("ADAPTER_INSERTION\n");
            break;

        case FATAL_ERROR:
            printf("FATAL_ERROR\n");
            break;

        case TIMEOUT_ERROR:
            printf("TIMEOUT_ERROR\n");
            break;

        default:
            printf("Illegal Service Event returned: 0x%x\n",
                serviceEvent)
            break;
        }
    }
}
```

```
else {
    printf("bytesReturned %d != sizeof(SERVICE_EVENT) %d\n",
        bytesReturned, sizeof(SERVICE_EVENT) );
}
}
else {
    printf("IOCTL_NAI_SERVICE_EVENT failed with 0x%x\n",
        GetLastError());
}
}
```

IOCTL_HOT_SWAP_STATE

Structure

```
#include "nailib.h"

        BOOL DeviceIoControl(        handle,
        IOCTL_HOT_SWAP_STATE,
        NULL,
        0,
        hotSwapState,
        sizeof(HOT_SWAP_STATE),
        &bytesReturned,
        NULL );

HANDLE        handle;
PHOT_SWAP_STATE        hotSwapState;
DWORD        bytesReturned;
```

Usage

This IOCTL allows a thread to obtain the Hot Swap State of the CompactPCI adapter.

Returns

STATUS_SUCCESS - CompactPCI adapter's Hot Swap State was successfully returned.

STATUS_NOT_SUPPORTED - This request is not supported by this adapter (only NS300/NS301 CompactPCI adapters)

STATUS_BUFFER_TOO_SMALL - The buffer is too small to contain the HOT_SWAP_STATE. No information has been written to the buffer.

Notes

This device I/O control request is supported on CompactPCI adapters only.

HOT_SWAP_STATE

```
typedef enum {

    P0 = 0,          // Board is physically separate from the system
                    // (i.e., extracted).

    P1 = 1,          // Board is fully seated, but not powered, and not
                    // active on the PCI bus.

    H0 = 1,

    H1 = 2,          // Board has powered up and is sufficiently
                    // initialized to connect to the PCI bus.

    H1F = 3,         // The board has been commanded to power up and
                    // initialize and has failed,
                    // or the board has detected an error and
                    // disconnected itself from the PCI bus.
                    // The board is not suitable for connection to the
                    // PCI bus.

    H2 = 4,          // The board is powered, and enabled for access by
                    // the the PCI bus in

    S0 = 4,          // configuration space only. The boards's
                    // configuration space is not yet initialized.

    S1 = 5,          // The board is configured by the system.

    S2 = 7,          // The necessary supporting software (drivers,
                    // etc.) are loaded.
                    // The board is ready for use by the OS and/or
                    // Applications,
                    // but no operations involving the bard are active.

    S2Q = 6,         // This state is same as state S2, but no new
                    // operations
                    // are allowed to start. The board is quiesced.

    S3 = 9,          // The board is engaged in software operations.

    S3Q = 8          // The software is completing current operations,
                    // but is not allowed to start new ones.

} HOT_SWAP_STATE, *PHOT_SWAP_STATE;
```

Example

```
void HotSwapState( HANDLE fd ) {
    HOT_SWAP_STATE hotSwapState;
    DWORD bytesReturned;
    BOOL value;

    value = DeviceIoControl( fd,
        IOCTL_HOT_SWAP_STATE,
        NULL,
        0,
        &hotSwapState,
        sizeof(HOT_SWAP_STATE),
        &bytesReturned,
        NULL );

    if ( value ) {
        if ( bytesReturned == sizeof(HOT_SWAP_STATE) ) {
            switch ( hotSwapState ) {
                case P0:
                    printf("P0\n"); break;
                case P1:
                    printf("P1 - H0\n"); break;
                case H1:
                    printf("H1\n"); break;
                case H1F:
                    printf("H1F\n"); break;
                case H2:
                    printf("H2 - S0\n"); break;
                case S1:
                    printf("S1\n"); break;
                case S2:
                    printf("S2\n"); break;
                case S2Q:
                    printf("S2Q\n"); break;
                case S3:
                    printf("S3\n"); break;
                case S3Q:
                    printf("S3Q\n"); break;
                default:
                    printf("Illegal Hot Swap State returned: 0x%x\n",
                        hotSwapState); break;
            }
        }
        else {
            printf("bytesReturned %d != sizeof(HOT_SWAP_STATE) %d\n",
                bytesReturned, sizeof(HOT_SWAP_STATE) );
        }
    }
}
```

```
        }  
    }  
    else {  
        printf("IOCTL_HOT_SWAP_STATE failed with 0x%x\n",  
GetLastError());  
    }  
}
```

IOCTL_EXTRACTION_AUTHORIZED

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(      handle,
IOCTL_EXTRACTION_AUTHORIZED,
NULL,
0,
NULL,
0,
&bytesReturned,
NULL );

HANDLE   handle;
DWORD   bytesReturned;
```

Usage Hot Swap aware Win32 applications inform the IRP Device Driver that it has prepared for the removal of the CompactPCI adapter. This is called **ONLY** in response to receiving an **EXTRACTION_REQUESTED** Hot Swap Event.

Returns **STATUS_SUCCESS** – The **IOCTL_EXTRACTION_AUTHORIZED** request was successful.

STATUS_NOT_SUPPORTED - This request is not supported by this adapter (only NS300/NS301 CompactPCI adapters).

STATUS_INVALID_HANDLE - **IOCTL_EXTRACTION_AUTHORIZED** was issued by a handle that did not issue an **IOCTL_NAI_SERVICE_EVENT** request.

Notes

1. This device I/O control request is supported on CompactPCI adapters only.
2. The handle used to issue this request must be the same handle that was used to issue the **DeviceIoControl(fd, NAI_HOT_SWAP_EVENT, ...)** on which the **EXTRACTION_REQUESTED** Hot Swap Event was received.

Example

```
void AuthorizeExtraction( HANDLE fd )
{
    BOOL value;
    DWORD bytesReturned;

    value = DeviceIoControl( fd,
                             IOCTL_EXTRACTION_AUTHORIZED,
                             NULL,
                             0,
                             NULL,
                             0,
                             &bytesReturned,
                             NULL);

    if ( value ) {
        printf("EXTRACTION AUTHORIZED\n");
    }
    else {
        printf("IOCTL_EXTRACTION_AUTHORIZED failed with 0x%x\n",
              GetLastError());
    }
}
```

IOCTL_HOT_SWAP_EXTRACT_ADAPTER

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(    handle,
    IOCTL_HOT_SWAP_EXTRACT_ADAPTER,
    NULL,
    0,
    NULL,
    0,
    &bytesReturned,
    NULL );

HANDLE handle;
DWORD bytesReturned;
```

Usage Hot Swap aware Win32 applications can request that the NT 4.0 IRP device driver extract a CompactPCI adapter.

Returns

STATUS_SUCCESS – The IOCTL_HOT_SWAP_EXTRACT_ADAPTER request was successful (i.e., the CompactPCI adapter was “extracted”).

STATUS_NOT_SUPPORTED - This request is not supported by this adapter (only NS300/NS301 CompactPCI adapters).

STATUS_ILLEGAL_FUNCTION - Extraction is not possible given current Hot Swap State.

Notes

1. This device I/O control request is supported on CompactPCI adapters only.
2. This is a software interface that allows the programmer to initiate the orderly extraction of the CompactPCI adapter (e.g., as if the administrator had opened the CompactPCI adapter’s lower ejector handle).
3. The adapter’s Blue LED will light up.

Example

```
void ExtractAdapter( HANDLE fd )
{
    BOOL value;
    DWORD bytesReturned;

    value = DeviceIoControl( fd,

IOCTL_HOT_SWAP_EXTRACT_ADAPTER,
NULL,
0,
NULL,
0,
&bytesReturned,

NULL);

    if ( value ) {
        printf("IOCTL_HOT_SWAP_EXTRACT_ADAPTER
successful\n");
    }
    else {
        printf("IOCTL_HOT_SWAP_EXTRACT_ADAPTER failed
with 0x%x\n", GetLastError());
    }
}
```

IOCTL_HOT_SWAP_INSERT_ADAPTER

Structure

```
#include "nailib.h"

BOOL DeviceIoControl(    handle,
    IOCTL_HOT_SWAP_INSERT_ADAPTER,
    NULL,
    0,
    NULL,
    0,
    &bytesReturned,
    NULL );

HANDLE handle;
DWORD bytesReturned;
```

Usage Hot Swap aware Win32 applications can request that the NT 4.0 IRP device driver insert a CompactPCI adapter.

Returns

STATUS_SUCCESS – The IOCTL_HOT_SWAP_INSERT_ADAPTER request was successful. (i.e., the CompactPCI adapter was “inserted”).

STATUS_NOT_SUPPORTED - This request is not supported by this adapter (only NS300/NS301 CompactPCI adapters).

STATUS_ILLEGAL_FUNCTION - Insertion is not possible given current Hot Swap State.

Notes

1. This device I/O control request is supported on CompactPCI adapters only.
2. This is a software interface that allows the programmer to initiate the orderly insertion of the CompactPCI adapter (e.g., as if the administrator had inserted the CompactPCI adapter into the system chassis and closed the ejector handles).
3. The adapter's Blue LED will turn off.

Example

```
void InsertAdapter( HANDLE fd )
{
    BOOL value;
    DWORD bytesReturned;

    value = DeviceIoControl( fd,

    IOCTL_HOT_SWAP_INSERT_ADAPTER,
    NULL,
    0,
    NULL,
    0,
    &bytesReturned,
    NULL);

    if ( value ) {
        printf("IOCTL_HOT_SWAP_INSERT_ADAPTER
successful\n");
    }
    else {
        printf("IOCTL_HOT_SWAP_INSERT_ADAPTER failed
with 0x%x\n", GetLastError());
    }
}
```




Library Services

Introduction

This section describes the Windows 2000 Driver Library available for use in application development. Tables 4.1 through Table 4.4 lists each library function and describes its purpose; the functions appear throughout this section.

The library routines described here have all been implemented with overlapped I/O. Hence, it is possible to issue all of these requests simultaneously on different threads within a process (although some combinations won't make any sense). It is expected that the standard application will want to issue send and receives simultaneously, so this library was written to try and shield the programmer from some of the quirks of 32 bit Windows programming.

Note: This does not mean these calls are asynchronous. In fact, the majority of these call are synchronous, however they may be issued simultaneously on separate threads.

Table 4-1. Control Messaging

Adapter Type	Control Message Handling	Purpose
All	NaiControlRead	Control message is read from the adapter.
All	NaiControlWrite	Control message is written to the adapter.

Table 4-2. VTTY Handling

Adapter Type	VTTY Handling	Purpose
All	NaiVttyRead	Data is read from the VTTY port on the adapter, up to the length of bytes requested.
All	NaiVttyWrite	Data is written to the VTTY port on the adapter, up to the length of bytes available.
All	NaiSetVTTY	Selects this handle as being the VTTY recipient for the adapter.
All	NaiClearVTTY	Releases this particular handle as being the VTTY recipient for the adapter.

Table 4-3. Data Messaging

Adapter Type	Data Handling	Purpose
All	NaiDataRead	Data is read from the data channel descriptor, up to the length of bytes requested.
All	NaiDataWrite	Data is written to the data channel descriptor, up to the length of bytes requested.
PCI/CPCI	NaiTxHalt	Stops an outgoing data stream on the data channel descriptor.
PCI/CPCI	NaiTxResume	Restarts an outgoing data stream on the data channel descriptor.
All	NaiDataReadNoPend	Receive data events with call back.
All	NaiDataWriteNoPend	Transmit data events with call back.
All	NaiReadWillPend	Returns true if next ReadFile() are likely to respond.

Table 4-4. Device Management

Adapter Type	Device Management	Purpose
All	NaiOpenAdapter	Opens the adapter card and associates it with a particular handle.
All	NaiCloseAdapter	Close and release a previously open handle.
All	NaiResetAdapter	Places adapter in a reset mode.
All	NaiDownloadAdapter	Downloads IISDN software to adapter.
All	NaiReadCrashData	Reads the crash dump area on the adapter.
All	NaiOpenChannel	Opens an adapter and associates a data channel descriptor.
PCI/CPCI	NaiCloseChannel	Closes and release a previously opened handle.
All	NaiRunDiags	Starts execution of diagnostics.
All	NaiGetDeviceType	Returns PCI or CompactPCI
All	NaiGetVersionIISDN	Returns version of IISDN the driver and library are built with, and the version of IISDN running on the adapter.
NS30x	NaiLedControl	Blink LEDs on NS300/NS301 adapters, or return them to normal operation.

Functional Library

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

The library provides a simple and straight forward interface for the user to program. To this end, the library is responsible for handling all issues relating to Windows 32 File I/O, such as Overlapped I/O.

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 control messages (L4L3 and L3L4) to and from the adapter.

NaiControlRead

Structure

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

Usage

This call reads a 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(NaiDataHandle, &msg)) {
    error_out("Can't read control message.");
}
switch (msg.msgtype) {
    case L3L4mALERTING:
        ...
}
```

Returns

```
= 0 Success.
> 0 Failed. The return value is the thread's last-error code value (i.e. which was obtained from GetLastError()).
```

NaiControlWrite

Structure

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

Usage

This call writes a control message to the adapter specified by the given handle. The call will pend until room is available on the SMI queue.

Example

```
L4_to_L3_struct  msg;

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

Returns

```
= 0    Success.
> 0    Failed. The return value is the thread's last-error code value
        (i.e. which was obtained from GetLastError()).
```

VTTY Handling

These messages are used for reading, writing and managing the IISDN VTTY port.

NaiVttyRead

Structure

```
#include "nailib.h"
int      NaiVttyRead(NaiVttyHandle, buffer, length);
HANDLE   NaiVttyHandle;
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 'handle'. Data is stored in 'buffer'. Upon completion, 'length' contains the actual number of bytes read. This call will pend until data arrives.

Example Output

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

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

Returns

```
= 0   Success.
> 0   Failed. The return value is the thread's last-error code value
      (i.e. which was obtained from GetLastError()).
```

NaiVttyWrite

Structure

```
#include "nailib.h"
int      NaiVttyWrite(NaiVttyHandle, buffer, length);
HANDLE   NaiVttyHandle;
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 'handle'. 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(NaiVttyHandle, &ch, &length)) {
error_out("Can't write data.");
}
printf("Wrote %d byte(s).\n", length);
```

Returns

```
= 0    Success.
> 0    Failed. The return value is the thread's last-error code value
        (i.e. which was obtained from GetLastError()).
```

NaiSetVTTY

Structure

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

Usage

This call selects this handle as being the VTTY recipient for the adapter specified by the handle.

Returns

```
= 0    Success.
> 0    Failed. The return value is the thread's last-error code value
        (i.e. which was obtained from GetLastError()).
```

NaiClearVTTY

Structure

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

Usage

This call releases this particular handle as being the VTTY recipient for the adapter specified by the handle.

Returns

```
= 0      Success.
> 0      Failed. The return value is the thread's last-error code value
         (i.e. which was obtained from GetLastError()).
```

Data Messaging

These messages are used to send and receive data on a particular data channel 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(NaiDataHandle,buffer, length);
HANDLE   NaiDataHandle
char     *buffer;
int      *length
```

Usage

This call reads up to 'length' bytes of data from the dchannel descriptor specified by the given 'handle'. Data is stored in 'buffer'. Upon completion, 'length' contains the actual number of bytes read. This call will pend until data arrives.

Example Output

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

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

Returns

```
= 0   Success.
> 0   Failed. The return value is the thread's last-error code value
      (i.e. which was obtained from GetLastError()).
```

NaiDataWrite

Structure

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

Usage

This call writes up to 'length' bytes of data to the dchannel descriptor specified by the given 'handle'. Data is stored in 'buffer'. Upon completion, 'length' contains the actual number of bytes written. This call will pend until the data has been transmitted.

Example Output

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

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

Returns

```
= 0    Success.
> 0    Failed. The return value is the thread's last-error code value
        (i.e. which was obtained from GetLastError()).
```

NaiTxHalt

Structure

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

Usage

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

Returns

```
= 0    Success.
> 0    Failed. The return value is the thread's last-error code value
        (i.e. which was obtained from GetLastError()).
```

NaiReadWillPend

Structure	<pre>#include nailib.h" int NaiReadWillPend(NaiDataHandle); HANDLE NaiDataHandle;</pre>
Usage	<p>This routine returns a boolean value indicating if the next <code>NaiDataRead</code> call will likely pend.</p>
Returns	<pre>=0 Next read will not pend. =1 Next read may pend. < 0 Failed. The return value is the negative of the thread's last-error code value (i.e. which was obtained from GetLastError()).</pre>

NaiTxResume

Structure

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

Usage

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

Returns

```
= 0      Success.
> 0      Failed. The return value is the thread's last-error code
value (i.e. which was obtained from GetLastError()).
```

Callback Routine

Structure

```
void      (callback) (NaiCallbackHandle, ARG, error, bytes, event)
HANDLE    NaiCallbackHandle;
void      *ARG;
DWORD     error;
DWORD     bytes;
HANDLE    event;
```

Usage This routine is called in response to the completion of a previous `NaiDataReadNoPend()` or `NaiDataWriteNoPend()`. Callback routines can only be run during a point when a thread is in an "alertable state", such as during `SleepEx()`, `WaitSingleObjectEx()`, `WaitMultipleObjectEx()`, etc.

Returns None.

Example

```
....
....
    length = 1024;
    NaiDataReadNoPend(NaiCallbackHandle, buffer, &length, &out_handle,
        ReadDone, buffer);
....
....
void ReadDone (Handle fd, void*arg, DWORD ERR, DWORD bytes,
    Handle event)
{
    printf ("Read on buffer %x done!\n", ARG),
}
```

NaiDataWriteNoPend

Structure

```
#include "nailib.h"

int      NaiDataWriteNoPend(NaiDataHandle, buff, len,
                           out_handle, callback,
                           ARG);

HANDLE   NaiDataHandle;

chan     *buff
int      *len
int      *out_handle;
void     (*callback)();
void     *ARG
```

Usage

This call writes up the "len" bytes of data to the dchannel description specified by "NaiDataHandle". Data is stored in "buff". Upon completion, "len" contains the number of bytes written and out_handle contains a pointer to an OVERLAPPED structure that can be used in other Win32 function calls. The data buffer specified by buff can not be re-used or modified until the callback routine has been run.

Note: The callback routine will only be run when the thread issuing the NaiDataWriteNoPend() enters as "alertable-state"; for more information, refer to the Microsoft documentation. A null callback can be specified and the returned HANDLE (out_handle) can be used in any Microsoft call that waits for an event to be signaled (e.g. WaitForSingleObject()).

Returns

```
= 0      Success.
> 0     Failed. The return value is the thread's last-error code
value (i.e. which was obtained from GetLastError()).
```

NaiDataReadNoPend

Structure

```
#include "nailib.h"

int      NaiDataReadNoPend (NaiDataHandle, buff, len, out_handle,
                           callback,
                           ARG);

HANDLE   NaiDataHandle;

char     *buff;

int      *len;

int      *out_handle

void     (*callback)();

void     *ARG;
```

Usage

This call reads up to 'len' bytes of data from the data channel descriptor specified by 'NaiDataHandle'. Data is stored in 'buff'. Upon completion, "len" contains the number of bytes read and out_handle contains a pointer to an OVERLAPPED structure that can be used in other Win32 function calls. This call will return immediately. If data was read, it will return a 0 and the callback routine will be run. If the read pends, a handle is returned and the callback run at a later time.

The data buffer specified by buff can not be re-used or modified until the callback routine has been run.

Note: The callback routine will only be run when the thread issuing the NaiDataReadNoPend() is in an "alertable-state"; for more information, refer to the Microsoft documentation. A null call can be specified and the returned HANDLE (out_handle) used in any Microsoft call that waits for event to be signaled (e.g. WaitForSingleObject()).

Example

```
char  buff[256];
HANDLE read_handle;

int   len;
int   err;

len = size of (buff)
err = NaiDataReadNoPend(NaiDataHandle, buff, &len, &read_handle,
                       callback, buff);
```

Return

```
= 0 Success

> 0 Failed. The return value is the thread's last-error code
value (i.e. which was obtained from GetLastError()).
```

Device Management

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

NaiOpenAdapter

Structure

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

Usage

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

Returns

```
>= 0  Successful open, return value is the handle.
<=   INVALID_HANDLE_VALUE Open failed. Call GetLastError() for
      reason.
```

NaiRunDiags

Structure

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

Usage

This call starts a diagnostic sequence on PCI cards.

Returns

```
=0      Success
>0      Failed. The return value is the thread's last-error code value
        (i.e. which was obtained from GetLastError()).
```

NaiCloseAdapter

Structure

```
#include "nailib.h"
int      NaiCloseAdapter(NaiCtrlHandle);
HANDLE   NaiCtrlHandle;
```

Usage

This call closes and releases a previously open HANDLE.

Returns

Always returns 0.

NaiResetAdapter

Structure

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

Usage

This call will place the adapter associated with 'NaiCtrlHandle' into a reset state.

Returns

```
= 0      Success.
> 0      Failed. The return value is the thread's last-error code
value (i.e. which was obtained from GetLastError()).
```

NaiDownloadAdapter

Structure

```
#include "nailib.h"
int      NaiDownloadAdapter(NaiCtlrHandle, fileSpecification);
HANDLE   NaiCtlrHandle;
char     *fileSpecification;
```

Usage This routine downloads the adapter. The “fileSpecification” is a fully qualified path name to either an S-Record file or a binary image (microcode) file. **NaiDownloadAdapter** will test the first line of the file to determine whether it is a S-Record or binary image file.

Notes

- NaiDownloadAdapter will pend until the file has been downloaded and the adapter is running.
- Downloading of binary files is supported on PCI and cPCI adapters only.
- Brooktrout Technology distributes S-Record files. An S-Record file can be converted to a binary image file by using the SREC2BIN utility. The source code for the SREC2BIN utility is available on the FTP site: ftp.netacc.com/support/utills/srec2bin..

Example Output *Example 1: Download S-Record file.*

```
HANDLE      ctlrHandle;
int          ret;

      ctlrHandle = NaiOpenAdapter(busNumber, slotNumber);
      if (ctlrHandle == Invalid_Handle_Value) {
          error_out("Can't open adapter", ctlrHandle, busNumber,
slotNumber);
      }

      if ((ret = NaiDownloadAdapter(ctlrHandle, "C:\program
files\Brooktrout Technology\Windows 2000 IRP
Kit\irp\images\naii.0"));
          error_out("Can't download adapter.", ret);
      }
```

Example 2: Download binary image file for PCI & cPCI adapter only.

```

HANDLE    ctrlHandle;
          int        ret;

          ctrlHandle = NaiOpenAdapter(busNumber, slotNumber);
          if (ctrlHandle == Invalid_Handle_Value) {
              error_out("Can't open adapter", ctrlHandle, busNumber,
slotNumber);
          }
// Download the binary image file (e.g., created with the SREC2BIN
utility).
          if ((ret = NaiDownloadAdapter(ctrlHandle, "C:\temp\pri.bin");
              error_out("Can't download adapter.", ret);
          }

```

Returns

```

= 0    Download successful.
> 0    Download failed. The return value is the thread's last-
error code value
       (i.e., which was obtained from GetLastError()).

```

Returns

```

= 0    Download successful.
> 0    Download failed. The return value is the thread's last-
error code value (i.e. which was obtained from GetLastError()).

```

NaiReadCrashData

Structure

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

Usage

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

Returns

```
= 0    Success.
> 0    Failed. The return value is the thread's last-error code
value (i.e. which was obtained from GetLastError()).
```

NaiOpenChannel

Structure

```
#include "nailib.h"
HANDLE 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 handle.
< 0  Open failed. The return value is the negative of the
thread's last-error code value (i.e. which was obtained from
GetLastError()).
```

NaiCloseChannel

Structure

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

Usage This call closes and releases a previously open handle.

Note: Developers will need to disable protocols on links before they issue the `NaiCloseChannel()` command.

Returns Always returns 0.

NaiGetDeviceType

Structure

```
#include "nailib.h"
```

```
int      NaiGetDeviceType(NaiCtrlHandle);  
HANDLE   NaiCtrlHandle;
```

Usage

This call returns NAI_PCI_CARD or NAI_CPCI_CARD. It indicates what kind of device is associated with the handle.

Returns

```
> 0 Success. One of NAI_PCI_CARD or NAI_CPCI_CARD.  
<=0 Failed. Return is -GetLastError().
```

NaiGetVersionIISDN

Structure

```
#include "nailib.h"
```

```
int NaiGetDeviceType(NaiCtrlHandle pVersionInfo);  
HANDLE NaiCtrlHandle;  
PVERSION_INFO pVersionInfo;
```

Usage

This function is used to obtain the IISDN version information for:

- adapter (i.e., version running)
- driver (i.e., version built with)
- library (i.e., version built with).

Returns

=0 Success. The VERSION_INFO structure contains the version of IISDN that the driver and library were built with, and the version running on the adapter.

!0 Failed. Return is GetLastError().

NaiLedControl

Structure	<pre>#include "nailib.h" int NaiLedControl(NaiCtrlHandle blink); HANDLE NaiCtrlHandle; ULONG blink;</pre>
Usage	<p>This function is used control the state of the NS300 and NS301 LEDs, where;</p> <ul style="list-style-type: none">blink=1 blink the adapter's LEDsblink=0 return the adapter's LEDs to normal operation.
Returns	<p>=0 Success. The adapter's LEDs were configured as requested.</p> <p>!0 Failed. Return is GetLastError().</p>
Notes	<p>This function is supported on NS300 and NS301 adapters only. In addition, the adapter must be running the flash code (not the firmware).</p>



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.

Contacting Brooktrout Customer Support

Purchase of a Brooktrout Developer's Kit entitles you to 12-months of unlimited technical support during standard business hours. Our Technical Assistance Center also offers a wide variety of additional fee-based support services, including Extended 24-hour Support Plans, Training Classes, 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, 8:30am to 5:30pm, 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	(781) 433-9600 8:30 am - 5:30 pm EST, Mon - Fri (781) 449-9009 (Fax) 24 hours
eMail	techsupport@brooktrout.com

Additional Brooktrout Support Services

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, FAQ files – you can even open a trouble ticket with a Brooktrout Support Engineer.

Appendix B

Running the Virtual TTY

This section will briefly describe the operations and activities of the Virtual TTY program test.

The Virtual TTY functions and features apply only to Brooktrout PCI and CPCI controllers.

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 *Section 2* for instructions on conducting that operation.

To run the Virtual TTY, the following procedure should be performed:

1. Through the Windows Explorer program or via DOS command screen, go to the following directory location in your Windows NT operating system:

**Program Files\Brooktrout Technology\Windows 2000 IRP
Kit\Sample\VTTY\Release**

In the **Release** directory an application, **VTTY.EXE** is stored. Double click the icon to active the Virtual TTY program if you have gone through the graphical interface or type `vtty.exe` if you are in a DOS command screen.

2. The Virtual TTY will prompt you to select an installed Brooktrout controller.

Note: A Brooktrout controller must be installed in order for the Virtual TTY program to function. If only one adapter is installed in this machine, the dialog box will not appear.

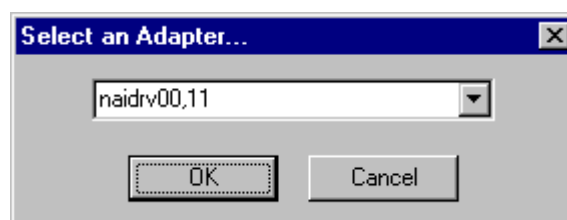


Figure B-1. Adapter Selection Window

3. Select the appropriate adapter to run Virtual TTY and then click **OK** to continue.
4. The Virtual TTY window will then appear, as shown below.

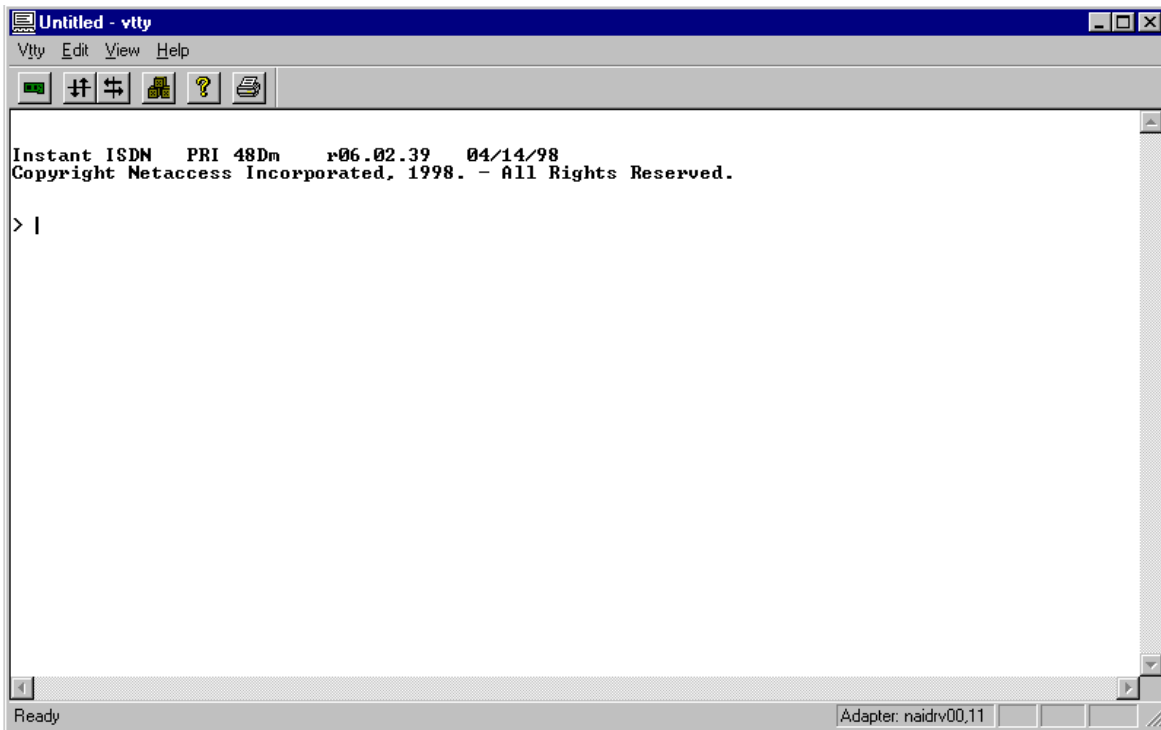


Figure B-2. Basic Virtual TTY Command Window

5. At the top of the Virtual TTY command screen, a series of buttons reside on the toolbar. A balloon function is operational for these buttons, as described below.







Toolbar Button	Toolbar Button Name	Toolbar Button Function
	Select Adapter	Allows the user to swap VTTY to another active controller.
	SMI Trace	Toggle on/off. Displays all messages between L4 (Host) and L3 (Controller). Only messages will be displayed, this function does not go into great detail. Refer to the <i>Instant ISDN Simple Message Interface Reference Manual</i> for more details.
	L2 Trace	Toggles on/off. Link layer trace of messages between central office and local dchannel.
	Inventory	This button prints an inventory of the controller hardware.
	Help	This button displays the current set of debug commands interpreted by the controller.
	Print Button	Enables the user to print the current screen and the displayed information.

Figure B-3. Toolbar Function Display

Displaying Memory

The board initialization process downloads Instant ISDN Software and configures the board's memory map as shown in *Figure B-4*. The shared memory area starts at the address given by the command within a VTTY screen, which will display the shared memory base. The shared memory is then determined by the shared memory base plus f0000.

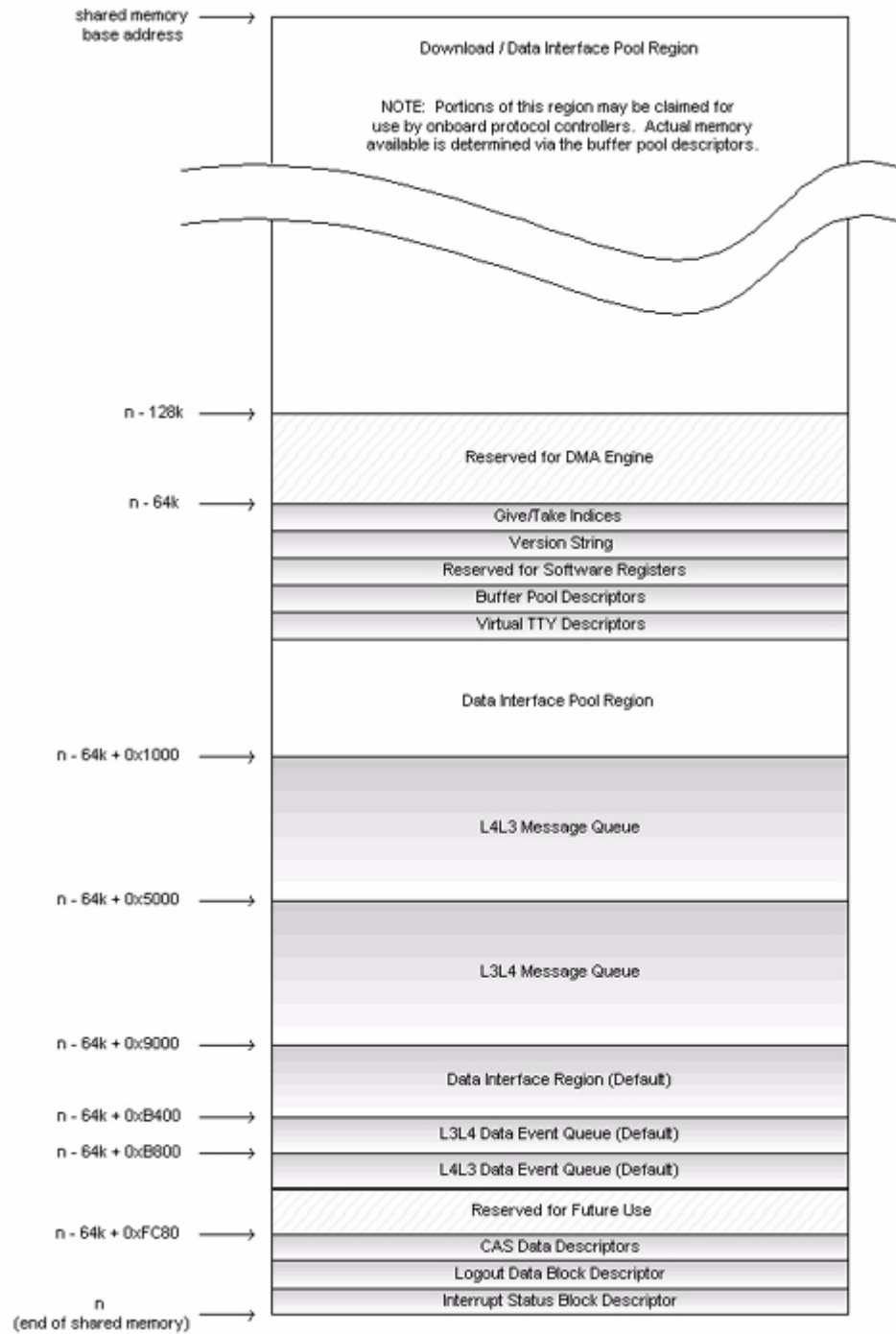


Figure B-4. Memory Map for PCI Boards

To display a 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**. A memory display example is shown in *Figure B-5*

```

Untitled - vty
Vty Edit View Help
Instant ISDN PRI 48Dm r06.02.39 04/14/98
Copyright Netaccess Incorporated, 1998. - All Rights Reserved.

> d fe0f9000 40
fe0f9000 00 71 d1 68 00 67 7e e8 00 06 00 06 05 00 00 00 .q.h.g~.....
fe0f9010 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 ..
fe0f9020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
fe0f9030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
> d 71d168 100
0071d168 28 7f 67 80 00 00 02 00 01 68 17 48 00 00 02 00 <.g.....h.H...
0071d178 01 68 15 28 00 00 02 00 01 95 30 08 00 00 02 00 .h.<.....0....
0071d188 01 95 3d e8 00 00 02 00 01 95 3b c8 00 00 01 00 ..=.....;....
0071d198 00 00 00 00 00 00 00 02 00 01 02 46 61 74 20 ..Fat
0071d1a8 68 cb 86 81 68 85 85 80 48 80 85 80 00 00 00 00 h...h...H...
0071d1b8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 ..
0071d1c8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
0071d1d8 00 00 00 00 00 00 00 02 00 01 01 46 61 74 20 ..Fat
0071d1e8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
0071d1f8 00 00 00 00 00 00 00 01 00 01 01 46 61 74 20 ..Fat
0071d208 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
0071d218 00 00 00 00 10 00 00 84 01 00 01 01 46 61 74 20 ..Fat
0071d228 88 1b 7a 80 00 00 00 00 00 00 00 00 00 00 00 00 ..z.....Uad
0071d238 00 00 00 00 06 00 00 01 00 01 02 56 61 64 20 ..
0071d248 00 00 78 00 ff 1f 78 00 48 09 72 80 48 ef 71 80 ..x...H.r.H.q.
0071d258 a8 88 75 80 00 00 80 01 68 38 76 80 48 de 5e e1 ..u.....h8v.H...
>
Ready Adapter: naidrv00,11

```

Figure B-5. Memory Display Example

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 menu appears as shown in *Figure B-6*.

```

> ?
InstantISDN r06.02.39  command line help
b>ond # - bonding debug control
d>ump <address> <length>
rb/rw/rl address - peek at memory location
wb/ww/wl address value - poke a memory location
debug - enter the system debugger
l>ap # - link layer trace control
t>race - toggle SMI tracing
s>how - display channel or module info <try show ?>
>
>
>
>
>
Ready Adapter: naidrv00,11

```

Figure B-6. Diagnostics Menu

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



CAUTION: 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 fe0f0010 and fe0f0020. Each hexadecimal byte in the string represents an ASCII character code. A sample version string for a PCI controller running Instant ISDN Software 6.0 is shown below:

```

>
>
>
>
> d fe0f0000 40
fe0f0000 00 01 00 02 00 01 00 01 00 00 00 00 00 00 00 00 00
fe0f0010 50 52 49 20 34 38 44 6d 20 20 72 30 36 2e 30 32  PRI 48Dm r06.02
fe0f0020 2e 33 39 20 20 30 34 2f 31 34 2f 39 38 00 20 20  .39 04/14/98.
fe0f0030 00 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00
> |
Ready Adapter: naidrv00,11

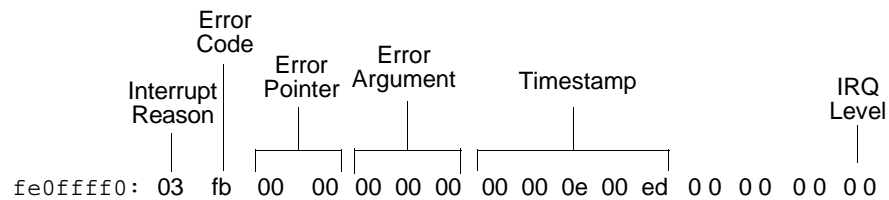
```

Figure B-7. Version String Display on Virtual TTY Screen

Interrupt Status Block

The Interrupt Status Block (ISB) is located at 0xFE0FFF0. 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 IISDNerrUNKNOWN_INTERRUPT) on a PCI board is shown below:



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 **L1** and press **Return**. To stop the trace, type **L0** and press **Return**. Values that appear during the trace are defined in *Figure B-8*. *Figure B-8* 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-8. Level 1 Trace Example

Table B-1. Trace Values & Meanings

Value	Meaning
Ch#	HDLC channel used for formatting; possible values are 0 – 64 ^a
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 ^b	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-9* 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-9. 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.

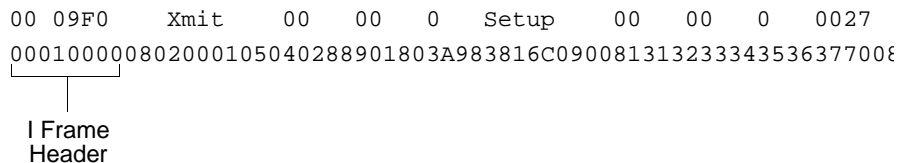


Figure B-10 compares the general format for an I Frame against the I Frame for an example SETUP message, and illustrates the following points:

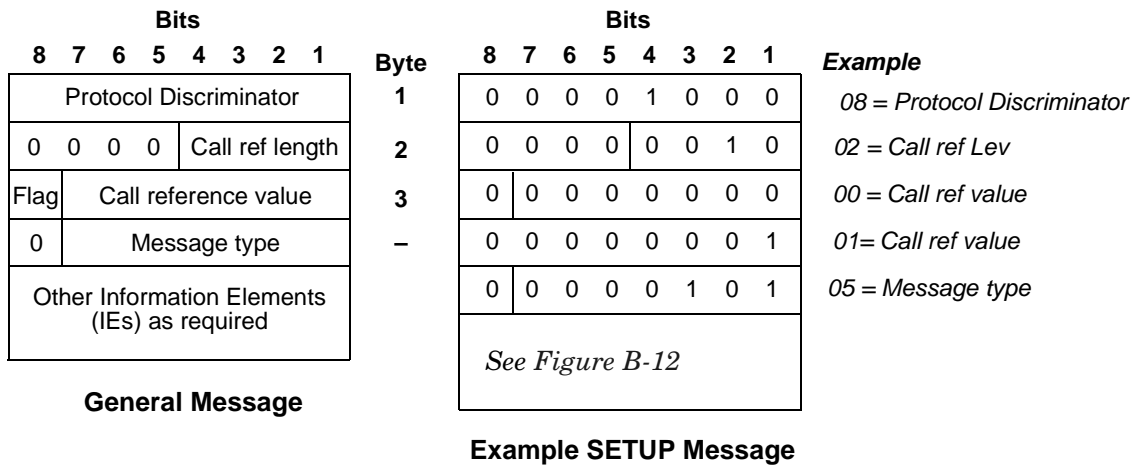


Figure B-11. Message Structures

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.

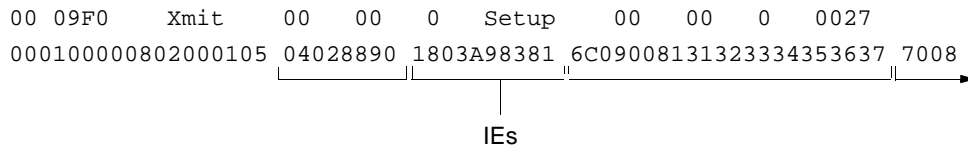


Figure B-12 compares the general IE format against the first IE contained in the example SETUP message, and illustrates the following points:

- 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).

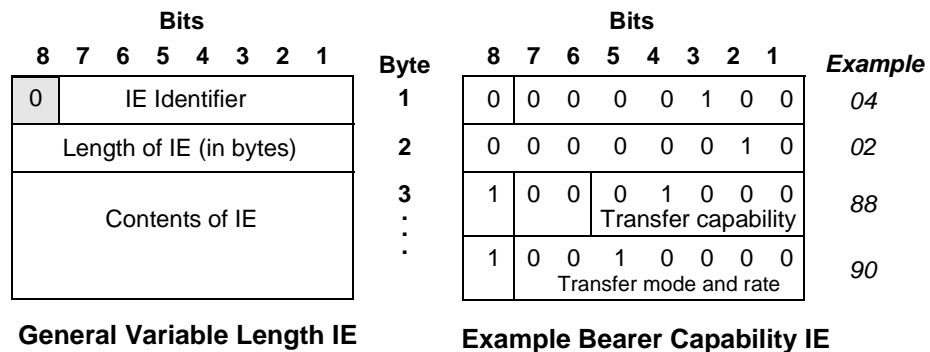


Figure B-12. IE Formats

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



Appendix C

Understanding the Sample Applications

This section will briefly illustrate sample applications that are provided with the Windows 2000 IRP kit for your Windows 2000 system. These sample applications are Test, Samp and the VTTY.

Several sample applications are shipped with the Windows 2000 IRP Kit, and consists of various levels of complexity. During this *Appendix C*, the description of basic design choices will be discussed and detailed for your use.

Sample Application: Test

The sample application TEST is a Win32 console application that downloads the adapter and passes data on up to N channels.

Initialization

The input parameters to Test application are passed on the command line, in the form:

```
> Test -b {bus number} -s {slot number} -f {IISDN file} -n {Number of channels to exercise}
```

The Test application will initialize the adapter as follows:

6. Open the adapter via `NaiOpenAdapter()`.
7. Resetting the adapter via `NaiResetAdapter()`.
8. Downloading the IISDN image into the adapter via `NaiDownLoadAdapter()`.

For adapters that have modems, the test application will:

9. Download the request modem image file via `download_modem_image()`.
10. Connect the first half of the modems to the second half using the TSI matrix via `set_tsi_loopback()`.

The Test application will then create the `read_write()` threads which execute forever reading and writing to the data channels.

```
switch (card_type & PRIbdMODEM_CARD) {  
case PRImodem8:    limit = 4; break;  
case PRImodem24:  limit = 12; break;  
case PRImodem30:  limit = 15; break;  
}  
for (i = 0, j = 0; i < limit; i++) {  
l43msg.data.tsi_data.tsi_map[j].destination = PRItsiMODEM_0 + i;
```

```

l43msg.data.tsi_data.tsi_map[j++].source    = PRItsiMODEM_0 + i + limit;

l43msg.data.tsi_data.tsi_map[j].destination = PRItsiMODEM_0 + i + limit;
l43msg.data.tsi_data.tsi_map[j++].source    = PRItsiMODEM_0 + i;
}
l43msg.data.tsi_data.num_mappings = j;

```

For PCI PRI cards we loop all 32 channels back onto themselves.

```

for (i = 0; i < 32; i++) {
    l43msg.data.tsi_data.tsi_map[i].destination = PRItsiHDL_0 + i;
    l43msg.data.tsi_data.tsi_map[i].source     = PRItsiHDL_0 + i;
}
l43msg.data.tsi_data.num_mappings = i;

```

```

for (i = 0, j = 0; i < 32; i++) {
    l43msg.data.tsi_data.tsi_map[j].destination = PRItsiHDL_0 + i;
    l43msg.data.tsi_data.tsi_map[j++].source   = PRItsiDSi0 + i;

    l43msg.data.tsi_data.tsi_map[j].destination = PRItsiDSi0 + i;
    l43msg.data.tsi_data.tsi_map[j++].source   = PRItsiHDL_0 + i;
}
l43msg.data.tsi_data.num_mappings = j;

```

For PCI BRI cards, there is no TSI matrix. Instead the loopback at the framer is set via a L4L3mSET_HARDWARE.

```

l43msg.msgtype = L4L3mSET_HARDWARE;
for (i = 0; i < PRI_MAX_LINES; i++) {
    l43msg.data.hardware_data.line_data[i].briL1_cmd = PRicmdL1_LOOP;
    l43msg.data.hardware_data.line_data[i].bri_loop  = PRIlloopBRI_LOCAL;
}

```

At this point, the application enables the data channel. Applications typically use HDLC to transmit and receive data.

To start an HDLC channel:

```

memset(&l43msg, 0, sizeof(l43msg));
l43msg.msgtype = L4L3mENABLE_PROTOCOL;
l43msg.lapidid = channel;
l43msg.data.enable_protocol.level1.l1_mode = PRIl1modHDL;
l43msg.data.enable_protocol.level2.l2_mode = PRIl2modDISABLED;
l43msg.data.enable_protocol.level2.data_interface.enable = 1;
l43msg.data.enable_protocol.level3.l3_mode = PRIl3modDISABLED;

```

To start a MODEM channel:

```

switch (card_type & PRIbdMODEM_CARD) {
    case PRImodem8:    limit = 4; break;

```

```

        case PRImodem24:  limit = 12; break;
        case PRImodem30:  limit = 15; break;
        case PRImodemNONE:
    }

    //
    // Write an ENABLE_PROTOCOL for the calling side...
    //
    memset(&l43msg, 0, sizeof(l43msg));
    l43msg.msgtype = L4L3mENABLE_PROTOCOL;
    l43msg.lapdid = 32 + channel;
    l43msg.data.enable_protocol.level1.l1_mode = PRl1modMODEM;
    l43msg.data.enable_protocol.level1.modem.originate = 0;
    l43msg.data.enable_protocol.level2.l2_mode = PRl2modDISABLED;
    l43msg.data.enable_protocol.level2.data_interface.enable = 1;
    l43msg.data.enable_protocol.level3.l3_mode = PRl3modDISABLED;
    printf("Enable channel %d (fd = %x)...", 32 + channel, fd);
    err = NaiControlWrite(fd, &l43msg);
    if (err != 0) {
        printf("start_modem: NaiControlWrite failed with %d\n", err);
        exit(1);
    }

    //
    // Write an ENABLE_PROTOCOL for the listening side...
    //
    memset(&l43msg, 0, sizeof(l43msg));
    l43msg.msgtype = L4L3mENABLE_PROTOCOL;
    l43msg.lapdid = 32 + channel + limit;
    l43msg.data.enable_protocol.level1.l1_mode = PRl1modMODEM;
    l43msg.data.enable_protocol.level1.modem.originate = 0;
    l43msg.data.enable_protocol.level2.l2_mode = PRl2modDISABLED;
    l43msg.data.enable_protocol.level2.data_interface.enable = 0;
    l43msg.data.enable_protocol.level3.l3_mode = PRl3modDISABLED;
    printf("Enable channel %d (fd = %x)...", 32 + channel + limit, fd);
    err = NaiControlWrite(fd, &l43msg);
    if (err != 0) {
        printf("start_modem: NaiControlWrite failed with %d\n", err);
        exit(1);
    }
}

```

download_modem_image

The `download_modem_image()` thread is created when a request to download the modems (`L3L4mREQ_DOWNLOAD`) is received by `process_l34()`. This thread will open and read into memory the entire requested modem image file (i.e., `i990803.pkg` or `T211p38.s37`). The modem image is loaded into the card in 384 byte chunks. The thread accomplishes this by calling `NaiControlWrite()` to send `L4L3mDOWNLOAD` messages to the card. The thread will terminate when the entire modem image file has been sent to the adapter. The following is the source code for `download_modem_image()`.

```

void download_modem_image( void * ptr )
/*+++

```

Routine Description:

This thread is created when a request to download the modems (L3L4mREQ_DOWNLOAD) is received by process_l34(). This thread will open and read into memory the entire requested modem image file (i.e., i990803.pkg or T211p38.s37). The modem image is loaded into the card in 384 byte chunks. The thread accomplishes this by calling NaiControlWrite() to send L4L3mDOWNLOAD messages to the card. The thread will terminate when the entire modem image file has been sent to the adapter.

Arguments:

ptr - unused...

Return Value:

None.

```
--*/
{
    L4_to_L3_struct l43;
    FILE * fd;
    int seqnum, numRead, idx, err;
    char * bigBuffer;
    int bigBufferSize = 1048 * 1024;

    printf("\n request for download:  %s\n", modem_filename);
    fflush(stdout);
    bigBuffer = (char *) malloc(bigBufferSize);
    if (bigBuffer == 0) {
        printf("unable to malloc %d bytes\n", bigBufferSize);
        exit(-1);
    }

    if ((fd = fopen( modem_filename, "r")) == NULL) {
        printf("Unable to open modem download file\n");
        fflush(stdout);
        exit(-2);
    }

    numRead = fread( bigBuffer, 1, bigBufferSize, fd);
    if ((numRead == -1) || (numRead == bigBufferSize)) {
        printf("buffer read failure\n");
        fflush(stdout);
        exit(-3);
    }
    printf("%d bytes read\n", numRead);
    fflush(stdout);

    memset( &l43, 0, sizeof(L4_to_L3_struct));
    l43.msgtype = L4L3mDOWNLOAD;
    l43.data.download_data.module_id = modem_module_id;
    l43.data.download_data.total_length = numRead;

    seqnum = 0;
```

```

    idx = 0;
    while (numRead) {
        memcpy( &l43.data.download_data.buffer[0], &bigBuffer[idx], 384);
        l43.data.download_data.length = 384;
        l43.data.download_data.seq_num = seqnum++;
        if (numRead < 384) {
            l43.data.download_data.length = numRead;
            numRead = 0;
        }
        else {
            numRead -= 384;
            idx += 384;
        }
        if (numRead == 0) {
            l43.data.download_data.last = 1;
            seqnum--;
            modem_last_seqnum = seqnum;
            printf("\n%d download msgs sent\n", seqnum);
        }
        err = NaiControlWrite(Brd.fd, &l43);
        if (err != 0) {
            printf("set_tsi: NaiControlWrite failed with %d\n", err);
            exit( -4);
        }
        else {
            // printf(".");// Debug printf
        }
        SleepEx(1, TRUE); // give process_l34 time to run
    }

    free( bigBuffer);
    return;
}

```

Synchronous (or pended) mode

With the card in Synchronous (or pended) mode, the driver will spin a thread for each channel. This enables an explicit separation of data and control. Also, it makes handling N channels easier, because each task represents a unique channel. The sample application will then to write a packet and read it back to itself.

```

for (;;) {
    //
    // Write a block
    //
    length = 0x100;
    memset(buffer, 0xaa, length);
    NaiDataWrite(fd, buffer, &length);

    //
    // Read it back
    //
    length = 1024;
}

```

```

        NaiDataRead(fd, buffer, &length);
    }

```

Asynchronous mode

Note: The test application can be built to use Asynchronous data transfer by setting `USE_CALLBACKS` to true.

Alternatively, the application can use Asynchronous routines to pass data. The application starts by writing a packet. During the write complete callback, it will also read a packet. In return, during a read complete callback, the application will write a packet.

```

void WriteDone(HANDLE fd, void *arg, DWORD error, DWORD bytes_read, HANDLE event)
{
    printf("WriteDone: call read...\n");
    length = 1024;
    NaiDataReadNoPend(fd, buffer, &length, ReadDone, buffer);
}

void ReadDone(HANDLE fd, void *arg, DWORD error, DWORD bytes_read, HANDLE event)
{
    printf("ReadDone: call write...\n");
    length = 1024;
    memset(buffer, 0xaa, length);
    NaiDataWriteNoPend(fd, buffer, &length, WriteDone, buffer);
}

```

Please note that when issuing callbacks, the main routine calls `SleepEx()` with an argument of `TRUE`, because Win32 only executes callbacks when the thread that caused the callbacks is in an “alertable wait” state. `SleepEx()` is an example of this type of callback execution, as well as `WaitForSingleObjectEx()` and `WaitForMultipleObjectsEx()`.

```

//
// Issue the first write to kick things off...
//
length = 1024;
memset(buffer, 0xaa, length);
NaiDataWriteNoPend(fd, buffer, &length, WriteDone, buffer);

//
// Then just sleep...
//
for (;;) {
    SleepEx(1, TRUE);
}

```

Sample Application: VTTY

The Virtual TTY or VTTY sample application is an MFC GUI application, which acts as a diagnostics port. This application reads and writes characters to the *Virtual TTY*. This application only works on PCI and CompactPCI controllers.

Overall

Within the VTTY, two threads are used, one is used to manage input from the user and manage the screen and the other accepts characters from the VTTY on the adapter. For simplicity, only a single window is managed. Multiple cards can be examined, but each requires that the current view be stopped. Pended calls are used where-ever possible to keep the application short and to the point.

Classes

There are three important classes within the Virtual TTY; **CSelAdapter**, **CVttyView** and **CVttyReader**.

The **CSelAdapter** class inventories the active cards by looking for `naidrv*` entries. If there is more than one entry, a dialog choice box will come up. With only one entry, the class will return that name. If zero entries are found it will ask if the driver is started.

The **CVttyView** class manages the on-screen view and reads characters from the user, as well as writes them to the VTTY port. This will also accept characters from the `VttyReader` task and print them. The user input is taken at `CVttyView::OnChar()`. TTY characters are taken at `CVtty::PrintVTTYChars`. The application will strip backspace characters when printing characters.

The **CVttyReader** class runs as a separate thread because it must pend on reads of the VTTY port. The body of this class is in `CVttyReader::Run`. When it gets characters, it calls the `CVttyReader::PrintVTTYChars`. If the application receives an `INVALID_FILE_OPERATION_HANDLE`, it sleeps and tries again. In most cases, this means the user is switching to a different adapter. If the application receives an `ERROR_OPERATION_ABORTED` message, it indicates the main application is shutting down, so this class terminates.

Sample Application: Samp

The sample application `Samp` is an MFC GUI application, which also uses the MFC SDI model. In addition to sharing many attributes to the VTTY application described in the previous section, it also does the following:

- Downloads cards,
- Runs diagnostics,
- Starts a d-channel
- Makes phone calls, and
- Passes data.

Overall

There are three threads that are used within this application:

- One to Manage VTTY characters,
- One to Manage L3L4 SMI messages,
- One to Manage the screen and menus.

During this application, a single window is managed. The application can only access a single card at a time. Pended calls are used where ever possible to keep the application short and to the point.

Classes

Some of the classes of this application are functionally identical to those in the VTTY application, others are purely user interface orientated and have no interaction with the driver or the controller.

Of those classes that remain important to this sample application, the following list includes:

The **CBriConfig** class gathers BRI Dchannel configuration specifics (SPIDS, etc.), where as the **CDchanConfig** class gathers common Dchannel configuration.

The **CBChan** class gather the selection of a Bchannel for call placement. The **CSetHardware** class gathers various L4L3mSET_HARDWARE settings. All these classes manage an associated dialog box that contains settings for a related SMI operation.

The **CSelAdapter**, **CSampView** classes and **CVttyReader** all match their Vtty counterparts.

The **CMainFrame** class manages the menu structure, which in turn enables and disables menu options depending on the current menu choice and driver responses. After a download the application will identify the current card type and issue a device specific L4L3mSET_HARDWARE and L4L3mSET_TSI SMI message.

`CMainFrame::OnDchanEnable()` actually builds and sends

L4L3mENABLE_PROTOCOL messages. Lapidid and contents are determined by the board and configuration give to `CDchanConfig/CBRICConfig`.

`CMainFrame::OnCallsMake()` actually build the L4L3mCALL_REQUEST SMI message.

The **CSMIReader** class runs as a separate thread, as well as parses SMI messages and currently only changes some lights or menu options. This application could also pass L3L4 message events to a call state machine.



Appendix D

Integrating with the Installation

This section will briefly illustrate installation integrating for developers. This appendix is intended for developers who will customize the driver to their own installation.

Files

There are only a few files which the developer needs to be concerned with while running the driver. They are detailed in Table D-1 below.

Table D-1. Installation Files and Directories

File	Directory	Description
i990803.pkg	C:\Program Files\Brooktrout Technology\Windows 2000 IRP Kit\images	Download image for Rockwell modems.
naid.0	C:\Program Files\Brooktrout Technology\Windows 2000 IRP Kit\images	Diagnostics for Brooktrout PCI cards.
naid.1	C:\Program Files\Brooktrout Technology\Windows 2000 IRP Kit\images	Diagnostics for Brooktrout cPCI cards.
naii.0	C:\Program Files\Brooktrout Technology\Windows 2000 IRP Kit\images	IISDN for Brooktrout PCI cards.
naii.1	C:\Program Files\Brooktrout Technology\Windows 2000 IRP Kit\images	IISDN for Brooktrout cPCI cards.
t211p38.s37	C:\Program Files\Brooktrout Technology\Windows 2000 IRP Kit\images	Download image for ADI modems.
naidrv.sys	%WINDIR%\System32\Drivers	Windows 2000 IRP Device Driver binary image.
nailib.dll	%WINDIR%\System32	Brooktrout Library.

nailib.dll is only needed if you are using the Brooktrout provided library in your application. If you are writing directly to Win32, this file will not be necessary.

The image *.S37 (the name will change from revision to revision) is the download image required for downloading the 8 wide V.90 modem mezzanine.

The image i*.pkg (the name will change from revision to revision) is the download image required for downloading the ADI V.90 modems on the high density modem cards.

Registry Parameters

The Windows 2000 IRP driver uses a number of registry keys to control its operation and behavior. The following keys and values are recognized by the driver and Windows 2000 to control the driver.

Also please be aware that these keys and values reside under **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Naidrv**

Table D-2. Registry Keys and Values

Key	Name	Type	Range
Naidrv	ErrorControl	DWORD	1
	Start	DWORD	0 = Boot 1 = System 2 = Automatic (suggested) ^a 3 = Manual (default) 4 = Disabled
	Stop	DWORD	1
	DisplayName	String	Name that appears in Devices control panel (default is "Windows 2000 IRP Driver").
Naidrv\Tuneables	DebugLevel	DWORD	0 - 0xffffffff (default is 0)
	LowWaterMark	DWORD	0 - NumTxBufs (default is 0)
	NumRxBufs	DWORD	over 3 (default is 32)
	NumTxBufs	DWORD	over 3 (default is 32)
	RxBufSize	DWORD	over 64 (default is 1520)
	TxBufSize	DWORD	over 64 (default is 1520)

- a. Automatic is suggested for a full product installation. The development kit is shipped as manual because it is a development environment.

B

- Bellcore standards B-15
- Brooktrout Customer Support A-1
- Brooktrout I/O Request Packet (IRP) Driver
 - Layout 1-2
- bytes_returned 3-11

C

- CBChan C-8
- CBriConfig C-8
- CDchanConfig C-8
- Classes C-7, C-8
 - CBChan C-8
 - CBriConfig C-8
 - CDchanConfig C-8
 - CMainFrame C-8
 - CSelAdapter C-7, C-8
 - CSELVttyView C-7
 - CSetHardware C-8
 - CSMIReader C-8
 - CVttyReader C-8
- CMainFrame C-8
- Command/Response (C/R) bit B-10
- control interface B-7
- control message queues B-5, B-7
- control messages
 - SMI 1-1
- Control Messaging 4-1, 4-4
- Control Path 4-3
- CSampView C-8
- CSelAdapter C-7, C-8
- CSetHardware C-8
- CSMIReader C-8
- CVttyReader C-7, C-8
- CVttyView C-7

D

- data interface B-7
- Data Messaging 4-2, 4-10
- D-Channel descriptor 4-25
- Device Management 4-2, 4-18
- DeviceIoControl 3-1, 3-3, 3-7, 3-8, 3-9, 3-10

- diagnostic utilities
 - tracing messages B-8
- DIRECT_IO 1-3

F

- Frame Check Sequence (FCS) B-12
- Functional Library 4-2

G

- GetLastError 3-3, 3-8, 3-9, 3-10
- Give/Take indexes B-5, B-7

I

- I Frame header B-11
- i990803.pkg D-1
- Information Elements (IEs) B-11, B-13, B-14
- Installation and Setup of the Windows 2000 IRP Driver 2-1
- Installation Overview 2-1
- Instant ISDN Software
 - version string B-6
- Interrupt Status Block (ISB) B-5, B-8
- INVALID_FILE_OPERATION_HANDLE C-7
- IOCTL_EXTRACTION_AUTHORIZED 3-2
- IOCTL_HOT_SWAP_EXTRACT_ADAPTER 3-2
- IOCTL_HOT_SWAP_INSERT_ADAPTER 3-2
- IOCTL_HOT_SWAP_STATE 3-2
- IOCTL_NAI_CAS_READ 3-2
- IOCTL_NAI_CAS_WRITE 3-2
- IOCTL_NAI_CHECK_READ 3-2
- IOCTL_NAI_CLEAR_VTTY 3-1, 3-9
- IOCTL_NAI_CTL_READ 3-1, 3-12
- IOCTL_NAI_CTL_WRITE 3-1, 3-13
- IOCTL_NAI_DEBUG 3-1, 3-15
- IOCTL_NAI_DEVICE_TYPE 3-2
- IOCTL_NAI_DOWNLOAD 3-1, 3-3, 3-4
- IOCTL_NAI_GET_VERSION_IISDN 3-1
- IOCTL_NAI_LED_CONTROL 3-2
- IOCTL_NAI_READ_ISB 3-1, 3-14
- IOCTL_NAI_RESET 3-1, 3-3
- IOCTL_NAI_RUN_TEST 3-2, 3-21
- IOCTL_NAI_RX_HALT 3-24

IOCTL_NAI_RX_RESUME 3-24
 IOCTL_NAI_SEND_SRECORD 3-1, 3-3, 3-7
 IOCTL_NAI_SERVICE_EVENT 3-2
 IOCTL_NAI_SET_DCHAN 3-2, 3-20
 IOCTL_NAI_SET_LOW_WATER 3-2, 3-24
 IOCTL_NAI_SET_VTTY 3-1, 3-8
 IOCTL_NAI_TX_HALT 3-2, 3-22
 IOCTL_NAI_TX_RESUME 3-2, 3-23
 IOCTL_NAI_VTTY_READ 3-1, 3-10
 IOCTL_NAI_VTTY_WRITE 3-1, 3-11
 IRP Driver Library 1-2
 ISDN PRI signaling B-8

L

L4L3mCALL_REQUEST C-8
 L4L3mENABLE_PROTOCOL 3-20, C-8
 L4L3mSET_HARDWARE C-8
 L4L3mSET_TSI C-8
 LAP-B message tracing B-8
 LowWaterMark 3-24

M

message queues B-5
 Multi-Vendor Integration Protocol (MVIP) bus
 bus mappings B-10

N

NaiAcquireVTTY 4-8
 NaiClearVTTY 4-1
 NaiCloseAdapter 4-2, 4-20
 NaiCloseChannel 4-2, 4-26
 NaiControlRead 4-1, 4-4
 NaiControlWrite 4-1, 4-5
 naid.0 D-1
 naid.1 D-1
 NaiDataRead 4-2, 4-10
 NaiDataReadNoPend 4-2
 NaiDataWrite 4-2, 4-11
 NaiDataWriteNoPend 4-2
 NaiDownloadAdapter 4-2, 4-22
 Naidrv D-2
 NAIDRV.SYS 1-2
 naidrv.sys D-1
 NaiGetDeviceType 4-2
 NaiGetVersionIISDN 4-2
 naii.0 D-1
 naii.1 D-1
 NaiLedControl 4-2
 NAILIB.DLL 1-2
 nailib.dll D-1
 NaiOpenAdapter 4-2, 4-18

NaiOpenChannel 4-2, 4-25
 NaiReadCrashData 4-2
 NaiReadLogoutBlock 4-24
 NaiReadWillPend 4-2
 NaiReleaseVTTY 4-9
 NaiResetAdapter 4-2, 4-21
 NaiRunDiags 4-2
 NaiRxHalt 4-16
 NaiSetVTTY 4-1
 NaiTxHalt 4-2, 4-12
 NaiTxResume 4-2, 4-14
 NaiVttyRead 4-1, 4-6
 NaiVttyWrite 4-1, 4-7

O

OnCallsMake C-8
 OnDchanEnable C-8
 Overlapped I/O 4-3

P

parameters 1-2
 board 1-2
 Poll/Final (P/F) bit B-10
 Preinstallation Requirements 2-1

Q

Q.931 message tracing B-8, B-12

R

Registry Editor 1-2
 Registry Keys and Values D-2
 Registry Parameters D-2
 related publications 1-1

S

Samp Sample Application C-7
 Sample Application
 Samp C-7
 VTTY C-6
 Sample Applications
 Test C-1
 Service Access Point Identifier (SAPI) B-10, B-12
 SleepEx C-6
 SMI 1-1, 3-20
 SMI messages B-7
 S-Record 3-7
 system requirements 2-1

T

t211p38.s37 D-1
Terminal Endpoint Identifier (TEI) B-10, B-12
test programs 2-11
Test sample application C-1
tracing
 general B-8
 running a Level 1 trace B-8
 running a Level 2 trace B-10
tunable parameters 2-18

U

uninstall the driver 2-10
Uninstalling the Windows 2000 IRP Driver 2-10

V

version string B-6
VTTY Handling 4-1, 4-6
VTTY Sample Application C-6

W

WaitForMultipleObjectsEx C-6
WaitForSingleObjectEx C-6
Win32 1-3
Windows 2000 1-1
 registry 1-2
Windows 2000 Driver Library 4-1
Windows 32 File I/O 4-3

