Newton Driver
Development Kits

Lantern™ Data Link Layer Tools

Beta Draft 1.0

September 13, 1997

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Figures, Tables, and Listings

About This Book

This book introduces the Lantern Data Link Layer Tools Development Kit, which you can use to add data link layer drivers to the Newton.

How to Use This Book

This book is both a tutorial introduction and a reference guide to the Lantern Data Link Layer Tools Development Kit. You use this book in conjunction with *An Introduction to Newton Driver Development Kits*, which describes the components and functionality common to all Newton driver development kits.

This book contains five chapters:

- Chapter 1, "Introduction," introduces data link layer drivers.
- Chapter 2, "How to Write a Lantern Driver,"provides a tutorial guide to building a data link layer driver.
- Chapter 3, "Data Link Layer Tool Reference,"provides a reference listing for the constants, data types, and functions that you use to develop data link layer drivers for the Newton.
- Chapter 4, "Building a Data Link Layer Driver,"describes the steps that you need to take to build a data link layer driver for the Newton.

Related Books

This book is one in a set of books that describe the Newton driver development kits. Each kit requires that you have a good understanding of the Newton programming environment. The other books that you need to use are:

- *An Introduction to Newton Driver Development Kits*. This book describes how to build Newton drivers, including a description of how p-classes and packages work.
- *Newton Programmer's Guide*. This book is the definitive guide and reference for Newton programming with the NewtonScript language. It explains how to write Newton programs and describes the system software routines that you can use to do so.
- *The NewtonScript Programming Language*. This book describes the NewtonScript programming language.

Conventions Used in This Book

This book uses the following conventions to present various kinds of information.

Special Fonts

This book uses the following special fonts:

- **Boldface**. Key terms and concepts appear in boldface on first use. These terms are also defined in the Glossary.
- Code typeface. Code listings, code snippets, and special identifiers in the text such as predefined system frame names, slot names, function names, method names, symbols, and constants are shown in the Code typeface to distinguish them from regular body text. If you are programming, items that appear in Code typeface should be typed exactly as shown.
- *Italic typeface*. Italic typeface is used in code to indicate replaceable items, such as the names of function parameters, which you must replace with your own names. The names of other books are also shown in italic type, and *rarely*, this style is used for emphasis.

Developer Products and Support

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order product or to request a complimentary copy of the *Apple Developer Catalog* contact

Apple Developer Catalog
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
World Wide Web	http://www.devcatalog.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For Newton-specific information, see the Newton developer World Wide Web page at:

http://devworld.apple.com/dev/newtondev.shtml

Introduction

This chapter provides an introduction to the Lantern Driver Development Kit, which you use to create data link layer drivers for Newton devices.

About Data Link Layer Drivers

The data link layer in communications is the layer that translates data into a format that can be used with a specific hardware device. Newton data link layer devices are typically PC Cards. A Lantern driver interfaces a data link layer device to the Newton Operating System.

You can write a driver for a specific hardware device using the functions and methods defined in this book. NewtonScript and C++ applications written for the Newton can then use your driver to access that hardware device. To implement a data link layer driver, you need to do the following:

- Implement a version of the TLanternDriver p-class, which interfaces your device with the Newton Operating System. The functions and methods in this p-class implement the instantiation, data management, and event handling operations of your driver.

- Add event-handling functions for any driver-specific events that your driver supports. For example, an ethernet card driver needs to handle ethernet-specific or card-specific events.
- Implement a version of protoLanternDriver to configure and/or provide a user interface for your driver and to define device-specific events for protoLanternClients.

What a Data Link Layer Driver Does

A Lantern driver translates an input or output data stream into a format for the device being controlled by the driver. The driver can perform these actions:

- translate an output data packet into a format that the hardware device transmits out from the Newton
- translate input data received by the device into input data packets for use on the Newton

A single driver can perform both actions. A client application requests the services of a driver and then makes calls into the driver to send or receive data.

The Newton Operating System requires a data link layer driver for each hardware device that is used to transmit data over a network. Some examples of hardware devices that require data link layer drivers are:

- wireless Ethernet cards
- wired Ethernet cards
- non-Ethernet, wireless packet devices
- data acquisition PC Card

How Your Driver is Used

When a card device is inserted into the Newton, the system software determines which driver is to be loaded for that card and instantiates the driver. Client applications can then send requests to the driver, and the

driver notifies applications of certain conditions by sending events to the applications.

Application Requests

To use your driver, a client application sends requests to the driver with Remote Procedure Calls (RPC). An RPC is a function call from one task to a subroutine in another task. The driver handles the request and sends a reply back to the client.

The Newton Operating System defines a set of client RPCs for data link layer drivers. You implement a version of each RPC in your driver.

Note

The client application waits for your driver to reply before continuing, which means that you must reply to each request as soon as possible. However, your driver can reply and then subsequently service the request. Beware that any data structure passed from the client application may be disposed after the reply, which means that your driver must save the data in its own data space. u

Driver Events

When certain conditions arise or change, your driver needs to notify its client applications, so that the applications can respond to those conditions. For example, when some failure occurs in your driver, the application may need to notify the user or retry an operation.

The Newton Operating System defines a set of events that it uses to notify Lantern clients of state and condition changes. Your driver sends these events when appropriate. The application provides an event handler for each event.

In addition, your driver can define its own, driver-specific set of events. If you do this, you need to publish information for application developers, informing them of the specifics of these events. The driver needs to provide an event handler for these events and applications can send these events to your driver.

Driver Configuration

You must implement a configuration module for your driver. This module is an instance of protoLanternDriver, which provides several configuration and setup functions.

The configuration module communicates with your driver to determine certain characteristics, and informs the system of those characteristics.

Data Link Layer Driver Packet Types

NIE 2.0 and AppleTalk have been adapted to use the Newton Operating System data link layer software to support ethernet devices. Any ethernet data link layer driver installed on a Newton can be used by AppleTalk and NIE 2.0.

In addition to supporting NIE 2.0 and AppleTalk, you driver can be used by any NewtonScript application (in the form of a protoLanternClient) to send and receive specific kinds of input and output packets. It is your responsiblity to publish the details required to use your driver, such as the driver-specific events you may want to provide as an API.

Newton Data Link Layer Architecture

The Newton data link layer architecture is designed to support devices without having to know any details about the hardware or the clients who use the hardware. Figure 1-1 shows an overview of the architecture, and Figure 1-2 provides a more detailed picture of the Lantern task architecture.

Figure 1-1 Overview of the data link layer architecture

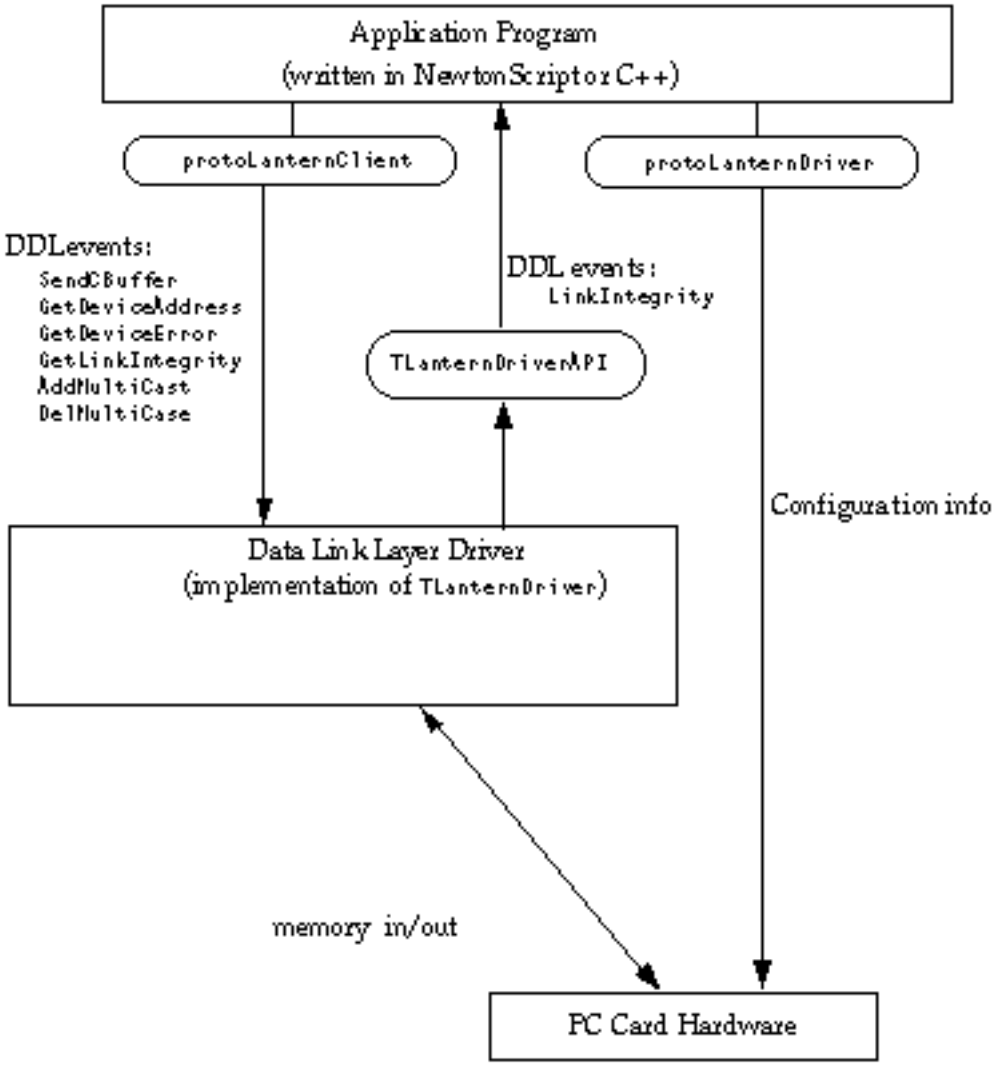
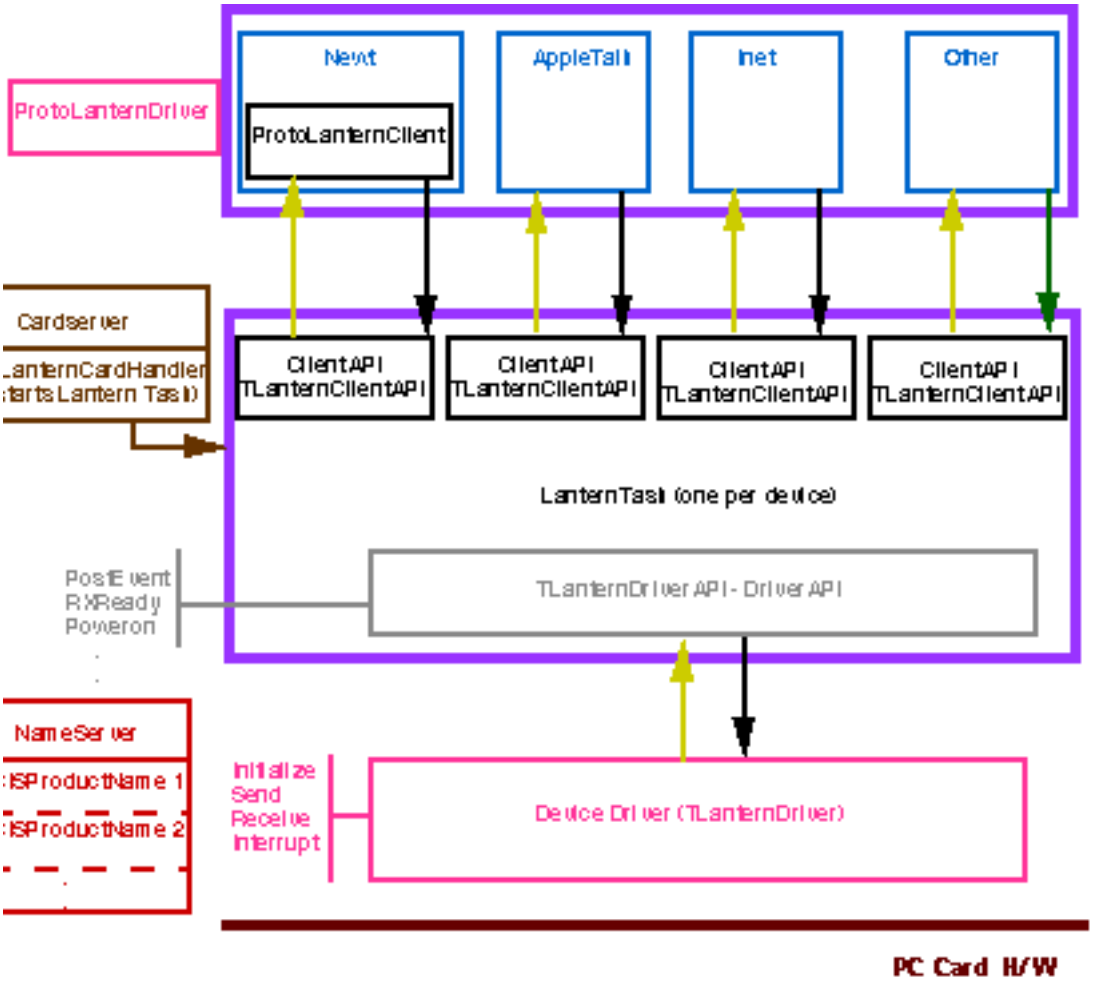


Figure 1-2 shows a more detailed version of the Lantern architecture.

Figure 1-2 Detailed Lantern task architecture



About Newton Tasks

The Newton operating system is multitasking, with Lantern as a task controlled by the operating system. Each data link layer driver runs within the Lantern task.

Lantern receives requests from and sends requests (or replies) to other tasks. Lantern makes sure that each driver request is sent to the appropriate driver. The data link layer drivers use the Remote Procedure Call (RPC) mechanism to create an interface with other software components of the system. RPC is implemented as a request–reply mechanism: one task sends a RPC request to another task, which sends a RPC reply back to the requester after it completes the request.

Events and event handlers work with the RPC mechanism. Client applications can communicate with your driver by using remote procedure calls and by calling methods in the driver p-class.

This section provides a brief description of Newton tasks and provides an overview of how communications tools fit into the system software architecture.

About Newton Tasks and Ports

Each driver is running in a separate Newton task. In the Newton system software, each task

- has its own heap
- has its own machine state, which is preserved across activations of the task
- is scheduled according to a priority value
- can be preempted
- has a port

The task uses its port to receive RPC requests from other tasks, and to send replies and requests to other tasks. Each data link driver task allocates a port for itself when it is started.

When Your Driver is Called

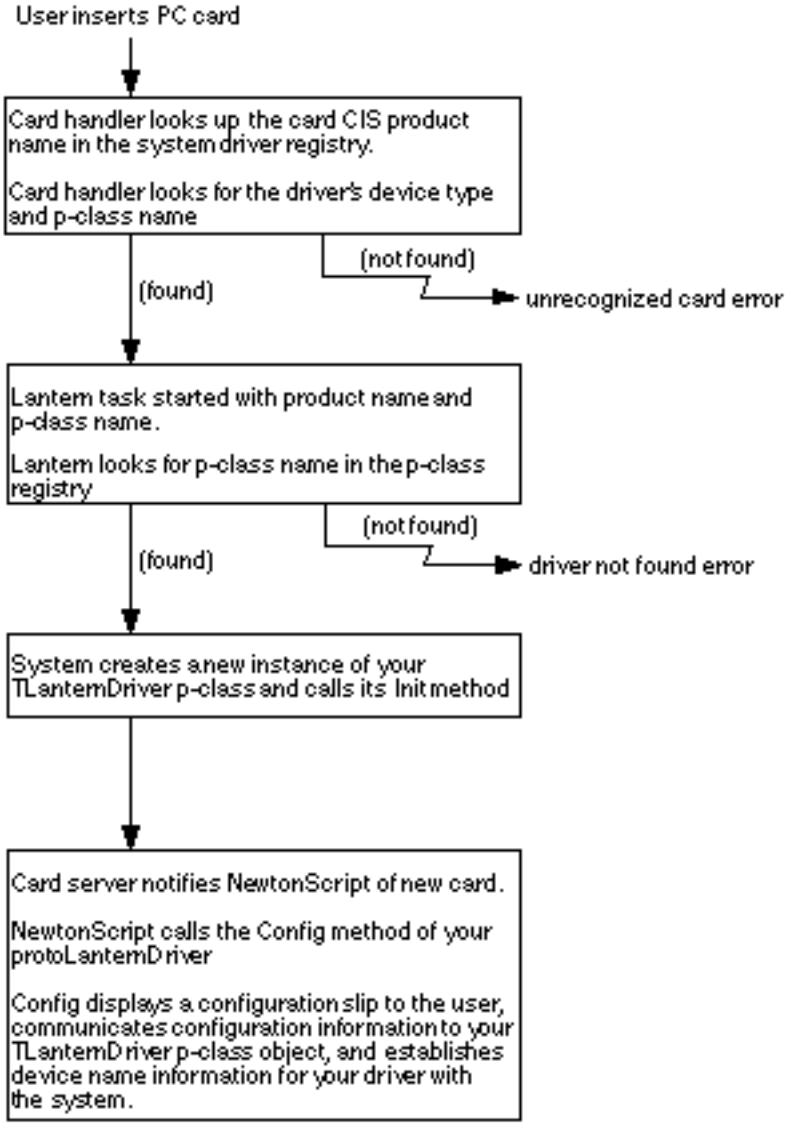
Your data link layer driver is called by the Newton OS when

- the PC card is inserted into the Newton device
- an application client attaches your driver
- an application client sends communications requests to your driver
- an application client finishes with your driver
- the PC card is removed from the Newton device

Card Insertion

When the user inserts a PC card with a device that uses your driver, the Newton OS first checks the manufacturer and product strings in the CIS on the card, instantiates the appropriate driver, and then calls methods in your protoLanternDriver to communicate with the driver, configure it, and identify it to the system. Figure 1-3 shows the flow of control when the PC card is inserted.

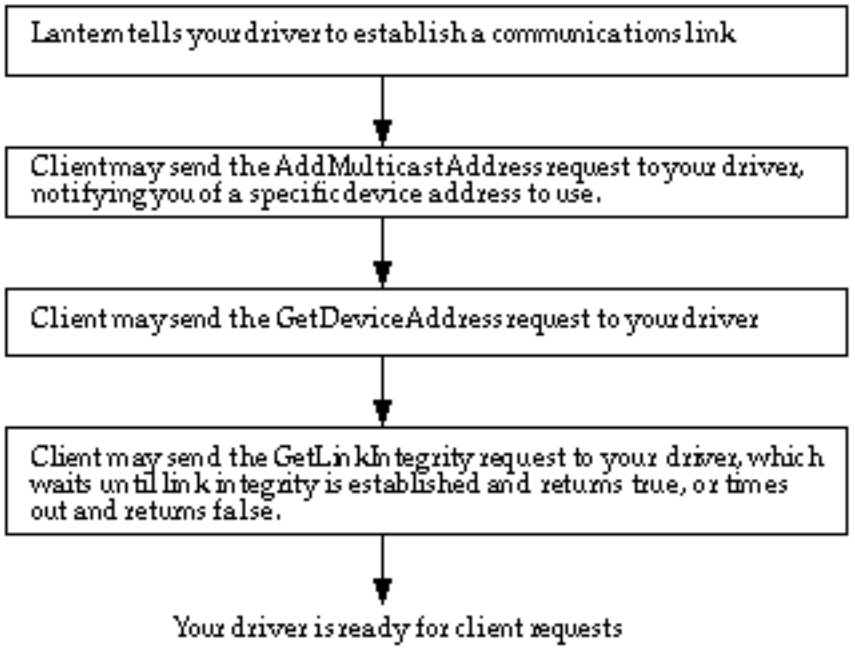
Figure 1-3 Flow of control when a PC card is inserted



Application Attaches Your Driver

When an application client attaches your driver, the Newton OS sends the attach request to your driver. At this time, the client and the operating system send several requests to your driver to get a link established and ready your driver for client use.

Figure 1-4 Flow of control for attaching an Appletalk ethernet device driver



Note that you cannot rely upon the ordering of the calls shown in Figure 1-4, because some client applications may use a different order.

Client Requests

Once the link is established and your driver is ready, the client application sends requests to your driver to send and receive data.

Card Removal or Application Completion

When the client application is done with your driver, or when the card is removed, the client sends your driver the DeleteMulticastAddress request, and then the Newton OS tells your driver to close the communications link. When the card is removed, your driver's p-class instance is deleted.

Implementing Your Own Driver

To implement your driver, you need to implement one p-class and one prototype:

- Implement a protoLanternDriver that allows NewtonScript and C++ clients to access your driver. This proto allows Newton applications to use your driver like an endpoint.
- Implement a TLanternDriver p-class to handle RPCs from clients of your driver and manage the internal state of the driver.

Chapter 2, "How to Write a Lantern Driver" provides a description and sample code for implementing your driver p-class and prototype.

How to Write a Lantern Driver

This chapter provides a detailed description of how to write a Lantern driver, using a simple driver template to guide you through the process.

Data Link Layer Model Driver Code

The model driver code in this chapter is a model Lantern driver for an Ethernet card. You can use as a shell for implementing your own driver. The source files for this code are found on the CD-ROM that accompanies this book.

Chapter 3, "Data Link Layer Tool Reference," contains complete descriptions of the protocols, classes, method, functions, and constants that you use to implement your driver.

This chapter begins with the declaration of the model driver p-class, and then describes the model driver implementation in the following sections:

- driver event definitions
- p-class creation and destruction methods

- Lantern task service methods
- Lantern client service methods

This chapter concludes with a list of driver implementation notes and limitations that you need to consider.

Data Link Layer Model Driver P-class

This section shows the declaration of the TTemplateDriver p-class. This is the p-class for the model Ethernet data link layer driver described in this chapter.

```
#define kPacketAddrSize             (6)
#define kTimerTime                  (100)           // Milliseconds

PROTOCOL TTemplateDriver : public TLanternDriver// Protocol class
    PROTOCOLVERSION(1.0)                            // Protocol class version
{
    public:

        //      Protocol Class Interface

                // DO NOT change the following line (Required)
        PROTOCOL_IMPL_HEADER_MACRO(TTemplateDriver);

        TLanternDriver*         New(void);
        void                    Delete();

            //      Task services

                                    // driver initialization
        NewtonErr               Init(void);
                                    // start or resume card operations
        NewtonErr               Enable();
                                    // stop or suspend card operations
        NewtonErr               Disable();
                                    // handle a card interrupt
        void                    InterruptHandler()


    private:
            //  Client Services(from event handlers of TLanterDriverAPI)
```

```
                                        // transmit a buffer
        void                    SendBuffer(Ptr thePacket, Size size);
                                        // transmit a CBufferList
        void                    SendCBufferList(CBufferList* thePacket);
                                        // return the device address
        void                    GetDeviceAddress(UByte* addr, Size size);
                                        // add a multicast address
        void                    AddMulticastAddress(UByte* addr);
                                        // delete a multicast address
        void                    DelMulticastAddress(UByte* addr);
                                        // get the link integrity flag
        void                    GetLinkIntegrity(void);


            // Optional services
                                        // Set to receive in promiscuous mode
        void                    SetPromiscuous(ULong promiscuous);
                                        // Get a throughput statistic
        void                    GetThroughput(void);


            // Private Services for this driver
                                        // Called when driver timer expires
        void                    TimerExpired(void);

            // Internal variables
                                        // internal status
        ULong       fStatus;
                                        // Ethernet address
        UChar       fEthernetAddr[kPacketAddrSize];
                                        // link integrity status
        Boolean     fLinkIntegrity;
                                        // is a timer event posted?
        Boolean     fTimerPosted;
                                        // the time for the next timer event
        TTime*      fTickTime;

};

            // Internal status definitions
enum kStatus
{
                                        // no status yet
    kStatusNoStatus             =   0x00000000,
                                        // card power is turned on
    kStatusCardPowerOn          =   0x00000001,
```

```
};
```

Driver event definitions

This section shows the model driver code for defining an event (a timer event) for driver use.

```
#define kTemplateTimerEvent          'tmpl','timr',0                    // Timer event
```

P-class Creation and Destruction Methods

This section describes the TLanternDriver protocol methods that you need to provide in your driver to manage creation and destruction of the p-class. The protocol for the model driver is named TTemplateDriver.

```
PROTOCOL_IMPL_SOURCE_MACRO(TTemplateDriver)          // DON'T CHANGE THIS!
```

New

The New method of your p-class needs to allocate and initialize any driver variables that you use.

This method is called when the user inserts the card into the Newton device. Note that the fCardSocket, fCardPCMCIA, and fDriverAPI variables have not yet been set at the time of this call. They are available in the Init method.

The model driver implementation of New creates a timer event record.

```
TLanternDriver* TTemplateDriver::New()
{
      fTickTime = new TTime;


      return this;                                          // Always return this
}
```

WARNING

The Newton p-class handler does not automatically call imbedded class constructors. If you define an imbedded class in your p-class, you must explicitly call the constructor for that class. The code segment below shows an example. u

```
PROTOCOL TTemplateDriver: ...
{
        ...
        TImbedClass fImbed;
        ...
}

...

TTemplateDriver::New()
{
        ...
        new(&fImbed) TImbedClass;
        ...
}
```

Delete

Your p-class destructor method, Delete, is called when the card is removed from the Newton device. Your implementation needs to perform any needed clean-up operations, and free any memory that you have allocated.

```
void TTemplateDriver::Delete()
{
        delete fTickTime;
        fTickTime = nil;
}
```

Lantern Task Service Methods

This section describes the methods that you need to provide in your driver p-class to implement task service level methods.

Init

Your driver's Init method is called when the card has been inserted into the Newton device and a client has requested the services of your driver. You use this method to initialize your driver and set up the card.

The model driver implementation of Init registers the appropriate method for each Lantern event request.

```
NewtonErr TTemplateDriver::Init()
```

```
{
    NewtonErr err = noErr;

    //••• add  code here to check if the card is working correctly


        //      add Lantern request functions
    fDriverAPI->AddEventHandler(kLanternSendBuffer,
                        (DriverProcPtr) &TTemplateDriver::SendBuffer);
    fDriverAPI->AddEventHandler(kLanternSendCBufferList,
                    (DriverProcPtr) &TTemplateDriver::SendCBufferList);
    fDriverAPI->AddEventHandler(kLanternGetDeviceAddress,
                    (DriverProcPtr) &TTemplateDriver::GetDeviceAddress);
    fDriverAPI->AddEventHandler(kLanternAddMulticastAddress,
                    (DriverProcPtr) &TTemplateDriver::AddMulticastAddress);
    fDriverAPI->AddEventHandler(kLanternDelMulticastAddress,
                    (DriverProcPtr) &TTemplateDriver::DelMulticastAddress);
    fDriverAPI->AddEventHandler(kLanternGetLinkIntegrity,
                    (DriverProcPtr) &TTemplateDriver::GetLinkIntegrity);

        // add Optional handlers
    fDriverAPI->AddEventHandler(kLanternSetPromiscuous,
                    (DriverProcPtr) &TTemplateDriver::SetPromiscuous);
    fDriverAPI->AddEventHandler(kLanternGetThroughput,
                    (DriverProcPtr) &TTemplateDriver::GetThroughput);

        // add private timer services handler
    fDriverAPI->AddEventHandler(kTemplateTimerEvent,
                    (DriverProcPtr) &TTemplateDriver::TimerExpired);
    fTimerPosted = false;

    return err;
}
```

Enable

You Enable method is called whenever a client requires the services of your card. This method is called for the first client that uses your driver's services; it is not called when subsequent clients ask for the services of your driver.

The model driver implementation of Enable turns on power to the card and posts a timer event to itself (the driver) to get its timer handling going.

```
NewtonErr TTemplateDriver::Enable()
{
    NewtonErr err = noErr;
```

```
    Boolean powerOn;

        // turn on card power
    fDriverAPI->PowerOn(&powerOn);          // turn on card power
    fStatus |= kStatusCardPowerOn;          // and remember that it is on

        //••• Add your code here

    if (!fTimerPosted)
    {
            // this posts the event to our own task so we get the event
        fTimerPosted = true;
        *fTickTime = GetGlobalTime() + TTime(kTimerTime, kMilliseconds);
        fDriverAPI->PostLocalEvent(fTickTime, kTemplateTimerEvent);
    }
    return err;
}
```

WARNING

You must not access card memory until after your driver is enabled. If you attempt to do so, the entire Lantern subsystem may halt. u

Disable

Your Disable method is called when the Newton OS determines that there are no clients requiring the services of your driver or that the card is no longer available. Your implementation needs to terminate any receive or transmit operations.

The model driver implementation of Disable turns off power to the card.

```
NewtonErr TTemplateDriver::Disable()
{
    NewtonErr err = noErr;

    fDriverAPI->PowerOff();             // turn off card power and remember it
    fStatus &= ~kStatusCardPowerOn;

        //••• Add your code here
    return err;
}
```

TimerExpired

The model driver creates a timer event type to allow it to periodically perform operations. The driver receives the timer event, performs some actions, and then sends the event to itself. You can use this code as a model for implementing timer handling in your driver.

You can implement any number of timers in your driver: add a new event type and a handler for that event for each timer you need. To define a timer, follow these steps:

1. Define the event type, as shown in "Driver event definitions" (page 2-4).
2. Add an event handler method for that event type, as shown in the model driver's implementation of the Init method (page 2-5).
3. Call PostLocalEvent with a delay to send the event.
4. Implement the method to handle your timer event.

It's a good idea to check link integrity in your timer event handler. Note that the model driver's timer event handler does not do this because link integrity cannot be determined on the card that it is driving.

```
void TTemplateDriver::TimerExpired(void)
{
    if (fStatus & kStatusCardPowerOn)
    {
    //••• add your code here

    // grab link integrity and post it here:
        // Boolean linkIntegrity = GetLinkIntegrity();
        // if (linkIntegrity != fLinkIntegrity)
        //     fDriverAPI->PostEvent(kLanternLinkIntegrity,
                                    (ULong) (fLinkIntegrity = linkIntegrity));

            // post the event to our own task so we get the event
        fTimerPosted = true;
        *fTickTime = *fTickTime + TTime(kTimerTime, kMilliseconds);
        fDriverAPI->PostLocalEvent(fTickTime, kTemplateTimerEvent);
    }
    else
    {
        // else timer services are cancelled.
        fTimerPosted = false;
    }
}
```

InterruptHandler

Your InterruptHandler method is called by Lantern from the task level after an interrupt line on the card is asserted. This method is dispatched as a priority event from the Newton OS interrupt handler, which means that it is not a true interrupt handler. This means that it does not have to execute as time-critical or out-of state code. However, the method is dispatched before any other pending RPCs waiting for your driver.

Lantern makes an assumption that a device can buffer inbound data for at least the amount of time required for the interrupt event to be dispatched (which can lead to some loss of packets). The current minimum time between a card interrupt and the execution of your InterruptHandler method is approximately 50 mSecs. This timing is subject to change in future releases.

```
void TTemplateDriver::InterruptHandler()
{
      NewtonErr err = noErr;
      //••• Add your code here

      fDriverAPI->InterruptDone();                    // optional and not really needed
}
```

Lantern Client Service Methods

This section describes the methods that you need to provide in your driver p-class to implement client service level methods.

SendBuffer

Your SendBuffer method is called when Lantern needs your driver to send data. You need to send the data asynchronously, and you must buffer the data.

```
void TTemplateDriver::SendBuffer(Ptr thePacket, Size packetSize)
{
      NewtonErr err = noErr;

            //••• Add your code here

      fDriverAPI->PostReply(err);
}
```

SendCBufferList

Your SendCBufferList method is called when Lantern needs your driver to send data. You need to send the data asynchronously, and you must buffer the data.

```
void TTemplateDriver::SendCBufferList(CBufferList* thePacket)
{
	NewtonErr err = noErr;

	//••• Add your code here

	fDriverAPI->PostReply(err);
}
```

GetDeviceAddress

Your driver receives the GetDeviceAddress request when a client needs to know the hardware address of your device.

The model driver implementation of this method returns the 6-byte device Ethernet hardware address.

```
void TTemplateDriver::GetDeviceAddress(UByte* addr, Size size)
{
	NewtonErr err = noErr;

		//••• change the following as needed

	if (size <= sizeof(fEthernetAddr)) {
		memcpy((char*)addr, (char*)fEthernetAddr,size);
	}
	else {
		err = eLANTERN_DriverValueRangeError;
	}

	fDriverAPI->PostReply(err);
}
```

AddMulticastAddress

Your driver receives the AddMulticastAddress request when a client wants you to add a multicast address.

```
void TTemplateDriver::AddMulticastAddress(UChar* addr)
```

```
{
    NewtonErr err = noErr;

    //••• Add your code here

    fDriverAPI->PostReply(err);                    // Reply
}
```

DelMulticastAddress

Your driver receives the DelMulticastAddress request when a client wants you to delete a multicast address.

```
void TTemplateDriver::DelMulticastAddress(UChar* addr)
{
    NewtonErr err = noErr;

    //••• Add your code here

    fDriverAPI->PostReply(err);                    // Reply
}
```

GetLinkIntegrity

Your driver receives the GetLinkIntegrity request when a client needs to determine the current status of the link integrity. If you cannot determine this for your device, return true.

```
void TTemplateDriver::GetLinkIntegrity()
{
    NewtonErr err = noErr;
    ULong linkIntegrity = true;

    //••• Add your code here
    // linkIntegrity = GetLinkIntegrity();                    // Get link status
    fDriverAPI->PostReply(err, 1, linkIntegrity);// Reply
}
```

SetPromiscuous

You can use promiscuous mode for an ethernet device driver to receive all packets on a network, regardless of the destination ethernet address. This effectively allows you to snoop the network.

When your driver receives this event, you need to enable the hardware on the card to use promiscuous mode. The model driver implementation does not support this mode and thus replies that the event was not handled.

```
void TTemplateDriver::SetPromiscuous(ULong promiscuous)
{
	NewtonErr err = noErr;
		//••• Change the following if supported

	err = eLANTERN_DriverUnhandledEvent;
	fDriverAPI->PostReply(err);// Reply
}
```

GetThroughput

Some devices maintain an average throughput value for their ethernet hardware. An application can send this event to your driver to retrieve the throughput value(s).

If your hardware does not support this capability, you need to reply to the event with an unhandled event error, as does the model driver.

You can return 0 for any throughput value that your hardware does not support. You can also return separate values for the receive, transmit, and overall throughput values.

The model driver implementation does not support this mode and thus replies that the event was not handled.

```
void TTemplateDriver::GetThroughput(void)
{
	NewtonErr err = noErr;

	//••• Change the following if supported

	err = eLANTERN_DriverUnhandledEvent;
	fDriverAPI->PostReply(err);// Reply
}
```

Data Link Layer Driver Notes and Limitations

This section provides a collection of implementation notes for data link layer drivers.

Busy Loops

- You should never use busy wait loops when performing input/output operations in your driver. Only use busy wait loops if the device registers need a read/write delay to allow for hardware latency.
- You can use short (100ms or less) busy wait loops for device state or control operations. Operations such as these are typically called synchronously from the client.

Packet Delivery

- The Newton system software considers a packet successfully delivered as soon as the request is received. If your driver fails to successfully deliver a packet, the packet must be discarded. The client is expected to implement protocols to deal with this situation.
- The Newton system software queues input packets strictly to provide efficient support of multiple clients. Packets are not queued indefinitely: they may be dropped arbitrarily before all clients have read them.
- If packet delivery failure is occurring due to a poor communications link, you can raise the LinkIntegrity event to signal the operating system that the link is inoperable. The client can then choose to continue, abort the link, or attempt to resolve.
- You must use asynchronous input/output when your driver implements any form of handshake with a remote device for delivery of a packet.
- Lantern discards newer packets while waiting for the oldest packets to be delivered.

Link Integrity Changes

- Your driver should not change state or behavior due to a change in link integrity. Input/output requests may still arrive (and should be dropped) while the link integrity is poor.

Exception Handling in Your Driver

- The Newton Operating System encloses all driver calls within an exception-handling block, which means that you do not need to perform your own exception handling in your implementation of the driver methods.

Multicast Clients

If your driver has two clients and one subscribes to a multicast group, the other client also receives those packets. This is a current limitation in the Lantern architecture.

Data Link Layer Tool Reference

This chapter describes the constants, data types, and p-classes that you use to implement a communications tool for the Newton.

Data Link Layer Tool Constants and Data Types

This section describes the constants and data types that you use with your communications tool.

Data Link Layer Error Codes

This section lists the error codes generated by the Data Link Layer of the Newton Operating System.

eLANTERN_DriverNotFound	= -61001
eLANTERN_DriverInstallFailed	= -61002
eLANTERN_DriverRemoveFailed	= -61003
eLANTERN_DriverUnhandledEvent	= -61004
eLANTERN_DriverPacketDropped	= -61005

```
eLANTERN_DriverException                = -61006
eLANTERN_DriverNewAsyncFailed           = -61007
eLANTERN_DriverCardNotInserted          = -61008
eLANTERN_DriverAlreadyReplied           = -61009
eLANTERN_DriverRequestNotRepliedTo      = -61010

eLANTERN_DriverValueRangeError          = -61200
eLANTERN_DriverHardwareFailure          = -61201
eLANTERN_DriverResourceFailure          = -61202

eLANTERN_ClientDispatchFailed           = -61300
eLANTERN_ClientAlreadyBound             = -61301
eLANTERN_ClientNotBound                 = -61302
eLANTERN_ClientNoMemory                 = -61303
eLANTERN_ClientNewAsyncMsgFailed        = -61304
eLANTERN_ClientInvalidTaskType          = -61305
```

Constant descriptions

eLANTERN_DriverNotFound

The driver could not be loaded.

eLANTERN_DriverInstallFailed

The driver installation failed.

eLANTERN_DriverRemoveFailed

The driver could not be removed.

eLANTERN_DriverUnhandledEvent

The driver did not support the specified request.

eLANTERN_DriverPacketDropped

The requested packet is no longer in the cache.

eLANTERN_DriverException

The driver caused an exception to be thrown. Note that this is usually the evt.ex.abt.perm exception.

eLANTERN_DriverNewAsyncFailed

An internal resource is depleted.

eLANTERN_DriverCardNotInserted

The card handler is not available because RemoveServices was called.

eLANTERN_DriverAlreadyReplied

The driver replied to an RPC twice.

eLANTERN_DriverRequestNotRepliedTo
: The driver did not reply to an RPC.

eLANTERN_DriverValueRangeError
: The supplied data values are out of range for the device.

eLANTERN_DriverHardwareFailure
: An unrecoverable hardware fault was detected.

eLANTERN_DriverResourceFailure
: The driver could not perform the operation due to lack of system resources.

eLANTERN_ClientDispatchFailed
: Dispatching events is not supported by this task.

eLANTERN_ClientAlreadyBound
: Attempt to bind a client that is already bound.

eLANTERN_ClientNotBound
: The operation requires the client to be bound.

eLANTERN_ClientNoMemory
: The client is out of memory.

eLANTERN_ClientNewAsyncMsgFailed
: An asynchronous request failed because an internal resource could not be accessed.

eLANTERN_ClientInvalidTaskType
: The requested operation is not supported by this task type.

Data Link Layer Driver RPCs and Events

This section describes the remote procedure calls (RPCs) and events with which your data link layer driver needs to work. Table 3-1 summarizes the events.

Table 3-1 Data link layer driver events

RPC or event name	Description
kLanternSendBuffer	Sent to notify your driver that data needs to be sent.
kLanternSendCBufferList	Sent to notify your driver that data needs to be sent.
KLanternGetDeviceAddress	Sent to notify your driver to return the hardware device address.
kLanternGetLinkIntegrity	Sent to your driver to determine the status of the link.
kLanternAddMulticastAddress	Sent to your driver to add a specified hardware ethernet address as a multicast address.
kLanternDelMulticastAddress	Sent to your driver to notify you to delete the specified address as a multicast address.
kLanternLinkIntegrity	Your driver sends this event to all of its clients to notify them that the status of the link has changed.
kLanternDriverFailure	Your driver sends this event to all of its clients to notify them of a driver error condition.

RPCs Sent to Your Driver

This section describes the remote procedure calls that are sent to your driver by the Newton Operating System or by your client programs.

kLanternSendBuffer

```
kLanternSendBuffer kLanternEventClass,'sndB',2
                                        // (Ptr ptr, Size size) : {}
```

The Newton Operating System sends the kLanternSendBuffer request when your driver needs to send data in the form of a pointer to a buffer.

Event Send Arguments

ptr	A pointer to the data to be sent. The size of the buffer pointed to by *ptr* must be equal to or greater than the minimum packet size.
size	The number of bytes of data to be sent.

Event Return Arguments

None.

kLanternSendCBufferList

```
kLanternSendCBufferList kLanternEventClass,'sndC',1
                                        // (CBufferList* data) : {}
```

The Newton Operating System sends the kLanternSendCBufferList request when your driver needs to send data in the form of a CBufferList.

Event Send Arguments

data	A pointer to a CBufferList that contains the data to be sent.

Event Return Arguments

None.

kLanternGetDeviceAddress

```
kLanternGetDeviceAddress kLanternEventClass,'gdva',2
                                        // (UByte* addr, Size size) : {}
```

The Newton Operating System sends the kLanternGetDeviceAddress request to request that your driver return the current hardware address. For ethernet address, this is a 6-byte ethernet hardware address.

Event Send Arguments

addr	A pointer to the address in which the hardware address should be stored.
size	The number of bytes to store.

Event Return Arguments

None.

kLanternGetLinkIntegrity

```
kLanternGetLinkIntegrity kLanternEventClass,'glit',0
                                                    // () : {ULong link}
```

A client sends the kLanternGetLinkIntegrity request to your driver to determine the current status of the link. You return true if the link is available.

For an ethernet card, you return true if you can detect that the device is connected to a LAN and false if you can detect that the device is not connected. If you are unable to detect, return true.

Event Send Arguments

None.

Event Return Arguments

link	An unsigned long value that specifies the status of the link. Return true if the link is available.

kLanternAddMulticastAddress

```
kLanternAddMulticastAddress kLanternEventClass,'amca',1
                                                    // (UByte* addr) : {}
```

A client sends the kLanternAddMulticastAddress request to notify you to add a specific hardware device address as a multicast address. For example, AppleTalk might need to add an AppleTalk broadcast address.

The driver is responsible for remembering any multicast addresses between calls to the Enable and Disable methods of your driver.

If your driver receives multiple requests to add a specific address as a multicast address, you are responsible for maintaining a count in coordination with kLanternDelMulticastAddress events. This is because two different clients can request addition of the same address.

Event Send Arguments

addr The multicast address to add.

Event Return Arguments

None.

kLanternDelMulticastAddress

```
kLanternDelMulticastAddress kLanternEventClass,'dmca',1
                                                    // (UByte* addr) : {}
```

A client sends the kLanternDelMulticastAddress request to notify you that you should remove the specified multicast address.

Note that your driver is responsible for maintaining a count of requests to use a specified multicast address, which means that you must only delete the address when the count reaches zero.

Event Send Arguments

data A pointer to a CBufferList that contains the data to be sent.

Event Return Arguments

None.

Events Your Driver Sends

This section describes the events that your driver sends to its clients.

kLanternLinkIntegrity

```
kLanternLinkIntegrity kLanternEventClass,'link',1
                                                    // (ULong link)
```

Your driver sends this event to notify clients that the status of the link has changed with regard to media being available or connected. For example, you send this event if the LAN cable is plugged in, or if a wireless connection exists over which packets can be transported.

Event Send Arguments

link The status of the link. Set this to true if the link is connected.

Event Return Arguments

None.

kLanternDriverFailure

```
kLanternDriverFailure kLanternEventClass,'fail',1
                                                    // (NewtonErr reason)
```

Your driver sends this event to notify clients that a driver error has occurred.

Event Send Arguments

err The driver-specific error code.

Event Return Arguments

None.

The Data Link Layer Driver P-Class

This section describes the *TLanternDriver* p-class, which you implement to create a data link layer driver. Listing 3-1 shows the TLanternDriver p-class.

Listing 3-1 The data link layer driver p-class

```
PROTOCOL TLanternDriver : public TProtocol
{
public:
      static TLanternDriver*                    New(char*);
```

```
    void                                    Delete();

    NewtonErr                               Init(void);

    NewtonErr       Enable(void);
    NewtonErr       Disable(void);
    NewtonErr       InterruptHandler(void);

protected:
friend class TLanternTask;
    TCardSocket*                fCardSocket;
    TCardPCMCIA                 fCardPCMCIA;
    TLanternDriverAPI*          fDriverAPI;
}
```

Data Link Layer Driver P-Class Methods

This section describes the methods of the TLanternDriver p-class.

WARNING

If your code attempts to access the card after it has been removed, an exception is thrown and the rest of your code will not execute. To protect against this, you should enclose your code in an exception block. u

New

static TLanternDriver::New(char*);

Is called to construct your p-class object.

return value None.

DISCUSSION

The New method is called to construct your p-class object. This constructor is different than the standard C++ constructor because of the glue code that the Newton Operating System uses to create the interface to this p-class.

IMPORTANT

Your implementation of the New method must return the class instance pointer, as follows:

```
    return this;

u
```

Delete

void TLanternDriver::Delete();

Is called to delete your p-class object.

return value None.

DISCUSSION

Your implementation of the Delete method should free any resources associated with your driver. For example, you can deallocate any memory that has been allocated in your p-class.

Init

NewtonErr TLanternDriver::Init();

Is called when the card being controlled by your driver is inserted into the Newton device or when the device is reset.

return value An error code.

DISCUSSION

Your implementation of the Init method can perform any initialization required to determine that you can properly work with the card. For example, you might test the card's memory in your Init method.

Another common operation of the Init method is to register your event handlers.

Enable

NewtonErr TLanternDriver::Enable();

Is called whenever the services of the card being controlled by your driver are required.

return value An error code.

DISCUSSION

The Enable method is called when a client requests the services of your driver or when the Newton is powered up after a prior power-off sequence. This method is called when the first client requires the services of your driver; subsequent clients do not generate another call to Enable.

Your implementation of the Enable method needs to call the PowerOn method to turn on Vcc to the card.

Note that the clients of an ethernet card can be one of the following:

- AppleTalk
- TCP/IP
- a protoLanternClient client in a NewtonScript application

You must maintain the current state of the device being controlled by your driver at all times. This means that you must remember situations and events such as the following:

- when your driver receives a request before being enabled, you may need to save the request and activate it after enabling. Do this for any requests that you cannot process without being enabled.
- when your driver receives a Disable request, you must save state so that your driver can be reenabled in the same state.
- any state information generated by special events sent to your driver by an instance of protoLanternDriver .

Disable

NewtonErr TLanternDriver::Disable();

Is called when the services provided by the card being controlled by your driver are no longer needed.

return value An error code.

DISCUSSION

The Disable method is called when there are no more clients using the services of your driver or when the card has been removed by the user.

Your implementation of the Disable method needs to turn off Vcc to the card.

InterruptHandler

void TLanternDriver::InterruptHandler();

Is called after an interrupt line on the card being controlled by your driver changes level.

return value None.

DISCUSSION

The InterruptHandler method is called whenever an interrupt line changes level. The Newton Operating System interrupt handler calls this method as a priority event.

Since this is not a true interrupt handler, you do not have to execute this method as time-critical or out-of-state code. However, it is a priority event, which means that your driver receives the InterruptHandler call before receiving any other pending events. The minimum time between a card interrupt and your InterruptHandler method getting called is approximately 50 milliseconds; however, the actual time could be longer.

The Newton Operating System assumes that the device being controlled by your driver is capable of buffering inbound data for at least as long as is required to dispatch an interrupt event. This can lead to a loss of data for high bitrate devices or on noisy networks.

The PowerOn method enables interrupts for the PC card by default. This means that any change in the IRQ level generates an interrupt. Some PC cards do not interrupt in this manner; for example, some cards require a positive edge IRQ change. If necessary for your card, you can call PowerOn and tell it to not enable interrupts; you then enable card interrupts as required for your card. Note that this method of handling interrupts will make your driver incompatible with future multi-function card drivers.

You can optionally call the TLanternDriverAPI::InterruptDone method from within your implementation of InterruptHandler. If you do not call InterruptDone, the system assumes that you are done with the interrupt when your InterruptHandler method returns. The Newton Operating System uses the InterruptDone method to reenable interrupts on the device, if necessary.

Data Link Layer Driver P-Class Fields

This section describes the fields of the TLantern p-class.

fCardSocket

TCardSocket* TLanternDriver::fCardSocket;

Is a pointer to the TCardSocket instance associated with the driver. You can use this to access methods of the TCardSocket class instance for your driver.

fCardPCMCIA

TCardPCMCIA* TLanternDriver::fCardPCMCIA;

Is a pointer to the TCardPCMCIA instance associated with the driver. You can use this to access methods of the TCardPCMCIA class instance for your driver.

fDriverAPI

TCardSocket* TLanternDriver::fDriverAPI;

Is a pointer to the TLanternDriverAPI instance associated with the driver. You can use this to access methods of the TLanternDriverAPI class instance for your card.

The Data Link Layer Driver API P-Class

This section describes the TLanternDriverAPI p-class, which provides the methods that you call from your TLanternDriver implementation. Listing 3-2 shows the TLanternDriverAPI p-class.

Listing 3-2 The data link layer driver API p-class

```
PROTOCOL TLanternDriverAPI : public TProtocol
{
public:
    static TLanternDriverAPI*          New(char*);
    void                               Delete();
```

```
    NewtonErr                       Init(TLanternTask*);

    NewtonErr       PowerOn(Boolean* powerOn, Boolean intEnable=true);
    NewtonErr       PowerOff(ULong msDelay=kDefaultPowerDownTime,
                            Boolean intDisable=true);

    Ptr             NewPacketPtr(Size size);
    NewtonErr       RxReady(Ptr buf, Size size=0);
    NewtonErr       InterruptDone(void);

    NewtonErr       CardHandlerSpecific(ULong selector, ...);

    NewtonErr       AddEventHandler(kLanternEventType, DriverProcPtr);
    NewtonErr       PostEvent(kLanternEventType, ...);
    NewtonErr       PostLocalEvent(kLanternEventType, ...);
    NewtonErr       PostLocalEvent(TTime*, kLanternEventType, ...);
    NewtonErr       PostReply(NewtonErr);
    NewtonErr       PostReply(NewtonErr, ULong nArgs, ...);
}
```

Data Link Layer Driver API P-Class Methods

This section describes the methods of the TLanternAPI p-class. The first three methods of this p-class are called by the Newton Operating system and are not of any use to you as a driver developer. These three methods are not described and must not be called directly by your driver:

- TLanternDriverAPI::New
- TLanternDriverAPI::Delete
- TLanternDriverAPI::Init

PowerOn

```
NewtonErr TLanternDriverAPI::PowerOn(Boolean *powerOn,
                                     Boolean intEnable);
```

Your driver calls PowerOn to power up and initialize the device that it is controlling.

powerOn	Upon return, this is false if the card was already powered up, and true if the card was not already powered up.
intEnable	If true, enables interrupt control of the device. If you set this to false, your driver will not be compatible with a multifunction PC Card.
return value	An error code.

DISCUSSION

The PowerOn method powers on the device being controlled by your driver. You typically call this in your implementation of the TLanternDriver::Enable method.

The PowerOn method performs the following actions:

1. powers on Vcc
2. resets the PCMCIA bus
3. waits for the card-ready line to be asserted
4. sets the bus access for 8-bit access

You can override this behavior in your driver by providing your own power-on function instead of using PowerOn; however, if you do so, you lose compatibility with multifunction PC Cards.

If powerOn is set to false when the method returns, it means that the card was already powered up. This can happen because of delays in powering the card off, or because the device is on a multifunction card on which other devices are powered up. If powerOn returns false, your driver probably does not need to reinitialize the card, which can save time.

If your card needs special processing to enable interrupts, you need to provide that code after the PowerOn method has completed. The PowerOn method enables IREQ on the bus, as shown here:

```
fSocket->EnableSocketInterrupt(kSocketCardIREQInt);
```

PowerOff

NewtonErr TLanternDriverAPI::PowerOff(ULong msDelay,
Boolean intDisable);

Your driver calls PowerOff to power down the device that it is controlling.

msDelay	The number of milleseconds to delay before powering down the device. Set this to the appropriate power-down time. You often want to leave the power on for a certain period of time before shutting down, in case the device's services are quickly requested again. This is useful for cards that have a long power-on time such as wireless LAN cards, which typically take time to acquire a network base station.
intDisable	If true, disables interrupt control of the device and clears out any pending interrupts. If false, your driver will not be compatible with multifunction PC cards.
return value	An error code.

DISCUSSION

The PowerOff method powers off the device being controlled by your driver. Your driver needs to call this method when it no longer needs to use the card; this is typically done in your TLanternDriver::Disable method.

The PowerOff method drops Vcc after the specified delay time and if no other drivers are using the card (for multifunction cards). If you need to override this behavior in your driver, you can provide your own power-off function instead of calling PowerOff; however, if you do so, you lose compatibility with multifunction PC Cards.

NewPacketPtr

Ptr TLanternDriverAPI::NewPacketPtr(Size *size*);

Your driver calls this method when it receives a new packet of data. NewPacketPtr allocates a block of memory for the data.

size	The number of bytes requested for the new packet.
return value	A pointer to the new packet.

DISCUSSION

The NewPacketPtr method allocates memory for packet data, using a memory allocation scheme that is optimized for packet memory requests.

If NewPacketPtr returns nil, your driver must discard the packet that it received from the PC Card and rely on higher level protocols to recover the data.

Note

You can only allocate one packet at a time. If you attempt to call NewPacketPtr a second time before calling RxReady, NewPacketPtr deletes the previous memory buffer and returns a new pointer to you. The Newton Operating System automatically cleans up memory after you call RxReady or when the p-class object is deleted.

RxReady

NewtonErr TLanternDriverAPI::RxReady(Ptr *buf*,
Size *size*);

Your driver calls RxReady after copying the packet data from the card buffer to the packet buffer. This makes the buffer ready for any clients of your driver.

buf	A pointer to the buffer contained the received data.
size	The size, in bytes, of the received data buffer.
return value	An error code.

DISCUSSION

When your driver receives a packet of data, you first call the NewPacketPtr method to allocate a buffer for that data, then copy the data to the buffer, and then call RxReady to make the data available to your clients.

Inbound data packets are selectively queued until all clients have read them, or until a certain amount of time has passed.

The Newton Operating System automatically discards the packet memory when it is no longer needed. This means that your driver should not do anything with *buf* once you have called RxReady.

InterruptDone

NewtonErr TLanternDriverAPI::InterruptDone();

Your driver calls this method to tell the Newton Operating System that you have finished handling an interrupt in your driver's InterruptHandler method.

return value An error code.

DISCUSSION

When you call InterruptDone, the Newton Operating System reenables interrupts for the device (if necessary). If your InterruptHandler method does not call InterruptDone, the Newton Operating System assumes that interrupts can be reenables upon return from your InterruptHandler.

CardHandlerSpecific

NewtonErr TLanternDriverAPI::CardHandlerSpecific(ULong selector, ...);

Your driver can use this method to access a CardSpecific method in the CardHandler class for the PC Card.

selector ???.

return value An error code.

DISCUSSION

AddEventHandler

NewtonErr TLanternDriverAPI::AddEventHandler(
kLanternEventType eventType,
DriverProcPtr ptr);

You call this method to specify a method that is a handler for an event. You must also call this method to handle Newton OS events sent to your driver.

eventType The type of event to be handled.

ptr A pointer to the function that handles the event.

return value An error code.

DISCUSSION

You can extend your driver to handle events that are not predefined by the Newton Operating System for data link layer drivers. If you document these events for your clients, they can send these events to your driver. To handle these events, you need to add a method to your TLanternDriver p-class implementation and register that method with AddEventHandler.

To define an event that your driver handles, you use a statement such as the following:

```
#define kdeviceNameFunctionName kLanternClass, 'XXXX', n
                                        // (call_arguments) :: (reply_arguments)
```

For example:

```
#define kLanternMyRPC kLanternClass, 'myEv', 1
                    // (CBufferList *) :: (Ulong result, MyStruct data;)
```

This example defines an RPC named MyRPC with identifier 'myEv'. This RPC takes one argument (a pointer to a CBufferList) and returns a structured reply. You specify the call and reply parameters in the comment at the end of the define statement.

If you define a method in your driver named MyRPC to handle this event, you would then register MyRPC as an event handler as shown here:

```
fDriverAPI->AddEventHandler(kLanternMyRPC, &myDriver::MyRPC)
```

Note

For more information about events, RPCs, and using these with your driver, see Chapter 1, “Introduction.” u

PostEvent

NewtonErr TLanternDriverAPI::PostEvent(kLanternEventType eventType, ...);

You call this method from your driver to post an event for all clients of your driver.

eventType	The type of event to post.
...	Event-specific data
return value	An error code.

DISCUSSION

You use the PostEvent method to post an event to all clients of your driver. This is an asynchronous call.

PostLocalEvent

```
NewtonErr TLanternDriverAPI::PostLocalEvent(
                                    TTime* evtTime,
                                    kLanternEventType eventType, ...);
NewtonErr TLanternDriverAPI::PostLocalEvent(
                                    kLanternEventType eventType, ...);
```

You call this method from your driver to post an event for your driver to handle. Only use PostLocalEvent to post events to yourself.

eventTime	The time at which to post a deferred event.
eventType	The type of event to post.
...	Event-specific data
return value	An error code.

DISCUSSION

You use the PostLocalEvent method to post an event to all clients that are connected to your driver. You can post a standard driver event or a private event that you have defined. You need to add an event handler (see "AddEventHandler" (page 3-18)) for any private events that you have defined for your driver.

You can use the first form of PostLocalEvent to post the event at a later time.

PostReply

```
NewtonErr TLanternDriverAPI::PostReply(NewtonErr err);
NewtonErr TLanternDriverAPI::PostReply(NewtonErr err,
                                       ULong nArgs,
                                       ...);
```

You call the PostReply method to reply to an event that your driver has handled, if the event requires a reply. Use the first form to reply to an event with no reply arguments.

err	The error code for the reply.
nArgs	The number of unsigned long values that follow in the reply-specific data.
...	Reply-specific data
return value	An error code.

DISCUSSION

You call PostReply when your driver has finished handling an event. Clients receive the reply and can examine and use the reply arguments.

You need to specify the number of unsigned long values that are in the reply in *nArgs*. You can use the macro SizeOfStructAsArgs() to determine the size of any structure that you are passing back as a reply.

Note
Data is copied from the stack back to the client reply buffer. Any pointers in the argument list are passed literally, which means that they may not be of any value to the client. u

TCardSocket Methods for Data Link Drivers

This section describes the methods of the TCardSocket p-class that you use in your driver. You can access these methods using the TLanternDriver:fCardSocket field in your driver's p-class.

This section describes these methods:

- AttributeMemBaseAddr
- CommonMemBaseAddr
- IOBaseAddr
- SelectIOInterface

- SetControl
- GetControl

AttributeMemBaseAddr

ULong TCardSocket::AttributeMemBaseAddr();

Returns the attribute memory base address for the card in the socket.

return value The attribute memory base address.

The attribute memory space for a PC-card is mapped linearly into Newon memory space. There is no 64K addressing window on the Newton, as there is on other platforms.

CommonMemBaseAddr

ULong TCardSocket::CommonMemBaseAddr();

Returns the common memory base address for the card in the socket.

return value The common memory base address.

The common memory space for a PC-card is mapped linearly into Newon memory space. There is no 64K addressing window on the Newton, as there is on other platforms.

IOBaseAddr

ULong TCardSocket::IOBaseAddr();

Returns the I/O base address for the card in the socket.

return value The I/O base address.

The I/O space for a PC-card is mapped linearly into Newon memory space. There is no 64K addressing window on the Newton, as there is on other platforms.

SelectIOInterface

void TCardSocket::SelectIOInterface();

Selects the I/O configuration on the PCMCIA bus.

IOBaseAddr

void TCardSocket::SetControl(ULong control);

Sets the bus control value for the card.

control	The bus control value.
return value	None.

GetControl

ULong TCardSocket::GetControl(ULong control);

Returns the current bus control value on the card.

return value	The bus control value.

TCardPCMCIA Fields and Methods for Data Link Drivers

This section describes the methods and fields of the TCardPCMCIA p-class that you use in your driver. You can access these methods and fields using the TLanternDriver:fCardPCMCIA field in your driver's p-class.

This section describes these fields and methods from the TCardPCMCIA p-class:

- GetCardManufacturer
- GetCardProduct
- GetCardV1String3
- GetCardV1String4
- GetCardConfiguration
- fRegisterBaseAddress
- fNumOfConfigEntry
- fManufactureId
- fManufactureIdIno
- fFunctionId

This section also describes these fields from the TCardConfiguration class, which you can access with the TCardPCMCIA::GetCardConfiguration method.

- fConfigurationNumber
- fIoAddresses

GetCardManufacturer

const char* TCardPCMCIA::GetCardManufacturer() const;

This method returns the string representation of the card manufacturer. This is the first string in the CIS Level 1 Version tuple.

return value A constant string.

GetCardProduct

const char* TCardPCMCIA::GetCardProduct() const;

This method returns the string representation of the card product name. This is the second string in the CIS Level 1 Version tuple.

return value A constant string.

GetCardV1String3

const char* TCardPCMCIA::GetCardV1String3() const;

This method returns the third string in the CIS Level 1 version tuple.

return value A constant string.

GetCardV1String4

const char* TCardPCMCIA::GetCardV1String4() const;

This method returns the fourth string in the CIS Level 1 Version tuple..

return value A constant string.

GetCardConfiguration

TCardConfiguration* TCardPCMCIA::GetCardConfiguration(ULong configNum);

This method returns the specified configuration object.

configNum	The number of the configuration entry that you want to access.
return value	A pointer to the configuration entry object.

fRegisterBaseAddress

ULong TCardPCMCIA::fRegisterBaseAddress;

Is the base address of the configuration registers for the card in attribute memory.

fNumOfConfigEntry

ULong TCardPCMCIA::fNumOfConfigEntry;

Is the number of configuration entries available.

fManufactureId

ULong TCardPCMCIA::fManufactureId;

Is the manufacturer code from the CIS Manufacturer Id tuple.

fManufactureIdInfo

ULong TCardPCMCIA::fManufactureIdInfo;

Is the manufacturer info value from the CIS Manufacturer Id tuple.

fFunctionId

ULong TCardPCMCIA::fFunctionId;

Is the function ID code from the CIS Function Id tuple.

fConfigurationNumber

UChar TConfiguration::fConfigurationNumber;

Is the configuration number for the configuration entry object.

fIoAddresses

ULong TConfiguration::fIoAddresses[0];

Is the I/O address for this configuration.

The Data Link Layer Driver Proto (protoLanternDriver)

This section describes protoLanternDriver, the NewtonScript proto that you need to provide for your driver. You use this proto to register your card and perform a few configuration operations.

You can also use this proto to provide extra configuration events that you can send to clients to communicate configuration information.

You can also use this proto to define custom external events that your clients can send to you.

Listing 3-3 shows the protoLanternDriver slots.

Listing 3-3 protoLanternDriver slots

```
{
Config:                    func(ConfigInfo, enforceConfig),
AppSymbol:                 Symbol,
DeviceType:                String,
DeviceDisplayName:         String          // or Array of Strings
DeviceCISProductName:      String          // or Array of Strings
DeviceClass:               Symbol,
DeviceProtocolClassName:   String,
ConfigBeforeUse:           Boolean,        // TRUE or NIL
AppleTalkAvailable:  :     Boolean,        // TRUE or NIL
EventDefs:                 Array,

    // optional slots:
DeviceInserted:            func(),
DeviceRemoved:             func(),
ConfigView:                View,
MapErrorCode:              func(ErrorCode),
StatusView:                View,
ConfigChanged:             func(ConfigInfo),
MakeMessageText:           func()
}
```

The protoLanternDriver slots

This section describes the slots of *protoLanternDriver*.

Config

Config: func(*ConfigInfo, enforceConfig*)

This method is called when the device is inserted. You can use this method to send the current configuration values to the driver; this allows you to put the driver into a state that matches the configuration values stored in the NewtonScript soup.

ConfigInfo	A frame containing the information entered in the ConfigView slip.
enforceConfig	If TRUE, you should return an error if the device cannot be configured.

AppSymbol

AppSymbol: Symbol

The application symbol (kAppSymbol) is used in the system registry to name your driver. Your symbol must not conflict with any other registered drivers.

DeviceType

DeviceType: String

The four-character identifier string used to uniquely identify the device to the Newton Operating System.

DeviceDisplayName

DeviceDisplayName: String

The device name string used when displaying information to the user. If your driver can work with multiple devices, specify an array of device display names.

DeviceCISProductName

DeviceCISProductName: String

The string contained in the device CIS (for PC Card devices). If your driver can work with multiple devices, specify an array of product name strings.

DeviceClass

DeviceClass: Symbol

The device class symbol. Currently, only 'ethernet is supported.

DeviceProtocolClassName

DeviceProtocolClassName: String

The string name of the p-class for your driver. For example, TTemplateDriver.

ConfigBeforeUse

ConfigBeforeUse: Boolean

A Boolean value. If TRUE, indicates that the device must be configured before use. This means that a user notification is generated when the card is inserted for the first time.

If NIL, indicates that the device does not need to be configured before use.

AppleTalkAvailable

AppleTalkAvailable: Boolean

A Boolean value. If TRUE, indicates that AppleTalk can be used with the device. If NIL, indicates that AppleTalk cannot be used with the device.

EventDefs

EventDefs: Array

An array of event definition arrays. Each event definition has the following format:

event symbol	The event name. For example: 'evGetMemory.
event class and ID	A two-element array that contains the event class and event ID. For example: ["netw", "gmem"].
send args array	An array that specifies the type of each argument posted with the event. An empty array indicates that no arguments are posted. For example: [ULong].
reply args array	An array that specifies the type of each argument received back from event posting. An empty array indicates that no arguments are received. For example: [ULong].

The following is an example of an event definition array:
['evGetMemory, ["netw", "gmem"], [ULong], [ULong]]

DeviceInserted

DeviceInserted: func()

You can optionally define this method, which is called when the device is inserted. You can use this method to provide an application-level function whenever the device is inserted.

DeviceRemoved

DeviceRemoved: func()

You can optionally define this method, which is called when the device is removed. You can use this method to provide an application-level function whenever the device is removed.

ConfigView

ConfigView: View

A view template that allows the user to enter values for all of the items that require configuration on the device.

MapErrorCode

MapErrorCode: func(*ErrorCode*),

You can optionally define this method to provide text descriptions of any device-specific error codes that your driver can return. Return a text string or NIL for each error code. The default version of this method maps the Lantern errors.

StatusView

StatusView: View,

You can optionally define this view to display device status or performance statistics.

ConfigChanged

ConfigChanged: func(*ConfigInfo*),

You can optionally define this method, which is called when the current emporium changes. This happesn when the user selects a new worksite.

ConfigInfo	A frame containing the information entered in the ConfigView slip.

The default version of this method calls Config.

MakeMessageText

MakeMessageText: func()

You can optionally define this method, which is called when your card slip is about to open. You can use this method to display information to the user about the card.

return value	A string.

The default version of this view displays the CISInfo product name, manufacturer name, and version information.

The Data Link Layer Client Proto (protoLanternClient)

This section describes protoLanternClient, the NewtonScript proto that clients of your driver use to access your driver.

Listing 3-4 shows the protoLanternDriver slots.

Listing 3-4 protoLanternDriver slots

```
{
Instantiate:          func(DeviceFrame),
Init:                 func(Options),
Attach:               func(Options),
Detach::              func(),
Delete:               func(),
State:                func(),
MapErrorCode:         func(ErrorCode),
PostDriverEvent:      func(EventSym, EventArgs),
RegisterForEvent:     func(EventSym, Receiver, Message)
UnregisterForEvent:   func(EventSym)
}
```

The protoLanternClient slots

This section describes the slots of protoLanternClient.

Instantiate

Instantiate: func(*DeviceFrame*)

The Instantiate method creates a NewtonScript object (a client) that interfaces to a data link layer driver.

DeviceFrame — A frame that specifies driver information. This frame contains the following slots:

- appSymbol — The application symbol.
- DeviceName — The text string of CIS Product Name

return value — NIL to indicate successful instantiation, or an error code if unsuccessful.

Init

Init: func(*Options*)

The Init method connects your client to an interface object. This is used for passive communication with the driver, such as sending configuration events. The card must be inserted, but the driver is not enabled.

Options — A frame containing options for the initialization (not currently used).

return value — NIL if the connection is sucessful, or an error code if unsuccessful.

Attach

Attach: func(*Options*)

The Attach method attaches your client to a driver, which allows you to request driver services.

Options — A frame containing options for the initialization (not currently used).

return value — NIL if the attach (open) is sucessful, or an error code if unsuccessful.

Detach

Detach: func()

The Detach method detaches your client from the driver, which means that you can no longer request driver services.

return value — NIL if the detach (close) is sucessful, or an error code if unsuccessful.

Delete

Delete: func()

The Delete method deletes your client.

return value — NIL if the deletion is sucessful, or an error code if unsuccessful.

State

State: func()

The State method returns a symbol that specifies the current state of your client.

return value — One of the following symbols:

'ClientNotReady	The client proto has been created but is not yet instantiated.
'ClientReady	The client proto is instantiated and ready to use.
'APIReady	The client is available for use.
'APIAttached	The client is enabled to make driver requests.

MapErrorCode

MapErrorCode: func(*ErrorCode*)

The MapErrorCode method returns a string that describes an error.

ErrorCode — An error code.

return value — A string describing the error or NIL if no description is found for the error.

PostDriverEvent

PostDriverEvent: func(*EventSym*, *EventArgs*)

The PostDriverEvent method sends an event from your client to the driver.

EventSym — A symbol representing the event to be sent.

EventArgs — An array of arguments used by the driver to process the event. The contents of this array is dependent on the event type being posted.

return value — A frame that contains the following two slots:

- Result — NIL if the event was successfully to the driver, or an error code if unsuccessful.
- Data — An array of the reply data, or NIL if no reply data was received.

RegisterForEvent

RegisterForEvent: func(*EventSym*, *Receiver*, *Message*)

The *RegisterForEvent* method registers a callback function to be invoked when a specific event occurs.

EventSym — A symbol representing the event.

Receiver — The receiver frame to which the *Message* is sent.

Message — The method of the receiver that is run when the Newton Operating System receiveds the event.

return value — NIL if registration is successful, or the error code eLANTERN_ClientProtoRFESSpecificEventDefError if the event is not defined or if registration fails.

UnregisterForEvent

UnregisterForEvent: func(*EventSym*)

The UnregisterForEvent method unregisters the callback function that you previously registered for a specific event type.

EventSym — A symbol representing the event.

return value — NIL if the callback is successfully unregistered. If not, one of the following two errors:

eLANTERN_ClientProtoURFENotRegisteredForEvent
: Your client has no callbacks registered for the event.

eLANTERN_ClientProtoURFEEventNotRegistered
: No callbacks are registered for the specified event.

Summary of Data Link Layer Reference

Data Link Layer Constants

Data Link Layer Error Codes

```
eLANTERN_DriverNotFound                 = -61001
eLANTERN_DriverInstallFailed            = -61002
eLANTERN_DriverRemoveFailed             = -61003
eLANTERN_DriverUnhandledEvent           = -61004
eLANTERN_DriverPacketDropped            = -61005
eLANTERN_DriverException                = -61006
eLANTERN_DriverNewAsyncFailed           = -61007
eLANTERN_DriverCardNotInserted          = -61008
eLANTERN_DriverAlreadyReplied           = -61009
eLANTERN_DriverRequestNotRepliedTo      = -61010

eLANTERN_DriverValueRangeError          = -61200
eLANTERN_DriverHardwareFailure          = -61201
eLANTERN_DriverResourceFailure          = -61202

eLANTERN_ClientDispatchFailed           = -61300
eLANTERN_ClientAlreadyBound             = -61301
eLANTERN_ClientNotBound                 = -61302
eLANTERN_ClientNoMemory                 = -61303
eLANTERN_ClientNewAsyncMsgFailed        = -61304
eLANTERN_ClientInvalidTaskType          = -61305
```

Data Link Layer Driver RPCs and Events

RPCs Sent to Your Driver

```
kLanternSendBuffer kLanternEventClass,'sndB',2
                                        // (Ptr ptr, Size size) : {}
```

```
kLanternSendCBufferList kLanternEventClass,'sndC',1
                                                    // (CBufferList* data) : {}

kLanternGetDeviceAddress kLanternEventClass,'gdva',2
                                                    // (UByte* addr, Size size) : {}

kLanternGetLinkIntegrity kLanternEventClass,'glit',0
                                                    // () : {ULong link}

kLanternAddMulticastAddress kLanternEventClass,'amca',1
                                                    // (UByte* addr) : {}

kLanternDelMulticastAddress kLanternEventClass,'dmca',1
                                                    // (UByte* addr) : {}
```

Events Your Driver Sends

```
kLanternLinkIntegrity kLanternEventClass,'link',1
                                                    // (ULong link)

kLanternDriverFailure kLanternEventClass,'fail',1
                                                    // (NewtonErr reason)
```

TLanternDriver P-Class

```
PROTOCOL TLanternDriver : public TProtocol
{
public:
      static TLanternDriver*                        New(char*);
      void                                          Delete();

      NewtonErr                                     Init(void);

      NewtonErr         Enable(void);
      NewtonErr         Disable(void);
      NewtonErr         InterruptHandler(void);

protected:
friend class TLanternTask;
      TCardSocket*                  fCardSocket;
      TCardPCMCIA                   fCardPCMCIA;
      TLanternDriverAPI*            fDriverAPI;
}
```

TLanternDriverAPI P-Class

```
PROTOCOL TLanternDriverAPI : public TProtocol
{
public:
      static TLanternDriverAPI*                    New(char*);
      void                                         Delete();

      NewtonErr                                    Init(TLanternTask*);

      NewtonErr          PowerOn(Boolean* powerOn, Boolean intEnable=true);
      NewtonErr          PowerOff(ULong msDelay=kDefaultPowerDownTime,
                                  Boolean intDisable=true);

      Ptr                NewPacketPtr(Size size);
      NewtonErr          RxReady(Ptr buf, Size size=0);
      NewtonErr          InterruptDone(void);

      NewtonErr          CardHandlerSpecific(ULong selector, ...);

      NewtonErr          AddEventHandler(kLanternEventType, DriverProcPtr);
      NewtonErr          PostEvent(kLanternEventType, ...);
      NewtonErr          PostLocalEvent(kLanternEventType, ...);
      NewtonErr          PostLocalEvent(TTime*, kLanternEventType, ...);
      NewtonErr          PostReply(NewtonErr);
      NewtonErr          PostReply(NewtonErr, ULong nArgs, ...);
}
```

TCardSocket Methods

ULong TCardSocket::AttributeMemBaseAddr();

ULong TCardSocket::CommonMemBaseAddr();

ULong TCardSocket::IOBaseAddr();

void TCardSocket::SelectIOInterface();

void TCardSocket::SetControl(ULong control);

ULong TCardSocket::GetControl(ULong control);

TCardPCMCIA Fields and Methods

const char* TCardPCMCIA::GetCardManufacturer() const;

const char* TCardPCMCIA::GetCardProduct() const;

const char* TCardPCMCIA::GetCardV1String3() const;

const char* TCardPCMCIA::GetCardV1String4() const;

TCardConfiguration* TCardPCMCIA::GetCardConfiguration(ULong configNum);

ULong TCardPCMCIA::fRegisterBaseAddress;

ULong TCardPCMCIA::fNumOfConfigEntry;

ULong TCardPCMCIA::fManufactureId;

ULong TCardPCMCIA::fManufactureIdInfo;

ULong TCardPCMCIA::fFunctionId;

UChar TConfiguration::fConfigurationNumber;

ULong TConfiguration::fIoAddresses[0];

protoLanternDriver Slots

```
{
Config:                     func(ConfigInfo, enforceConfig),
AppSymbol:                  Symbol,
DeviceType:                 String,
DeviceDisplayName:          String          // or Array of Strings
DeviceCISProductName:       String          // or Array of Strings
DeviceClass:                Symbol,
DeviceProtocolClassName:    String,
ConfigBeforeUse:            Boolean,        // TRUE or NIL
AppleTalkAvailable:  :      Boolean,        // TRUE or NIL
EventDefs:                  Array,

      // optional slots:
DeviceInserted:             func(),
DeviceRemoved:              func(),
ConfigView:                 View,
MapErrorCode:               func(ErrorCode),
StatusView:                 View,
```

```
ConfigChanged:          func(ConfigInfo),
MakeMessageText:        func()
}
```

protoLanternClient Slots

```
{
Instantiate:            func(DeviceFrame),
Init:                   func(Options),
Attach:                 func(Options),
Detach::                func(),
Delete:                 func(),
State:                  func(),
MapErrorCode:           func(ErrorCode),
PostDriverEvent:        func(EventSym, EventArgs),
RegisterForEvent:       func(EventSym, Receiver, Message)
UnregisterForEvent:     func(EventSym)
}
```

Building a Data Link Layer Driver

This chapter tells you how to build a data link layer driver.

Building the Sample Driver

This section describes the steps that you need to perform to build the sample data link layer driver that was shipped on your CD-ROM. Use similar steps to build your own driver.

To build the sample data link layer driver, you need to follow these steps:

5. Copy the contents of the NIE 2.0 DDK distribution CD-ROM to your hard drive.
6. Copy the files in the folder “NTK:Move to System” to your System folder.
7. Reboot your computer.
8. Launch MPW.
9. When prompted by MPW, select the “Sources” directory that you copied in step 1 as the NCT root directory.

10. Select “NE2000 Sample” from the “Newton C++ Tools” menu.
11. Launch NTK and open the “NE2K” sample project.
12. Build the sample project.
13. Load the package “Newton Devices.pkg” onto your Newton.
14. Download the package you just built—NE2K.pgk—onto your Newton.
15. Run AppleTalk with the sample driver.

Installing the NCT Additions

To develop a data link layer driver, you need to install the Newton C++ Toolbox (NCT) and then add in several additions. This section describes the components that you must install.

Newton C++ Toolbox (NCT)

The Newton C++ Toolbox provides a complete MPW environment for developing C++ code for the Newton.

DDK Addition

The DDK NCT Addition to NCT adds a collection of MPW scripts that extend MPW to include menu items for creating a DDK project.

The DDK NCT Addition also adds a collection of header files that you need to build a communications tool.

PC-Card Addition

The PCMCIA DDK Addition adds the header files that you need to access a PCMCIA card, including the TCardSocket and TCardPCMCIA classes.

Data Link Layer Addition

The data link layer addition adds the protocol interfaces that you need to build your data link layer driver.