OpenLDV™ Programmer's Guide

Release 2.1

ECHELON®
Corporation

078-0275-01C

Echelon Corporation
www.echelon.com

Purpose

This document describes Echelon's OpenLDV Release 2.1 software. OpenLDV is an API that you can use to create applications that send and receive low-level LonTalk® messages through Echelon's family of LONWORKS® network interface products. This includes local network interfaces such as the PCC-10 PC Card adapter and the PCLTA-20 PC LonTalk Adapter, as well as Internet-enabled network interfaces such as the *i*.LON® 10 Ethernet Adapter and the *i*.LON 100 Internet Server. The OpenLDV software is licensed for use exclusively with Echelon's family of network interface products.

For most Echelon customers, the development of PC-based LONWORKS network tools will be simpler and less time-consuming when done using Echelon's LNS® Network Operating System software. In addition, network tools that use LNS will have higher performance levels than those that use OpenLDV. You can find out more about LNS on Echelon's website at http://www.echelon.com/lns. Contact Echelon Sales at http://www.echelon.com/sales if you would like assistance in determining whether you should develop your network tools with LNS, or with OpenLDV.

Audience

This document is intended for software developers creating OpenLDV applications for use with Echelon's family of LONWORKS network interface products. Readers of this guide should be familiar with LONWORKS technology. An introduction to LONWORKS technology can be found in the *Introduction to the LONWORKS System* document, which can be downloaded from Echelon's website at:

http://www.echelon.com/support/documentation/manuals.

Hardware and Software Requirements

To install and use the OpenLDV software, your PC must meet the following minimum requirements:

- Intel® Pentium® III 366MHz processor
- 128MB RAM
- Microsoft Windows® 98SE, Windows 2000, Windows XP or Windows Server 2003
- 10MB of available hard-disk space
- 800x600 screen resolution

Table of Contents

Purpose....i
Audience....i
Hardware and Software Requirements....i
Table of Contents....ii

Introduction to the OpenLDV API.... 1
Introduction to the OpenLDV API....2
Installing the OpenLDV Software....4
Modifying the OpenLDV Installation Path....5
Modifying the Reboot Behavior When OpenLDV 2.1 Is a Nested Installation....6
Getting Started....6

Using the OpenLDV API.... 9
Referencing the OpenLDV Component....10
The OpenLDV API....10
ldv_get_version....10
ldv_open....11
ldv_close....12
ldv_read....12
ldv_write....14
ldv_register_event....14
OpenLDV Return Codes....16
Using the OpenLDV API with Multiple Threads and Multiple Processes....19

Sending and Receiving Messages With The OpenLDV API.... 21
Overview....22
OpenLDV Application Architecture....22
Application Layer....23
Presentation Layer....24
Interface Layer....24
Physical Layer....24
Link Layer....24
Constructing Link Layer Messages....25
Downlink Commands....26
Uplink Commands....27
Immediate Commands....27
Application Buffer Structure....33
Application Layer Header....33
Message Header....35
ExpMsgHdr....35
Network Address....36
SendAddrDtl....36
RcvAddrDtl....40
RespAddrDtl....42
Message Data....44
UnprocessedNV....44
ExplicitMsg....44
Sending Messages to the Network Interface....45
Receiving Messages from the Network Interface....45

The OpenLDV Developer Example.... 47
The OpenLDV Developer Example....48

Common Definitions 48
COpenLDVapi and COpenLDVtrace 48
COpenLDVni, Message Pumps, and Message Dispatchers 48
Toolkits and User Interface 49
Developer Example Diagram 49

1

Introduction to the OpenLDV API

This chapter introduces the OpenLDV API, and describes how you can use it to send and receive LonTalk messages through any of Echelon's network interface products. It also provides instructions to follow when installing the OpenLDV software.

Introduction to the OpenLDV API

OpenLDV is an API that you can use to write applications that send and receive low-level LonTalk messages through Echelon's family of LONWORKS network interface products. These messages can be used to initialize and terminate communication with a network interface, retrieve incoming LonTalk messages from a network interface, or transmit outgoing LonTalk messages through a network interface. OpenLDV supports simultaneous communication with multiple network interfaces through a single client, and it supports access to both local and remote (Internet-enabled) network interfaces.

The OpenLDV API is realized in the `LDV32.DLL` file. Prior to the release of OpenLDV Release 1.0, Echelon licensed the `LDV32.DLL` to certain third parties for use in their products. ***The OpenLDV Release 1.0 API and higher is backwards compatible with the API contained in all previous releases of Echelon's*** **`LDV32.DLL`**. If you are currently using a previous release of Echelon's `LDV32.DLL` file, Echelon requests that you convert to the OpenLDV Release 2.1 software as soon as possible. In order to avoid Windows DLL search path and naming conflicts when you do so, you should remove the previous release of `LDV32.DLL` that you have been using, and include the OpenLDV installer with your updated product installation.

OpenLDV may be useful when deployed in many network management, or monitor and control, systems. For example, if you are managing a self-installed system with hard-coded network addresses, you could use the OpenLDV API to create an application that sends LonTalk messages to test the devices on your network. This diagnostic application could also periodically send request messages to devices in the system to check their status. You could also use OpenLDV to create a data logging application to monitor and retrieve network variable values from the various devices on your network.

When used with an *i*.LON 10 Ethernet Adapter or an *i*.LON 100 Internet Server, network tools that use the OpenLDV API can establish *downlink* connections (session initiation from a PC to an *i*.LON 10 or *i*.LON 100), and they can accept uplink session requests (session initiation from an *i*.LON 10 or *i*.LON 100 to a PC). The *i*.LON devices and PCs communicate using Echelon's xDriver software subsystem, which is included with the OpenLDV Release 2.1 software.

xDriver is an extensible network driver that uses TCP/IP to establish connections with network interfaces such as Echelon's *i*.LON 10 Ethernet Adapter and *i*.LON 100 Internet Server. You can use the xDriver Profile Editor to create xDriver Profiles for use with your OpenLDV applications. An xDriver Profile is a set of configuration parameters that determines how the xDriver subsystem will manage connections with a group of remote networks. For example, you may have hundreds of remote networks, each of which has an *i*.LON 10 attached. At your service center, your monitoring tool could use the OpenLDV API and xDriver subsystem to listen for uplink session requests from these networks for when they need to report alarm conditions. You can configure each xDriver Profile to provide your application with information identifying the network interface that has requested an uplink session. This will allow you to program your application to quickly identify the source of an uplink session request, and respond to a variety of different alarm conditions. For more information on xDriver and the xDriver Profile Editor, see the *OpenLDV Programmer's Guide, xDriver Supplement.*

Before you begin developing your OpenLDV application, you should be aware that development with OpenLDV is complex. To create applications that properly use the OpenLDV API, you need to understand LonTalk message formats and network interface state management. You also need to be able to manage low-level LonTalk messaging details such as LonTalk reference IDs. Chapters 3 and 4 of this document describe some of the LonTalk message formats you can use with OpenLDV Release 2.1. In addition, the *Message Header* section of Chapter 3 includes some discussion of LonTalk reference IDs. Other documents you may find useful when performing these tasks include the *LONWORKS Host Application Programmer's Guide* and the *LONWORKS Microprocessor Interface Program (MIP) User's Guide,* which can be downloaded from Echelon's website at http://www.echelon.com/support/documentation/Manuals/default.htm, as well as the EIA/CEA 709.1-B-2002 protocol specification, which can be downloaded from http://global.ihs.com/.

You should note that Echelon's LNS software provides a high-level interface to LONWORKS networks that intentionally hides all of the complexity involved with managing network interfaces and the low-level communication details of the LonTalk protocol. LNS is a powerful, flexible network management platform you can use with high performance Layer 2 and Layer 5 network interfaces, as well as with LONWORKS/IP routers such as the *i*.LON 600 and *i*.LON 1000. LNS provides a wide variety of network management and monitor and control services, and allows multiple client access to the same network interface - which is not directly supported by OpenLDV.

For most customers, choosing to use the LNS software platform will result in a high-quality application that can be developed more quickly, and with far less knowledge of the low-level details of the LonTalk protocol, than with other network management platforms, including OpenLDV. However, with the introduction of the OpenLDV API, you now have another choice for writing PC-based LONWORKS software for use with Echelon's family of network interface products.

This document describes how to install the OpenLDV software, and how to write applications that use the OpenLDV API. For instructions on configuring the xDriver subsystem and on using the xDriver Profile Editor, see the *OpenLDV Programmer's Guide, xDriver Supplement,* which is included with the OpenLDV Developer's Kit described in the next section.

Installing the OpenLDV Software

This section describes how to install the OpenLDV software. To install and use the OpenLDV software, your PC must meet the following minimum requirements:

- Intel Pentium III 366MHz processor
- 128MB RAM
- Microsoft Windows 98SE, Windows 2000, Windows XP or Windows Server 2003
- 10MB of available hard-disk space
- 800x600 screen resolution

You can download the OpenLDV runtime installer (`OpenLDV210.exe`) from Echelon's website at http://www.echelon.com/downloads. The OpenLDV runtime installer installs the OpenLDV runtime components, including the LONWORKS Interfaces application in the Windows Control Panel, and the xDriver Profile Editor. Echelon designed the OpenLDV runtime installer to be incorporated directly into your OpenLDV application's installation, either as a standalone component that your end-users will install, or as a component that your overall software installer will install. Note that the OpenLDV runtime installer is based on Microsoft Installer 2.0. If the PC you are installing the OpenLDV runtime component on is using an outdated version of Microsoft Installer, the OpenLDV runtime installation will update that PC to use version 2.0.

You can also download the OpenLDV Release 2.1 readme file (`readme_OpenLDV.htm`) from http://www.echelon.com/downloads. You should review the readme file before executing the OpenLDV runtime installer, or developing your OpenLDV application.

If you are developing an OpenLDV application, you will also need the OpenLDV Developer's Kit (`OpenLDV210-DK.zip`), which is installed into the `C:\LonWorks` folder by the LNS Turbo Edition installation. You can also download the file from http://www.echelon.com/downloads. The OpenLDV Developer's Kit contains documentation, include files, and the OpenLDV Developer Example, which you will find useful when you begin developing your own OpenLDV application.

You can also download the OpenLDV Developer' Kit Release 2.1 readme file (`readme_OpenLDV-DK.htm`) from http://www.echelon.com/downloads. You should review this readme file before extracting the contents of the OpenLDV Developer's Kit archive, or developing your OpenLDV application.

NOTE: The "210" appended to OpenLDV in the above file names indicates that the file is for OpenLDV Release 2.1.

To install and use the OpenLDV runtime software, follow these steps:

1. Prior to the release of OpenLDV Release 1.0, Echelon licensed the LDV32.DLL file to certain third parties for use in their products. ***The OpenLDV Release 1.0 (and higher) API is backwards compatible with the API contained in all previous releases of Echelon's LDV32.DLL***. If the PC that you installing the OpenLDV runtime software on contains a previous version of the `LDV32.DLL` file, you should either delete the existing `LDV32.DLL` file, rename it, or move it out of the search path that Windows uses to find components. The OpenLDV software will malfunction if an older version of `LDV32.DLL` exists in the Windows search path, and is found before the newly-installed OpenLDV version of the `LDV32.DLL` file.

2. Download the `readme_OpenLDV.htm` and `OpenLDV210.exe` files from Echelon's website at http://www.echelon.com/downloads.

3. After reviewing the readme file, double-click the `OpenLDV210.exe` file to begin the OpenLDV runtime installation. The Welcome to the InstallShield Wizard for Echelon OpenLDV 2.1 window will open. Click **Next** to continue. This opens the License Agreement window.

4. Read the terms of the license agreement, and if you agree to the terms, click the **I accept…** button to continue. This opens the Installing Echelon OpenLDV 2.1 window. The installer will now install the OpenLDV runtime software. A completion dialog will appear when the installation is complete. If the OpenLDV runtime is not installed successfully, a dialog will appear to notify you of this.

5. If you are using the *i*.LON 10 Ethernet Adapter or the *i*.LON 100 Internet Server, you may need to modify the configuration for those devices with the LONWORKS Interfaces application in the Windows Control Panel and the xDriver Profile Editor before using them with your OpenLDV application. You can use the LONWORKS Interfaces application in the Windows Control Panel to specify the Internet network addresses of the *i*.LON 10 or *i*.LON 100 that you will connect to with your OpenLDV application. Consult the online help for the LONWORKS Interfaces application for more information on this. You can use the xDriver Profile Editor to configure xDriver Profiles for use with your OpenLDV application. For more information on the xDriver Profile Editor, see Chapter 2 of the *OpenLDV Programmer's Guide, xDriver Supplement.*

To install and use the OpenLDV Developer's Kit, follow these steps:

1. Download the `readme_OpenLDV-DK.htm` and `OpenLDV210-DK.zip` files from Echelon's website at http://www.echelon.com/downloads.

2. After reviewing the readme file, extract the files contained in the `OpenLDV210-DK.zip` file to the LONWORKS folder of your PC. By default, this folder is `C:\LonWorks`. Although you may install the OpenLDV Developer's Kit to any folder on your PC, Echelon recommends that you install them into the LONWORKS folder. For more information on the LONWORKS folder, see the next section, *Modifying the OpenLDV Installation Path.*

3. You can now use the OpenLDV Developer's Kit to write applications that use the OpenLDV API. To do so, include the `ldv32.h` header file that was extracted into the `C:\LonWorks\OpenLDV\Include` folder, and link with the `ldv32.lib` library file that was extracted into the `C:\LonWorks\OpenLDV\Lib` folder, within your Windows application development environment. For more information on the OpenLDV API, see Chapter 2, *Using the OpenLDV API.*

Modifying the OpenLDV Installation Path

All of Echelon's software products are designed to install into a single folder tree on a PC. Echelon's software installers check for the existence of a Windows registry key to determine if the location of this tree (the LONWORKS Path) has been set. If it has been set, all of Echelon's software installers automatically install into this path.

By default the LONWORKS Path is `C:\LonWorks.` The Windows registry key that determines this path (`HKEY_LOCAL_MACHINE\Software\LonWorks\LonWorks Path`) must ***never*** be changed after it has been initially set. If the LONWORKS Path is changed after it has been initially set, some or all of the Echelon software installed on your machine will malfunction.

The OpenLDV runtime installer will install into the LONWORKS Path if it has already been set in the Windows registry. If the LONWORKS Path has not been set, the OpenLDV runtime installer will set the LONWORKS Path to the `C:\LonWorks` folder, and then install into this folder without prompting. If you want to modify this behavior, you may do so by first checking that the LONWORKS Path has not been set (the registry key is `HKEY_LOCAL_MACHINE\Software\LonWorks\LonWorks Path`), and then creating a string variable specifying a valid path on the PC. When you subsequently launch the OpenLDV installer, the installer will find the LONWORKS Path you have created, and install into it.

Since the LONWORKS Path cannot be modified after you initially set it, and because all subsequent Echelon software installations will use this path, Echelon recommends that you do not set it to install into your product's folder tree. The reason is that if an end-user uninstalls your software product, the uninstallation may remove Echelon software components as well.

Modifying the Reboot Behavior When OpenLDV 2.1 Is a Nested Installation

The OpenLDV installation may discover that some components of the OpenLDV product are in use, and the installation cannot complete without a Windows reboot. The default behavior of the OpenLDV 2.1 installation in this case is to display a dialog stating that a reboot is necessary, and then reboot immediately if you select **Yes**.

There are two potential problems here. One is that your product installation may not want the user to make a reboot choice until all installation is complete. The second is that your product installation checks the error return from the OpenLDV installation to determine whether it was successful, but a failed code will be returned if a reboot is required, even if the user selects **No** to defer the reboot.

The reboot choice dialog may be suppressed in the OpenLDV 2.1 installation by invoking it with the command-line option `REBOOT=R`. Note that this will suppress the choice dialog only, and will not prevent an "installation not complete" error code in the case where a reboot is necessary.

From your installation, if you need to determine whether the OpenLDV 2.1 sub-installation has succeeded after you have run it, your installation can look at the Windows Registry entry at `\HKEY_LOCAL_MACHINE\Software\Echelon\Echelon OpenLDV\Install Status`. If the string value there is `Success`, the installation succeeded and no reboot is required. If the value is Success-`RebootRequired`, the installation succeeded and a reboot is required. For any other value, or if this Registry entry is not present, the OpenLDV 2.1 installation failed.

Getting Started

An important factor you need to be aware of before developing or using any OpenLDV application is that OpenLDV is licensed for use only with Echelon's network interfaces, and for the PCC-10, PCLTA-10 and PCLTA-20, only with Layer 5 firmware.

Use the LONWORKS Plug 'n Play application in the Windows Control Panel to determine if your network interface is using a Layer 5 image. You can determine which image the network interface is using with the "NI Application" field. Table 1.1 lists the image you should select to ensure that your Echelon network interface is using a Layer 5 image.

NOTE: The LONWORKS Plug 'n Play application is installed with the network interface driver for the network interfaces listed in Table 1.1.

Table 1.1 NI Application Settings

Network Interface	NI Application Setting for Layer 5 Image
PCC-10	NSIPCC
PCLTA-10	NSIPCLTA
PCLTA-20	NSIPCLTA

The remainder of this document contains information you will need when creating your OpenLDV application. Echelon strongly recommends that you review this material before you begin developing of your OpenLDV application. This includes the following chapters:

Chapter 2, *Using the OpenLDV API:* This chapter describes each function that is included in the OpenLDV API. It also defines guidelines you need to follow when writing applications that use the OpenLDV API to access multiple network interfaces.

Chapter 3, *Sending and Receiving Messages With The OpenLDV API:* You can use the `ldv_write` and `ldv_read` functions described in Chapter 2 to send and receive message commands through a network interface. This chapter describes the various network interface commands your OpenLDV application can send and receive with these functions, as well as the application buffer structure each type of message requires.

Chapter 4, *The OpenLDV Developer Example:* This chapter introduces the OpenLDV Developer Example, which is installed with the OpenLDV Developer's Kit. It describes various classes implemented in the OpenLDV Developer Example. You should also note that the OpenLDV Developer Example contains comments you will find useful as you review its code

.

2

Using the OpenLDV API

This chapter describes each function that is included in the OpenLDV API. It also defines guidelines you need to follow when writing applications that use the OpenLDV API to access multiple network interfaces.

Referencing the OpenLDV Component

This chapter describes the OpenLDV API functions, including the input and output parameters associated with each function, and the return codes returned by each function.

You can develop applications that use the OpenLDV API with any Windows application development environment that supports the use of standard Windows DLL and COM components. Echelon has tested the OpenLDV software with Microsoft Visual Studio .NET® 2003, using the Microsoft Visual C++ component. Echelon will provide technical assistance for the OpenLDV API only if you are developing your application with Microsoft Visual Studio .NET 2003, using the Microsoft Visual C++ component.

In order to develop with the OpenLDV API, you must install the OpenLDV Developer's Kit (`OpenLDV-DK.zip`), as described in Chapter 1. During this procedure, you should have extracted the `ldv32.h` and `ldv32.lib` files to the `C:\LonWorks\OpenLDV\Include` and `C:\LonWorks\OpenLDV\Lib` folders of your PC.

To develop your OpenLDV application, you must instruct your Windows application development environment to include the `ldv32.h` header file, and to link to the `ldv32.lib` import library. Consult your development environment's documentation for information about linking to external libraries. Once you have performed these steps (and assuming that the OpenLDV210 runtime installer has been run on your PC), the OpenLDV interface (`LDV32.DLL`) will be automatically loaded and dynamically linked to your application when your application loads. End users of your OpenLDV application do not need any of the files included in the OpenLDV Developer's Kit installed on their PC to use their application. They only need to have the OpenLDV installer installed.

The OpenLDV Developer's Kit includes the OpenLDV Developer Example, which uses the functions described in this chapter. This example application should be useful to you as you begin writing your own OpenLDV application. You will need Microsoft Visual Studio .NET 2003, with the Microsoft Visual C++ component installed, to compile and debug the OpenLDV Developer Example. The example application will be extracted to the `C:\LonWorks\OpenLDV\Examples\Example1` folder of your PC when you install the OpenLDV Developer's Kit. The OpenLDV Developer Example contains numerous helpful comments. In addition, Chapter 4 of this document describes the architecture of the OpenLDV Developer Example, and the different classes it contains.

The OpenLDV API

This section describes each of the functions included in the OpenLDV API, and common return codes that may be returned by each function. The entire set of return codes that may be returned by these functions is described in the next section*, OpenLDV Return Codes,* on page 16.

ldv_get_version

Summary: Use this function to read the version number of the OpenLDV software installed on your PC.

Syntax:: `LPCSTR` ldv_get_version(void)

Remarks: This function returns the version number of the OpenLDV software being used as a constant string.

ldv_open

Summary: Use this function to establish communication between your application and a network interface. The function returns a unique handle that you can provide to the other OpenLDV functions to identify this instance of the network interface.

Syntax: `LDVCode ldv_open`(LPCSTR *id*, short* *handle*)

Element	Description
id	Identify the network interface to establish communication with by specifying its name as the *id* input parameter. For example, “x.Default.1MainStreet” could be used to identify an *i*.LON 10 that will be opened through xDriver. Or, “LON1” could be used to identify a PCLTA-10 or PCLTA-20 network interface.
handle	This output parameter will contain a pointer to a short integer that you will use to identify the network interface with the other OpenLDV functions. This handle is only valid if the function returns LDV_OK.

Remarks: This function returns LDV_OK if the network interface is successfully opened. In this case, the function will also return a handle that you will use to identify the network interface with the other OpenLDV functions. You can use the `ldv_close` function to close the session with the network interface.

When the `ldv_open` function returns the LDV_OK success code, the network interface device has been initialized, and has entered the initial flush state. To start using the network interface, the OpenLDV application must cancel the flush state with the `niFLUSH_CANCEL` immediate network interface command. For more information on immediate commands, see *Immediate Commands* on page 27.

For xDriver-based remote network interfaces that use the Default xDriver Profile, the name specified as the *id* parameter should match an entry created for a device with the LONWORKS Interfaces application in the Windows Control Panel. See Chapter 2 of the *OpenLDV Programmer’s Guide, xDriver Supplement* for more information on this. If you do not specify a valid network interface name as the *id* parameter when you call this function, or if the network interface referenced by the *id* parameter cannot be found, the LDV_INVALID_DEVICE_ID or LDVX_INVALID_XDRIVER return codes will be returned.

Each network interface can only be part of one OpenLDV session at a time on a given PC. If you invoke this function on a network interface that is being used by another process on your PC, the function will fail to execute, and the LDV_ACCESS_DENIED return code will be returned.

If you use xDriver to open a remote network interface while a remote client on another PC is using it, the call to `ldv_open` may initially appear to have succeeded. However, when you call `ldv_read` or `ldv_write` to read or write a message to the network interface later, the LDVX_READ_FAILED or LDVX_WRITE_FAILED failure codes will return, indicating that the session has failed. The timing of this depends on the setting of the Synchronous Timeout field of the xDriver Profile handling the session, as well as the setting of the `TcpMaxConnectRetransmissions` parameter on the PC running the application. For more information on xDriver Profiles, see Chapter 3 of the *OpenLDV Programmer's Guide, xDriver Supplement.*

ldv_close

Summary: Use this function to close a network interface that has been previously initialized with the `ldv_open` function.

Syntax: `LDVCode ldv_close`(short *handle*)

Element	Description
handle	Pass in a handle value identifying the network interface to be closed as the *handle* input parameter. This value was returned as the *handle* element when you opened the network interface with the `ldv_open` function.

Remarks: Use this function to close an OpenLDV session, which will end communication between your application and the network interface involved in the session. This frees any resources assigned to the network interface, and the *handle* assigned to the session. This function returns LDV_OK if the network interface is successfully closed. Once this happens, other processes on your PC will be able to access the network interface.

If you attempt to close a network interface that has not been previously opened, or has already been closed, the LDV_NOT_OPEN code will be returned. If the *handle* parameter you pass to the function is not valid, the LDV_INVALID_DEVICE_ID code will be returned.

ldv_read

Summary: Use this function to read the next uplink message from a network interface.

Syntax: `LDVCode ldv_read`(short *handle,* void* *msg_p,* short *len*)

Element	Description
handle	Pass in a handle value identifying the network interface to be read from as the *handle* input parameter. This value was returned as the *handle* element when you opened the network interface with the `ldv_open` function.

msg_p	A pointer to a buffer allocated by your application that will receive the next uplink message. You must program your application to ensure that a sufficiently large buffer is available to receive each message. The length of this buffer is specified by the *len* parameter. For information on the different uplink messages you might read with this function, and descriptions of the application buffer structure each one uses, see Chapter 3, *Sending and Receiving Messages With OpenLDV.*
len	You specify the length of the application buffer to receive the message, in bytes, as the input *len* parameter. The maximum length of a message is 257 bytes, and so Echelon recommends that you use a buffer length of at least 257 bytes.

Remarks: All messages from a network interface involved in an OpenLDV session are buffered in the OpenLDV driver until a client application reads them with this function. You can program your application to poll the network interface for incoming messages by periodically calling this function. The function will return LDV_OK to signal that it has successfully read a message from the network interface. Alternatively, you can use the `ldv_register_event` function to set up events that will signal the receipt of each new message. The `ldv_register_event` function is described later in this chapter.

Although incoming messages are buffered in the OpenLDV driver, the OpenLDV application must process these messages, and provide suitable responses to the LONWORKS network, in a timely fashion. The acceptable duration for this depends on many different attributes of the OpenLDV session, including the arrival rate of messages from the network, the number of buffers in the network interface driver involved, and the speed and current processing load of the PC running your application. Therefore, the best strategy is for the OpenLDV application to process all incoming message promptly, and with high priority.

The `ldv_read` function will return LDV_NO_MSG_AVAIL if no messages are currently available to be read from the network interface. It will return LDV_INVALID_BUF_LEN if the specified buffer is too small to contain the next incoming message. In this case, you should allocate a larger buffer to receive the message, and call the function again, specifying a larger value as the *len* input parameter. Note that the maximum length of a message is 257 bytes, and so Echelon recommends that you use a buffer length of at least 257 bytes.

If the *handle* parameter you pass to the function is not valid, the LDV_INVALID_DEVICE_ID code will be returned. If the network interface referenced by the *handle* parameter has not been opened by your process, then the LDV_NOT_OPEN code will be returned if the *handle* references a local network interface. If the *handle* references a remote network interface, the LDVX_READ_FAILED code will be returned.

ldv_write

Summary: Use this function to write a message to the network interface, or to send a message through the network interface to a device on the network.

Syntax: `LDVCode ldv_write`(short *handle,* void* *msg_p,* short *len*)

Element	Description
handle	Pass in a handle value identifying the network interface to be written to as the *handle* input parameter. This value was returned as the *handle* element when you opened the network interface with the `ldv_open` function.
msg_p	This input parameter should contain a pointer to a buffer which contains the message to be written to the network interface. For information on the different message commands you can send with this function, and descriptions of the application buffer structure each one requires, see Chapter 3, *Sending and Receiving Messages With OpenLDV.*
len	This input parameter should specify the length of the message to be written. Note that this might not match the length of buffer referenced by the *msg_p* parameter. The *len* parameter should reflect how many bytes will be written to the network interface, and should therefore be less than or equal to the length of the buffer referenced by the *msg_p* parameter.

Remarks: This function will return LDV_OK if the message is successfully written to the network interface.

If the *handle* parameter you pass to the function is not valid, the LDV_INVALID_DEVICE_ID code will be returned. If the network interface referenced by the *handle* parameter is not open, then the LDV_NOT_OPEN code will be returned if it is a local network interface. If it is a remote network interface, the LDVX_WRITE_FAILED code will be returned.

ldv_register_event

Summary: Use this function to register a Windows event object that will be signaled whenever a message is available to be read from a network interface.

Syntax: `LDVCode ldv_register_event`(short *handle,* HANDLE *event*)

Element	Description
handle	Pass in a handle value identifying the network interface that will cause the Windows event object to be signaled as the *handle* input parameter. This value would have been returned as the

	handle element when you opened the network interface with the `ldv_open` function.
event	The Windows event object that should be signaled each time a message is received. You can use the Windows *CreateEvent* and *CloseHandle* functions to create and destroy a Windows event object suitable for use with the `ldv_register_event` function.

Remarks: Use this function to register for notification of incoming messages from the network interface. When the network interface receives a message, the Windows event object referenced by the *event* parameter will be signaled. After that, you can use the `ldv_read` function to read the message.

Note that this event signals the availability of one or more messages to be read. When the Windows event object is signaled, the OpenLDV application should call the `ldv_read` function repeatedly, until all available uplink messages have been read.

To register another event, call `ldv_register_event` again with a new *event* parameter. You can also call the `ldv_register_event` function and specify NULL as the *event* parameter to disable event notifications for a network interface.

As an alternative to using events, you can program your application to periodically call `ldv_read` to check if there are any messages to be read from the network interface. The `ldv_read` function is described earlier in this chapter.

If the *handle* parameter you pass to the function is not valid, the LDV_INVALID_DEVICE_ID code will be returned. If the network interface referenced by the *handle* parameter is not open, then the LDV_NOT_OPEN code will be returned. If the function fails to register the Windows event object for any reason, the LDVX_REGISTER_FAILED code will be returned.

OpenLDV Return Codes

Table 2.1 describes the return codes that may be returned by the OpenLDV functions.

Table 2.1 OpenLDV Return Codes

Return Code	Numeric Value	Description
`LDV_OK`	0	The operation completed successfully.
`LDV_NOT_FOUND`	1	This code will be returned if you call the `ldv_open` function to open a network interface, but you do not specify a valid device name as the *id* parameter, or the device referenced by the *id* parameter cannot be found.
`LDV_ALREADY_OPEN`	2	This return code is obsolete.
`LDV_NOT_OPEN`	3	The network interface is not open. This code may be returned if you use the `ldv_read` or `ldv_write` functions to read or write a message to a network interface, or if you use the `ldv_close` function to close a session with a network interface, and the network interface has not yet been initialized with the `ldv_open` function (or the network interface has already been closed).
`LDV_DEVICE_ERR`	4	This code will be returned if a function fails to execute as a result of a failure to communicate with the network driver. If you encounter this return code, you should call `ldv_close` to close the network interface. This will release the resources assigned to the network driver. Once you have done so, you can re-open the network interface with the `ldv_open` function.
`LDV_INVALID_DEVICE_ID`	5	This code will be returned if you specify an invalid device name when opening a network interface with the `ldv_open` function, or an invalid handle when using any of the other OpenLDV functions. When you invoke the `ldv_open` function, you will use the *id* input parameter to specify the name of the device to be opened. Make sure you enter a valid device ID here. The `ldv_open` function will then open the network interface, and return a handle value you can use to identify the network interface with the other OpenLDV functions.

Return Code	Numeric Value	Description
`LDV_NO_MSG_AVAIL`	6	No message is available to be read. This code will be returned if you call `ldv_read`, and there are no uplink messages from the network interface that have not yet been read. You can use the `ldv_register_event` function described earlier in this chapter to receive notification events when messages are available to be read from the network interface.
`LDV_NO_BUFF_AVAIL`	7	No buffer is available. This code will be returned if you call `ldv_write`, and there is no available buffer on the local network interface to write the message to. In this case, you should wait until a buffer becomes available, and try writing the message again.
`LDV_NO_RESOURCES`	8	No resources are available. This code may be returned if the OpenLDV API has assigned too many session handles, or if the PC running your application is having memory allocation problems. If you encounter this return code, you should close any non-essential processes running on your PC, and try the operation again.
`LDV_INVALID_BUF_LEN`	9	This code will be returned if you call `ldv_read` to read a message from a network interface, and the buffer length you specify is not big enough to contain the next incoming message. In this case, you need to allocate a larger buffer to receive the message and then call `ldv_read` again, making sure to specify a larger value as the *len* input parameter. The message will remain as the next incoming message until you successfully read it with `ldv_read`. To avoid this error, Echelon recommends that you allocate a buffer of at least 257 bytes (the maximum size of an incoming message) each time you call `ldv_read`.
`LDV_NOT_ENABLED`	10	This return code is obsolete.
`LDVX_INITIALIZATION_FAILED`	11	The remote network interface could not be initialized. Generally, this code will be returned if there are configuration problems on the network interface you are opening, or on the PC running your application.

Return Code	Numeric Value	Description
`LDVX_OPEN_FAILED`	12	The remote network interface could not be opened.
`LDVX_CLOSE_FAILED`	13	The remote network interface could not be closed.
`LDVX_READ_FAILED`	14	The application failed to read the message as a result of a generic failure during the call to `ldv_read`. If you encounter this return code persistently, you should close the current session and start a new one, as the current session may have failed.
`LDVX_WRITE_FAILED`	15	The application failed to write the message as a result of a generic failure during the call to `ldv_write`. If you encounter this return code persistently, you should close the current session and start a new one, as the current session may have failed.
`LDVX_REGISTER_FAILED`	16	The application failed to register the Windows event object for event notification.
`LDVX_INVALID_XDRIVER`	17	This code will be returned if you attempt to open an xDriver network interface with the `ldv_open` function, and the xDriver Lookup Extension Component fails to find that network interface. You should use the LONWORKS Interfaces application in the Windows Control Panel to check that the network interface referenced by the *id* parameter exists. See the LONWORKS Interfaces application's online help for information on how to do so. For information on Lookup Extension Components, see the *OpenLDV Programmer's Guide, xDriver Supplement.*
`LDVX_DEBUG_FAILED`	18	This return code is reserved for future use.
`LDVX_ACCESS_DENIED`	19	This code will be returned if you call `ldv_open` to initialize a network interface that is already involved in another process on your PC. OpenLDV does not support concurrent access to network interfaces between multiple processes on the same PC. For more information on this, see the next section, *Using the OpenLDV API with Multiple Threads and Multiple Processes.*

Using the OpenLDV API with Multiple Threads and Multiple Processes

The OpenLDV software supports communication with multiple network interfaces at a time. However, there are certain restrictions you need to be aware of when writing applications that use the OpenLDV API to access multiple network interfaces simultaneously:

1. A single process can access multiple network interfaces simultaneously. However, concurrent thread access to each network interface should be limited to access by one writer thread, and by one reader thread. You must program your application to enforce this restriction, as it is not enforced by the OpenLDV software. See the OpenLDV Developer Example for a demonstration of the proper use of separate reader and writer threads.

2. Multiple processes on the same PC cannot access the same network interface simultaneously. Attempts to access the same network interface by more than one process on a PC will result in the LDV_ACCESS_DENIED failure code when the `ldv_open` function is called.

3. The *i*.LON 10 Ethernet Adapter and *i*.LON 100 Internet Server allow a single session at a time. If you attempt to open such a network interface while another session is active (usually from another PC), the call to `ldv_open` may initially appear to have succeeded. However, when you call `ldv_read` or `ldv_write` to read or write a message to the network interface later, the LDVX_READ_FAILED or LDVX_WRITE_FAILED return codes will return, indicating that the session has failed. The timing of this depends on the setting of the Synchronous Timeout field of the xDriver Profile handling the session, as well as the setting of the `TcpMaxConnectRetransmissions` parameter on the PC running the application. For more information on this, see the description of the `ldv_open` function earlier in this chapter.

These are the guidelines you need to follow when writing an application that will access multiple network interfaces simultaneously. If you follow these guidelines, you should not have a problem creating such an application.

3

Sending and Receiving Messages With The OpenLDV API

This chapter describes the various network interface message commands your OpenLDV application can send and receive through a network interface, as well as the application buffer structure that each type of message requires.

Overview

OpenLDV applications construct outgoing messages using application buffer structures, and send that data to the interface using the `ldv_write` function. OpenLDV applications use the `ldv_read` function to retrieve data from the interface, using the same application buffer structures.

This chapter begins with a discussion of the different layers of an OpenLDV application that handle the transmission and receipt of these messages, and then describes the formats of the application buffer structures used by those messages. Example code that defines the formats of these application buffer structures described in this chapter can be found in the `OpenLDVdefinitions.h` header file. The `OpenLDVdefinitions.h` file is included with the OpenLDV Developer Example, which is described in Chapter 4 of this document.

OpenLDV Application Architecture

An application that uses the OpenLDV API is called an *OpenLDV application.* The OpenLDV application architecture has several layers: the application layer, the presentation layer, the interface layer, the link layer, and the physical layer.

Figure 3.1 illustrates the different layers and interfaces contained within a typical OpenLDV application. The sections following Figure 3.1 describe each of these layers in detail.

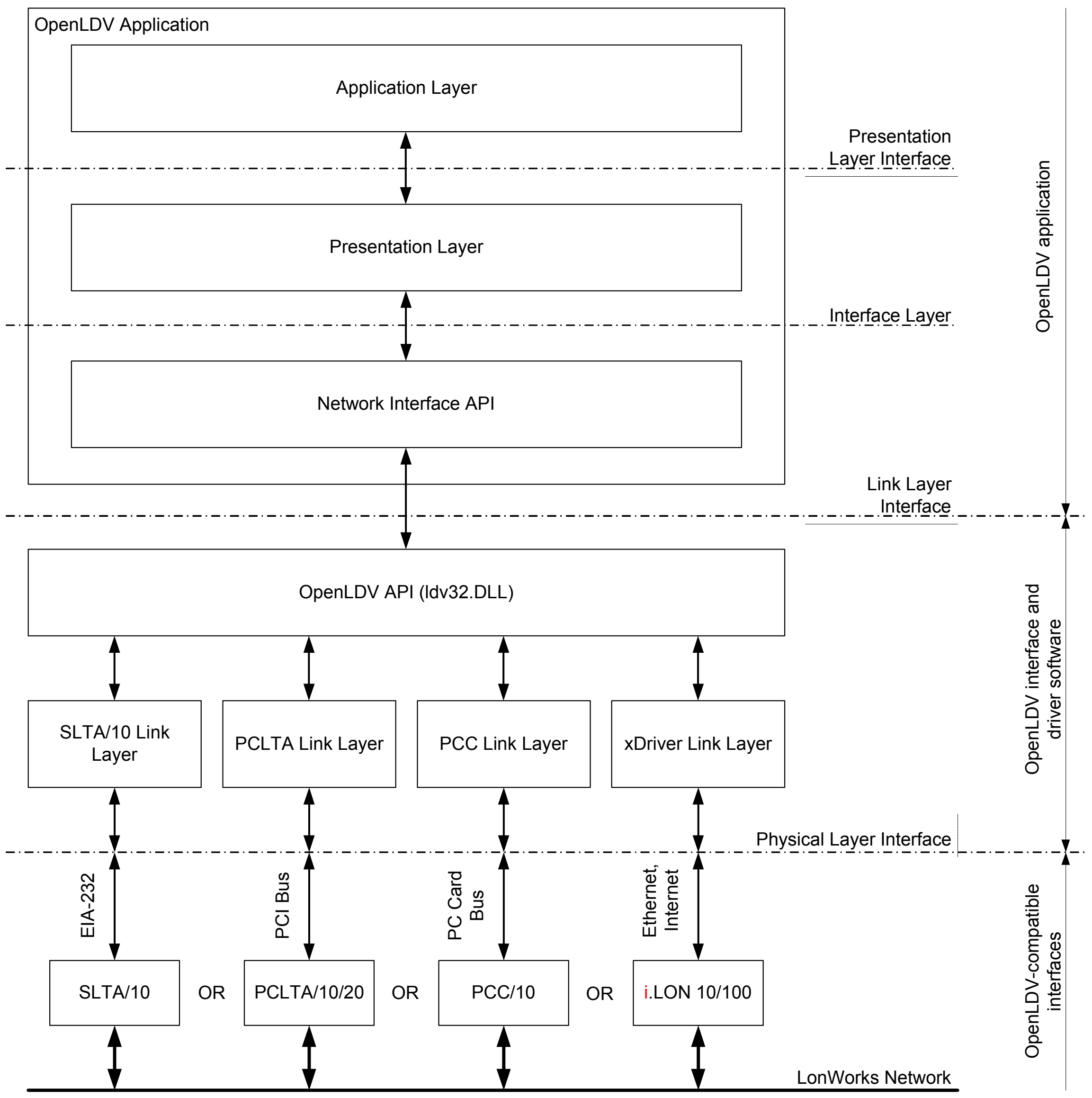


Figure 3.1 OpenLDV Application Architecture

Application Layer

The application layer represents layer 7 of the OSI (Open Systems Interconnection) reference model developed by ISO (International Standards Organization). This layer sends data to the LONWORKS network through output network variables and outgoing application messages, and receives LONWORKS network data through input network variables and incoming application messages. This layer is where an OpenLDV application's primary algorithm operates. In general, the bulk of an OpenLDV application's code is at the application layer.

The OpenLDV API does not explicitly support the application layer as a separate component. However, the OpenLDV Developer Example contains a fragment architecture that dispatches incoming messages to an application-specific message dispatcher. For more information on the OpenLDV Developer Example and the message dispatcher it employs, see Chapter 4, *The OpenLDV Developer Example.*

Presentation Layer

The presentation layer represents layer 6 of the OSI (Open Systems Interconnection) reference model developed by ISO (International Standards Organization). This layer translates messages between the lower layers implemented by OpenLDV (described in the following sections) and the easier-to-use presentation format used by the application layer. For example, all incoming network variable update messages from a network interface will be recognized by this layer. Following this, the application's network variable objects will be updated accordingly, and the application layer will be notified of the change.

The presentation layer also manages several network management messages and diagnostics services, such as the responses to QuerySI network management commands.

OpenLDV does not explicitly support the presentation layer as a separate component. However, the OpenLDV Developer Example contains a fragment architecture that provides a framework for a presentation layer implementation. The OpenLDV Developer Example also includes code that handles several network management commands and diagnostics.

Interface Layer

The interface layer, also referred to as the *Network Interface Layer*, is used to ensure reliable delivery of packets between the OpenLDV application and the network interface, and to execute complete network transactions.

Examples of these network transactions include sending a simple message and awaiting a transaction completion (success or failure) code, querying some external data and awaiting one or more responses to that request, replying to incoming requests, and operations that involve local management and diagnostics of a network interface.

The OpenLDV API does not explicitly implement the interface layer as a separate component. However, the OpenLDV Developer Example contains an example implementation of a network interface layer. It also contains code that supports execution of all transaction types mentioned in the previous paragraph.

Physical Layer

The physical layer interface is the interface between the OpenLDV application and the network interface. The physical layer interface for an *i*.LON 10 or *i*.LON 100 network interface is an Ethernet connection that uses the TCP/IP protocol. The physical layer interface for an SLTA-10 is an EIA-232 interface, as described in the *Serial LonTalk Adapter User's Guide*. Other network interfaces use a variety of local physical layer interfaces, such as the PCI bus or the PC Card bus.

Link Layer

The network interface driver implements the link layer to ensure the reliable delivery of messages between the OpenLDV application and the network interface. The link layer is hardware-dependent, and varies between the different network interfaces you might use with an OpenLDV application. For example, a local PCC-10 PC LonTalk Adapter requires a link layer implementation different than the implementation required to access an *i*.LON 100 across the Internet.

In addition, the link layer also implements commands specific to the network interface being used. For example, the xDriver link layer, which is used for Internet connections to *i*.LON 10 and *i*.LON 100 network interfaces, contains support for strong encryption of messages, as well as authentication of the communicating endpoints, between an OpenLDV application and a remote network interface.

Although the link layer implementation is different for most network interfaces, the OpenLDV application developer does not need to be concerned with the details and constraints of the various link layer implementations. This is because the driver software that is delivered and installed with each network interface implements the link layer interface.

And so regardless of the way the network interface you are using implements the link layer interface, all you need to do to use the link layer interface to send and receive messages through the network interface is to call the `ldv_read` and `ldv_write` functions of the OpenLDV API, using the application buffer structures described in this chapter to format the data you are sending and receiving.

Constructing Link Layer Messages

The `ldv_read` and `ldv_write` functions take a *msg_p* parameter, which is a pointer to the data that is to be sent for the `ldv_write` function, or a pointer to the buffer for data to be received for the `ldv_read` function. They also take a *len* parameter, which specifies the size of that data (or buffer) in bytes. The data exchanged at this level is commonly referred to as the *application buffer structure*, although the actual application layer is further up the stack.

In both cases, the application buffer structure begins with a simple header, the *Network Interface Header.* This header contains an indicator for a command and queue in the first byte, and the length of the data that follows the header in the second byte. An optional, variable-length data field (as indicated by the header's *length* byte) follows the header.

NOTE: If an incoming message contains "0x1a" in the first byte, then the message was sent from a network interface that is using a Layer 2 image. You cannot use Layer 2 network interface firmware with the OpenLDV software. For more information on the permitted uses of the OpenLDV software, see *Getting Started* on page 6.

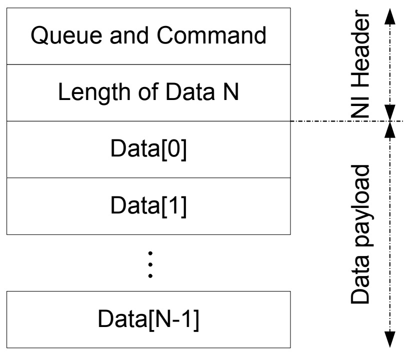


Figure 3.2 Application Buffer Structure

The queue bits indicate the path by which an incoming message was received, or by which an outgoing message should leave the network interface. For example, an outgoing message may use the standard, non-priority output queue or the priority output queue. Likewise, an incoming message might be received as a response to a pending request, or it might be an unexpected, normal, incoming message.

Commands that may be used with a specified queue include `niCOMM` for messages sent and received from the network, and `niNETMGMT` for local network management operation messages sent to the network interface.

Not all commands require queuing, however. Some commands affect the OpenLDV interface immediately, and therefore do not require queuing. These command are called *Immediate Commands* or *Non-Queue Commands*. For immediate commands, the queue selection in the NI header is not used to indicate the queue. Instead, the queue selection and command fields are combined as a single immediate command byte. A couple of examples of immediate commands are `niRESET` (to reset the network interface), and `niOFFLINE` (to set the network interface offline). For a complete list of the immediate commands you can use with OpenLDV, see *Immediate Commands* on page 27.

The OpenLDV Developer Example contains all relevant data type definitions, constants and enumerations you will require for the construction and understanding of link layer messaging. The complete application buffer structure is defined as a structure of type `ExpAppBuffer` in the `OpenLDVdefinitions.h` header file. For more information on the Open LDV Developer Example, see Chapter 4 of this document.

The remainder of this chapter describes the various message commands that are defined in the OpenLDVdefinitions.h header file included with the OpenLDV Developer Example, and the application buffer structure that you must use when sending each message command.

Downlink Commands

A downlink command is a message sent from an OpenLDV application to a network interface with the `ldv_write` function. There are several categories of downlink communication.

- Immediate commands do not require an application output buffer in the network interface, and are used to control the operation of the network interface itself. Immediate commands are sent with all queue selection bits cleared.
- Local network management commands are used to configure and control the Neuron Chip that is part of the network interface. They are sent with the `niNETMGMT` network interface command, and are not sent out on the network.
- The OpenLDV application sends application messages, network management and diagnostics messages, network variable updates, and network variable poll requests out on the network via the network interface using the `niCOMM` network interface command.
- The OpenLDV application also sends messages to the OpenLDV interface that it generates in response to uplink request messages. These include responses to uplink network variable poll messages.
- The OpenLDV application sends messages to the OpenLDV interface in response to certain uplink network management messages that it receives for processing.

For a list of the immediate commands you can use with OpenLDV Release 2.1, and descriptions of the application buffer structures those messages require, see *Immediate Commands* on page 25. For information on the application buffer structures used by all other messages you might send to a network interface with the OpenLDV, see *Application Buffer Structure* on page 33.

Uplink Commands

An uplink command is a message read from a network interface by an OpenLDV application with the `ldv_read` function. There are several classes of uplink communication.

- Immediate commands are sent to the OpenLDV application by the network interface to indicate the current operational status of the network interface.
- Local network management responses are sent to the OpenLDV application when it issues a local network management request to the network interface.
- The OpenLDV interface passes certain network management messages to the OpenLDV application for processing.
- The network interface passes uplink application messages, network variable updates and network variable poll requests to the OpenLDV interface when they are received from the network.
- The network interface also passes completion events to the OpenLDV interface at the conclusion of every downlink message initiated with the `niCOMM` network interface command. If the downlink message was a request message, the network driver also passes up any responses it may have received from the network.

For a list of the immediate commands you can use with OpenLDV Release 2.1, and descriptions of the application buffer structures those messages require, see *Immediate Commands* on page 25. For information on the application buffer structures used by all other messages you might read from a network interface with the OpenLDV API, see *Application Buffer Structure* on page 33.

Immediate Commands

Most immediate commands are just two bytes long. This includes a command byte followed by a trailing zero, which indicates there is no further data in the command (i.e. the Data Payload section of the message, as depicted in Figure 3.2, is empty). However, some of these commands, such as `niXDRVESC` (*xDriver escape command)*, do require additional data after the first two bytes.

Immediate commands may be sent to the OpenLDV interface using the `ldv_write` function. In addition, the OpenLDV Developer Example also includes an example implementation of a network interface API. The `NiSendImmediate` function, which is part of this example API, may be used to send immediate commands more conveniently.

Literals for the supported immediate commands are defined by the enumeration type definition `NI_NoQueueCmd` used in the field `NI_Hdr.q` of the application layer header. Table 3.1 lists the immediate network interface commands you can use with OpenLDV Release 2.1.

Table 3.1 Immediate Network Interface Commands

Network Interface Command	***Value***	***Uplink (U) or Downlink (D)***	***Description***
niRESET	0x50	U+D	This code is sent uplink whenever the network interface has executed a hardware or software reset. When this code is sent downlink, the network interface resets immediately.
niFLUSH_CANCEL	0x60	D	This code cancels any flush operation posted in the network interface with the niFLUSH command. The OpenLDV application should issue this command following a successful completion of the ldv_open function. The NiInit function, which is part of the OpenLDV Developer Example described in Chapter 4 of this document, may be used to open a connection to a network interface more conveniently.
niFLUSH_COMPLETE	0x60	U	This code informs the OpenLDV application that a flush operation posted with the niFLUSH command has completed.

Network Interface Command	***Value***	***Uplink (U) or Downlink (D)***	***Description***
`niONLINE`	0x70	D	This code sets the network interface's online flag, and sets the device to the online state. This code should be sent by the OpenLDV application whenever it goes online. Generally, the OpenLDV application will receive an uplink network management message indicating that it should go online, and the `niONLINE` command should be sent, from a network management tool or plug-in. This uplink message is a standard Set Node Mode network management command (message code 0x6C) with mode set to ONLINE.
`niOFFLINE`	0x80	D	This code clears the network interface's online flag, and sets the device to the offline state. This code should be sent by the OpenLDV application whenever it goes offline. Generally, the OpenLDV application will receive an uplink network management message indicating that it should go offline, and the `niOFFLINE` command should be sent, from a network management tool or plug-in. This uplink message is a standard Set Node Mode network management command (message code 0x6C) with mode set to OFFLINE.

Network Interface Command	Value	Uplink (U) or Downlink (D)	Description
`niFLUSH`	0x90	D	This code causes the network interface to enter the FLUSH state, which will cause it to send any pending downlink messages. Once all pending downlink messages are completed, the network interface will respond with the `niFLUSH_COMPLETE` command. No further downlink messages can be processed until the OpenLDV application cancels the flush state with the `niFLUSH_CANCEL` command.
`niFLUSH_IGN`	0xA0	D	Obsolete.
`niSLEEP`	0xB0	D	Obsolete.
`niSERVICE`	0xE6	D	When this code is sent downlink, the network interface sends out a service pin message in exactly the same way as it would if the service pin were grounded. Some network interfaces, such as the *i*.LON 100, do not support this command.

Network Interface Command	Value	Uplink (U) or Downlink (D)	Description
`niXDRVESC`	0xEF	U+D	This code applies to xDriver network interfaces only. Unlike most of the other immediate network interface commands, this message must contain a data field, in addition to the command byte and the length byte. The first byte of the data field denotes an xDriver-specific command. The xDriver-specific commands are described in more detail in Table 3.2. For information on other immediate commands that are specific to a particular network interface, see the documentation of that network interface. For example, the *Power Line SLTA Adapter and Power Line PSG/3 User's Guide* contains descriptions of commands specific to the SLTA/PSG interface products that can be used to control dial-up connections via a modem.

Table 3.2 describes the xDriver-specific commands that you can use with the `niXDRVESC` immediate command. The `niXDRVESC` immediate command is described in Table 3.1.

NOTE: When using these commands to enable encryption, you should be aware that encryption is very processor-intensive for the network interface, and should only be used if you are transferring highly secret information, such as LonTalk authentication keys, over the network. Other security features, such as MD5 authentication and sequence numbers, can be used to protect the link between your application and a network interface from random access and replay attacks.

Table 3.2 xDriver Specific Commands

xDriver Command	Description
LDVX_NICMD_ENCRYPTION_ON_SEND=0x02	Use this command to turn on RC4 encryption for all subsequent messages sent to the network interfaces. Once you send this command, all subsequent messages will be encrypted until the `LDVX_NICMD_ENCRYPTION_OFF_SEND` command is sent, or the session is terminated. On behalf of the calling application, the xDriver subsystem will probe the network interface to determine if it supports RC4 encryption. If the network interface does not support RC4 encryption, this command will be silently ignored. In addition, this command has no effect if encryption has already been turned on.
LDVX_NICMD_ENCRYPTION_OFF_SEND=0x03	Use this command to turn off RC4 encryption for all subsequent messages sent to the network interface. This command has no effect if encryption has already been turned off.
LDVX_NICMD_ENCRYPTION_ON_RECEIVE=0x04	Use this command to turn on RC4 encryption for all subsequent messages sent from the network interface. Once you send this command, all subsequent messages will be encrypted until the `LDVX_NICMD_ENCRYPTION_OFF_RECEIVE` command is sent, or the session is terminated. This command has no effect if encryption has already been turned on. On behalf of the calling application, the xDriver subsystem will probe the network interface to determine if it supports RC4 encryption. If the network interface does not support RC4 encryption, this command will be silently ignored. In addition, this command has no effect if encryption has already been turned on

xDriver Command	Description
LDVX_NICMD_ENCRYPTION_OFF_RECEIVE=0x05	Use this command to turn off RC encryption for all subsequent messages sent by a network interface. This command has no effect if encryption has already been turned off.

Application Buffer Structure

Most immediate commands use only the first byte—the `cmd` and `queue` fields—of the application buffer. Some immediate commands, such as `niXDRVESC`, also include additional data, as specified by that network interface command.

All other downlink and uplink message commands use the complete application buffer structure, which is described in this section. This structure is shown in Figure 3.3. The sections following Figure 3.3 describe the various parts of the application buffer structure in detail.

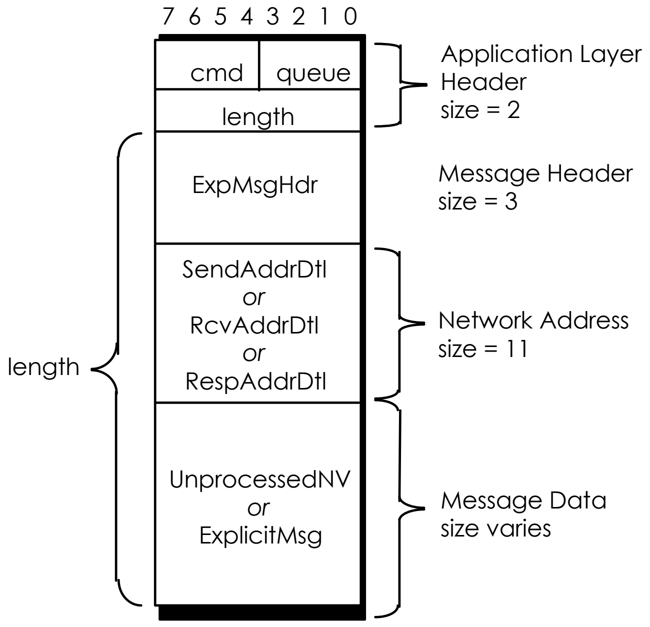

7 6 5 4 3 2 1 0
cmd
queue
length
ExpMsgHdr
SendAddrDtl *or* RcvAddrDtl *or* RespAddrDtl
UnprocessedNV *or* ExplicitMsg
Application Layer Header size = 2
Message Header size = 3
Network Address size = 11
Message Data size varies
length

Figure 3.3 Application Buffer Structure

Application Layer Header

For non-immediate commands, the application layer header contains the network interface command, and a byte indicating the length of the rest of the message. The most significant nibble of the network interface command contains the command code `niCOMM` (for network messages) or `niNETMGMT` (for local network interfaces messages), and the least significant nibble contains the queue code. These nibbles combine to form the command/queue byte, which is the network interface command.

The OpenLDV application sends these commands using the `ldv_write` function, and receives them via the `ldv_read` function. In addition, the OpenLDV Developer Example contains an example implementation of a network interface API. The `NiSendMsgWait` and `NiSendResponse` functions, part of this example API, may be used to send queued commands more conveniently, and the application-specific message dispatcher, also implemented as part of the OpenLDV Developer Example, may be used to receive these messages.

The command codes are defined by the enumeration type definition `NI_QueueCmd` used in the field `NI_Hdr.q.q_cmd` of the application layer header, and the queue codes are defined by the enumeration type definition `NI_Queue` used in the field `NI_Hdr.q.queue`. The OpenLDV Developer Example contains a utility function, `COpenLDVni::msgHdrInit`, that computes the correct value for the command/queue byte based on the address type (local or remote), the service type, and the priority attribute of the message.

Table 3.3 lists the various command codes that are used with OpenLDV Release 2.1.

Table 3.3 Command and Queue Values for the Application Layer Header

Bits 7..4	***Bits 3..0***	***Code***	***Uplink (U) or Downlink (D)***	***Description***
1		`niCOMM`	U+D	Used for messages sent to and received from the network.
2		`niNETMGMT`	U+D	Used for messages sent to and received from the network interface.
	2	`niTQ`	D	Used for downlink non-priority messages using acknowledged, request and repeated services.
	3	`niTQ_P`	D	Used for downlink priority messages using acknowledged, request and repeated services.
	4	`niNTQ`	D	Used for downlink non-priority messages using unacknowledged service, as well as responses.
	5	`niNTQ_P`	D	Used for downlink priority messages using unacknowledged service, as well as responses.
	6	`niRESPONSE`	U	Used for uplink response messages and completion codes.
	8	`niINCOMING`	U	Used for uplink messages received from the network or the network interface.

Message Header

The message header describes the various attributes of the LonTalk message contained in the data field. The message header field is defined by the structure `ExpMsgHdr`, which is displayed below. For details on the various services at layers 2 through 5 of the LonTalk protocol, see the ANSI/EIA/CEA 709.1 protocol specification.

ExpMsgHdr

7	6	5	4	3	2	1	0
0	service type		auth	tag			
Priority	Path	compl code		addr mode	alt path	pool	resp
length							

The *service type* field contains one of the following values, depending on which LonTalk protocol service is to be used for delivery of the message: `ACKD` (0) for the acknowledged messaging service, `UNACKD_RPT` (1) for the unacknowledged, repeat messaging service, `UNACKD` (2) for the unacknowledged messaging service, or `REQUEST` (3) for the request/response messaging service.

The *auth* field is set on a downlink message header if the receiver must authenticate the message using LonTalk authentication. It is set on an uplink message header if the message has been authenticated by the network interface. If authentication is not enabled on the network interface involved in the session, this field should be set to 0.

The OpenLDV application uses the *tag* field on a downlink message to correlate returned responses and completion events. For explicitly addressed messages, this may be set to any value in the range 0-14, and the same value is returned in the corresponding responses and completion events. In this case, the tag is also known as the reference ID. In a downlink implicitly addressed message, the *tag* field is used as an index into the address table of the Neuron Chip on the network interface to indicate the destination address of the message. In this case, the tag must be in the range of 0-14. For more details on the address table, see the ANSI/EIA/CEA 709.1 protocol specification.

For an uplink message, the *tag* field indicates the index into the receive transaction database for acknowledged, repeated and request messages. When the OpenLDV application generates a response to an uplink request message, it must save the tag value from the request, and return the same tag value in the downlink response message.

The *priority* field is set to indicate a message delivered with priority media access, either uplink or downlink. When the OpenLDV application generates a response to an uplink request message, it saves the priority attribute from the request, and returns the response with the same priority. If the network interface is configured without priority buffers, and a priority request is received, the OpenLDV application sets the priority bit in the response, but sends the response in a non-priority buffer.

The *path* field is set to one if the message should use the alternate path, and zero if it should use the primary path. This feature is enabled only if the *alternate path* bit, described later in this section, is set. Alternate path is a feature of certain special-purpose mode LONWORKS transceivers.

The *completion code* field is set in an uplink completion event buffer. Completion code events are returned to the OpenLDV application for every downlink (`niCOMM`) network message sent previously. The `MSG_SUCCEEDS` (1) value indicates that the message was successfully delivered. The `MSG_FAILS` (2) value indicates that the message failed to be delivered. The completion code field should be set to `MSG_NOT_COMPL` (0) for application layer buffers that are not completion events. Messages sent to the network driver with the `niNETMGMT` network interface command do not have associated completion events.

The *address mode* bit should be set to one for an explicitly addressed downlink message, in which case the network address field should have a `SendAddrDtl` structure defined in it. This is described in the next section. The address mode field should be set to zero for an implicitly addressed downlink message, in which case the network address field (also described in the next section) is ignored, although it must be present. In this case, the *tag* field is used as the index into the address table of the Neuron Chip in the network interface for the destination address. The address mode field should be set to zero for downlink responses to uplink request messages and network variable polls. For more details on the address table, see the ANSI/EIA/CEA 709.1 protocol specification. The address mode bit is ignored for local network management (`niNETMGMT`) messages.

If the *alternate path* bit is set, the message will be delivered on the path specified in the path bit, otherwise it will be delivered on the default path. See the description of the *path* bit earlier in this section for more details.

The *pool* bit should be set to zero for a downlink message.

The *response* bit should be set to one in a downlink response message, and clear otherwise. If it is set in an uplink message, the message is a response to a previously sent request.

The *length* field in the message header is distinct from the length field in the application layer header. The length field the message header tells how many bytes there are in the application buffer, which is the size of the network address field plus the size of the data field.

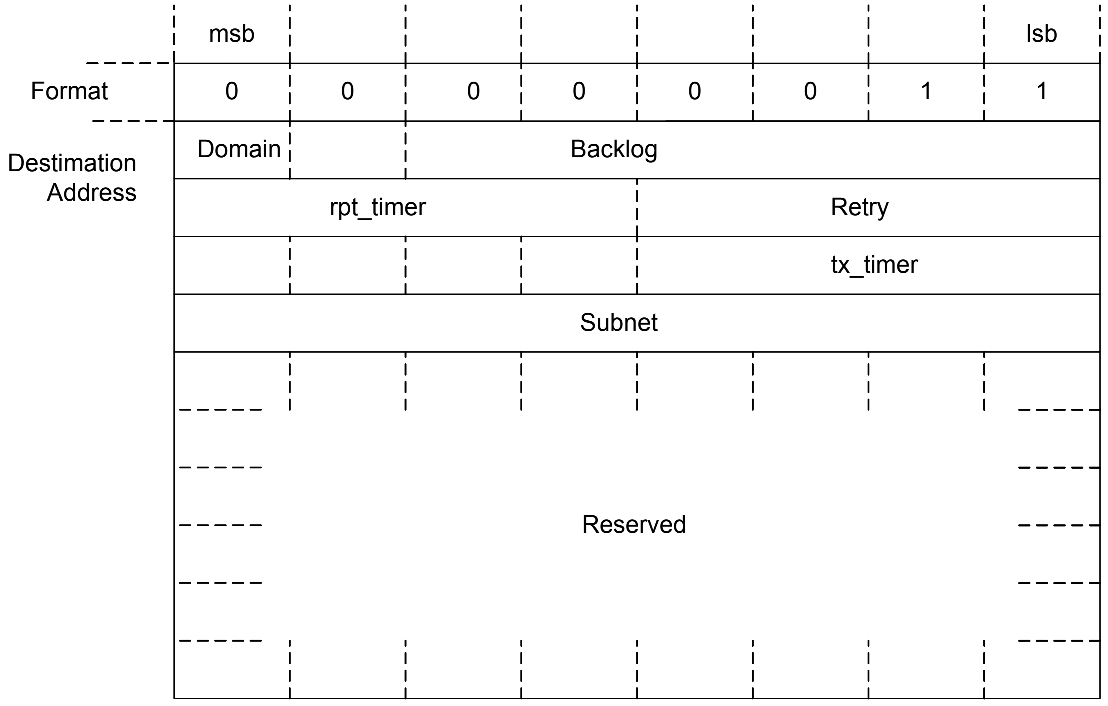
Network Address

The network address specifies the address for network (`niCOMM`) messages, which includes application messages as well as network variable messages. The network address is not used for local (`niNETMGMT`) messages, or for implicitly addressed downlink messages, but it must be present in the application buffer. The type definition `ExplicitAddr` is a union of three structures, depending on the type of message buffer. For more details on address modes and the corresponding structures, see the ANSI/EIA/CEA 709.1 protocol specification.

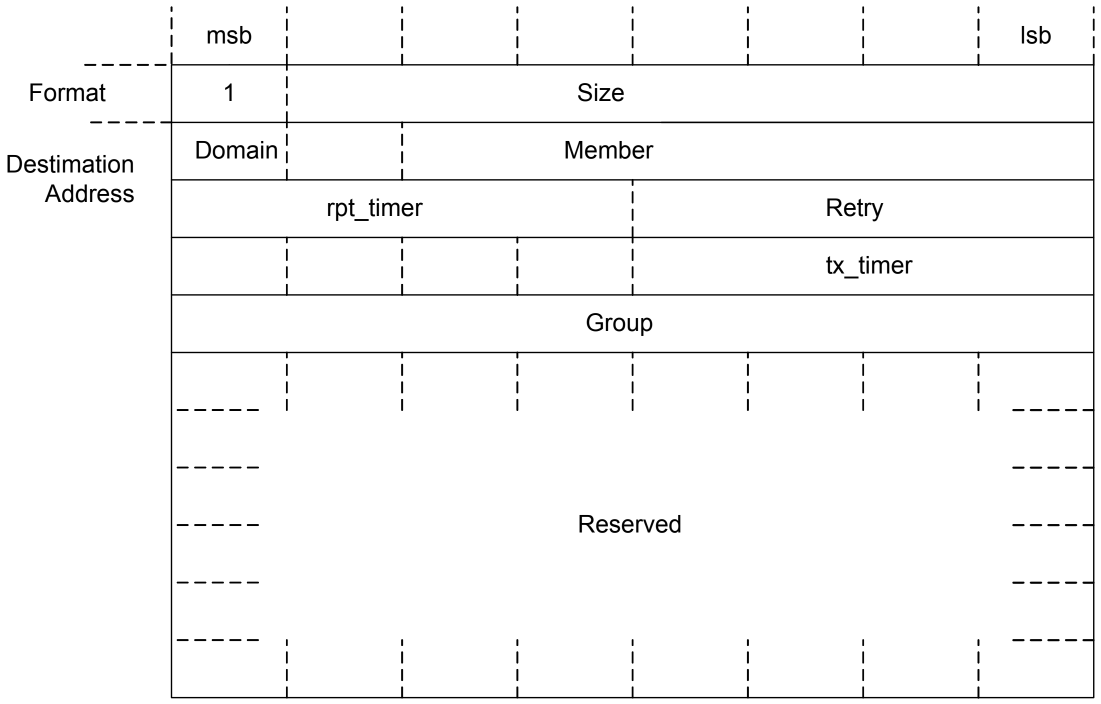
The OpenLDV Developer Example also contains an example definition of the related structures in the `OpenLDVdefinitions.h` header file.

SendAddrDtl

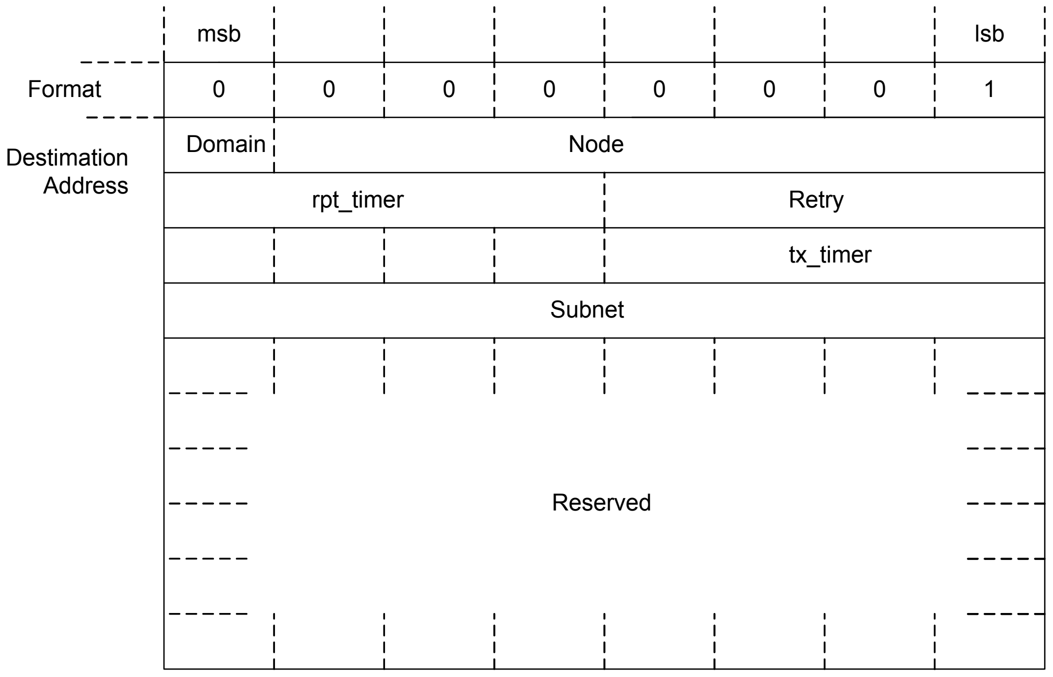
This structure is used for a downlink, explicitly addressed message, and contains the destination address of the downlink message in one of four formats, depending on the address mode. The address modes for sending explicitly addressed messages are broadcast, group, subnet/node, Neuron ID, local and implicit. The `SendAddrDtl` formats for uplink messages sent using each of these address modes are displayed below.



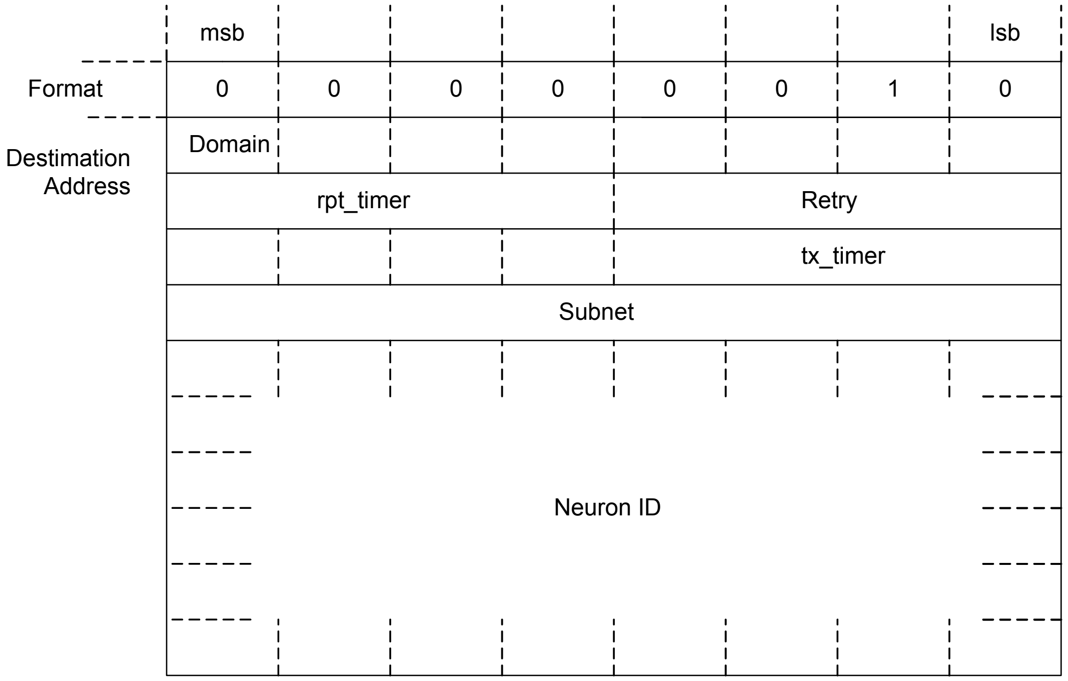
SendAddrDtl
Destination Address For Broadcast Addressing



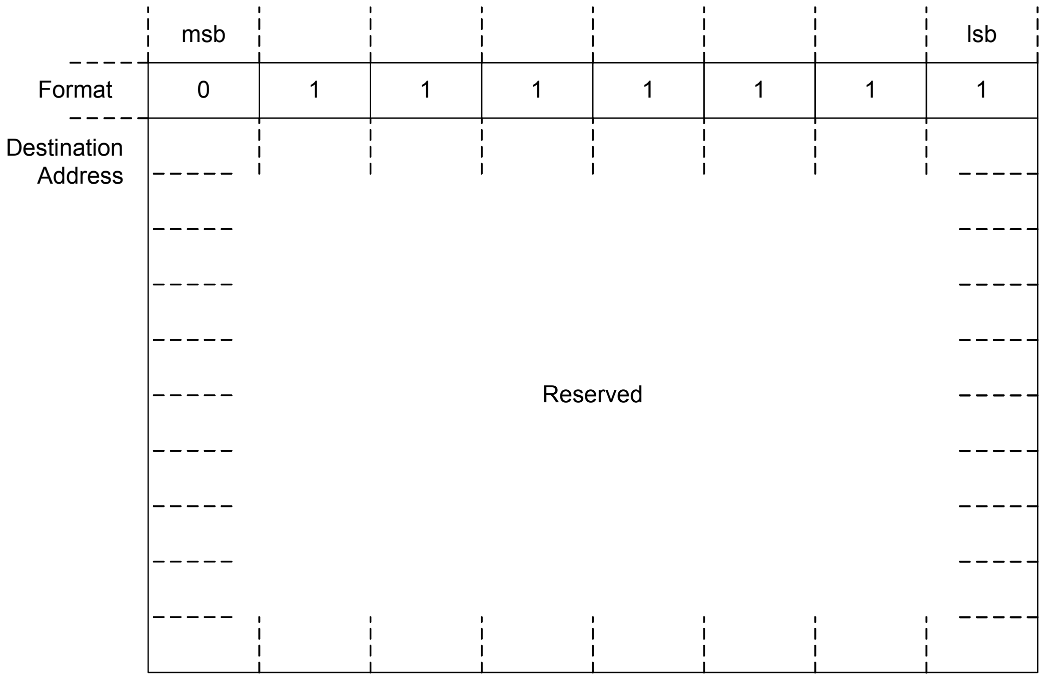
SendAddrDtl
Destination Address For Group
Addressing



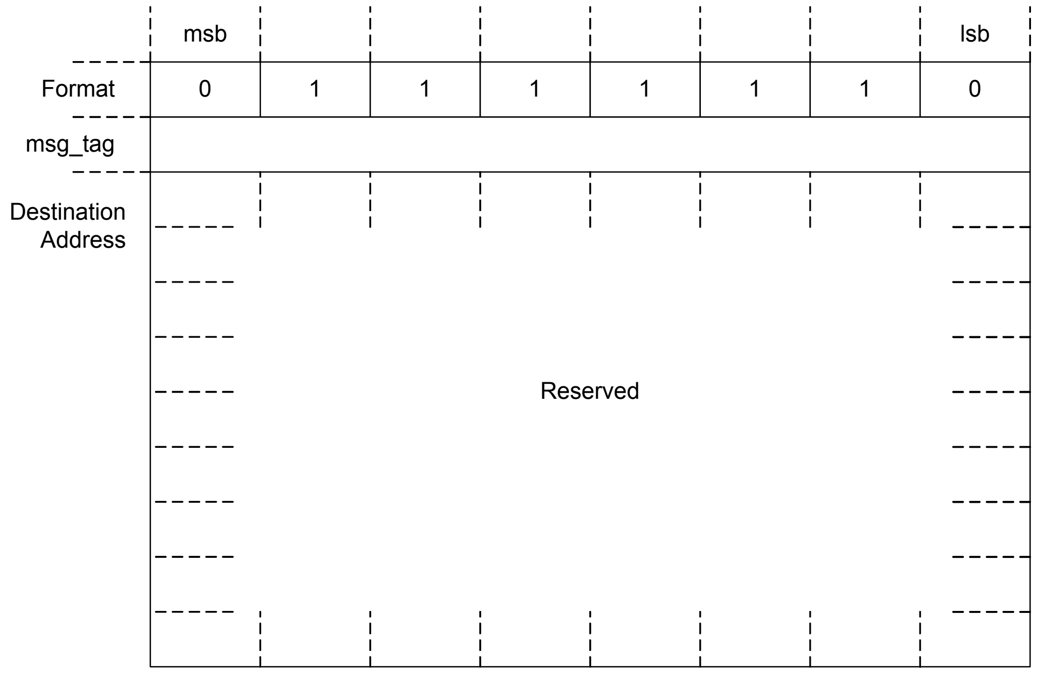
SendAddrDtl
Destination Address For Subnet/Node
Addressing



SendAddrDtl
Destination Address For Neuron ID
Addressing



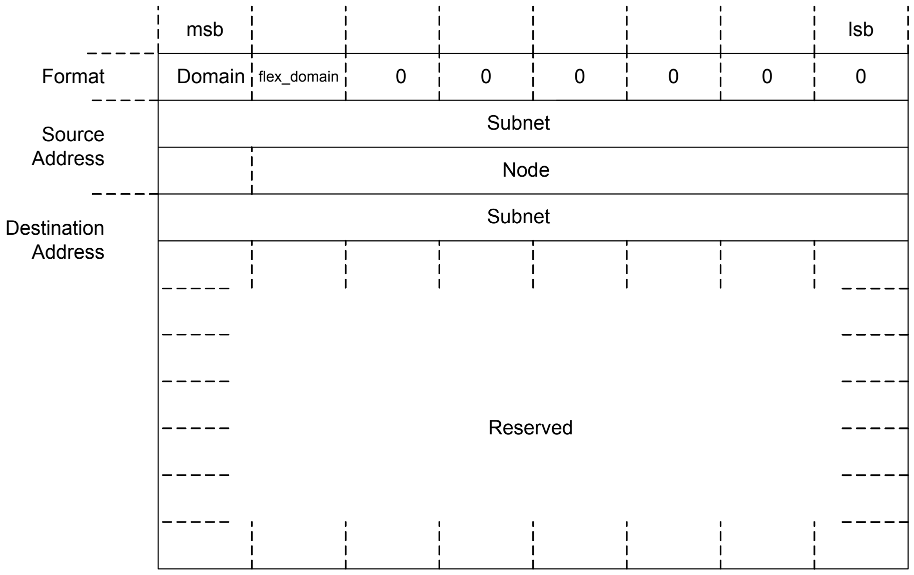
SendAddrDtl
Destination Address For Local
Addressing



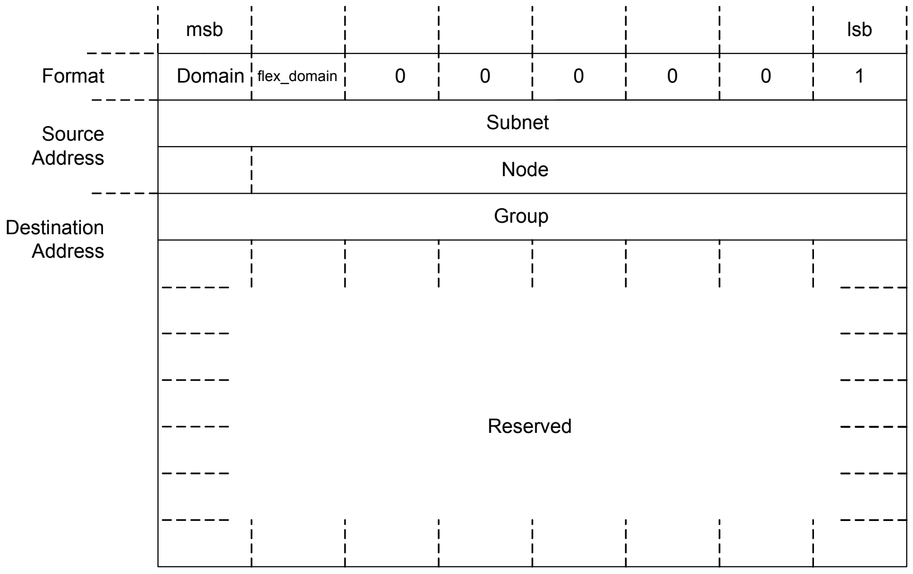
SendAddrDtl
Destination Address For Implicit Addressing

RcvAddrDtl

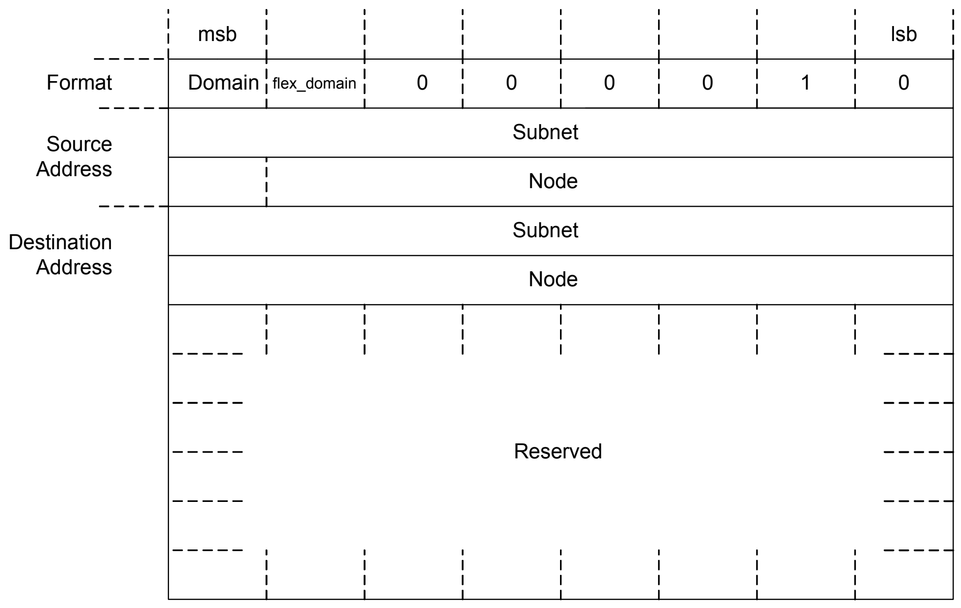
This structure is used for uplink messages addressed to the network interface and intended for the OpenLDV application. The structure contains the source address of the node sending the message, and the destination address of the uplink message in one of four formats, depending on the address mode. The address modes for received addresses are broadcast, group, subnet/node, and Neuron ID. The `RcvAddrDtl` structures for uplink messages sent using each of these address modes are displayed below.



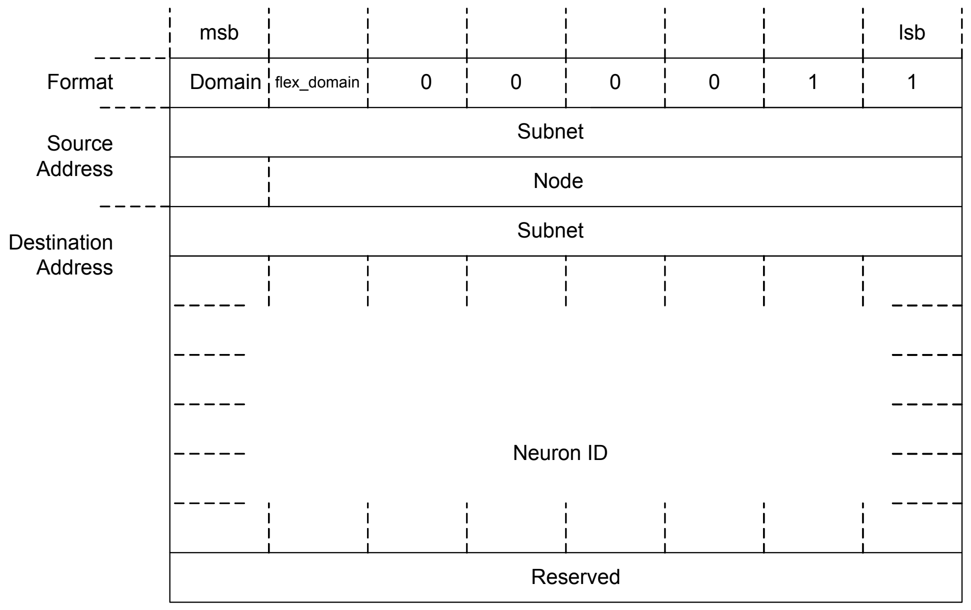
RcvAddrDtl
Received Address For Broadcast Addressing



RcvAddrDtl
Received Address For Group Addressing



RcvAddrDtl
Received Address For Subnet/Node
Addressing

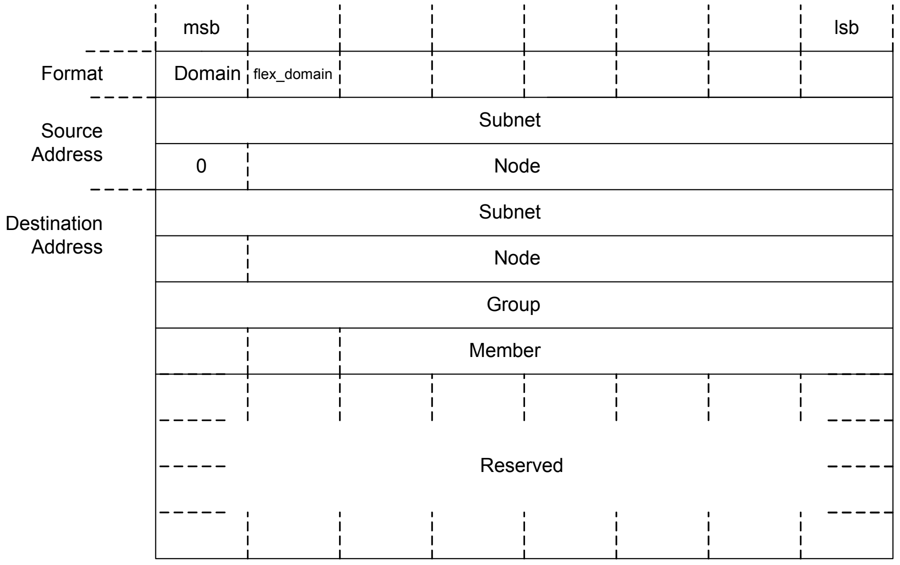


RcvAddrDtl
Received Address For Neuron ID
Addressing

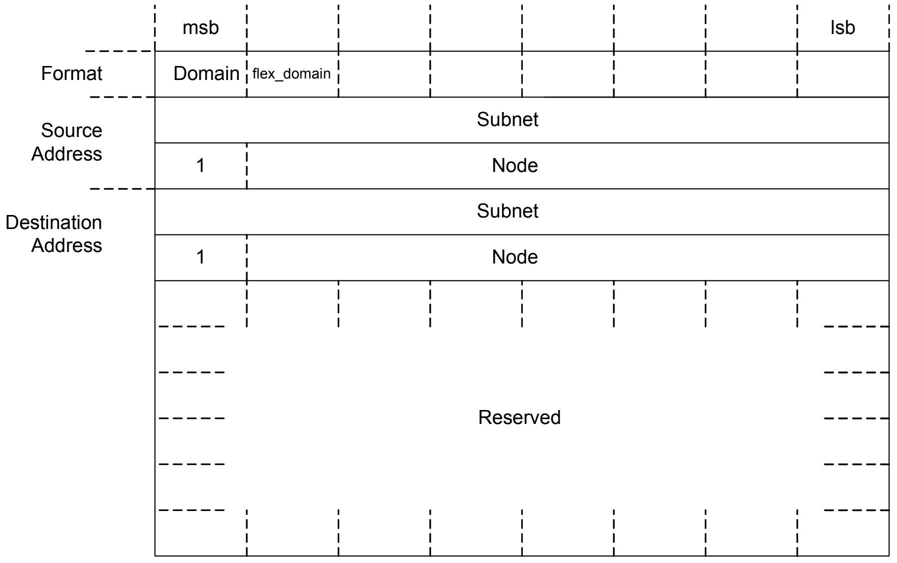
RespAddrDtl

This structure is used for an uplink message in response to a previous downlink request. This field contains the source address of the node sending the response, and the destination address of the uplink message in one of two formats, depending on the address mode. The address modes for received responses are group and subnet/node. The

`RespAddrDtl` structures for response messages sent using each of these address modes are displayed below.



RespAddrDtl
Response Address For Group
Addressing



RespAddrDtl
Response Address For Subnet/Node
Addressing

Message Data

The data field contains the application data to be transferred within a message. The format depends on the type of message, and is defined by either the `UnprocessedNV` or `ExplicitMsg` structures.

UnprocessedNV

7	6	5	4	3	2	1	0
1	dir	NV selector hi					
NV selector lo							
NV data							

Depending on the context, this form of the data field is used for network variable update messages, poll messages, poll responses or completion events. A network variable update message and a poll response contains 1-31 bytes of network variable data. A network variable poll request message and a completion event contain no data, only the selector in the first two bytes.

The *direction* bit should be set to one when polling an output network variable, and zero when updating or polling an input network variable.

An OpenLDV application that sends a downlink network variable message must retrieve the appropriate *network variable selector* from its network variable configuration table or alias table. Similarly, when an uplink network variable message arrives, the OpenLDV application looks up the network variable selector from the message in its network variable configuration table or alias table to determine which network variable was addressed.

For more extensive details on network variable configuration, messages, and alias tables, see the ANSI/EIA/CEA 709.1 protocol specification.

ExplicitMsg

7	6	5	4	3	2	1	0
0	Message code						
Message data							

Depending on the context, this form of the data field is used for downlink messages, uplink messages, or completion events. A downlink or uplink message contains 0-228 bytes of data. A completion event contains only the message code and the first byte of the data. Message codes for non-response messages are allocated as listed in Table 3.4.

Table 3.4 Message Codes for Application Messages

Message type	***Message codes (hex)***
Application message	00 .. 3F

Message type	***Message codes (hex)***
Foreign message	40 .. 4F
Network diagnostic message	50 .. 5F
Network management message	60 .. 73
Router configuration message (not used by the network interface)	74 .. 7C
Network management escape code	7D
Router far side escape code (not used by the network interface)	7E
Service pin message	7F

Sending Messages to the Network Interface

Some messages may be sent to the network interface itself. For example, the `NM_leave_domain (0x64)` message may be sent to the network interface itself using the `niNETMGMT` network interface command. This message is useful when preparing for the termination of an OpenLDV application, and causes the interface to be de-configured as the OpenLDV application terminates.

Receiving Messages from the Network Interface

Application, foreign, and network variable messages are passed unchanged to the OpenLDV application. Most network management messages received are handled by the network interface itself. However, the network management messages listed in Table 3.5 may be passed to the OpenLDV application, which should respond appropriately. See the ANSI/EIA/CEA 709.1 protocol specification for details about these network management and diagnostics messages.

The OpenLDV Developer Example also contains an example framework for recognizing and processing these messages. See the discussion of the application-specific message dispatcher in the next chapter for details.

Table 3.5 Network Management Messages Passed to the OpenLDV Application

Message	***Code***	***Comments***
Query NV Config	0x68	OpenLDV application responds with data from the network variable configuration table or alias table.
Update NV Config	0x6B	OpenLDV application writes its own network variable configuration or alias table, respectively.
Set Node Mode	0x6C	On-line and off-line only. OpenLDV application sends corresponding immediate command (`niONLINE` or `niOFFLINE`) to the network interface.

Message	*Code*	*Comments*
Wink	0x70	OpenLDV application indicates receipt of message to user, or handles a request to manage its self-documentation data.
Query SI	0x72	OpenLDV application responds with self-identification and self-documentation data.
NV Fetch	0x73	OpenLDV application responds with network variable data.

4

The OpenLDV Developer Example

This chapter introduces the OpenLDV Developer Example included with OpenLDV Release 2.1, and describes the various classes implemented in the example.

The OpenLDV Developer Example

OpenLDV Release 2.1 includes the OpenLDV Developer Example, an example application that uses the OpenLDV API. The example application will be placed in the `C:\LonWorks\OpenLDV\Examples\Example1` folder of your PC when you extract the files included in the OpenLDV Developer's Kit. The example application contains comments that should assist you when reviewing the code. This chapter describes the structure of the example application, and the different classes it contains.

The OpenLDV Developer Example is a simple, dialog-based, Windows application written in C++ with Microsoft Foundation Classes (MFC), using Microsoft Visual Studio .NET 2003. The example illustrates how a Windows application can access the OpenLDV API, and demonstrates a wide range of simple to complex network operations.

Common Definitions

The OpenLDV API functions are specified in the standard OpenLDV header file `ldv32.h`. The OpenLDV Developer Example provides additional definitions of constants, enumerations, and aggregated types in the `OpenLDVdefinitions.h` header file. These definitions are used throughout the remainder of the example application.

COpenLDVapi and COpenLDVtrace

A `COpenLDVapi` class is implemented in the example application to wrap the OpenLDV API functions. The `COpenLDVapi` class provides a simple interface via four methods: `Open, Close, Read and Write`. Compared to the standard OpenLDV functions (`ldv_*` functions), this class provides thread-safe, synchronized access to the downlink message path (`ldv_write`), and implements a reader thread `COpenLDVreader`, which queries the uplink message path (`ldv_read`) and supplies data to a protected queue. The `COpenLDVapi::Read` function queries that queue, thereby providing coordinated access to both uplink and downlink messaging.

The example application also implements a `COpenLDVtrace` class. The `COpenLDVtrace` class illustrates how an OpenLDV application may provide hooks to debugging or tracing into the low level portion of the OpenLDV application. The example implementation, `COpenLDVtrace`, provides a packet dump of all incoming and outgoing packets that is shown in the application's user interface.

The related header files, `OpenLDVapi.h` and `OpenLDVtrace.h`, contain extensive details about these classes and their usage.

COpenLDVni, Message Pumps, and Message Dispatchers

A `COpenLDVni` class is also provided. This class implements the core functions of a network interface API. The functions included in this class are `NiInit, NiSendMsgWait, NiSendImmediate, NiGetNextResponse, NiSendResponse, NiClose,` or `NiEncryption.`

The `OpenLDVni.h` header file contains extensive details about this class and its usage.

The `COpenLDVni` class also implements and controls a worker thread, `COpenLDVmessagePump`. This thread operates as a message pump, receiving and dispatching uplink messages from the `COpenLDVapi` class.

To dispatch an incoming message, a message dispatcher must understand the message, take the appropriate action local to the OpenLDV application, and then respond accordingly to the network. For example, the incoming message might describe an update to an input network variable. The message dispatcher for the application receiving this message must recognize the message as a network variable update message, and route the new network variable data to the relevant application storage. Other message types might also cause interaction with the network. For example, the application might receive a network variable fetch message. In this case, the dispatcher will have to obtain the current value of the network variable in question, and report the value to the network by constructing an appropriate response message.

The message pump thread in this example application uses the functions provided by the `COpenLDVni` and `COpenLDVapi` classes to retrieve and dispatch messages. Ultimately, these messages will be sent via a `NiDispatch` method. The `COpenLDVni` class specifies, but does not implement, such a `NiDispatch` method. Therefore, the `COpenLDVni` class is an abstract C++ class.

The OpenLDV Developer Example implements an example for an application-specific message dispatcher, `COpenLDVexampleDispatcher`, which is derived from `COpenLDVni` and that implements the `NiDispatch` function.

The example dispatcher implements handlers for a variety of messages including handlers for selected network management and diagnostics messages such as `HandleQuerySnvt, HandleSetNodeMode,` or `HandleServicePin`.

When writing an OpenLDV application, the `COpenLDVexampleDispatcher` class may serve as an example, but the dispatcher must be re-written and adopted to each specific OpenLDV application.

The header and implementation files `OpenLDVexampleDispatcher.h` and `OpenLDVexampleDispatcher.cpp`, respectively, contain extensive comments describing the details of the implementation.

Toolkits and User Interface

The OpenLDV Developer Example provides a simple user interface based on a single dialog. The `OpenLDV ExampleDlg.cpp` implementation file contains event handlers related to that user interface, such as the various “click” event handlers related to buttons. The same `COpenLDV ExampleDlg` class also provides example instantiation of the above classes.

For most operations, however, the dialog uses the `COpenLDVtools` class as a toolkit. `COpenLDVtools` provides a simple interface, implementing selected operations such as `QueryDomain, LeaveDomain`, or `UpdateDomain`. The `COpenLDvtools` class also implements a `FindDevices` function, demonstrating the implementation of multi-transaction sequences within the context of this framework.

Developer Example Diagram

Figure 4.1 describes the hierarchy of the classes introduced in this chapter:

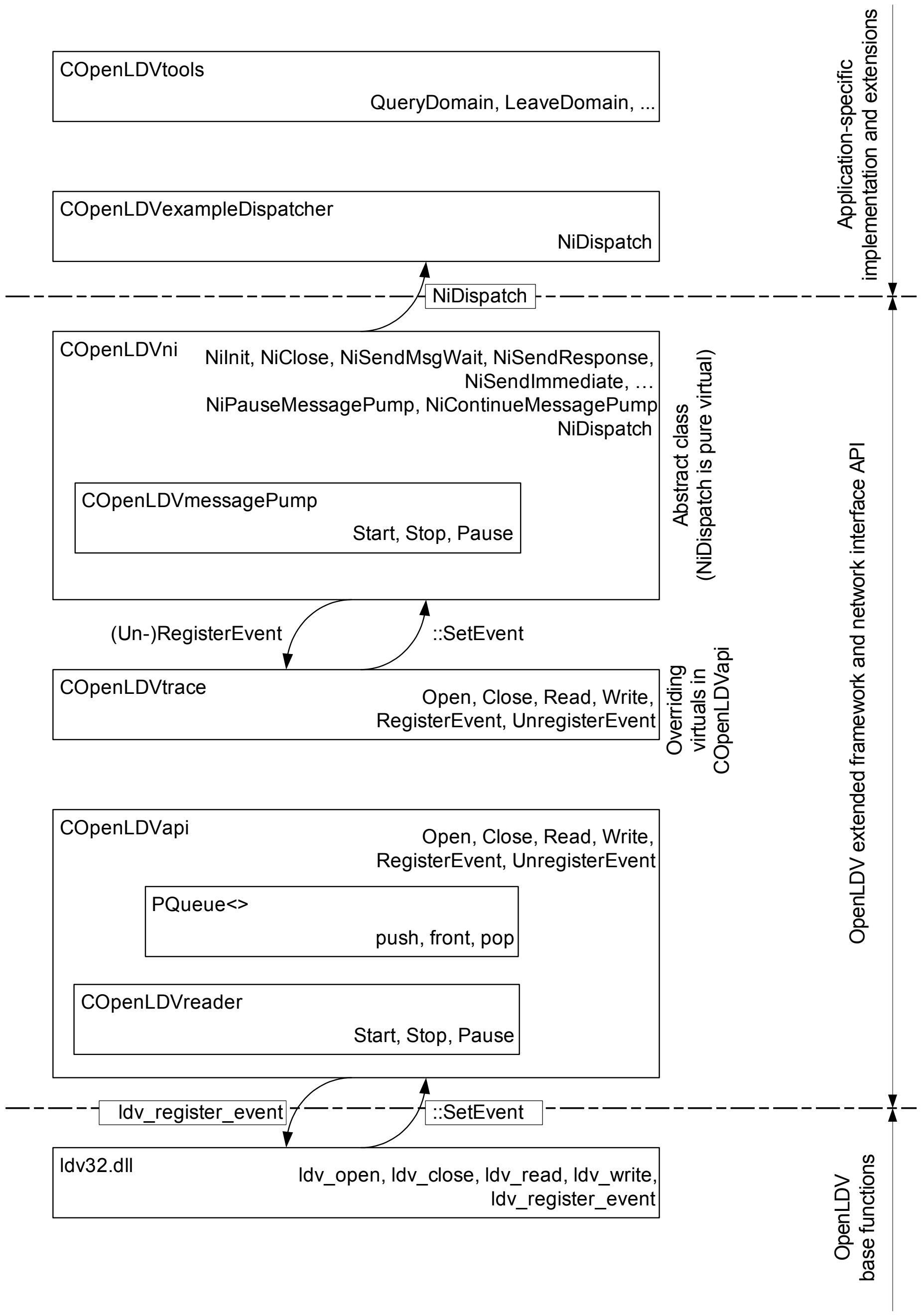

COpenLDVtools
QueryDomain, LeaveDomain, ...
COpenLDVexampleDispatcher
NiDispatch
NiDispatch
COpenLDVni
NiInit, NiClose, NiSendMsgWait, NiSendResponse, NiSendImmediate, …
NiPauseMessagePump, NiContinueMessagePump
NiDispatch
COpenLDVmessagePump
Start, Stop, Pause
(Un-)RegisterEvent
::SetEvent
COpenLDVtrace
Open, Close, Read, Write, RegisterEvent, UnregisterEvent
COpenLDVapi
Open, Close, Read, Write, RegisterEvent, UnregisterEvent
PQueue<>
push, front, pop
COpenLDVreader
Start, Stop, Pause
ldv_register_event
::SetEvent
ldv32.dll
ldv_open, ldv_close, ldv_read, ldv_write, ldv_register_event
Application-specific implementation and extensions
Abstract class (NiDispatch is pure virtual)
Overriding virtuals in COpenLDVapi
OpenLDV extended framework and network interface API
OpenLDV base functions

Figure 4.1 OpenLDV Developer Example Class Hierarchy