ECHELON®

Open LDV Programmer's Guide

Describes how to use the OpenLDV driver, an open driver for Microsoft® Windows® operating systems that enables Windows applications to send and receive low-level ISO/IEC 14908-1 messages through compatible Echelon and third-party network interfaces.

078-0275-01E

Echelon, *i*.LON, IzoT, LonMaker, LONMARK, LonTalk, LONWORKS, LNS, Neuron, NodeBuilder, 3120, 3150, LonScanner, OpenLDV, and the Echelon logo are trademarks of Echelon Corporation that may be registered in the United States and other countries.

Other brand and product names are trademarks or registered trademarks of their respective holders.

Smart Transceivers, Neuron Chips, and other OEM Products were not designed for use in equipment or systems, which involve danger to human health or safety, or a risk of property damage and Echelon assumes no responsibility or liability for use of the Smart Transceivers or Neuron Chips in such applications.

Parts manufactured by vendors other than Echelon and referenced in this document have been described for illustrative purposes only, and may not have been tested by Echelon. It is the responsibility of the customer to determine the suitability of these parts for each application.

ECHELON MAKES AND YOU RECEIVE NO WARRANTIES OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR IN ANY COMMUNICATION WITH YOU, AND ECHELON SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Echelon Corporation.

Printed in the United States of America.
Copyright © 2004, 2011, 2015 Echelon Corporation.

Echelon Corporation
www.echelon.com

Welcome

This document describes Echelon's OpenLDV™ Release 4.0 Network Driver and Software Development Kit (SDK). The OpenLDV driver is an open driver for Microsoft® Windows® operating systems that enables Windows applications to send and receive low-level ISO/IEC 14908-1 messages through compatible Echelon and third-party network interfaces. The OpenLDV SDK provides example source code that demonstrates how to use the OpenLDV driver. Supported network interfaces include local network interfaces (such as the Echelon U10, U20, or U60 USB Network Interface, PCC-10 PC Card, PCLTA-21 PC LonTalk® Adapter), and Internet-enabled network interfaces (such as the Echelon SmartServer and i.LON® 600 IP-852 Router).

The OpenLDV driver includes an extensible remote network interface (RNI) component, called *xDriver*. The OpenLDV driver uses xDriver to connect an OpenLDV application, such as an LNS® Server, to remote LONWORKS® networks. This document describes how to configure and extend xDriver, including descriptions of the software tools that you use to configure and extend xDriver, and provides programming samples.

The OpenLDV SDK provides a low-level application programming interface (API) for network applications. For most LONWORKS application developers, using Echelon's LNS Network Operating System software provides an interface to the OpenLDV driver that is easier to use and that significantly reduces the time required to develop LONWORKS applications and tools for Windows environments. Network tools that use LNS software generally have much higher performance than those that use only the OpenLDV API. You can find out more about LNS software on Echelon's Web site at www.echelon.com/lns. Contact Echelon Sales at www.echelon.com/sales if you would like assistance in determining whether you should develop your network tools with LNS software or with the OpenLDV API.

Programming with the OpenLDV API requires knowledge of the ISO/IEC 14908-1 Control Network Protocol. Echelon's implementation of the ISO/IEC 14908 Control Network Protocol is called the *LonTalk protocol*. Echelon has implementations of the LonTalk protocol in several product offerings, including the Neuron firmware, LNS Server, i.LON 600 IP-852 Router, and SmartServer Energy Managers. This document refers to the ISO/IEC 14908 Control Network Protocol as the "LonTalk protocol", although other interoperable implementations exist.

Audience

This guide is intended for software developers creating OpenLDV applications for use with OpenLDV compatible network interface products. Readers of this guide should be familiar with LONWORKS technology.

This guide is also intended for software developers creating xDriver extensions. Programming samples in this document are written in C++ and Microsoft® Visual Basic® .NET. However, extensions for xDriver can be written in any language that supports Component Object Model (COM) components or ActiveX® controls.

Developers of xDriver extensions should have programming experience in such a language, as well as familiarity with LONWORKS technology and COM concepts.

Examples

Throughout this guide, C++, Visual Basic, and other language programming samples are used to illustrate concepts. To make these samples more easily understood, they have been simplified. Error checking has generally been removed, and in some cases, the examples are only fragments that might not compile without errors or warnings.

Related Documentation

The following manuals are available from the Echelon Web site (www.echelon.com) and provide additional information that can help you develop LONWORKS and LNS applications:

- *Introduction to the LONWORKS Platform* (078-0391-01B). This manual provides an introduction to the ISO/IEC 14908 (ANSI/CEA-709.1 and EN14908) Control Network Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.
- *LNS Programmer's Guide* (078-0177-01F). This manual describes how to write powerful LNS applications, and how to get those applications to market quickly.

All of the Echelon documentation is available in Adobe® PDF format. To view the PDF files, you must have a current version of the Adobe Reader®, which you can download from Adobe at: get.adobe.com/reader.

In addition to the Echelon documentation, the following specification can help you develop LONWORKS and LNS applications:

- International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) standard ISO/IEC 14908 Control Network Protocol

You can purchase copies of ISO standards from the Information Handling Services (IHS) Global page at: global.ihs.com.

Table of Contents

Welcome ... iii
Audience ... iii
Examples ... iv
Related Documentation ... iv

Introduction ... 1

Introduction to OpenLDV Networking ... 2
Client Applications ... 3
OpenLDV Driver ... 3
Network Interfaces ... 4
Installing the OpenLDV Software ... 5
Hardware and Software Requirements ... 5
Downloading the OpenLDV Software ... 5
Installing the OpenLDV Driver ... 6
Installing the OpenLDV SDK ... 6
Getting Started with the OpenLDV Driver ... 7
Getting Started with the xDriver Component ... 8

Using the OpenLDV API ... 11

Introduction to OpenLDV Programming ... 12
OpenLDV Application Architecture ... 13
Application Layer ... 14
Presentation Layer ... 15
Overview of the OpenLDV API ... 15
Referencing the OpenLDV Component ... 16
Using Multiple Threads or Multiple Processes ... 16
The OpenLDV API ... 17
Working with Devices and Drivers ... 18
Using the OpenLDV API ... 19
ldv_close() ... 20
ldv_free_device_info() ... 20
ldv_free_driver_info() ... 21
ldv_free_matching_devices() ... 22
ldv_get_device_info() ... 22
ldv_get_driver_info() ... 23
ldv_get_matching_devices() ... 24
ldv_get_version ... 24
ldv_locate_sicb() ... 25
ldv_open() ... 26
ldv_open_cap() ... 27
ldv_read() ... 29
ldv_register_event() ... 31
ldv_set_device_info() ... 32
ldv_set_driver_info() ... 33
ldv_write() ... 34
ldv_xlate_device_name() ... 35
ldvx_open() ... 36
ldvx_register_window() ... 38
ldvx_shutdown() ... 39
Structures and Enumerations for the Device API ... 39
LDVDeviceInfo Structure ... 39
LDVDevices Structure ... 40

LdvCombineFlags Enumeration 41
LdvDeviceCaps Enumeration 42
Structures and Enumerations for the Driver API 44
LDVDriverInfo Structure 44
LdvDriverID Enumeration 45
LdvDriverType Enumeration 46
Windows Messages for Session Notifications 46
OpenLDV API Return Codes 47
Example: A Simple OpenLDV Application 54
Sending and Receiving Messages with the OpenLDV API 59
Constructing Messages 60
Application Buffer Structure 60
Layer 2 Buffer Structure 63
Application Layer Header 63
Layer 2 Header 63
NPDU 63
CRC 64
Layer 5 Buffer Structure 64
Application Layer Header 64
Message Header 64
Network Address 69
Message Data 75
Sending Messages to the Network Interface 77
Receiving Messages from the Network Interface 77
Using the Network Interface Command Interface 78
Downlink Commands 78
Uplink Commands 79
Immediate Commands 79
Network Interface Commands 79
The OpenLDV Developer Example 91
Overview 92
Common Definitions 92
COpenLDVapi and COpenLDVtrace 92
COpenLDVni, Message Pumps, and Message Dispatchers 92
Toolkits and User Interface 93
Developer Example Diagram 94
Using the xDriver Default Profile 95
Configuring an xDriver Profile 96
LNS Applications for xDriver 100
Extending xDriver 101
Extending xDriver 102
xDriver Sessions 102
Downlink Sessions 102
Uplink Sessions 106
Session Control Object 110
Authentication Key Handling 113
Setting the Current Authentication Key 113
Changing the Current Authentication Key 114
Creating a Custom Lookup Extension in C++ 115
Create a New Visual Studio Project 116
Add a COM Object 118

Implement the ILdvxLookup Interface 121
Add the Extension to the Component Category 123
Build and Register the COM Server 123
Create a Custom xDriver Profile 124
Test the Lookup Extension 126
Optional Steps 126
Creating a Custom Lookup Extension in Visual Basic 127
Create a New Visual Studio Project 127
Add a Reference to the xDriver Type Library 129
Add a COM Class 129
Delete the Project Default Class 130
Import xDriver Types to Your System Namespace 131
Implement the ILdvxLookup Interface 131
Build and Register the Lookup Extension 132
Create a Custom xDriver Profile 132
Test the Lookup Extension 134
Sample Lookup Extension Component 134
xDriver Profiles 136
Starting the Connection Broker 137

LNS Programming with xDriver 139

Downlink Sample Applications 140
Opening a Single Remote Network With xDriver 140
Opening Multiple Remote Networks for Downlink 141
Uplink Sample Application 144

Custom Network Interfaces 149

Overview 150
Working with a Custom Network Interface 150
Windows Registry Entries 152

LNS Methods and Events for xDriver Support 155

xDriver Methods and Events 156
AcceptIncomingSession 156
BeginIncomingSessionEvents 157
EndIncomingSessionEvents 157
NetworkInterfaces.Item( ) 158
OnIncomingSessionEvent 159
ReleasePendingUpdates 160

Custom Lookup Extension Component Programming 163

Overview 164
ILdvxConfigure Interface 164
SetInstance Method 164
SetOptions Method 165
ILdvxLookup Interface 166
DownlinkLookup Method 166
UpdateLookup Method 167
UplinkLookup Method 167
ILdvxSCO Interface 168
GetAdditionalDownlinkPacketHeader Method 169
GetAdditionalDownlinkPacketTrailer Method 169
GetAuthenticationFlag Method 170
GetCurrentAuthenticationKey Method 170
GetDownlinkKey Method 171

GetEncryptionType Method 172
GetLNSNetworkName Method 172
GetNextAuthenticationKey Method 173
GetSessionControlObjectID Method 173
GetUplinkKey Method 174
SetAdditionalDownlinkPacketHeader Method 174
SetAdditionalDownlinkPacketTrailer Method 175
SetAuthenticationFlag Method 175
SetCurrentAuthenticationKey Method 176
SetDownlinkKey Method 177
SetEncryptionType Method 177
SetLNSNetworkName Method 178
SetNextAuthenticationKey Method 179
SetUplinkKey Method 180
ILdvxSCO_TCP Interface 180
GetRemoteTCPAddress Method 181
GetRemoteTCPPort Method 181
SetRemoteTCPAddress Method 182
SetRemoteTCPPort Method 182
ILdvxSCO2 Interface 183
GetNeuronID Method 183

1

Introduction

This chapter introduces the OpenLDV driver and how you can use it to send and receive LonTalk messages through any OpenLDV compatible network interface.

This chapter also introduces the xDriver component.

Introduction to OpenLDV Networking

The OpenLDV driver allows a Windows application to communicate with a LONWORKS network through a locally attached network interface or a remote network interface. **Figure 1** shows the basic components for communicating with a LONWORKS network:

- One or more OpenLDV client applications (for example, an LNS Server with one or more LNS clients, the Echelon LonScanner™ Protocol Analyzer, or some other client that does not use an LNS Server), which use the OpenLDV API
- The OpenLDV driver
- One or more local network interfaces, which use a Windows device driver provided with the network interface
- One or more remote network interfaces (usually Internet enabled), which use the xDriver component of the OpenLDV driver

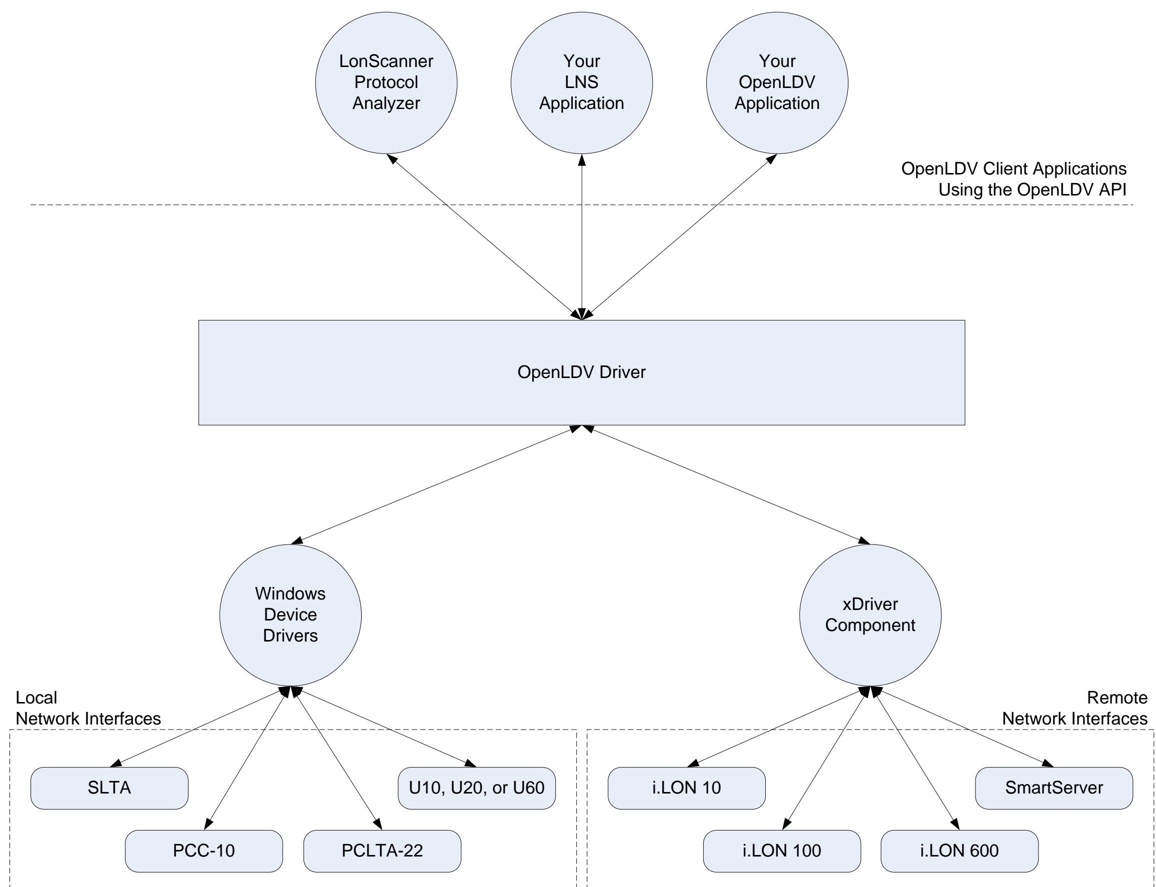


Figure 1. OpenLDV Programming Model

Client Applications

OpenLDV applications, such as the LNS Server and the LonScanner Protocol Analyzer, use the OpenLDV application programming interface (API) to communicate with LONWORKS network interfaces.

Echelon's LNS software provides a high-level interface to LONWORKS networks that simplifies managing network interfaces. The LNS software is a powerful, flexible network management platform you can use with high-performance Layer-2 or Layer-5 network interfaces, as well as with IP-852 routers (such as the i.LON 600 or SmartServer). The LNS Server provides a wide variety of network management and monitor and control services, and allows multiple client access to the same network interface.

For most customers, choosing the LNS software platform will result in a high-quality application that can be developed more quickly, requiring less knowledge of low-level details, than with other network management platforms, including the OpenLDV platform. However, the OpenLDV API provides a low-level interface for writing Windows based LONWORKS software for use with LONWORKS network interfaces.

You can use the OpenLDV API to write your own application that sends and receives messages through LONWORKS network interfaces, using either a LonTalk Layer 2 or Layer 5 interface. These messages can initialize and terminate communications with the network interface, retrieve incoming messages, or transmit outgoing messages. See Chapter 2, *Using the OpenLDV API*, on page 11, and Chapter 3, *Sending and Receiving Messages with the OpenLDV API*, on page 59, for information about these topics.

OpenLDV Driver

You can use the OpenLDV runtime with network management or monitoring and control applications. For example, for a self-installed system with fixed network addresses, you can use the OpenLDV API to create an application that sends messages to test the devices on your network. This diagnostic application could periodically send request messages to devices in the system to check their status. You can also use the OpenLDV API to create a data logging application to monitor and retrieve network variable values from the various devices on your network.

The OpenLDV 4.0 driver and API are backward-compatible with previous versions of the driver and the API (but see *Hardware and Software Requirements* on page 5 for the OpenLDV 4.0 requirements, which differ from those of prior releases).

To develop an OpenLDV application, you must 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 that you can use with the OpenLDV API. In addition, the section *Message Header* on page 64 includes some discussion of LonTalk reference IDs. See the ISO/IEC 14908 Control Network Protocol specification for detailed information about the LonTalk protocol.

Network Interfaces

A local network interface (one that is physically connected to the computer running the OpenLDV driver) uses its own Windows device driver. Echelon and third parties provide a number of network interface products; see the specific documentation about the network interface for more information.

You can also develop your own custom OpenLDV compatible local network interface. To make your network interface compatible with the OpenLDV driver, you must also develop a Windows device driver for it; see Appendix A, *Custom Network Interfaces*, on page 149, for additional information about working with a custom network interface.

A remote network interface (one that is connected to the computer through an IP network, typically the Internet) uses the xDriver component of the OpenLDV driver as a virtual device driver. The xDriver component is an extensible network driver that uses IP to establish connections between OpenLDV applications and network interfaces (such as an Echelon SmartServer).

xDriver can provide authenticated connections from an OpenLDV application to hundreds or even thousands of remote LONWORKS networks through RNI devices (such as a SmartServer). As shown in **Figure 2**, the OpenLDV application accesses the SmartServer, and the LONWORKS channel that the SmartServer is connected to, through an IP connection.

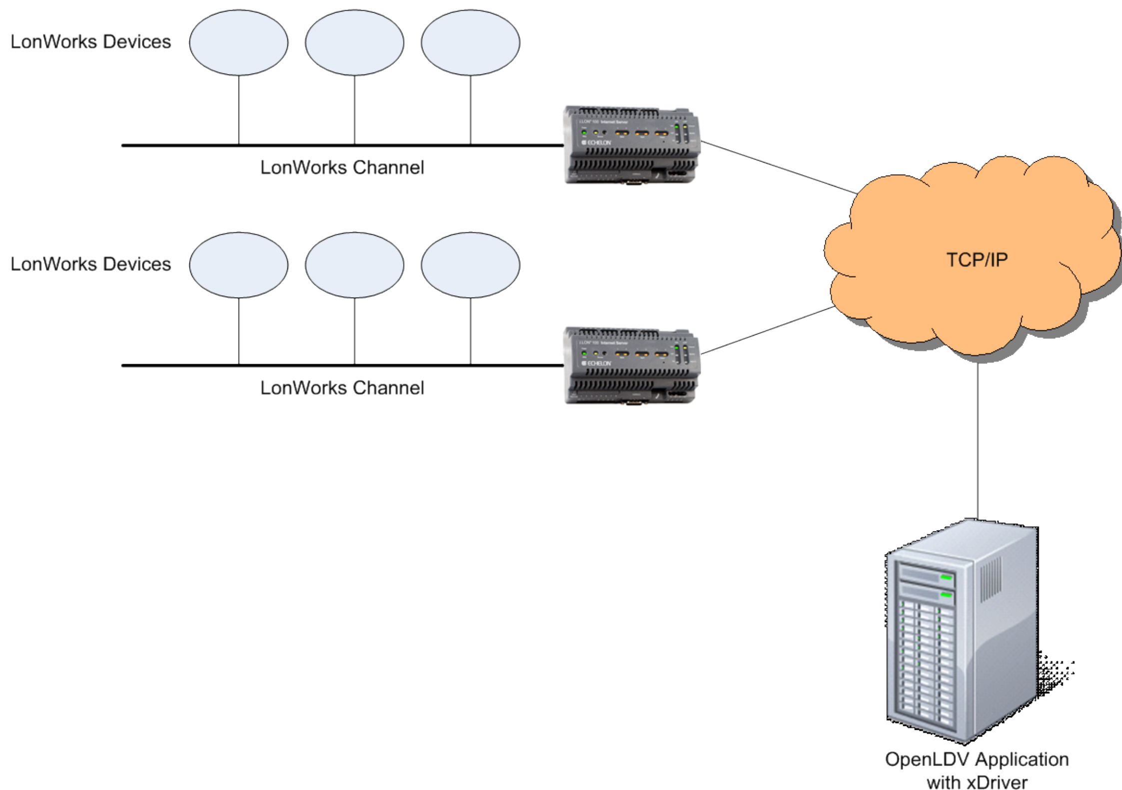


Figure 2. An OpenLDV Application Using xDriver to Manage LonWorks Devices

You configure the xDrvier component with an *xDriver profile*. An xDriver profile is a set of configuration parameters that determines how xDriver manages

connections with a group of remote networks. For example, you could have hundreds of remote networks, each of which has a SmartServer attached. At your service center, your monitoring tool could use the OpenLDV driver to listen for session requests from these networks and send messages to remote devices.

The OpenLDV driver includes a default xDriver profile. You can use the default xDriver profile for your OpenLDV applications, or you can use the xDriver Profile Editor to create a custom xDriver profile for your OpenLDV applications; see Chapter 5, *Using the xDriver Default Profile*, on page 95.

You can configure each xDriver profile to provide your application with information identifying the network interface that has requested a network session. Thus, you can program your application to quickly identify the source of the session request, and respond to a variety of different alarm conditions. See *Configuring an xDriver Profile* on page 96 for more information about the xDriver profile.

Installing the OpenLDV Software

The following sections describe the requirements for downloading the OpenLDV software, installing it, and using the OpenLDV software.

Hardware and Software Requirements

To install and use the OpenLDV 4.0 software, your computer must meet the following minimum requirements, in addition to those imposed by your operating system:

- 512 MB RAM (or the Windows operating system minimum requirement)
- Microsoft Windows 7 (32-bit or 64-bit), Windows Server 2008 R2 (64-bit), Windows Vista® (32-bit), Windows Server 2003 (32-bit), or Windows XP SP3
- Microsoft .NET Framework 3.5 SP1
- 50 MB of available hard-disk space
- 1024x768 screen resolution

Downloading the OpenLDV Software

You can download the OpenLDV driver and SDK from Echelon's Web site at www.echelon.com/downloads.

The OpenLDV driver installer (**OpenLDV400.exe**) installs the OpenLDV driver, the LONWORKS Interfaces application in the Windows Control Panel, and the xDriver Profile Editor.

You can incorporate the OpenLDV driver installer 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. The OpenLDV runtime installer is based on Microsoft Installer 3.1. If the computer onto which you are installing the OpenLDV driver uses an older version of Microsoft Installer, the OpenLDV driver installation will update the computer to use version 3.1.

See the OpenLDV 4.0 ReadMe document for updates to the OpenLDV driver documentation.

To develop an OpenLDV application or xDriver extension, install the OpenLDV SDK (**OpenLDV400-SDK.exe**) in addition to the OpenLDV driver. The OpenLDV SDK contains documentation, source files, and several examples, which you can use when developing your own OpenLDV application or xDriver extension.

See the OpenLDV SDK 4.0 ReadMe document for updates to the OpenLDV SDK documentation.

Installing the OpenLDV Driver

To install and use the OpenLDV driver, perform the following steps:

1. Download the OpenLDV Driver **OpenLDV400.exe** file from Echelon's Web site at www.echelon.com/downloads.
2. Double-click the **OpenLDV400.exe** file to begin the OpenLDV driver installation. The OpenLDV driver installer window opens.
3. Follow the installation dialogs to install the OpenLDV driver onto your computer.

If you are using a SmartServer or i.LON network interface:

1. Use the LONWORKS Interfaces application in the Windows Control Panel to specify the IP network addresses of the SmartServer or i.LON network interface. See the online help for the LONWORKS Interfaces application for information about how to use it.
2. If you are using a SmartServer or i.LON network interface and you are developing a custom xDriver profile, use the xDriver Profile Editor in the OpenLDV program folder to configure an xDriver profile for use with your OpenLDV application. For more information on the xDriver Profile Editor, see Chapter 5, *Using the xDriver Default Profile*, on page 95.
3. See the installation instructions for your network interface for any additional instructions for configuring the network interface.

Installing the OpenLDV SDK

To install and use the OpenLDV SDK, perform the following steps:

1. Download and install the OpenLDV driver, as described in *Installing the OpenLDV Driver*.
2. Download the OpenLDV SDK **OpenLDV400-SDK.exe** file from Echelon's Web site at www.echelon.com/downloads.
3. Double-click the **OpenLDV400-SDK.exe** file to begin the OpenLDV SDK installation. The OpenLDV SDK installer window opens.
4. Follow the installation dialogs to install the OpenLDV SDK onto your computer.
5. You can now use the OpenLDV SDK to write applications that use the OpenLDV API or create xDriver extensions. For information about the OpenLDV API, see Chapter 2, *Using the OpenLDV API*, on page 11. For

information about xDriver extensions, see Chapter 6, *Extending xDriver*, on page 101.

Getting Started with the OpenLDV Driver

An OpenLDV application can use a Layer 2 network interface or a Layer 5 network interface:

- *Layer 2 Network Interface* – A network interface that communicates at Layer 2 of the LonTalk protocol. This type of interface transports LonTalk packets without processing them, and does not filter by network address. It is typically used for applications that implement layers 3 through 7 of the LonTalk protocol, such as an LNS Server, and is also used for protocol analyzers that log and display network traffic. Implementing layers 3 through 7 on a Windows computer, rather than in the Neuron core or other processor of a local network interface, can provide significantly higher performance. For example, the LNS Server includes an implementation of layers 3 through 7 that provides significantly higher performance when used with a Layer 2 network interface.
- *Layer 5 Network Interface* – A network interface that communicates at Layer 5 of the LonTalk protocol. This type of interface transports incoming LonTalk packets that are addressed to the network interface, and transports outgoing packets that are addressed to other devices. It is typically used for remote network interfaces (such as a SmartServer or an i.LON network interface) because these interfaces typically implement layers 3 through 5 on a high performance processor within the network interface, and it allows an uplink session to be initiated when the host receives a particular message addressed to it. This type of interface requires handling of NI resources, such as reference IDs, at a software layer above the OpenLDV layer. For example, the LNS Server manages NI resources when used with a Layer 5 network interface.

A typical OpenLDV application uses Layer 5 interfaces so that it need not implement layer 3-5.

You can use the LONWORKS Interfaces application in the Windows Control Panel to determine if your network interface provides a Layer 2 or Layer 5 image or supports switching between Layer 2 and Layer 5:

- Echelon U10, U20, and U60 USB Network Interfaces can operate as either a Layer 2 or Layer 5 interface, switchable within an OpenLDV application
- Remote Network Interfaces (SmartServer or i.LON) can be configured to operate as a Layer 5 interface or as a read-only Layer 2 interface (for use with protocol analyzers, such as the LonScanner Protocol Analyzer)
- IP-852 devices always operate as Layer 2 interfaces, as defined by the ISO/IEC 14908-4 standard

For PCC-10, PCLTA-20, or PCLTA-21 network interfaces, **Table 1** on page 8 lists the application image that you can select to operate at either Layer 2 or Layer 5, as needed. See the documentation for your network interface for additional information.

Table 1. NI Application Settings

Network Interface	NI Application Setting for Layer 2 Image	NI Application Setting for Layer 5 Image
PCC-10	PCC10VNI	NSIPCC
PCLTA-20	PCL10VNI	NSIPCLTA
PCLTA-21	PCLTA21VNI	PCLTA21NSI

The LONWORKS Interfaces application is installed with the OpenLDV driver.

For additional information about developing an OpenLDV application, see the following chapters:

- Chapter 2, *Using the OpenLDV API*, on page 11. This chapter describes each function that is included in the OpenLDV API. It also defines guidelines for writing applications that use the OpenLDV API to access multiple network interfaces.
- Chapter 3, *Sending and Receiving Messages with the OpenLDV API*, on page 59. 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 that your OpenLDV application can send and receive with these functions, as well as the application buffer structure for each type of message.
- Chapter 4, *The OpenLDV Developer Example*, on page 91. This chapter introduces the OpenLDV Developer Example, which is installed with the OpenLDV SDK. It describes various classes implemented in the OpenLDV Developer Example. In addition to reviewing the code, you should also review the code comments in the example.

Getting Started with the xDriver Component

The xDriver component is included with the OpenLDV driver. xDriver supports scalable access to many network interfaces. The default xDriver implementation uses a Lookup component that uses the Windows Registry to store a database containing the information that it requires to connect to each device. For small-scale deployments, the Windows Registry is an efficient information store for the xDriver database.

However, for larger deployments (more than 50 network interfaces), you can improve performance by extending the default xDriver component to use a database as your information store.

If you do not plan to extend the default xDriver component to use a database, you can begin using the default xDriver component, as described in Chapter 5, *Using the xDriver Default Profile*, on page 95.

If you plan to extend the default xDriver component, see Chapter 6, *Extending xDriver*, on page 101. Most developers will not need to extend xDriver.

2

Using the OpenLDV API

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

Introduction to OpenLDV Programming

An application that uses the OpenLDV API is called an *OpenLDV application*. The communications protocol used for OpenLDV applications is the ISO/IEC 14908-1 (ANSI/CEA 709.1-B and EN14908.1) Control Network Protocol. This protocol is an international standard seven-layer protocol that has been optimized for control applications, and is based on the Open Systems Interconnection (OSI) Basic Reference Model (the OSI Model, ISO standard 7498-1). The OSI Model describes computer network communications through the seven abstract layers described in **Table 2**. The implementation of these layers in a LONWORKS device provides standardized interconnectivity for devices within a LONWORKS network.

Table 2. LONWORKS Network Protocol Layers

OSI Layer		Purpose	Services Provided
7	Application	Application compatibility	Network configuration, self-installation, network diagnostics, file transfer, application configuration, application specification, alarms, data logging, scheduling
6	Presentation	Data interpretation	Network variables, application messages, foreign frame transmission
5	Session	Control	Request/response, authentication
4	Transport	End-to-end communication reliability	Acknowledged and unacknowledged message delivery, common ordering, duplicate detection
3	Network	Destination addressing	Unicast and multicast addressing, routers
2	Data Link	Media access and framing	Framing, data encoding, CRC error checking, predictive carrier sense multiple access (CSMA), collision avoidance, priority, collision detection
1	Physical	Electrical interconnect	Media-specific interfaces and modulation schemes

Echelon's implementation of the ISO/IEC 14908 Control Network Protocol is called the *LonTalk protocol*. Echelon has implementations of the LonTalk protocol in several product offerings, including the Neuron firmware, LNS Server, SmartServers, and various network interfaces. This document refers to the ISO/IEC 14908-1 Control Network Protocol as the "LonTalk protocol", although other interoperable implementations exist.

An OpenLDV application can work with Layer 2 network interfaces, Layer 5 network interfaces, or LonScanner Protocol Analyzer interfaces. **Figure 3** shows

the seven layers of the OSI Model and which OSI layers a Layer 2 or Layer 5 network interface handles.

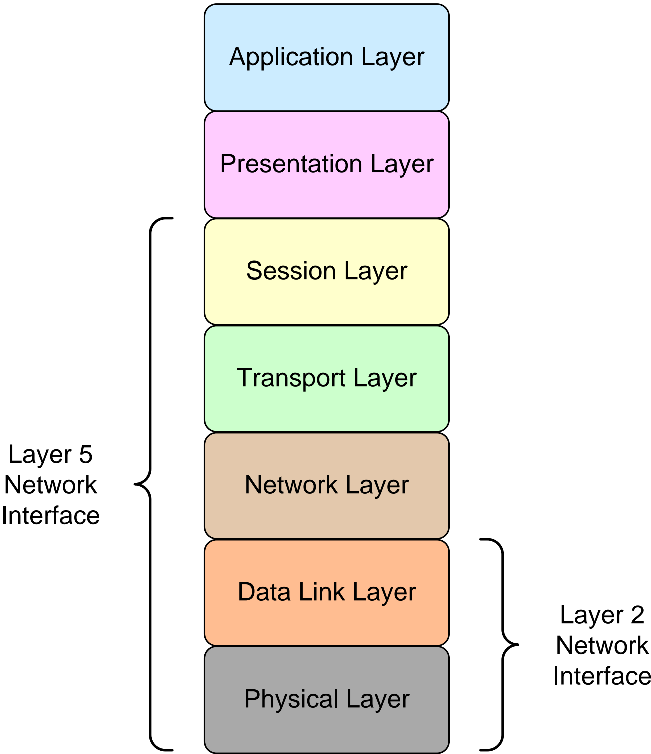


Figure 3. Network Interfaces and the Seven Layers of the OSI Model

A Layer 2 network interface handles only the first two layers of the OSI Model, and thus only sends and receives network packets; an application must implement the rest of the ISO/IEC 14908-1 protocol to communicate with the network.

A LonScanner Protocol Analyzer uses a Layer 2 network interface. The LonScanner Protocol Analyzer implements the rest of the ISO/IEC 14908-1 protocol to communicate with and analyze the network.

A Layer 5 network interface handles the first five layers of the OSI Model, and thus not only sends and receives network packets, but also implements layers 1 through 5 of the ISO/IEC 14908-1 protocol to communicate with the network.

Most OpenLDV applications use Layer 5 network interfaces so that they need not implement Layers 3 to 5 of the ISO/IEC 14908-1 protocol.

OpenLDV Application Architecture

The OpenLDV application architecture also uses the OSI Model, and provides a programming framework for communicating with the network, as shown in **Figure 4** on page 14.

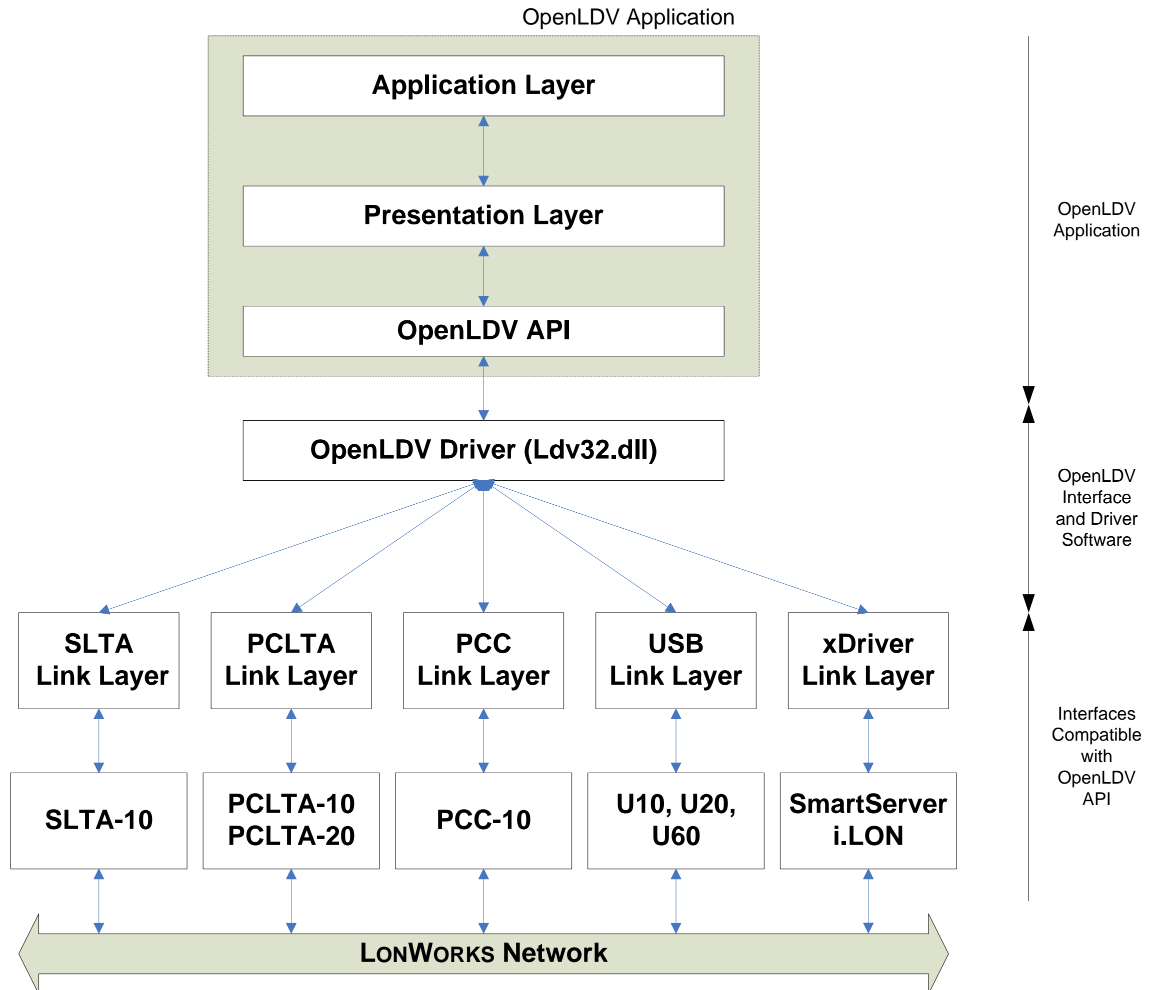


Figure 4. OpenLDV Application Architecture

An OpenLDV application implements the application and presentation layers, and uses the OpenLDV API (with the OpenLDV driver, **Ldv32.dll**) as an interface layer to communicate with a network interface. The network interfaces communicate with the LONWORKS network. Although the figure shows Echelon network interfaces, most third-party network interfaces can also use the OpenLDV driver.

An OpenLDV application that uses the OpenLDV API can establish *downlink* connections (session initiation from the application to the local or remote network interface), and it can accept *uplink* session requests (session initiation from the remote network interface, using the xDriver Broker service, to the application).

Application Layer

The application layer is responsible for sending data to the LONWORKS network through output network variables and outgoing application messages, and receiving LONWORKS network data through input network variables and incoming application messages. Typically, the bulk of an OpenLDV application's code is at the application layer.

The OpenLDV API does not include an application layer. However, the OpenLDV Developer Example demonstrates how to integrate an application layer (which dispatches incoming messages to an application-specific message dispatcher) with the OpenLDV API. For more information about the OpenLDV Developer Example and the message dispatcher it employs, see Chapter 4, *The OpenLDV Developer Example*, on page 91.

Presentation Layer

The presentation layer is responsible for translating messages between the lower layers, providing an easier-to-use presentation format used by the application layer. For example, all incoming network variable update messages from a network interface are recognized by this layer, updating the application's network variable values and notifying the application layer of the change.

The presentation layer can also manage network management messages and diagnostics services, such as the responses to Query SI network management commands (see *Receiving Messages from the Network Interface* on page 77).

The OpenLDV API does not include a presentation layer. However, the OpenLDV Developer Example demonstrates how to integrate a presentation layer with the OpenLDV API. The OpenLDV Developer Example also includes code that handles several network management commands and diagnostics.

Overview of the OpenLDV API

The OpenLDV API is implemented by a 32-bit C DLL that is compatible with both 64- and 32-bit versions of Windows. All data types defined for the API assume a 32-bit compiler; for example, pointers reference 32-bit addresses, each `enum` is a 32-bit type, `size_t` is an unsigned 32-bit type, and so on. The OpenLDV header file, **ldv32.h**, includes the following type definitions for native data types, rather than using compiler-dependent types:

```
typedef          void*   PVOID;
typedef          short   SHORT;    /*   signed 16-bit */
typedef          long    LONG;     /*   signed 32-bit */
typedef unsigned char    BYTE;     /* unsigned  8-bit */
typedef unsigned short   WORD;     /* unsigned 16-bit */
typedef unsigned long    DWORD;    /* unsigned 32-bit */
typedef          char*   LPSTR;
typedef const    char*   LPCSTR;
typedef          void*   HANDLE;
```

Each function of the OpenLDV API uses `extern "C"` and `__stdcall` calling conventions.

An alternative to the OpenLDV API is the **COpenLDVapi** class provided with the OpenLDV developer example described in Chapter 4. This class provides a COM interface with thread-safe, synchronized, access to downlink messages (**ldv_write()**). The same interface also implements a reader thread, **COpenLDVreader**, which queries uplink messages (**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 messages.

Referencing the OpenLDV Component

You can develop applications that use the OpenLDV API with any Windows application development environment that supports the use of standard Windows DLL components and (for xDriver Extensions) COM components. Echelon has tested the OpenLDV software with Microsoft Visual Studio 2008, using the Microsoft Visual C++®, Visual C#®, and Visual Basic® components.

To develop an applicatikon with the OpenLDV API, first install the OpenLDV driver and the OpenLDV SDK. During the installation, the **ldv32.h** and **ldv32.lib** files are copied to the LONWORKS **\OpenLDV SDK\Include** and **\OpenLDV SDK\Lib** folders.

To develop your OpenLDV application, include the **ldv32.h** header file in your application and link it with the **ldv32.lib** library. See your development environment's documentation for information about linking to external libraries. End users of your OpenLDV application do not need to install any of the files included in the OpenLDV SDK; they need only install the OpenLDV driver.

The OpenLDV SDK includes the OpenLDV Developer Example, which uses many of the functions described in this chapter. To compile and debug the OpenLDV Developer Example, install Microsoft Visual Studio 2008 (or later), with the Microsoft Visual C++ component. The example application is available from the **Examples & Tutorials** folder in the **Echelon OpenLDV 4.0 SDK** program folder; it is also installed as a ZIP file to the **\LonWorks\OpenLDV SDK\SourceArchive** folder. The OpenLDV Developer Example contains numerous helpful comments. See Chapter 4, *The OpenLDV Developer Example*, on page 91, for a description of the architecture of the OpenLDV Developer Example and of the different classes that it contains.

Using Multiple Threads or Multiple Processes

The OpenLDV software supports communication with multiple network interfaces at the same time, with the following restrictions:

1. A single process can access multiple network interfaces simultaneously. However, a single process should access a network interface with at most one writer thread and one reader thread. You must program your application to enforce this restriction, because 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 cannot access the same network interface simultaneously. Attempts to access the same network interface by more than one process result in the **LDVX_ACCESS_DENIED** failure code from the **ldv_open()** functions.
3. The SmartServer and i.LON network interfaces allow a single session at a time. If you attempt to open such a network interface while another session is active (usually from another computer), the call to **ldv_open()** might initially appear to have succeeded (the connection is established in the background). However, when you call **ldv_read()** or **ldv_write()** to read or write a message to the network interface, the functions return the **LDVX_READ_FAILED** or **LDVX_WRITE_FAILED** return codes, which indicate that the session has failed. See *ldv_open()* on page 26 for more information.

The OpenLDV API

This section describes the functions included in the OpenLDV API. **Table 3** summarizes these functions. See *Structures and Enumerations for the Driver API* on page 44 for descriptions of the structures and enumerations used by the OpenLDV API. See *OpenLDV API Return Codes* on page 46 for descriptions of the return codes.

Table 3. OpenLDV API Functions

Function	Description	Added in OpenLDV Version
ldv_close()	Closes an open session.	1.0
ldv_free_device_info()	Releases the resources allocated by a call to **ldv_get_device_info()** or **ldv_get_matching_devices()**.	2.0
ldv_free_driver_info()	Releases the resources allocated by a call to **ldv_get_driver_info()**.	2.0
ldv_free_matching_devices()	Releases the resources allocated by a call to **ldv_get_matching_devices()**.	2.0
ldv_get_device_info()	Retrieves information about a LONWORKS interface device.	2.0
ldv_get_driver_info()	Retrieves information about a LONWORKS interface device driver class.	2.0
ldv_get_matching_devices()	Retrieves information about the LONWORKS interface devices that match a set of capabilities.	2.0
ldv_get_version()	Retrieves the version number of the OpenLDV API.	1.0
ldv_locate_sicb()	Locates the **SICB** portion of the data within an **LdvEx** (or **SICB**) formatted message, if present.	2.0
ldv_open()	Opens a network interface device.	1.0
ldv_open_cap()	An extended version of **ldv_open()** and **ldvx_open()** that additionally allows the caller to request specific capabilities for the device.	2.0

Function	Description	Added in OpenLDV Version
ldv_read()	Reads a message from an open session.	1.0
ldv_register_event()	Registers a Windows Event object to receive notification of the availability of new messages.	1.0
ldv_set_device_info()	Creates or modifies the information about a LONWORKS interface device.	2.0
ldv_set_driver_info()	Creates or modifies the information about a LONWORKS interface device driver class.	2.0
ldv_write()	Writes a message to an open session.	1.0
ldv_xlate_device_name()	Translates a device's logical name to its physical name, that is, from the device's LONWORKS name (such as "LON1") to its Windows device name (such as "\\.\LON1.0").	1.0
ldvx_open()	Opens a network interface device, registers a Windows **HWND** object for receiving session change notifications.	1.0
ldvx_register_window()	Registers a Windows **HWND** object for receiving session change notifications.	1.0
ldvx_shutdown()	Cleanly shuts down OpenLDV and xDriver subsystem.	1.0

Working with Devices and Drivers

The OpenLDV API includes a number of functions that assist in working with devices and device drivers, for example:

- The **ldv_get_driver_info()** function allows you to identify the device driver for a particular device. With this function, an application could display a list of devices associated with a specific device driver.
- The **ldv_get_device_info()** function allows you to retrieve device information (such as the transceiver ID of the Smart Transceiver or Neuron Chip). For example, the transceiver ID allows an application to determine appropriate timer and channel settings for the device based on the available hardware.

- The **ldv_set_device_info()** function allows you to modify certain information for a device.
- The **ldv_get_matching_devices()** function allows you retrieve a list of devices that match a specified set of capabilities. For example, you can determine which devices operate at Layer 2 or Layer 5, you can determine which devices are IP-852 devices or channels, or you can determine which devices are protocol analyzers.
- The **ldv_free_device_info()**, **ldv_free_driver_info()**, and **ldv_free_matching_devices()** functions release resources allocated by the **ldv_get_device_info()**, **ldv_get_driver_info()**, and **ldv_get_matching_devices()** functions.
- The **ldv_open_cap()** function allows you to open a device with a specified capability enabled, if the capability is supported by the device. For example, you could open a U10, U20, or U60 USB network interface with Layer 2 operation or with Layer 5 operation. However, only local devices and static xDriver RNI devices that use the default xDriver lookup extension can supply their device capabilities. xDriver RNIs that use a custom lookup extension are either not available or report as having unknown capabilities.

See *The OpenLDV API* on page 17 for a description of these functions and the rest of the OpenLDV functions.

Using the OpenLDV API

The basic code flow for an OpenLDV application includes the following four functions, which comprise the basis of all OpenLDV applications:

- **ldv_open()**
- **ldv_read()**
- **ldv_write()**
- **ldv_close()**

An application can call the **ldv_read()** and **ldv_write()** functions in any order, and at any time after opening the network interface, but before closing the network interface.

The following sections describe each of the API functions in detail. The syntax for each function includes a table that describes the following information:

- Each parameter required for the function. The parameter data type does not appear in the table, but is shown in the syntax for the function.
- The direction for each parameter:
  - Input: Input parameters provide data to the OpenLDV function. You must supply an appropriate value for each input parameter.
  - Output: Output parameters provide data to your application from the OpenLDV function. You must supply an appropriately sized variable or buffer for each output parameter.
- A description of each parameter.

See *Structures and Enumerations for the Driver API* on page 44 for descriptions of the structures and enumerations used by the OpenLDV API. See *OpenLDV API Return Codes* on page 46 for descriptions of the return codes.

ldv_close()

Call this function to close a network interface that has been previously opened with the **ldv_open()** function.

Syntax

```
LDVCode ldv_close(
      LdvHandle handle
)
```

Table 4. ldv_close() Parameters

Parameter	Direction	Description
handle	Input	The network interface to be closed. This value was returned as the *handle* parameter when you opened the network interface with one of the open functions (**ldv_open()**, **ldv_open_cap()**, or **ldvx_open()**).

Remarks

Use this function to close an OpenLDV session and end communication between your application and the network interface involved in the session. This function also 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; if so, other processes on your computer can access the network interface.

Each successful **ldv_open()**, **ldv_open_cap()**, or **ldvx_open()** call (including nested ones) must have a matching **ldv_close()** call. The OpenLDV driver does not close a network interface until the last **ldv_close()** function is called. See also *ldvx_shutdown()* on page 39.

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

Recommendation: Add a delay of at least one second before you call the **ldv_open()** function after calling **ldv_close()** for a particular network interface.

ldv_free_device_info()

Call this function to release resources allocated by the **ldv_get_device_info()** function.

Syntax

```
LDVCode ldv_free_device_info(
      const LDVDeviceInfo* pDeviceInfo
)
```

Table 5. ldv_free_device_info() Parameters

Parameter	Direction	Description
pDeviceInfo	Input	A pointer to an **LDVDeviceInfo** structure (that was returned by the **ldv_get_device_info()** function) to be freed.

Remarks

Use this function to release resources allocated by the **ldv_get_driver_info()** function. This function returns **LDV_OK** if the resources are successfully released.

See *Structures and Enumerations for the Device API* on page 39 for a description of the **LDVDeviceInfo** structure.

---

ldv_free_driver_info()

Call this function to release resources allocated by the **ldv_get_driver_info()** function.

Syntax

```
LDVCode ldv_free_driver_info(
      LDVDriverInfo* pDriverInfo
)
```

Table 6. ldv_free_driver_info() Parameters

Parameter	Direction	Description
pDriverInfo	Input	A pointer to an **LDVDriverInfo** structure (that was returned by the **ldv_get_driver_info()** function) to be freed.

Remarks

Use this function to release resources allocated by the **ldv_get_driver_info()** function. This function returns **LDV_OK** if the resources are successfully released.

See *Structures and Enumerations for the Driver API* on page 44 for a description of the **LDVDriverInfo** structure.

ldv_free_matching_devices()

Call this function to release resources allocated by the **ldv_get_matching_devices()** function.

Syntax

```
LDVCode ldv_free_matching_devices(
      LDVDevices* pDevices
)
```

Table 7. ldv_free_matching_devices() Parameters

Parameter	Direction	Description
pDevices	Input	A pointer to an **LDVDevices** structure (that was returned by the **ldv_get_matching_devices()** function) to be freed.

Remarks

Use this function to release resources allocated by the **ldv_get_matching_devices()** function. This function returns **LDV_OK** if the resources are successfully released.

See *Structures and Enumerations for the Device API* on page 39 for a description of the **LDVDevices** structure.

ldv_get_device_info()

Call this function to retrieve device information about a LONWORKS interface device.

Syntax

```
LDVCode ldv_get_device_info(
            LPCSTR           szDevice,
      const LDVDeviceInfo** ppDeviceInfo
)
```

Table 8. ldv_get_device_info() Parameters

Parameter	Direction	Description
szDevice	Input	The name of the LONWORKS interface device for which you are requesting information.
ppDeviceInfo	Output	A pointer to an **LDVDeviceInfo** pointer that receives the information of the requested device.

Remarks

Use this function to retrieve device information about a LONWORKS interface device. This function returns **LDV_OK** if the device information is successfully retrieved. After you retrieve the device information and no longer need it, you must free the device information resources by calling the **ldv_free_device_info()** function.

The contents of the returned structure is constant (read-only) and cannot be modified.

See *Structures and Enumerations for the Device API* on page 39 for a description of the **LDVDeviceInfo** structure.

---

ldv_get_driver_info()

Call this function to retrieve driver information about a LONWORKS interface device driver.

Syntax

```
LDVCode ldv_get_driver_info(
      LDVDriverID     nDriverId,
      LDVDriverInfo** ppDriverInfo
)
```

Table 9. ldv_get_driver_info() Parameters

Parameter	Direction	Description
nDriverId	Input	The driver ID of the driver for which you are requesting information.
ppDriverInfo	Output	A pointer to an **LDVDriverInfo** pointer that receives the information of the requested driver.

Remarks

Use this function to retrieve driver information about a LONWORKS interface device driver. This function returns **LDV_OK** if the driver information is successfully retrieved. After you retrieve the driver information and no longer need it, you must free the driver information resources by calling the **ldv_free_driver_info()** function.

The contents of the returned structure should be treated as constant (read-only) and cannot be modified.

See *Structures and Enumerations for the Driver API* on page 44 for a description of the **LDVDriverID** enumeration values and the **LDVDriverInfo** structure.

ldv_get_matching_devices()

Call this function to retrieve information about all LONWORKS interface devices that match a set of capabilities.

Syntax

```
LDVCode ldv_get_matching_devices(
      LDVDeviceCaps   nCaps,
      LDVCombineFlags nCombine,
      LDVDevices*     pDevices
)
```

Table 10. ldv_get_matching_devices() Parameters

Parameter	Direction	Description
nCaps	Input	An **LDVDeviceCaps** value for the device capabilities to match.
nCombine	Input	The criterion for how the match should be performed. The criterion is a bitwise combination of one of the **LdvCombineFlags** values.
pDevices	Output	A pointer to an **LDVDevices** structure for the devices whose capabilities match those requested.

Remarks

Use this function to retrieve information about all LONWORKS interface devices that match a specified set of device capabilities. This function returns **LDV_OK** if the device information is successfully retrieved. After you retrieve the device information, you must free the device information resources by calling the **ldv_free_matching_devices()** function. Do not call the **ldv_free_device_info()** function for each retrieved device.

The contents of the returned structure should be treated as constant (read-only) and cannot be modified.

See *Structures and Enumerations for the Device API* on page 39 for a description of the **LDVDevices** and **LDVDeviceCaps** structures and the **LdvCombineFlags** enumeration.

ldv_get_version

Call this function to read the version number of the OpenLDV driver.

Syntax

```
LPCSTR ldv_get_version(
      VOID
)
```

Remarks

This function returns a string for the version number of the OpenLDV driver being used:

- OpenLDV 1.0 5.308.09
- OpenLDV/LNS 5.320.122
- OpenLDV 2.0 5.321.034
- OpenLDV 2.1 5.322.002
- OpenLDV 3.3 5.330.036
- OpenLDV 3.4 5.340.016
- OpenLDV 4.0 5.400.102

ldv_locate_sicb()

Call this function to locate the serial interface control block (SICB) portion of the data within an **LdvEx** (or **SICB**) formatted message, if present.

Syntax

```
LDVCode ldv_locate_sicb(
      PVOID pData,
      WORD  nDataLen,
      WORD* pnSicbOff,
      WORD* pnSicbLen
)
```

Table 11. ldv_locate_sicb() Parameters

Parameter	Direction	Description
pData	Input	A pointer to a buffer containing an **LdvEx** (or **SICB**) message.
nDataLen	Input	The length of the buffer containing the **LdvEx** (or **SICB**) message.
pnSicbOff	Output	A pointer to a variable to receive the offset (in bytes) of the start of the SICB portion of the specified message.
pnSicbLen	Output	A pointer to a variable to receive the length (in bytes) of the SICB portion of the specified message.

Remarks

Use this function to locate the **SICB** portion of the data within an **LdvEx** (or **SICB**) formatted message, if present. Data processed by most OpenLDV functions use the **SICB** format; data processed by the **ldv_open_cap()** function can request to use the extended **LdvEx** format (which includes the **SICB** data,

along with other data, such as timestamp data, that could be useful for some applications). See *Application Buffer Structure* on page 60 for a description of the **SICB** and **LdvEx** formats.

This function returns **LDV_OK** if the **SICB** data could be located and returned. If an **LdvEx** packet does not contain an **SICB** message, the error **LDV_NOT_FOUND** is returned. If the packet is not well formed (for example, too short), the error **LDV_INVALID_DATA_FORMAT** is returned.

This function accepts either **LdvEx** or **SICB** formatted messages. For an **SICB** formatted message, this function returns a zero offset and a decoded length.

ldv_open()

Call this function to establish communications between your application and a network interface. This function returns a unique handle that you can provide to the other OpenLDV functions to identify this network session.

Syntax

```
LDVCode ldv_open(
      LPCSTR     id,
      LdvHandle* handle
)
```

Table 12. ldv_open() Parameters

Parameter	**Direction**	**Description**
id	Input	The network interface with which to establish communication. For example, “LON1” could be used to identify a U10, U20, U60, PCLTA-10, or PCLTA-21 network interface. Or, “X.Default.1MainStreet” could be used to identify a SmartServer that will be opened through xDriver.
handle	Output	A pointer to a variable that receives a handle which you can use to identify the network interface with the other OpenLDV functions. This handle is valid only if the function returns **LDV_OK**. Note that zero is a valid handle.

Remarks

This function returns **LDV_OK** if the network interface is successfully opened. In this case, the function also returns a handle that you can use to identify the network interface with the other OpenLDV functions. To close the session with the network interface, use the **ldv_close()** function.

Each successful **ldv_open()**, **ldv_open_cap()**, or **ldvx_open()** call (including nested ones) must have a matching **ldv_close()** call.

For local network interfaces, after the **ldv_open()** function returns the **LDV_OK** success code, the network interface device has been initialized (see below for information about remote network interfaces). For some network interface types,

the network interface enters an initial quiet mode (flush state) after a reset. To start using such a network interface, the OpenLDV application must cancel the quiet mode with the **niFLUSH_CANCEL** immediate network interface command. For more information about immediate commands, see *Immediate Commands* on page 79.

The OpenLDV API clears old data from internal buffers during processing of the **ldv_open()** function before retrieving new data. Thus, your application does not need to perform this task.

For xDriver-based remote network interfaces that use the xDriver default lookup extension, 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 5, *Using the xDriver Default Profile*, on page 95, for more information. For xDriver-based remote network interfaces that use a custom (non-default) xDriver profile with a custom lookup extension, the name specified as the *id* parameter must exist in the custom database. See Chapter 6, *Extending xDriver*, on page 101, for more information.

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 code is returned.

Each network interface can only be part of one OpenLDV session at a time on a particular computer. If you call this function for a network interface that is being used by another process on your computer, the function will fail, and the **LDV_ACCESS_DENIED** return code is returned.

If you use xDriver to open a remote network interface while a remote client on another computer is using it, the call to **ldv_open()** might initially appear to succeed. 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 code is returned, indicating that the session has failed. The timing of the failure depends on the setting of the Synchronous Timeout field of the xDriver profile that is handling the session. For more information about xDriver profiles, see *xDriver Profiles* on page 136.

ldv_open_cap()

Call this function to establish communications between your application and a network interface. Additionally, you can request an operational mode for the network interface so that it opens in the specified mode. This 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_cap(
      LPCSTR         szDevice,
      LdvHandle*     pHandle,
      LDVDeviceCaps nDeviceCaps,
      HWND           hWnd,
      LONG           tag
)
```

Table 13. ldv_open_cap() Parameters

Parameter	Direction	Description
szDevice	Input	The network interface with which to establish communication. For example, "LON1" could be used to identify a U10, U20, U60, PCLTA-10, or PCLTA-21 network interface. Or, "X.Default.1MainStreet" could be used to identify a SmartServer that will be opened through xDriver.
pHandle	Output	A pointer to a variable that you can use to identify the network interface with the other OpenLDV functions. This handle is valid only if the function returns **LDV_OK**.
nDeviceCaps	Input	The requested operational mode for the network interface. For example, a USB network interface can be opened as a Layer 2 or a Layer 5 network interface by specifying the appropriate **LDVDeviceCaps** value (**LDV_DEVCAP_L2** or **LDV_DEVCAP_L5**).
hWnd	Input	The Windows handle for the session state change or attachment notification messages, where available (for example, from an xDriver device or a USB network interface). If **NULL**, no notification messages are sent.
tag	Input	Correlates notification messages with sessions. This tag is supplied as the **LPARAM** parameter of all session state change messages.

Remarks

This function is an extended version of the **ldv_open()** and **ldvx_open()** functions that additionally allows you to request the operational mode in which to open the specified device. If supported by the network interface, this function allows you to request Layer 2 or Layer 5 operational mode, or request that communications with the device (the **ldv_read()** and **ldv_write()** functions) use the **SICB** format or the extended **LdvEx** format (which includes the **SICB** data). See *Application Buffer Structure* on page 60 for a description of the **SICB** and **LdvEx** formats.

This function returns **LDV_OK** if the network interface is successfully opened. In this case, the function also returns a handle that you can use to identify the network interface with the other OpenLDV functions. Use the **ldv_close()** function to close the session with the network interface.

If the device does not support the requested capability, the error **LDV_CAPABILITY_NOT_SUPPORTED** is returned, and the device is not opened. Note that remote network interfaces (RNIs) are Layer 5 only. Protocol analyzer usage (supported by SmartServers and i.LON 600 RNI devices) is similar to Layer 2 (but receive-only). IP-852 channels cannot be opened by an OpenLDV application.

For local network interfaces, after the **ldv_open_cap()** function returns the **LDV_OK** success code, the network interface device has been initialized (see below for information about remote network interfaces). For some network interface types, the network interface enters an initial quiet mode (flush state) after reset. To start using the network interface, the OpenLDV application must cancel the quiet mode with the **niFLUSH_CANCEL** immediate network interface command. For more information about immediate commands, see *Immediate Commands* on page 79.

Each successful **ldv_open()**, **ldv_open_cap()**, or **ldvx_open()** call (including nested ones) must have a matching **ldv_close()** call.

For xDriver-based remote network interfaces that use the xDriver default lookup extension, the name specified as the *szDevice* parameter should match an entry created for a device with the LONWORKS Interfaces application in the Windows Control Panel. See Chapter 5, *Using the xDriver Default Profile*, on page 95, for more information. For xDriver-based remote network interfaces that use a custom (non-default) xDriver profile with a custom lookup extension, the name specified as the *id* parameter must exist in the custom database. See Chapter 6, *Extending xDriver*, on page 101, for more information.

If you do not specify a valid network interface name as the *szDevice* parameter when you call this function, or if the network interface referenced by the *szDevice* parameter cannot be found, the **LDV_INVALID_DEVICE_ID** or **LDVX_INVALID_XDRIVER** return code is returned.

Each network interface can only be part of one OpenLDV session at a time on a particular computer. If you call this function for a network interface that is being used by another process on your computer, the function will fail, and the **LDV_ACCESS_DENIED** return code is returned.

If you use xDriver to open a remote network interface while a remote client on another computer is using it, the call to **ldv_open_cap()** might initially appear to succeed. 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 code is returned, indicating that the session has failed. The timing of the failure depends on the setting of the Synchronous Timeout field of the xDriver profile that is handling the session, as well as the setting of the **TcpMaxConnectRetransmissions** parameter on the computer that is running the application. For more information about xDriver profiles, see *xDriver Profiles* on page 136.

ldv_read()

Call this function to read an uplink message from a network interface.

Syntax

```
LDVCode ldv_read(
      LdvHandle handle,
      PVOID     msg_p,
      SHORT     len
)
```

Table 14. ldv_read() Parameters

Parameter	Direction	Description
handle	Input	The network interface device to be read. This value was returned as the *handle* parameter when you opened the network interface with one of the open functions (**ldv_open()**, **ldv_open_cap()**, or **ldvx_open()**).
msg_p	Output	A pointer to a buffer allocated by your application that will receive the next uplink message. You must provide a sufficiently large buffer to receive each message. The length of this buffer is specified by the *len* parameter. For information about the different uplink messages you might read with this function, and descriptions of the application buffer structure that each one uses, see Chapter 3, *Sending and Receiving Messages with the OpenLDV API*, on page 59.
len	Input	The length of the application buffer to receive the message, in bytes. For communications that use the SICB format, the maximum length of a message is 257 bytes. When possible, use a buffer length of at least 257 bytes. For communications that use the LdvEx format, allocate additional buffer space.

Remarks

All messages from a network interface are buffered by the OpenLDV runtime 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 returns **LDV_OK** when it has successfully read a message from the network interface, or returns **LDV_NO_MSG_AVAIL** if no messages are currently available. Alternatively, you can use the **ldv_register_event()** function to set up an event to signal the receipt of each new message. For each poll loop within your application, you should call the **ldv_read()** function until you receive **LDV_NO_MSG_AVAIL**.

Most incoming messages will be responses to prior requests or unsolicited messages (such as network variable updates or application messages). Although incoming messages are buffered, the OpenLDV application must process these messages and provide suitable responses to the LONWORKS network, in a timely fashion. The acceptable duration for providing these responses depends on 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 computer running the application. Therefore, the OpenLDV application must process all incoming messages promptly, and with high priority.

The **ldv_read()** function returns **LDV_INVALID_BUF_LEN** if the specified buffer is too small to contain the next incoming message. In this case, allocate a larger buffer to receive the message, and call the function again, specifying a larger value for the *len* input parameter. For communications that use the SICB

format, the maximum length of a message is 257 bytes, and so you should use a buffer length of at least 257 bytes to guarantee that any message can be buffered. If your device uses the LdvEx format, you must allocate additional buffer space.

If the *handle* parameter is not valid, the **LDV_INVALID_DEVICE_ID** code is returned. If the network interface referenced by the *handle* parameter has not been opened by your process, then the **LDV_NOT_OPEN** code is returned if the *handle* references a local network interface. If the *handle* references a failed remote network interface, or if the session has failed, the **LDVX_READ_FAILED** code is returned.

ldv_register_event()

Call this function to register a Windows event object to be signaled whenever a message is available to be read from a network interface.

Syntax

```
LDVCode ldv_register_event(
      LdvHandle handle,
      HANDLE    event
)
```

Table 15. ldv_register_event() Parameters

Parameter	Direction	Description
handle	Input	The network interface device that will cause the Windows event object to be signaled. This value was returned as the *handle* parameter when you opened the network interface with one of the open functions (**ldv_open()**, **ldv_open_cap()**, or **ldvx_open()**).
event	Input	The Windows event object to 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 is signaled.

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 de-register a current event and register a new event, call **ldv_register_event()**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.

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

ldv_set_device_info()

Call this function to create or modify the information for a specified LONWORKS interface device. Applications that configure network interfaces (such as the Echelon LONWORKS Interfaces control panel application) can use this function.

Syntax

```
LDVCode ldv_set_device_info(
            LPCSTR          szDevice,
      const LDVDeviceInfo* pDeviceInfo
)
```

Table 16. ldv_set_device_info() Parameters

Parameter	Direction	Description
szDevice	Input	The name of the LONWORKS interface device that you are creating or modifying.
pDeviceInfo	Input	A pointer to an **LDVDeviceInfo** structure that contains the information for the created or modified device.

Remarks

Use this function to create or modify information for a LONWORKS interface device. This function returns **LDV_OK** if the device information is successfully updated. Otherwise, it returns a failure code (such as **LDV_DEVICE_INFO_INVALID**).

Before you call this function, you must initialize the **LDVDeviceInfo** structure:

- Set the *size* field equal to the size of the **LDVDeviceInfo** structure.
- The *driver* field is ignored by this function. Use the *driverId* field to specify the driver for the device.
- The *name* field is ignored by this function. The logical name of the device is read only and cannot be modified by this function.
- Set the *physName* field to the physical name of the new device, or set it to **NULL** or an empty string if the physical device name is not to be modified. For a custom network interface that uses a Windows driver,

specify the Windows device path, for example, `\\.\MYLON1.0`. See Appendix A, *Custom Network Interfaces*, on page149, for more information about custom network interfaces.

- Set the *desc* field to the description of the new device, or set it to **NULL** or an empty string if the device description is not to be modified.
- Set the *caps* and *capsMask* fields according to the capabilities of the device.
- Set the *transId* field to the transceiver ID for the device.
- Set the *driverId* field to an **LdvDriverID** value that corresponds to the driver ID of the device that you are creating or modifying, or set it to -1 if the driver ID is not to be modified.

See *Structures and Enumerations for the Device API* on page 39 for a description of the **LDVDeviceInfo** structure. See *Structures and Enumerations for the Driver API* on page 44 for a description of the **LDVDriverID** enumeration values.

To modify individual fields (read-modify-write) for an existing device, perform the following steps:

1. Retrieve the current information using the **ldv_get_device_info()** function.
2. Allocate and initialize a new **LDVDeviceInfo** structure.
3. Copy unchanging fields from the old structure into the new. For strings that are not being modified, set them to **NULL**.
4. Set the fields to be changed in the new structure.
5. Call the **ldv_set_device_info()** function.
6. Deallocate the new structure.
7. Release the old resources by calling the **ldv_free_device_info()** function.

See *Working with a Custom Network Interface* on page 150 for an example of how to use this function.

ldv_set_driver_info()

Call this function to create or modify the information for a specified LONWORKS interface device driver.

Syntax

```
LDVCode ldv_set_driver_info(
            LDVDriverID    nDriverId,
      const LDVDriverInfo* pDriverInfo
)
```

Table 17. ldv_set_driver_info() Parameters

Parameter	Direction	Description
nDriverId	Input	The driver ID of the driver that you are creating or modifying.
pDriverInfo	Input	A pointer to an **LDVDriverInfo** structure that contains the information for the created or modified driver.

Remarks

Use this function to create or modify information for a LONWORKS interface device driver. This function returns **LDV_OK** if the driver information is successfully updated. Otherwise, it returns a failure code (such as **LDV_DRIVER_INFO_INVALID**, **LDV_DRIVER_UPDATE_FAILED**, or **LDV_STD_DRIVER_TYPE_READ_ONLY**).

Before you call this function, you must initialize the **LDVDriverInfo** structure:

- Set the *size* field equal to the size of the **LDVDriverInfo** structure.
- The *id* field is ignored by this function. The driver ID is read only, and cannot be modified using this function.
- Set the *type* field to an **LdvDriverType** value that corresponds to the new driver type, or set it to -1 if the driver type is not to be modified. The driver type is read-only for standard driver IDs, and cannot be modified using this function; however, non-standard drivers can have their driver type set.
- Set the *name* field to the name of the new driver type, or set it to **NULL** or an empty string if the driver name is not to be modified.
- Set the *desc* field to the description of the new driver type, or set it to **NULL** or an empty string if the driver description is not to be modified.

See *Structures and Enumerations for the Driver API* on page 44 for a description of the **LDVDriverID** enumeration values and the **LDVDriverInfo** structure.

See *Working with a Custom Network Interface* on page 150 for an example of how to use this function.

ldv_write()

Call 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(
      LdvHandle handle,
      PVOID     msg_p,
      SHORT     len
)
```

Table 18. ldv_write() Parameters

Parameter	**Direction**	**Description**
handle	Input	The LONWORKS interface device to be written. This value was returned as the *handle* parameter when you opened the network interface with one of the open functions (**ldv_open()**, **ldv_open_cap()**, or **ldvx_open()**).
msg_p	Input	A pointer to a buffer that contains the message to be written to the network interface. For information about the different message commands that you can send with this function, and descriptions of the application buffer structure that each one requires, see Chapter 3, *Sending and Receiving Messages with the OpenLDV API*, on page 59.
len	Input	The length of the message to be written. This length 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 returns **LDV_OK** if the message is successfully written to the network interface.

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

Be sure to use the message format (**SICB** or **LdvEx**) that corresponds to the format specified when the network interface was opened.

ldv_xlate_device_name()

Call this function to retrieves the physical name of the Windows device that is associated with a (logical) network interface name.

Legacy device drivers (for example, for PCC-10 or PCLTA-21 network interfaces) can use the translated name to access the driver directly. In general, however, using the translated name is not recommended.

Syntax

```
LDVCode ldv_xlate_device_name(
      LPCSTR device_name,
      LPSTR  driver_name,
      int*   driver_name_len
)
```

Table 19. ldv_xlate_device_name() Parameters

Parameter	Direction	Description
device_name	Input	The name of the network interface.
driver_name	Output	A pointer to a buffer to receive the physical name.
driver_name_len	Input Output	A pointer to the length of the buffer that will receive the physical name. On return, set to the length of the returned name.

Remarks

This function returns **LDV_OK** if the physical name for the network interface is successfully retrieved.

ldvx_open()

Call this function to establish communications between your application and a network interface. This function also registers a Windows **HWND** object for receiving session state change notifications and returns a unique handle that you can provide to the other OpenLDV functions to identify this instance of the network interface.

Syntax

```
LDVCode ldvx_open(
      LPCSTR     id,
      LdvHandle* handle,
      HWND       hWnd,
      LONG       tag
)
```

Table 20. ldvx_open() Parameters

Parameter	Direction	Description
id	Input	The LONWORKS interface device with which to establish communication. For example, “LON1” could be used to identify a U10, U20, U60, PCLTA-10, or PCLTA-21 network interface. Or, “X.Default.1MainStreet” could be used to identify a SmartServer that will be opened through xDriver.
handle	Output	A pointer to a variable that you can use to identify the network interface with the other OpenLDV functions. This handle is valid only if the function returns **LDV_OK**.
hWnd	Input	The Windows handle for the session state change or attachment notification messages, where available (for example, from an xDriver device or a USB network interface). If **NULL**, no notification messages are sent. See *Windows Messages for Session Notifications* on page 46.
tag	Input	Correlates notification messages with sessions. This tag is supplied as the **LPARAM** parameter of all session state change messages.

Remarks

This function is an extended version of the **ldv_open()** function that additionally allows you to specify a Windows handle and tag for session state change or attachment notificatgion messages. This function returns **LDV_OK** if the network interface is successfully opened. In this case, the function also returns a handle that you can 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.

For local network interfaces, after the **ldvx_open()** function returns the **LDV_OK** success code, the network interface device has been initialized (see below for information about remote network interfaces). For some network interface types, the network interface enters an initial quiet mode (flush state) after reset. 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 about immediate commands, see *Immediate Commands* on page 79.

Each successful **ldv_open()**, **ldv_open_cap()**, or **ldvx_open()** call (including nested ones) must have a matching **ldv_close()** call.

For xDriver-based remote network interfaces that use the default xDriver lookup extension, the name specified as the *id* parameter must match an entry created for a device with the LONWORKS Interfaces application in the Windows Control Panel. See Chapter 5, *Using the xDriver Default Profile*, on page 95, for more

information. 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 code is returned.

Each network interface can only be part of one OpenLDV session at a time on a particular computer. If you call this function for a network interface that is being used by another process on your computer, the function will fail, and the **LDV_ACCESS_DENIED** return code is returned.

If you use xDriver to open a remote network interface while a remote client on another computer is using it, the call to **ldvx_open()** might initially appear to succeed. 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 code is returned, indicating that the session has failed. The timing of the failure depends on the setting of the Synchronous Timeout field of the xDriver profile that is handling the session, as well as the setting of the **TcpMaxConnectRetransmissions** parameter on the computer that is running the application. For more information about xDriver profiles, see *xDriver Profiles* on page 136.

ldvx_register_window()

Call this function to register a Windows **HWND** object for receiving session change notifications.

Syntax

```
LDVCode ldvx_register_window(
      LdvHandle handle,
      HWND      hWnd,
      LONG      tag
)
```

Table 21. ldvx_register_window() Parameters

Parameter	Direction	Description
handle	Input	The session handle for the network interface. This value was returned as the *handle* parameter when you opened the network interface with one of the open functions (**ldv_open()**, **ldv_open_cap()**, or **ldvx_open()**).
hWnd	Input	The Windows handle for the session state change or attachment notification messages, where available (for example, from an xDriver device or a USB network interface). This handle replaces any previously set handle (such as one from one of the open functions). If **NULL**, OpenLDV stops sending notifications. See *Windows Messages for Session Notifications* on page 46.

Parameter	Direction	Description
tag	Input	Correlates notification messages with sessions. This tag is supplied as the **LPARAM** parameter of all session state change messages.

Remarks

Use this function to register a Windows **HWND** object for receiving session change notifications. This handle is the same as that passed to one of the open functions (**ldv_open()**, **ldv_open_cap()**, or **ldvx_open()**).

ldvx_shutdown()

Call this function to cleanly shut down the OpenLDV driver before exiting your application.

Syntax

```
LPCSTR LDVAPI ldvx_shutdown(
      VOID
)
```

Remarks

Call this function once before allowing the application to exit to avoid delays on shutdown. After you call this function, you cannot access the OpenLDV driver again.

Structures and Enumerations for the Device API

This section describes the structures and enumerations defined for the OpenLDV device API.

LDVDeviceInfo Structure

The device API functions refer to the **LDVDeviceInfo** structured data type:

```
typedef struct LDVDeviceInfo
{
    DWORD                       size;
    const LDVDriverInfo*        driver;
    LPCSTR                      name;
    LPCSTR                      physName;
    LPCSTR                      desc;
    LDVDeviceCaps               caps;
    LDVDeviceCaps               capsMask;
    BYTE                        transId;
    LDVDriverID                 driverId;
}   LDVDeviceInfo;
```

```
/* (read-only) */
typedef const LDVDeviceInfo*    LDVDeviceInfoPtr;
```

The **LDVDeviceInfo** structure contains information that describes a specific LONWORKS interface device (identified by its name). **Table 22** describes the **LDVDeviceInfo** structure's fields.

Table 22. LDVDeviceInfo Structure

Field	Description
size	The size (in bytes) of this structure. This field **must** be set before calling any of the *set* functions that pass this structure as a parameter.
driver	A pointer to a driver information object that describe the driver for the device. See the **LDVDriverInfo** in *Structures and Enumerations for the Driver API* on page 44. Ignored by the **ldv_set_device_info()** function.
name	A string that contains the name of the logical device.
physName	A string that contains the name of the physical device, if applicable. If not applicable, returns the logical name.
desc	A string that contains the description of the device driver, if available.
caps	A combination of **LDVDeviceCaps** values that describes the capabilities of this device, where known. See **LDVDeviceCaps** in **Table 25**.
capsMask	A combination of **LDVDeviceCaps** values that describes which of the capability bits are valid. See **LDVDeviceCaps** in **Table 25**.
transId	The transceiver ID of the device, if known. A value of -1 signifies "unknown" or "don't change".
driverId	The device driver ID of the associated driver. See **LdvDriverID** in **Table 27**.

See the following functions for their use of this structure: *ldv_get_device_info()* on page 22, *ldv_set_device_info()* on page 32, and *ldv_free_device_info()* on page 20.

LDVDevices Structure

The device API functions refer to the **LDVDevices** structured data type:

```
typedef struct LDVDevices
{
    DWORD             nInfos;
    LDVDeviceInfoPtr* pInfos;

}   LDVDevices;
```

The **LDVDevices** structure contains information that describes a set of LONWORKS interface devices. **Table 23** describes the **LDVDevices** structure's fields.

Table 23. LDVDevices Structure

Field	Description
nInfos	The number of Device Info pointers in the array.
pInfos	An array of Device Info pointers. See *LDVDeviceInfo Structure* on page 39 for information about the **LDVDeviceInfo** structure.

See the following functions for their use of this structure: *ldv_get_matching_devices()* on page 24 and *ldv_free_matching_devices()* on page 22.

LdvCombineFlags Enumeration

Table 24 describes the enumerated values for the **LdvCombineFlags** device capability combination flags. The device capability combination flags specify how multiple bits are combined when determining device capability support.

Table 24. OpenLDV Device Capability Combination Flags (LdvCombineFlags)

Driver Type	Numeric Value	Description
LDV_COMBINE_DEFINITELY_ALL	0	All of the specified capabilities must definitely exist
LDV_COMBINE_POSSIBLY_ALL	1	All of the specified capabilities must possibly exist
LDV_COMBINE_DEFINITELY_ANY	2	Any of the specified capabilities must definitely exist
LDV_COMBINE_POSSIBLY_ANY	3	Any of the specified capabilities must possibly exist

The two enumeration values *_ALL specify an AND of the device capabilities, whereas the two *_ANY values specify an OR of the device capabilities.

See the *ldv_free_matching_devices()* function on page 22 for its use of this enumeration.

Example: To return all devices that are currently defined as protocol analyzers, use the **ldv_get_matching_devices()** function:

1. Set the `nCaps` to **LDV_DEVCAP_PA** to specify protocol analyzer capability.
2. Set `nCombine` to **LDV_COMBINE_DEFINITELY_ALL** (or **LDV_COMBINE_DEFINITELY_ANY**) to specify that the function should return all devices that definitely are defined as protocol analyzers.

   If you want the function to return all devices that are definitely protocol analyzers along with devices that might be protocol analyzers (devices with multiple capabilities), specify **LDV_COMBINE_POSSIBLY_ALL** (or **LDV_COMBINE_POSSIBLY_ANY**).
3. Prepare the `pDevices` output buffer for the returned results.

```
LDVCode rc = ldv_get_matching_devices(
                LDV_DEVCAP_PA,
                LDV_COMBINE_DEFINITELY_ALL,
                *myPA_List);
```

LdvDeviceCaps Enumeration

Table 25 describes the enumerated values for the **LdvDeviceCaps** device capabilities, which describes the capabilities of a LONWORKS interface device. These constants can be OR'ed together for LONWORKS interface devices that support multiple capabilities.

See *ldv_open_cap()* on page 27 for information about how to specify device capabilities.

Table 25. OpenLDV Device Capabilities (LdvDeviceCaps)

Device Capability	Numeric Value	Description
LDV_DEVCAP_UNKNOWN	0x00000000	An unknown device, or a device whose capabilities cannot be determined
LDV_DEVCAP_L5	0x00000001	The device can operate as a Layer 5 network interface.
LDV_DEVCAP_L2	0x00000002	The device can operate as a Layer 2 network interface.
LDV_DEVCAP_LWIP	0x00000010	The device can operate as an IP-852 device or channel. These types of devices are implemented within the LNS Server and cannot be opened using the OpenLDV API.
LDV_DEVCAP_PA	0x00000020	The device can operate as a protocol analyzer interface.

Device Capability	Numeric Value	Description
LDV_DEVCAP_XDRIVER	0x00000040	The device is an xDriver-based device. The network interface is physically remote from the host computer.
LDV_DEVCAP_SICB	0x00000100	The device uses SICB-formatted packets.
LDV_DEVCAP_LDVEX	0x00000200	The device uses LdvEx-formatted packets.
LDV_DEVCAP_NOSTATUS	0x00000800	The device does not support the **niSTATUS** command.
LDV_DEVCAP_SWITCHABLE	0x00001000	The device can switch between operations as a Layer 5 or a Layer 2 network interface.
LDV_DEVCAP_ATTACHABLE	0x00002000	The device can be attached or detached from the host computer, for example, by connecting to or disconnecting from a USB hub. You can receive attachment and detachment notifications for this device by registering a Windows handle using the **ldvx_register_window()** function.
LDV_DEVCAP_CURRENTLY_L5	0x00010000	The device is currently operating as a Layer 5 network interface.
LDV_DEVCAP_CURRENTLY_L2	0x00020000	The device is currently operating as a Layer 2 network interface.
LDV_DEVCAP_CURRENTLY_ATTACHED	0x20000000	The device is currently attached to the host computer. This capability applies to devices that can be physically removed and maintain their device entries remain. Such devices include U10, U20, and U60 USB network interfaces.

Device Capability	Numeric Value	Description
LDV_DEVCAP_CURRENTLY_AVAILABLE	0x40000000	The device is currently not in use by any process on this computer. If it is not is use by another computer, the device is available for use.

Structures and Enumerations for the Driver API

This section describes the structure and enumerations defined for the OpenLDV driver API.

LDVDriverInfo Structure

The driver API functions refer to the **LDVDriverInfo** structured data type:

```
typedef struct LDVDriverInfo
{
    DWORD           size;
    LDVDriverID     id;
    LDVDriverType   type;
    LPCSTR          name;
    LPCSTR          desc;
}   LDVDriverInfo;

/* (read-only) */
typedef const LDVDriverInfo*    LDVDriverInfoPtr;
```

The **LDVDriverInfo** structure contains information that describes a specific LONWORKS interface device driver (identified by its driver ID). **Table 26** describes the **LDVDriverInfo** structure's fields.

Table 26. LDVDriverInfo Structure

Field	Description
size	The size (in bytes) of this structure. This field **must** be set before calling any of the *set* functions that pass this structure as a parameter.
id	The LONWORKS interface device driver ID. See *LdvDriverID Enumeration* on page 45.
type	The device driver type. See *LdvDriverType Enumeration* on page 46.
name	A string that contains the name of the device driver.
desc	A string that contains the description of the device driver, if available.

See the following functions for their use of this structure: *ldv_get_driver_info()* on page 23, *ldv_set_driver_info()* on page 33, and *ldv_free_driver_info()* on page 21.

LdvDriverID Enumeration

Table 27 describes the enumerated values for the **LdvDriverID** driver identifier (ID). The driver ID describes the driver class of an associated network interface.

LdvDriverID enumeration values less than 127 define Echelon network interfaces. To define your own network interface type, use an enumeration value greater than 128.

Table 27. OpenLDV Driver Identifier (LdvDriverID)

Driver ID	Numeric Value	Description
LDV_DRIVERID_UNKNOWN	0	An unknown network interface type, or a type that cannot be determined
LDV_DRIVERID_ILON	1	An undetermined i.LON Ethernet Adapter
LDV_DRIVERID_ILON10	2	An i.LON 10 Ethernet Adapter
LDV_DRIVERID_ILON100	3	A SmartServer or i.LON 100 Internet Server
LDV_DRIVERID_ILON600	4	An i.LON 600 IP-852 Router
LDV_DRIVERID_LWIP	5	An IP-852 device or channel implemented by an LNS Server
LDV_DRIVERID_USBLTA	6	A U10, U20, or U60 USB Network Interface
LDV_DRIVERID_SLTA10	7	An SLTA-10 Serial LonTalk Adapter
LDV_DRIVERID_PCC10	8	A PCC-10 PC Card Network Interface
LDV_DRIVERID_PCLTA10	9	A PCLTA-10 PC LonTalk Adapter
LDV_DRIVERID_PCLTA20	10	A PCLTA-20/SMX PCI Network Interface
LDV_DRIVERID_PCLTA21	11	A PCLTA-21 PCI Network Interface
LDV_DRIVERID_TA	12	A turnaround channel
LDV_DRIVERID_RNISIM	13	A remote network interface (RNI) simulator
LDV_DRIVERID_STD_MAX	127	The maximum enumeration value for Echelon network interface types

LdvDriverType Enumeration

Table 28 describes the enumerated values for the **LdvDriverType** driver type. The driver type describes the driver type of an associated network interface.

LdvDriverType enumeration values less than 127 define types of Echelon network interface drivers. To define your own network interface type, use an enumeration value greater than 128.

Table 28. OpenLDV Driver Type (LdvDriverType)

Driver Type	Numeric Value	Description
LDV_DRIVERTYPE_UNKNOWN	0	An unknown network interface type, or a type that cannot be determined
LDV_DRIVERTYPE_LNI	1	A local network interface driver (Windows)
LDV_DRIVERTYPE_RNI	2	A remote network interface driver (xDriver)
LDV_DRIVERTYPE_USB	3	A USB LonTalk Adapter driver
LDV_DRIVERTYPE_STD_MAX	127	The maximum enumeration value for Echelon network interface types

Windows Messages for Session Notifications

Table 29 describes the defined values for the Windows messages for session change notifications used by the **ldvx_open()** and **ldvx_register_window()** functions.

Table 29. Windows Messages for Session Change Notifications

Message	Definition	Value
LDVX_WM_CLOSED	LDVX_APP + 0	34408
LDVX_WM_CONNECTING	LDVX_APP + 1	34409
LDVX_WM_ESTABLISHED	LDVX_APP + 2	34410
LDVX_WM_FAILED	LDVX_APP + 3	34411
LDVX_WM_DETACHED	LDVX_APP + 4	34412
LDVX_WM_ATTACHED	LDVX_APP + 5	34413
Note: LDVX_APP is defined as 34408 (WM_APP + 1640).		

OpenLDV API Return Codes

Table 30 describes the return codes that can be returned by the OpenLDV API functions. These codes are defined in the **LDVCode** enumeration.

Table 30. OpenLDV Return Codes

Return Code	Numeric Value	Description
LDV_OK	0	The operation completed successfully.
LDV_NOT_FOUND	1	This code is returned if you call any of the open functions to open a LONWORKS interface device, but you do not specify a valid device name as the *id* parameter, or the device referenced by the *id* parameter cannot be found. This code is also returned for the **ldv_locate_sicb()** function if an **LdvEX** packet does not contain an **SICB** message.
LDV_ALREADY_OPEN	2	This return code is obsolete.
LDV_NOT_OPEN	3	The LONWORKS interface device is not open. This code is returned if you use the **ldv_read()** or **ldv_write()** functions to read or write a message to a network interface device, or if you use the **ldv_close()** function to close a session with a network interface, and the network interface has not yet been opened with the **ldv_open()** function (or the network interface has already been closed).

Return Code	Numeric Value	Description
LDV_DEVICE_ERR	4	This code is returned if a function fails to execute as a result of a failure to communicate with the network driver. Call **ldv_close()** to close the network interface and release the resources assigned to the network driver. Then, re-open the network interface with one of the open functions.
LDV_INVALID_DEVICE_ID	5	This code is returned if you specify an invalid device name when opening a LONWORKS interface device with one of the open functions, or an invalid handle when using any of the other OpenLDV functions. Ensure that the *id* input parameter for the name of the device is valid.
LDV_NO_MSG_AVAIL	6	No message is available to be read. This code is 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 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 is returned if you call **ldv_write()**, and there is no available buffer on the local network interface to which to write the message. Wait until a buffer becomes available and try writing the message again.

Return Code	Numeric Value	Description
LDV_NO_RESOURCES	8	No resources are available. This code is returned if the OpenLDV API has assigned too many session handles, or if the computer running your application has memory allocation problems. Close any non-essential processes running on your computer and try the operation again.
LDV_INVALID_BUF_LEN	9	This code is returned if you call **ldv_read()** to read a message from a LONWORKS interface device, and the specified buffer is not big enough to contain the next incoming message. Allocate a larger buffer to receive the message and then call **ldv_read()** again with a larger value for the *len* input parameter. The message remains as the next incoming message until you successfully read it with the **ldv_read()** function. **Recommendation**: Allocate a buffer of at least 257 bytes (the maximum size of an SICB format incoming message) each time you call **ldv_read()**. If your device uses the LdvEx format, you must allocate additional buffer space.
LDV_NOT_ENABLED	10	This return code is obsolete.
LDVX_INITIALIZATION_FAILED	11	The remote network interface (RNI) could not be initialized. Generally, this code is returned if there are configuration problems on the network interface that you are opening or on the computer that is running your application.

Return Code	Numeric Value	Description
LDVX_OPEN_FAILED	12	The remote network interface (RNI) could not be opened.
LDVX_CLOSE_FAILED	13	The remote network interface (RNI) 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, close the current session and start a new one, because the current session might 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, close the current session and start a new one, because the current session might have failed.
LDVX_REGISTER_FAILED	16	The application failed to register the Windows event object for event notification.

Return Code	Numeric Value	Description
LDVX_INVALID_XDRIVER	17	This code is returned if you attempt to open an xDriver LONWORKS interface device with the **ldv_open()** function, and the xDriver lookup extension component fails to find that network interface. For devices that use the default profile, use the LONWORKS Interfaces application in the Windows Control Panel to verify that the network interface referenced by the *id* parameter exists. For information about lookup extension components, see Appendix C, *Custom Lookup Extension Component Programming*, on page 163.
LDVX_DEBUG_FAILED	18	This return code is reserved.
LDVX_ACCESS_DENIED	19	This code is returned if you call **ldv_open()** to initialize a LONWORKS interface device that is already opened by another process on your computer. OpenLDV does not support concurrent access to network interfaces between multiple processes on the same computer. For more information on this, see *Using Multiple Threads or Multiple Processes* on page 16.
LDV_CAPABLE_DEVICE_NOT_FOUND	20	No OpenLDV LONWORKS interface device was found for the **ldv_get_matching_devices()** or **ldv_open_cap()** function.
LDV_NO_MORE_CAPABLE_DEVICES	21	No additional OpenLDV devices were found for the **ldv_get_matching_devices()** function.

Return Code	Numeric Value	Description
LDV_CAPABILITY_NOT_SUPPORTED	22	The capability specified for the **ldv_open_cap()** function is not supported by the device.
LDV_INVALID_DRIVER_INFO	23	The driver information specified for the **ldv_set_driver_info()** function is not valid.
LDV_INVALID_DEVICE_INFO	24	The device information specified for the **ldv_set_device_info()** function is not valid.
LDV_DEVICE_IN_USE	25	The device is in use and cannot be opened with any of the open functions.
LDV_NOT_IMPLEMENTED	26	The OpenLDV interface is not implemented on the LONWORKS interface device being opened.
LDV_INVALID_PARAMETER	27	An invalid parameter was specified.
LDV_INVALID_DRIVER_ID	28	The driver identifier is not valid.
LDV_INVALID_DATA_FORMAT	29	This code is returned for the **ldv_locate_sicb()** function if the data packet is not well formed (for example, too short).
LDV_INTERNAL_ERROR	30	The OpenLDV API experienced in an internal error. Contact Echelon Support.
LDV_EXCEPTION	31	The OpenLDV API experienced in an internal error. Contact Echelon Support.
LDV_DRIVER_UPDATE_FAILED	32	The driver information specified for the **ldv_set_driver_info()** function could not be updated.

Return Code	Numeric Value	Description
LDV_DEVICE_UPDATE_FAILED	33	The device information specified for the **ldv_set_device_info()** function could not be updated.
LDV_STD_DRIVER_TYPE_READ_ONLY	34	The driver information specified for the **ldv_set_driver_info()** function could not be updated because the driver type is read only.
LDV_OUTPUT_BUFFER_SIZE_MISMATCH	40	Output buffer sizes (for both priority and non-priority buffers) must be the same. Applies to APP/NET buffers on the network interface.
LDV_INVALID_BUFFER_PARAMETER	41	The specified buffer parameter is not valid (for example, the specified size is too large). Applies to APP/NET buffers on the network interface.
LDV_INVALID_BUFFER_COUNT	42	The specified buffer count is not valid. Applies to APP/NET buffers on the network interface.
LDV_PRIORITY_BUFFER_COUNT_MISMATCH	43	All priority buffers must have the same count. For example, if one of the priority output buffer counts is zero, then both must be zero. Applies to APP/NET buffers on the network interface.
LDV_BUFFER_SIZE_TOO_SMALL	44	The specified buffer size is too small to support subsequent buffer configuration changes. Applies to APP/NET buffers on the network interface.

Return Code	Numeric Value	Description
LDV_BUFFER_CONFIGURATION_TOO_LARGE	45	The specified buffer configuration is too large to fit in the available space. Applies to APP/NET buffers on the network interface.
LDV_WARNING_APP_BUFFER_SIZE_MISMATCH	46	Warning message that the buffer size mismatch might cause problems. Applies to APP/NET buffers on the network interface.

Example: A Simple OpenLDV Application

The following code sample shows a very simple OpenLDV application that works with an Echelon USB Network Interface. In this example, the application performs the following tasks:

1. Opens the network interface
2. Writes a Query Status command to the interface to retrieve the interface's current state; from the current state, extracts the network interface's error log and prints a message
3. Reads the response to the Query Status command
4. Closes the network interface
5. Shuts down the OpenLDV driver

A real application would perform the same basic set of tasks, but typically for additional devices beyond just the network interface. Also, a real application would query the local device database to determine which network interface to use; this example simply uses LON1. An application might also define the network address for the network interface and for network devices; this example uses local network addressing.

This example uses definitions found in the **OpenLDVdefinitions.h** header file, which is included with the OpenLDV Developer Example; see Chapter 4, *The OpenLDV Developer Example*, on page 91. It also includes the **NetMgmt.h** header file, which is installed to the LONWORKS **\NeuronC\Include** directory with the NodeBuilder FX Developer's Kit or the Mini FX Evaluation Kit.

```
// Include Windows header file
#include <windows.h>

// Include OpenLDV header file
#include “ldv32.h”

// Include Neuron C Network Management header file
//   (contains definition for ND_query_status_response)
#include “netmgmt.h”
```

```
// Include the header file from the OpenLDV API Example
//   (contains definitions for ExpMsgHdr, ExpAppBuffer, niNTQ, niNETMGMT)
#include "OpenLdvDefinitions.h"

// Define Network Management commands
// from ISO/IEC 14908 Control Networking Protocol spec
#define LonNdQueryStatus 0x51           // Query Status command
#define LonNdQueryStatusSuccess 0x31    // Success Response for Query Status

//
// Variable Definitions
//

// Handle for Windows event calls
HANDLE          hEvent = NULL;

// Return code for ldv_* calls.
LDVCode         rc = LDV_OK;   // Assume OK result

// Handle used for ldv_* calls. -1 (minus 1) if not valid.
short           m_OpenLdvHandle;

// Flag to signal whether it's Ok to work with the network interface
Bool            ldvCmdOk = TRUE;  // Assume OK result

// Input and output buffers:
ExpAppBuffer    m_msgIn;     // Incoming message buffer
ExpAppBuffer    m_msgOut;    // Outgoing message buffer

// Size of the message to send
short           msgsize = sizeof(m_msgOut.data.code);

//
// Set Up Windows Event
//
hEvent = CreateEvent(NULL, FALSE, TRUE, NULL);


//
// Application Code
//

// Open the network interface; assume LON1
rc = ldv_open("LON1", &m_OpenLdvHandle);

if (rc != LDV_OK) {
  m_OpenLdvHandle = -1;  // Mark handle invalid
  ldvCmdOk = FALSE;      // Don't try to work with the network interface
  printf("Could not open the network interface.\n");
}
else {
  // Register Windows event with this network interface
  rc = ldv_register_event(m_OpenLdvHandle, hEvent);

  if (rc != LDV_OK) {
    ldvCmdOk = FALSE;      // Don't try to work with the network interface
    printf("Could not register an event for the network interface.\n");
  }
  else {
    ldvCmdOk = TRUE;       // Ok to work with the network interface
  }
}
```

```
if (ldvCmdOk) {

  // Build message to send to NI
  m_msgOut.ni_hdr.q.queue        = niNTQ;
  m_msgOut.ni_hdr.q.q_cmd        = niNETMGMT;
  m_msgOut.ni_hdr.q.length       = sizeof(ExpMsgHdr) + sizeof(ExplicitAddr) +
                                    msgSize;
  m_msgOut.msg_hdr.exp.tag       = 1;       // Assume message tag 1
  m_msgOut.msg_hdr.exp.auth      = 0;
  m_msgOut.msg_hdr.exp.response  = 1;       // Make it a response
  m_msgOut.msg_hdr.exp.st        = 3;       // Make it a REQUEST
  m_msgOut.msg_hdr.exp.pool      = 0;       // Must be zero
  m_msgOut.msg_hdr.exp.alt_path  = 0;       // Use default path
  m_msgOut.msg_hdr.exp.addr_mode = 0;       // Implicit addressing
  m_msgOut.msg_hdr.exp.cmpl_code = 0;       // MSG_NOT_COMPL
  m_msgOut.msg_hdr.exp.path      = 0;       // Use primary path
  m_msgOut.msg_hdr.exp.priority  = 0;
  m_msgOut.msg_hdr.exp.length    = msgSize;
  m_msgOut.addr.snd.lc           = NI_LOCAL;
  m_msgOut.data.code             = LonNdQueryStatus;
  // Omitting "m_msgOut.data.data" because Query Status command has no data

  // Write query status command to the network interface
  rc = ldv_write(m_OpenLdvHandle, m_msgOut, offsetof(ExpAppBuffer, code) +
                 msgSize);

  if (rc != LDV_OK) {
    ldvCmdOk = FALSE;      // Don't try to read from the network interface
    printf("Could not write to network interface.\n");
  }
  else {
    ldvCmdOk = TRUE;       // Ok to read from the network interface
  }
}

if (ldvCmdOk) {
  // Wait for network interface event
  WaitForSingleObject (hEvent, 1000);  // Wait 1 second

  // Read response to query status command
  rc = ldv_read(m_OpenLdvHandle, &m_msgIn, sizeof(m_msgIn));

  if (rc != LDV_OK) {
    printf("Could not read from network interface.\n");
  }
  else {
    // Read return code from the LonNdQueryStatus command
    if (m_msgIn.data.code == LonNdQueryStatusSuccess) {
      // Success response from Query Status command;
      // Read error log as something interesting to do
      BYTE errLog = (ND_query_status_response *)(&m_msgIn.data.data)->
                    error_log;
      if (errLog == 0) printf("No error in network interface.\n");
      else printf("Network interface error was: %d\n", errLog);
    }
    else {
      // Failure response from Query Status command; do something else
      printf("Failure response from Query Status cmd: %d\n",m_msgIn.data.code);
    }
  }
}
```

```
// Perform the following tasks regardless of ldvCmdOk value:

// Deregister event for NI
ldv_register_event(m_OpenLdvHandle, NULL);

// Close the network interface
rc = ldv_close(m_OpenLdvHandle);

if (rc != LDV_OK) {
  printf("Could not close network interface.\n");
}
else {
  m_OpenLdvHandle = -1;  // Mark handle invalid
}

Close(hEvent);  // Close the event

ldvx_shutdown();  // Cleanly shutdown the OpenLDV driver
```

3

Sending and Receiving Messages with the OpenLDV API

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

Constructing Messages

You can construct outgoing messages for OpenLDV application using application buffer structures, and send that data to the network interface using the **ldv_write()** function. Use the **ldv_read()** function to retrieve data from the network interface, using the same application buffer structures. The following section describes the application buffer structure.

The **OpenLDVdefinitions.h** header file contains example code that defines the formats of these application buffer structures for Layer 5 devices. This header file is included with the OpenLDV Developer Example; see Chapter 4, *The OpenLDV Developer Example*, on page 91.

The **ldv_read()** and **ldv_write()** functions take a *msg_p* parameter, which is a pointer to a buffer for the data that is to be received or sent. These functions also take a *len* parameter, which specifies the size (in bytes) of the buffer or data to write. See *The OpenLDV API* on page 17 for a description of these functions.

Application Buffer Structure

Figure 5 on page 61 shows the application buffer structures used by OpenLDV LONWORKS interface devices. All OpenLDV LONWORKS interface devices support the serial interface control block (SICB) buffer format. Some LONWORKS interface devices also support an extended (LdvEx) buffer format. You can use the **ldv_open_cap()** function to specify which format to use; see *ldv_open_cap()* on page 27.

The SICB buffer format begins with a simple header, the *Network Interface Header*. The structure of this header depends on the type of command being processed:

- For commands that use a message queue, the header includes 4 bits for the queue type, four bits for the command, and a byte for the length of the payload.
- For commands that do not use a message queue (such as the immediate commands), the header includes a byte for the command and a byte for the length of the payload.

For some commands, the value of the length byte can be zero. An optional, variable-length data field (as indicated by the header's *length* byte) follows the header.

The **LdvEx** buffer format encapsulates the SICB format; it adds extra timestamp information and extended data (where applicable; this extended data is not interpreted by the OpenLDV software). Unless specifically stated, an "application buffer" refers to the SICB format buffer.

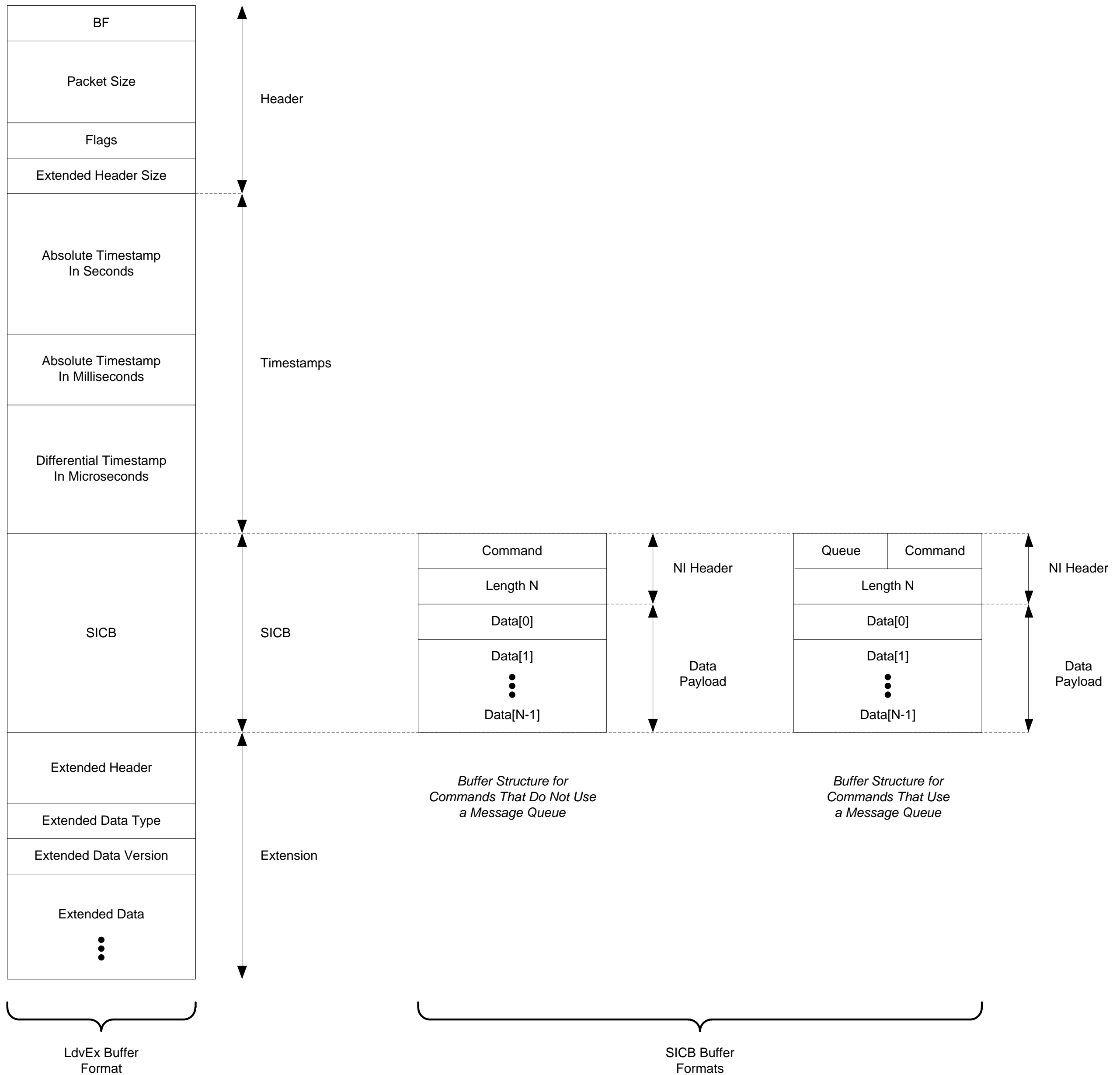


Figure 5. Application Buffer High-Level Structure

For commands that use a message queue, the queue bits in the SICB buffer 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 can 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 a normal incoming message. See *Network Interface Commands* on page 79 for a description of the queue and command-code values.

The OpenLDV Developer Example contains relevant data type definitions, constants, and enumerations for messaging using the SICB buffer. The complete

SICB application buffer structure is defined as a structure of type *ExpAppBuffer* in the **OpenLDVdefinitions.h** header file.

Some immediate commands use only the first byte of the SICB buffer—the *cmd* field—of the application buffer, with no data payload. Other immediate commands also include a data payload.

All other downlink and uplink message commands use the complete SICB application buffer structure, as shown in **Figure 6**. The following sections describe the application buffer structures.

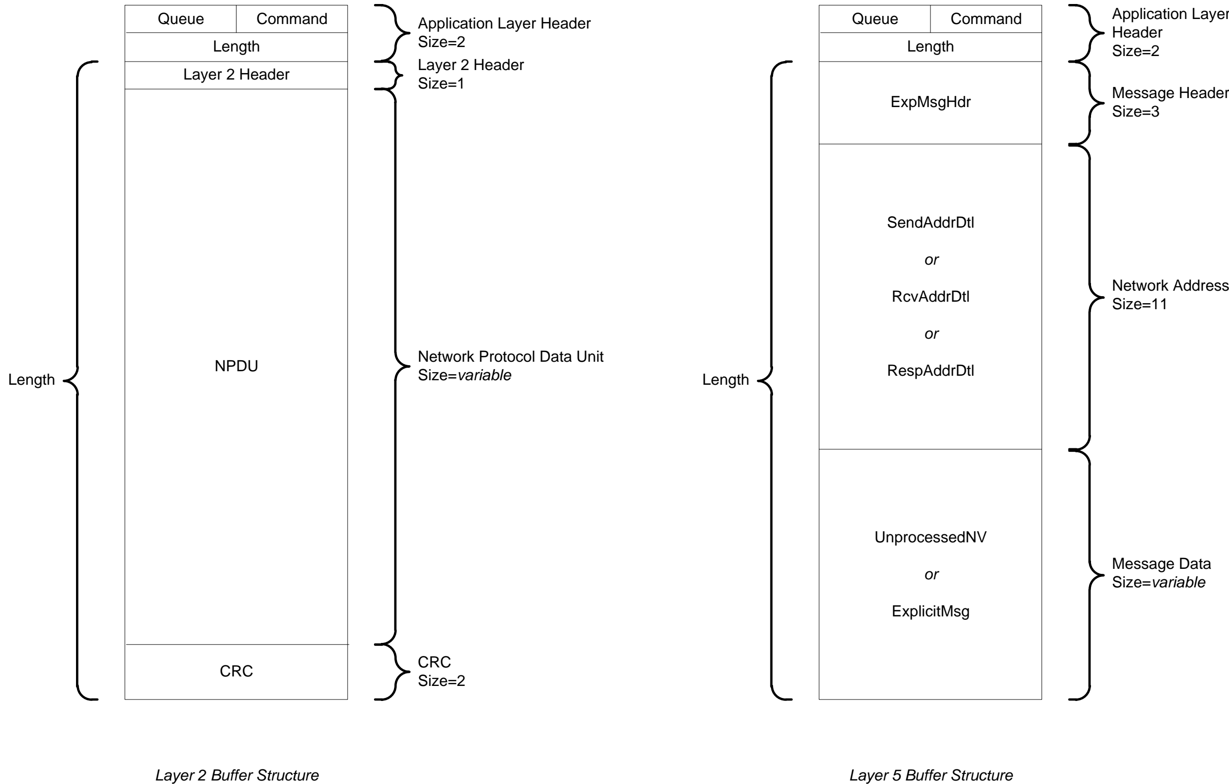


Figure 6. SICB Application Buffer Detailed Structure

As described in *Getting Started with the OpenLDV Driver* on page 7, an OpenLDV application can use a Layer 2 network interface or a Layer 5 network interface:

- *Layer 2 Network Interface* – A network interface that communicates at Layer 2 of the LonTalk protocol. This type of interface transports LonTalk packets without processing them, and does not filter by network address.
- *Layer 5 Network Interface* – A network interface that communicates at Layer 5 of the LonTalk protocol. This type of interface transports incoming LonTalk packets that are addressed to the network interface, and transports outgoing packets that are addressed to other devices.

Layer 2 Buffer Structure

The following sections provide an overview of the Layer 2 buffer structure shown in **Figure 6**. Bit transmission order within a byte is "most significant first", meaning that the most significant bit is transmitted first. Byte transmission order is also "most significant first", meaning that the most significant byte of a field is transmitted first.

A Layer 2 network interface uses the Layer 2 buffer structure for most messages. However, local network management messages use the Layer 5 buffer structure, regardless of which layer the network interface uses for network messages.

For a more complete description of the Layer 2 buffer structure, see the *ISO/IEC 14908-1 Interconnection of information technology equipment – Control Network Protocol - Part 1: Protocol Stack*.

Application Layer Header

The application layer header contains the network interface command (and queue) and a byte that indicates the length of the rest of the message. The most significant nibble of the network interface command contains the command code (for example, **niCOMM** for network messages), and the least significant nibble contains the queue code, if any. These nibbles combine to form the command/queue byte, which is the network interface command.

An OpenLDV application sends these commands using the **ldv_write()** function, and receives them using the **ldv_read()** function. See *Network Interface Commands* on page 78 for a description of the network interface commands.

Layer 2 Header

The Layer 2 header is a single byte that includes the following fields:

- A 1-bit field to specify the priority of the data packet. 0 = Normal; 1 = Priority.
- A 1-bit field to specify the channel to use, primary or alternate. This field allows transceivers that have the ability to transmit on two different channels and receive on either one, without the need to instruct the transceiver to explicitly receive on a specific channel. The transport layer sets this bit.
- A 6-bit unsigned field (≥ 0) to specify the channel backlog increment to be generated as a result of delivering this packet. The backlog represents the number of messages that the packet shall cause to be generated upon reception. This value is used by the Smart Transceiver or Neuron Chip MAC algorithm.

NPDU

The *Network Protocol Data Unit* (NPDU) encapsulates the physical packet data. The NPDU includes the following fields:

- Protocol version (2 bits)
- Physical packet type (2 bits)
- Address format (2 bits)

- Domain length (2 bits)
- Data (variable length, depending on the packet type and the data)

CRC

The *cyclic redundancy check* (CRC) is computed over the NPDU and the Layer 2 Header. The CRC is generated using the ITU-T (CCITT) CRC-16 standard polynomial. When the link layer receives a packet with a CRC error, or a packet that is less than 8 bytes in length, a transmission error statistic is incremented; if a packet is received that is too long for the input buffer, or if there are no input buffers, the missed packet statistic is incremented.

Layer 5 Buffer Structure

The following sections describe the Layer 5 buffer structure shown in **Figure 6**. Bit transmission order within a byte is "most significant first", meaning that the most significant bit is transmitted first. Byte transmission order is also "most significant first", meaning that the most significant byte of a field is transmitted first.

For a more complete description of the Layer 5 buffer structure, see the *ISO/IEC 14908-1 Interconnection of information technology equipment – Control Network Protocol - Part 1: Protocol Stack*.

Application Layer Header

The application layer header contains the network interface command (and queue) and a byte that indicates 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.

An OpenLDV application sends these commands using the **ldv_write()** function, and receives them using the **ldv_read()** function. The OpenLDV Developer Example contains an example implementation of a network interface API. You can use the **NiSendMsgWait()** and **NiSendResponse()** functions, included as part of this example API, to send enqueued commands more conveniently. You can also use the application-specific message dispatcher, also implemented as part of the OpenLDV Developer Example, to receive these messages.

See *Network Interface Commands* on page 78 for a description of the network interface commands.

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 union of two structures:

- **ExpMsgHdr** — for sending and receiving explicit messages that are not processed by the network interface

- **NetVarHdr** — for sending and receiving network variables that are processed by the network interface

ExpMsgHdr

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

msgtype

The *msgtype* field is set to 0 for the **ExpMsgHdr**.

service type

The *service type* field contains one of the following values, depending on which LonTalk protocol messaging service is to be used for delivery of the message:

- ACKD (0) for the acknowledged messaging service
- UNACKD_RPT (1) for the repeated messaged service
- UNACKD (2) for the unacknowledged messaging service
- REQUEST (3) for the request/response messaging service

auth

The *auth* field is set to 1 for a downlink message (sent to a network interface) if the receiver must authenticate the message using LonTalk authentication. A network interfaces might require authentication for local network management commands.

It is set to 1 for an uplink message (read from a network interface) if the message has been authenticated by the network interface.

If authentication is not enabled on the network interface, this field should be set to 0.

tag

The OpenLDV application uses the *tag* field for a downlink message (sent to a network interface) to correlate returned responses and completion events. For explicitly addressed messages (those that use the full 3-layer address), you can segt the *tag* to any value in the range 0-14, and the same value is returned with the corresponding responses and completion events. In this case, the tag is also known as the *reference ID*. For a downlink implicitly addressed message (one that specifies an entry in the address table), the *tag* field is used as an index into the address table of the Smart Transceiver or Neuron Chip in the network interface to indicate the destination address of the message. For more information about the address table, see the ISO/IEC 14908-1 protocol specification.

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

priority

The *priority* field is set to indicate a message delivered with priority media access, either uplink or downlink. When an OpenLDV application generates a response to an uplink request message, it must save the priority attribute from the request, and sets 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.

path

The *path* field is set to 1 if the message should use the alternate path, and 0 if it should use the primary path. This feature is enabled only if the *alternate path* bit is set. Alternate path is a feature of certain special-purpose mode LONWORKS transceivers, such as power line (PL) transceivers.

completion code

The *completion code* field is set for an uplink completion event. Completion code events are returned to the OpenLDV application for every downlink (**niCOMM**) network message sent:

- 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.
- Set the completion code field 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.

address mode

Set the *address mode* bit to 1 for an explicitly addressed downlink message, and specify the network address field as a **SendAddrDtl** structure (see *SendAddrDtl* on page 69).

Set the *address mode* field to 0 for an implicitly addressed downlink message, in which case the network address field is ignored, although it must be present. In this case, use the *tag* field as the index into the address table of the Smart Transceiver or Neuron Chip in the network interface for the destination address. For more information about the address table, see the ISO/IEC 14908-1 protocol specification.

Set the *address mode* to 0 for downlink responses to uplink request messages and network variable polls.

The *address mode* bit is ignored for local network management (**niNETMGMT**) messages.

alternate path

If the *alternate path* bit is set, the message is delivered on the path specified in the *path* bit, otherwise it is delivered on the default path.

pool

Set the *pool* bit to 0 for a downlink message.

response

Set the *response* bit to 1 for a downlink response message, and 0 otherwise. If it is set for an uplink message, the message is a response to a previously sent request.

length

The *length* field in the message header is distinct from the length field in the application layer header. The *length* field the message header indicates the number of bytes for the message data.

NetVarHdr

7	6	5	4	3	2	1	0
msgtype	poll	resvd		tag			
Priority	Path	compl code		addr mode	trnarnd	pool	resp
length							

msgtype

The *msgtype* field is set to 1 for the **NetVarHdr**.

poll

The *poll* field is set to 1 for a network variable poll message. For other network variable messages, it is set to 0 (zero).

reserved

The **NetVarHdr** includes two bits that are reserved for future use. Set this field to 0 for downlink messages.

tag

The OpenLDV application uses the *tag* field for a downlink message (sent to a network interface) to correlate returned responses and completion events. For explicitly addressed messages, the *tag* can be set to any value in the range 0-14, and the same value is returned with the corresponding responses and completion events. In this case, the tag is also known as the reference ID. For a downlink implicitly addressed message, the *tag* field is used as an index into the address table of the Smart Transceiver or Neuron Chip in the network interface to indicate the destination address of the message. For more information about the address table, see the ISO/IEC 14908-1 protocol specification.

For an uplink message (read from a network interface), 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.

priority

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.

path

The *path* field is set to 1 if the message should use the alternate path, and 0 (zero) if it should use the primary path. This feature is enabled only if the *alternate path* bit is set. Alternate path is a feature of certain special-purpose mode LONWORKS transceivers.

completion code

The *completion code* field is set for an uplink completion event. Completion code events are returned to the OpenLDV application for every downlink (**niCOMM**) network message sent:

- 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.
- Set the completion code field 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.

address mode

Set the *address mode* bit to 1 for an explicitly addressed downlink message, and specify the network address field as a **SendAddrDtl** structure (see *SendAddrDtl* on page 69).

Set the *address mode* field to 0 for an implicitly addressed downlink message, in which case the network address field is ignored, although it must be present. In this case, the *tag* field is used as the index into the address table of the Smart Transceiver or Neuron Chip in the network interface for the destination address. For more information about the address table, see the ISO/IEC 14908-1 protocol specification.

Set the *address mode* to 0 for downlink responses to uplink request messages and network variable polls.

The *address mode* bit is ignored for local network management (**niNETMGMT**) messages.

turnaround

If the *turnaround* bit is set, the message is a turnaround message, that is, a message sent from one network variable to another network variable on the same device.

pool

Set the *pool* bit should 0 for a downlink message.

response

Set the *response* bit to 1 for a downlink response message, and 0 otherwise. If it is set for an uplink message, the message is a response to a previously sent request.

length

The *length* field in the message header is distinct from the length field in the application layer header. The *length* field the message header indicates the number of bytes for the message data.

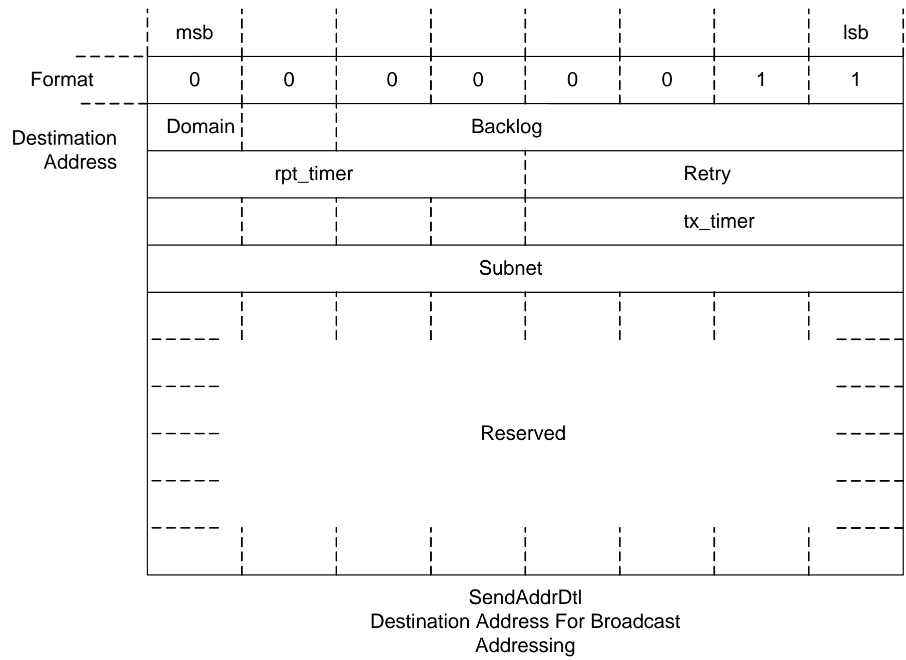
Network Address

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 for **ExplicitAddr** is a union of three structures, depending on the type of message buffer. For more information about address modes and the corresponding structures, see the ISO/IEC 14908-1 protocol specification.

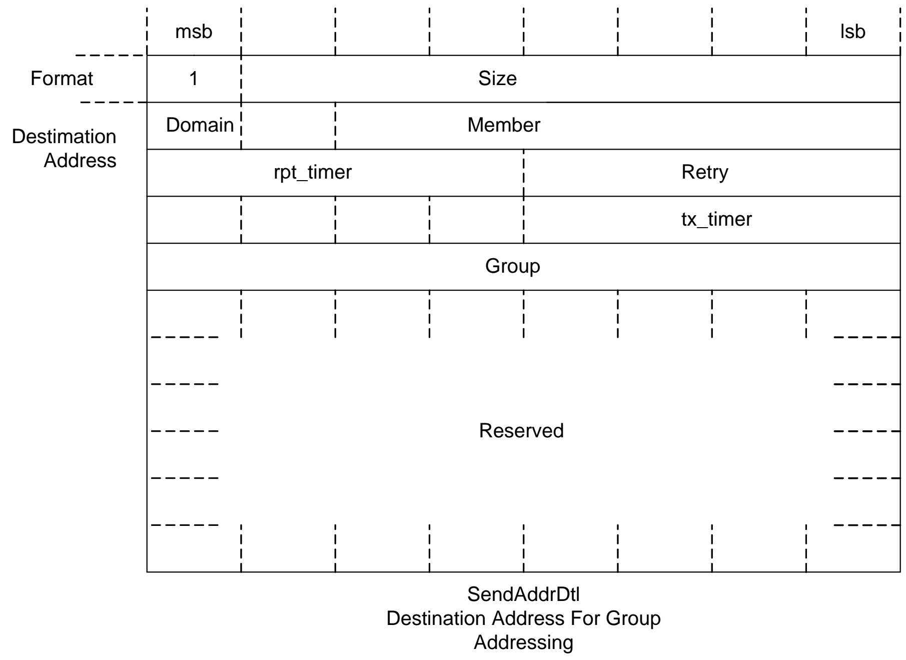
The OpenLDV Developer Example also contains an example definition of the related structures in the **OpenLDVdefinitions.h** header file.

SendAddrDtl

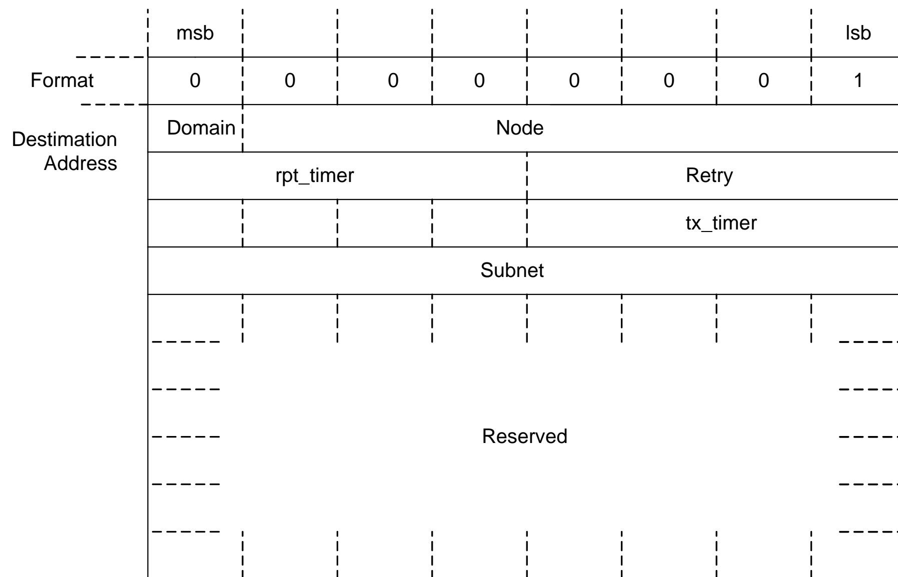
This structure is used for a downlink, explicitly addressed message, and contains the destination address of the downlink message in one of four formats, depending on the address mode. The address modes for sending explicitly addressed messages are broadcast, group, subnet/node, Neuron ID, local, and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



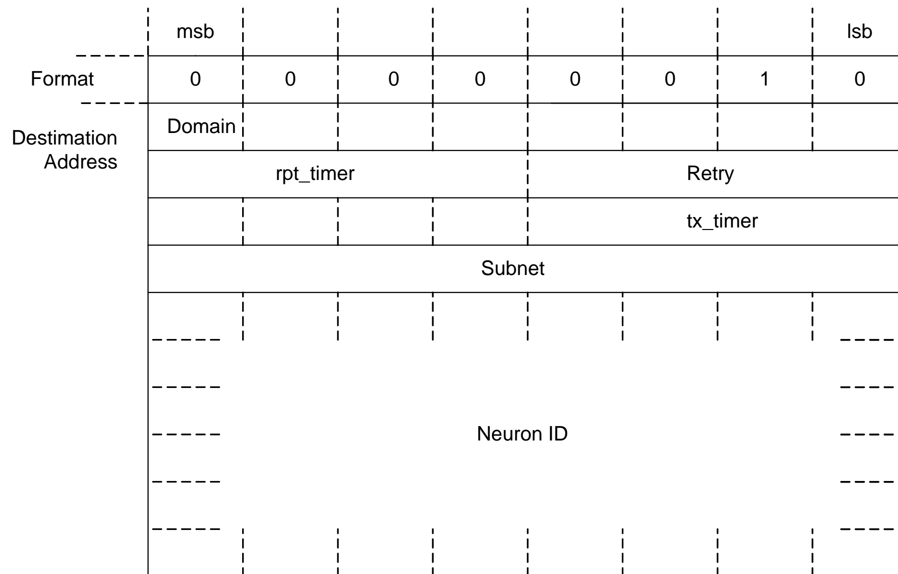
SendAddrDtl
Destination Address For Broadcast
Addressing



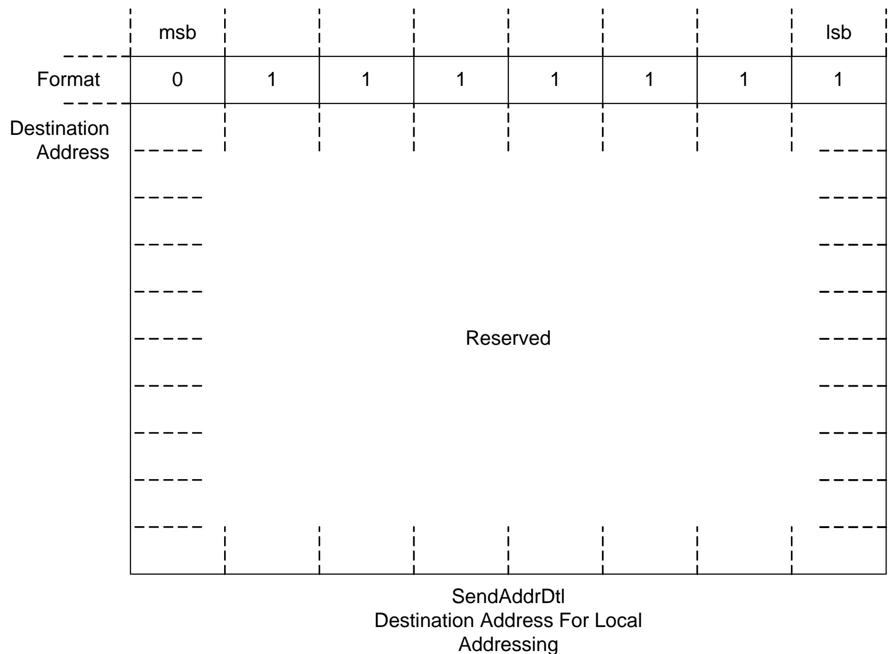
SendAddrDtl
Destination Address For Group
Addressing



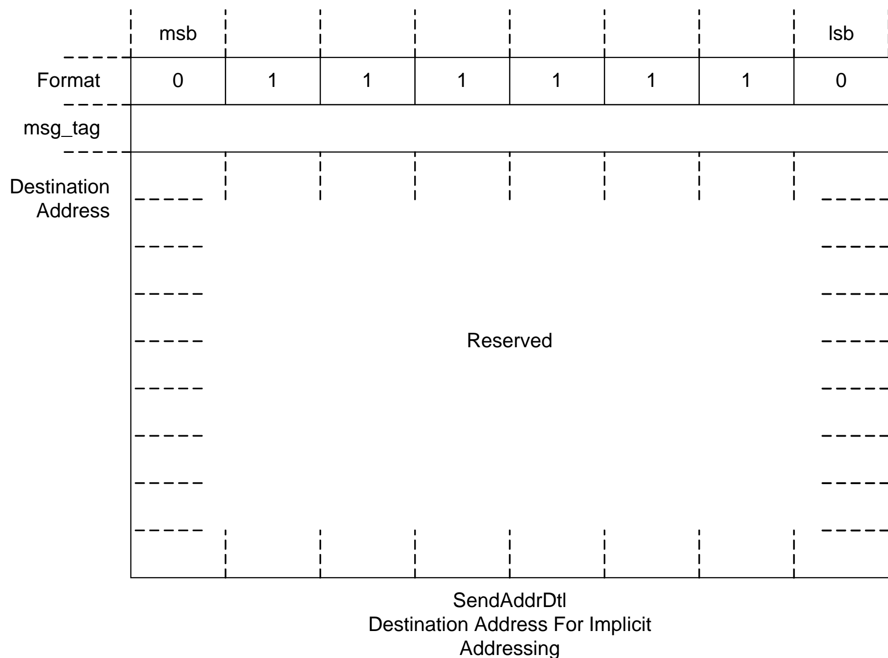
SendAddrDtl
Destination Address For Subnet/Node Addressing



SendAddrDtl
Destination Address For Neuron ID Addressing



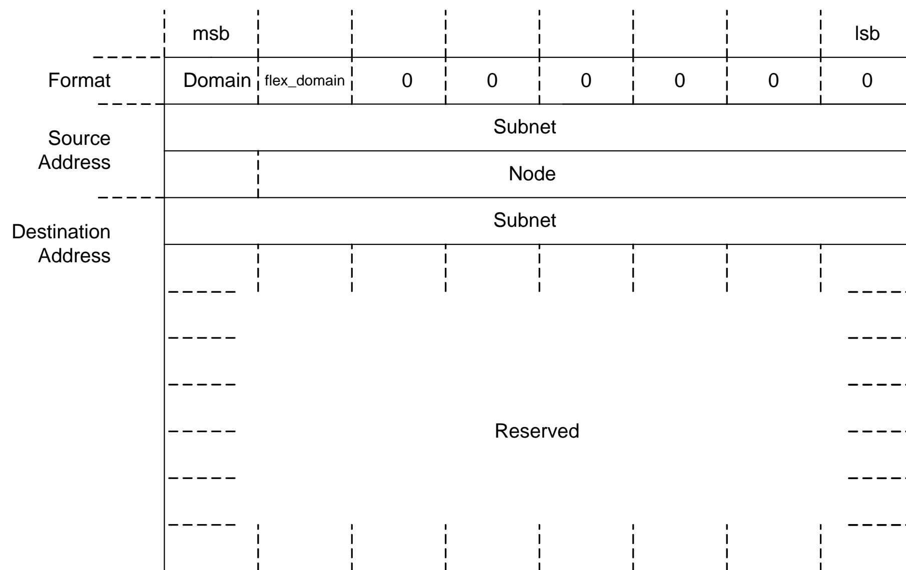
SendAddrDtl
Destination Address For Local Addressing



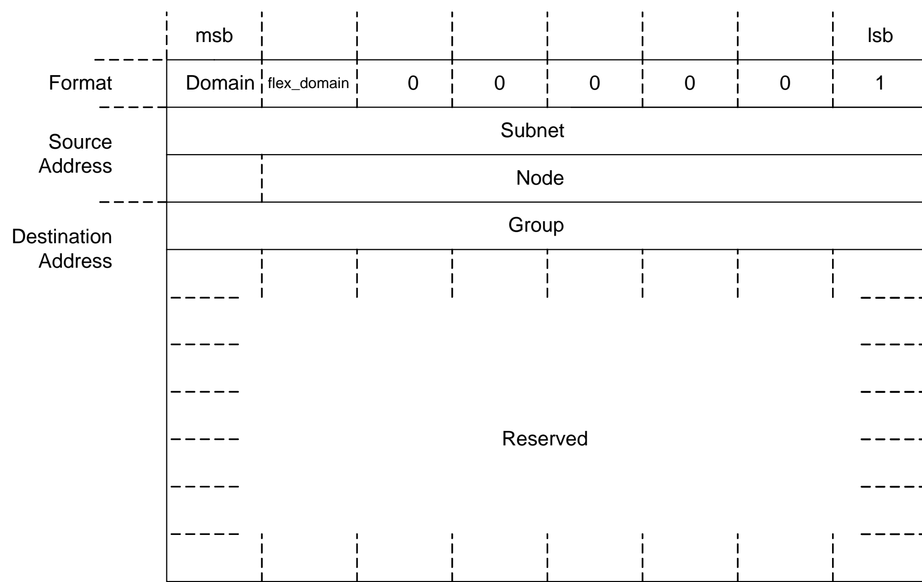
SendAddrDtl
Destination Address For Implicit Addressing

RcvAddrDtl

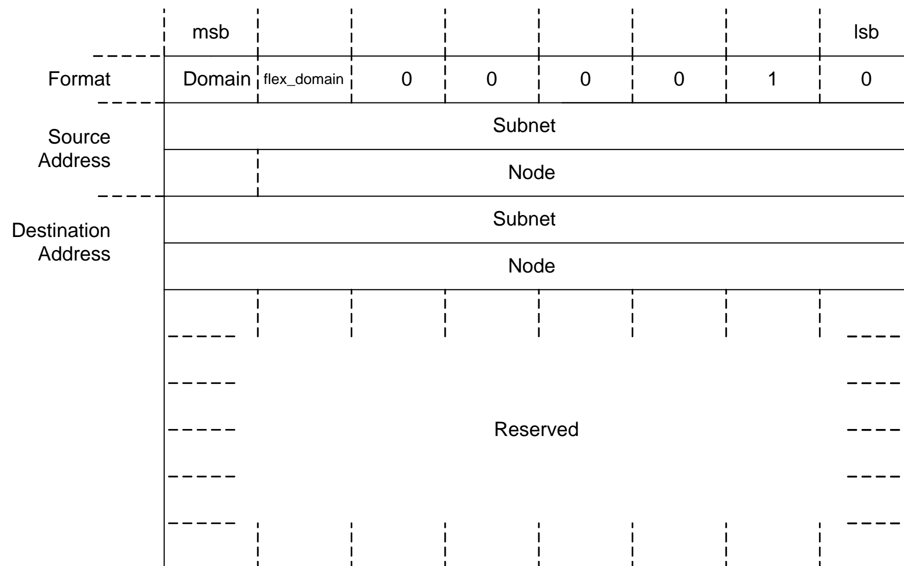
This structure is used for uplink messages addressed to the network interface and intended for the OpenLDV application. The structure contains the source address of the device sending the message and the destination address of the uplink message in one of four formats, depending on the address mode. 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.



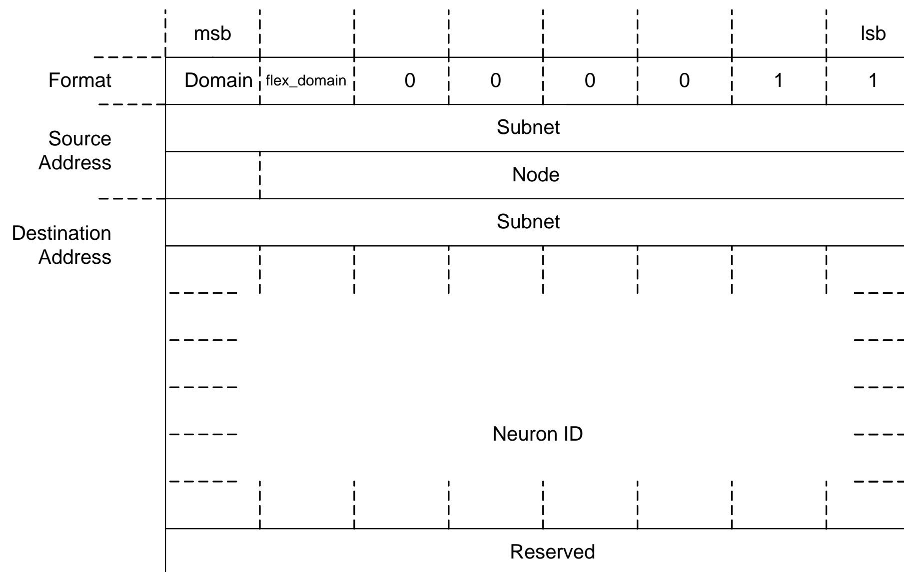
RcvAddrDtl
Received Address For Broadcast
Addressing



RcvAddrDtl
Received Address For Group Addressing



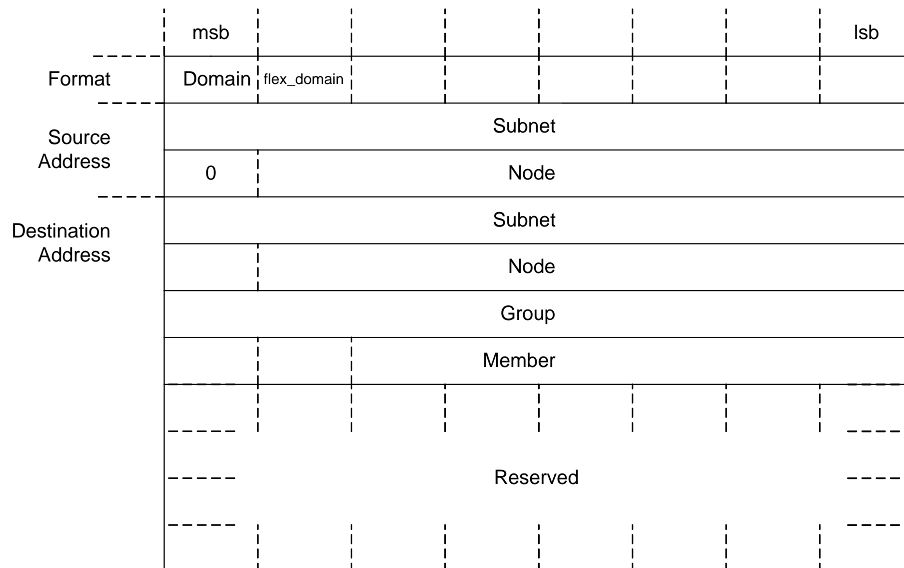
RcvAddrDtl
Received Address For Subnet/Node
Addressing



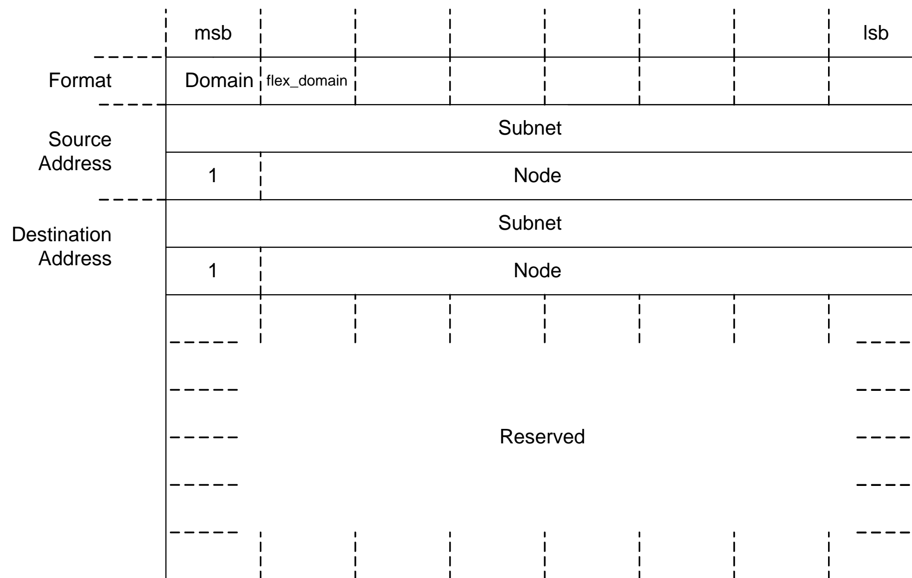
RcvAddrDtl
Received Address For Neuron ID
Addressing

RespAddrDtl

This structure is used for an uplink message in response to a previous downlink request. This field contains the source address of the device 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 or a poll response contains 1-31 bytes of network-variable data. A network-variable poll request message or a completion event contains no data, only the selector in the first two bytes.

Set the *direction* bit to 1 for polling an output network variable, and 0 for 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 information about network variable configuration, messages, and alias tables, see the ISO/IEC 14908-1 protocol specification.

ExplicitMsg

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

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 31**.

Table 31. Message Codes for Application Messages

Message Type	**Message Codes (Hex)**
User application message	00 .. 2F
Standard application message	30 .. 3E
Responder offline	3F
Foreign message	40 .. 4E

Message Type	Message Codes (Hex)
Foreign responder offline	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 can be sent to the network interface itself. For example, the **NM_update_domain** (0x63) message can be sent to the network interface using the **niNETMGMT** network interface command. This message sets the network address (domain ID, subnet ID and node ID) used by the network interface.

Some network interfaces require authentication for local network management commands. Thus, you should always set the *auth* bit of the **ExpMsgHdr** to 1 for local network management commands. See *ExpMsgHdr* on page 65 for information about the message header.

Receiving Messages from the Network Interface

Incoming application, foreign frame, 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 32** can be passed to the OpenLDV application, which must respond appropriately. See the ISO/IEC 14908-1 protocol specification for information about these network management and diagnostics messages.

The OpenLDV Developer Example contains an example framework for recognizing and processing these messages.

Table 32. Network Management Messages Passed to the OpenLDV Application

Message	Code	Comments
Query NV Config	0x68	OpenLDV application should respond with data from the network variable configuration table or alias table.
Update NV Config	0x6B	OpenLDV application should write its own network variable configuration or alias table, respectively.

Message	Code	Comments
Set Node Mode	0x6C	On-line and off-line only. OpenLDV application should send corresponding immediate command (**niONLINE** or **niOFFLINE**) to the network interface.
Wink	0x70	OpenLDV application should indicate receipt of message to user, or handles a request to manage its self-documentation data.
Query SI	0x72	OpenLDV application should respond with self-identification and self-documentation data.
NV Fetch	0x73	OpenLDV application should respond with network variable data.

Using the Network Interface Command Interface

The following sections describe the OpenLDV command interface.

Downlink Commands

A downlink command is a message sent to a network interface from an OpenLDV application with the **ldv_write()** function:

- The OpenLDV application sends application messages, network management and diagnostics messages, network variable updates, and network variable poll requests on the network through 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, including 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.

There are two 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 Smart Transceiver or Neuron Chip that is part of the network interface. They are sent with the **niNETMGMT** network interface command, and are not sent on the LONWORKS network.

Commands for Layer 5 devices that can be used with a specified queue include **niCOMM** for messages sent to the network, and **niNETMGMT** for local network management operation messages sent to the network interface. Local network management messages use the Layer 5 buffer structure, regardless of which layer the network interface uses for network messages.

Uplink Commands

An uplink command is a message read from a network interface by an OpenLDV application with the **ldv_read()** function:

- 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 might have received from the network.
- Layer 2 network interfaces send error codes for physical packet errors.

There are two 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.

Commands for Layer 5 devices that can be used with a specified queue include **niCOMM** for messages received from the network, and **niNETMGMT** for local network management operation messages received from the network interface. Local network management messages use the Layer 5 buffer structure, regardless of which layer the network interface uses for network messages.

Immediate Commands

Immediate commands can be sent to the OpenLDV interface using the **ldv_write()** function, and received using the **ldv_read()** function. Most immediate commands are just two bytes long. This includes a command byte followed by a trailing zero, which indicates there is no data payload for the command. However, some commands, such as **niXDRVESC** (*xDriver escape command)*, do require a data payload.

The OpenLDV Developer Example also includes an example implementation of a network interface API. The **NiSendImmediate()** function, which is part of this example API, can be used to send immediate commands.

Network Interface Commands

Table 33 on page 80 lists the network interface commands. Unless specifically described otherwise, the commands in the table apply to Layer 5 network interfaces only.

The command names listed in the table are suggestions; for the Layer 5 device commands, they are defined in the OpenLDV Developer Example 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.

Literals for the supported immediate commands are defined in the OpenLDV Developer Example by the enumeration type definition **NI_NoQueueCmd** used in the field *NI_Hdr.q* of the application layer header.

Table 33. Network Interface Commands

Network Interface Command	Value	Direction	Description
niL2_INCOMING	0x1A	Uplink	Specifies a Layer 2 incoming packet.
niL2_INC_M1	0x1B	Uplink	Specifies a Layer 2 Mode 1 incoming packet.
niL2_INC_M2	0x1C	Uplink	Specifies a Layer 2 Mode 2 incoming packet.
niCOMM + niTQ	0x12	Downlink	Used for downlink non-priority messages using acknowledged, request and repeated services. For Layer 2 devices, also used for unacknowledged messages. This command specifies **niCOMM** (for messages sent to and received from the network) as the *queue* value and **niTQ** as the *command* value. Applies to Layer 2 or Layer 5 devices. The command format is different for Layer 2 and Layer 5.

Network Interface Command	Value	Direction	Description
niCOMM + niTQ_P	0x13	Downlink	Used for downlink priority messages using acknowledged, request and repeated services. For Layer 2 devices, also used for unacknowledged messages. This command specifies **niCOMM** (for messages sent to and received from the network) as the *queue* value and **niTQ_P** as the *command* value. Applies to Layer 2 or Layer 5 devices. The command format is different for Layer 2 and Layer 5.
niCOMM + niNTQ	0x14	Downlink	Used for downlink non-priority messages using unacknowledged service, as well as responses. This command specifies **niCOMM** (for messages sent to and received from the network) as the *queue* value and **niNTQ** as the *command* value.
niCOMM + niNTQ_P	0x15	Downlink	Used for downlink priority messages using unacknowledged service, as well as responses. This command specifies **niCOMM** (for messages sent to and received from the network) as the *queue* value and **niNTQ_P** as the *command* value.
niCOMM + niRESPONSE	0x16	Uplink	Used for uplink response messages and completion codes. This command specifies **niCOMM** (for messages sent to and received from the network) as the *queue* value and **niRESPONSE** as the *command* value.

Network Interface Command	Value	Direction	Description
niCOMM + niINCOMING	0x18	Uplink	Used for uplink messages received from the network or the network interface. This command specifies **niCOMM** (for messages sent to and received from the network) as the *queue* value and **niINCOMING** as the *command* value.
niNETMGMT + niTQ	0x22	Downlink	Used for downlink non-priority messages using acknowledged, request and repeated services. Also used for a Layer 2 network interface to issue a local network management command. This command specifies **niNETMGMT** (for messages sent to and received from the network interface) as the *queue* value and **niTQ** as the *command* value.
niNETMGMT + niTQ_P	0x23	Downlink	Used for downlink priority messages using acknowledged, request and repeated services. Also used for a Layer 2 network interface to issue a local network management command. This command specifies **niNETMGMT** (for messages sent to and received from the network interface) as the *queue* value and **niTQ_P** as the *command* value.
niNETMGMT + niNTQ	0x24	Downlink	Used for downlink non-priority messages using unacknowledged service, as well as responses. Also used for a Layer 2 network interface to issue a local network management command. This command specifies **niNETMGMT** (for messages sent to and received from the network interface) as the *queue* value and **niNTQ** as the *command* value.

Network Interface Command	Value	Direction	Description
niNETMGMT + niNTQ_P	0x25	Downlink	Used for downlink priority messages using unacknowledged service, as well as responses. Also used for a Layer 2 network interface to issue a local network management command. This command specifies **niNETMGMT** (for messages sent to and received from the network interface) as the *queue* value and **niNTQ_P** as the *command* value.
niNETMGMT + niRESPONSE	0x26	Uplink	Used for uplink response messages and completion codes. Also used by a Layer 2 network interface to respond to a local network management command. This command specifies **niNETMGMT** (for messages sent to and received from the network interface) as the *queue* value and **niRESPONSE** as the *command* value.
niNETMGMT + niINCOMING	0x28	Uplink	Used for uplink messages received from the network or the network interface. This command specifies **niNETMGMT** (for messages sent to and received from the network interface) as the *queue* value and **niINCOMING** as the *command* value.
niL2_PKT_TIMEOUT	0x30	Uplink	Specifies a timeout error condition for a Layer 2 network interface.
niL2_PKT_CRC	0x31	Uplink	Specifies a CRC error condition for a Layer 2 network interface.
niL2_PKT_LONG	0x32	Uplink	Specifies a “Packet Too Long” error condition for a Layer 2 network interface.
niL2_PRE_LONG	0x33	Uplink	Specifies a “Preamble Too Long” error condition for a Layer 2 network interface.

Network Interface Command	Value	Direction	Description
niL2_PRE_SHORT	0x34	Uplink	Specifies a “Preamble Too Short” error condition for a Layer 2 network interface.
niL2_PKT_SHORT	0x35	Uplink	Specifies a “Packet Too Short” error condition for a Layer 2 network interface.
niL2_FREQ_RPT	0x40	Uplink	Specifies an incoming frequency report from a Layer 2 network interface.
niRESET	0x50	Uplink Downlink	Uplink: Specifies that the network interface has executed a hardware or software reset. Downlink: Requests a reset of the network interface. Applies to both Layer 2 and Layer 5 network interfaces.
niFLUSH_CANCEL	0x60	Downlink	Requests that the network interface cancel any flush operation posted with the **niFLUSH** command or caused by device reset. The OpenLDV application must issue this command after a successful completion of the **ldv_open()** function. You can use the **NiInit()** function, which is part of the OpenLDV Developer Example to open a connection to a network interface more conveniently.
niFLUSH_COMPLETE	0x60	Uplink	Specifies that a flush operation posted with the **niFLUSH** command has completed.

Network Interface Command	Value	Direction	Description
niONLINE	0x70	Downlink	Requests that the network interface set its online flag and enter the online state. The OpenLDV application must send this command whenever it goes online. Generally, the OpenLDV application receives an uplink network management message from a network management tool or plug-in requesting that the application go online and send the **niONLINE** command. The uplink message is a standard **Set Node Mode** network management command (message code 0x6C) with mode set to **ONLINE**.
niOFFLINE	0x80	Downlink	Requests that the network interface clear its online flag and enter the offline state. The OpenLDV application must send this command whenever it goes offline. Generally, the OpenLDV application receives an uplink network management message from a network management tool or plug-in requesting that the application go offline and send the **niOFFLINE** command. The uplink message is a standard **Set Node Mode** network management command (message code 0x6C) with mode set to **OFFLINE**.

Network Interface Command	**Value**	**Direction**	**Description**
niFLUSH	0x90	Downlink	Requests that the network interface enter quiet mode (the **FLUSH** state), which causes it to send any pending downlink messages. After all pending downlink messages are completed, the network interface responds 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	Downlink	Obsolete.
niSLEEP	0xB0	Downlink	Obsolete.
niLAYER	0xE5	Downlink Uplink	Sets the top-most protocol layer processed by the network interface. This command is used only for network interfaces that support switching between a Layer 2 and a Layer 5 interface, such as the Echelon U10, U20, and U60 USB Network Interfaces. This message can contain a single-byte data payload to specify the top-most protocol layer for the network interface: • 0 specifies Layer 5 • 1 specifies Layer 2 If no data payload is included, the network interface responds with two bytes: 0xE5 followed by 0 (if the interface is operating as a Layer 5 network interface) or 1 (if the interface is operating as a Layer 2 network interface).

Network Interface Command	Value	Direction	Description
niSERVICE	0xE6	Downlink	Requests that the network interface send a service pin message. This command has the same effect as activating the device's service pin. Some network interfaces might not support this command.
niXDRVESC	0xEF	Uplink Downlink	This command applies to xDriver network interfaces only. This message must contain a data payload in addition to the command and length bytes. The first byte of the data field denotes an xDriver-specific command; see **Table 34**. For information about other immediate commands that are specific to a particular network interface, see the documentation for 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 through a modem.

Table 34 on page 88 describes the xDriver-specific commands that you can use with the **niXDRVESC** immediate command. The **niXDRVESC** immediate command is described in **Table 33** above.

Table 34. xDriver Specific Commands

xDriver Command	Description
LDVX_NICMD_ENCRYPTION_ON_SEND=0x02	Use this command to enable RC4 encryption on the IP connection to the RNI for all subsequent messages sent to the network interface. All subsequent messages are encrypted until the **LDVX_NICMD_ENCRYPTION_OFF_SEND** command is sent, or the session is terminated. This command is ignored if encryption has already been enabled. The xDriver subsystem determines if the network interface supports RC4 encryption. If it does not, this command is silently ignored.
LDVX_NICMD_ENCRYPTION_OFF_SEND=0x03	Use this command to disable RC4 encryption on the IP connection to the RNI for all subsequent messages sent to the network interface. This command is ignored if encryption has already been disabled.
LDVX_NICMD_ENCRYPTION_ON_RECEIVE=0x04	Use this command to enable RC4 encryption on the IP connection to the RNI for all subsequent messages sent from the network interface. All subsequent messages are encrypted until the **LDVX_NICMD_ENCRYPTION_OFF_RECEIVE** command is sent, or the session is terminated. This command is ignored if encryption has already been enabled. The xDriver subsystem determines if the network interface supports RC4 encryption. If it does not, this command is silently ignored.

xDriver Command	Description
LDVX_NICMD_ENCRYPTION_OFF_RECEIVE=0x05	Use this command to disable RC4 encryption on the IP connection to the RNI for all subsequent messages sent from the network interface. This command is ignored if encryption has already been disabled.

4

The OpenLDV Developer Example

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

Overview

The OpenLDV Developer Example is an example application that uses the OpenLDV API. The example application is available from the **Examples & Tutorials** folder in the **Echelon OpenLDV 4.0 SDK** program folder.

The example application is also installed as a ZIP file in the LONWORKS **\OpenLDV SDK\SourceArchive** folder on your computer. The ZIP file is named **LdvApiExamplesSource_v*n.nn.nnn*.ZIP**, where the *n.nn.nnn* represents the version and build number for the OpenLDV release.

The OpenLDV Developer Example is a simple dialog-based Windows application written in C++ with Microsoft Foundation Classes (MFC). It is distributed in Microsoft Visual Studio 2008 project format. The example illustrates how a Windows application can access the OpenLDV API, and demonstrates a wide range of simple to complex network operations.

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 that it contains.

Common Definitions

The OpenLDV API functions are specified in the **ldv32.h** header file. 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

The example application implements a **COpenLDVapi** class to wrap the OpenLDV API functions. The **COpenLDVapi** class provides a simple interface through four methods: **Open**, **Close**, **Read**, and **Write**.

This class provides thread-safe, synchronized, access to downlink messages (**ldv_write()**), and implements a reader thread **COpenLDVreader**, which reads uplink messages (**ldv_read()**) and supplies data to a protected queue. The **COpenLDVapi::Read()** function reads that queue, thereby providing coordinated access to both uplink and downlink messages.

The example application also implements a **COpenLDVtrace** class. This class illustrates how an OpenLDV application can provide hooks for debugging or tracing into the low-level portion of the OpenLDV application. The example implementation provides a packet dump of all incoming and outgoing packets.

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

COpenLDVni, Message Pumps, and Message Dispatchers

The **COpenLDVni** class implements the core functions of a network interface API. The functions included in this class are **NiInit()**, **NiSendMsgWait()**,

NiSendImmediate(), **NiGetNextResponse()**, **NiSendResponse()**, **NiClose()**, and **NiEncryption()**.

The **OpenLDVni.h** header file contains 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 decodes the message, takes appropriate action local to the OpenLDV application, and responds 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 must 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. These messages are sent using an **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**), derived from the **COpenLDVni** class, which implements the **NiDispatch** method.

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**.

You can use the **COpenLDVexampleDispatcher** class as an example for your OpenLDV application, but you must adapt and rewrite the dispatcher for the application.

The **OpenLDVexampleDispatcher.h** header file and the **OpenLDVexampleDispatcher.cpp** implementation file contain comments that describe 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. This class provides a simple interface that implements selected operations such as **QueryDomain**, **LeaveDomain**, or **UpdateDomain**. The **COpenLDvtools** class also implements a **FindDevices()** function that demonstrates the implementation of multi-transaction sequences within the context of this framework.

Developer Example Diagram

Figure 7 shows the hierarchy of the classes described in this chapter.

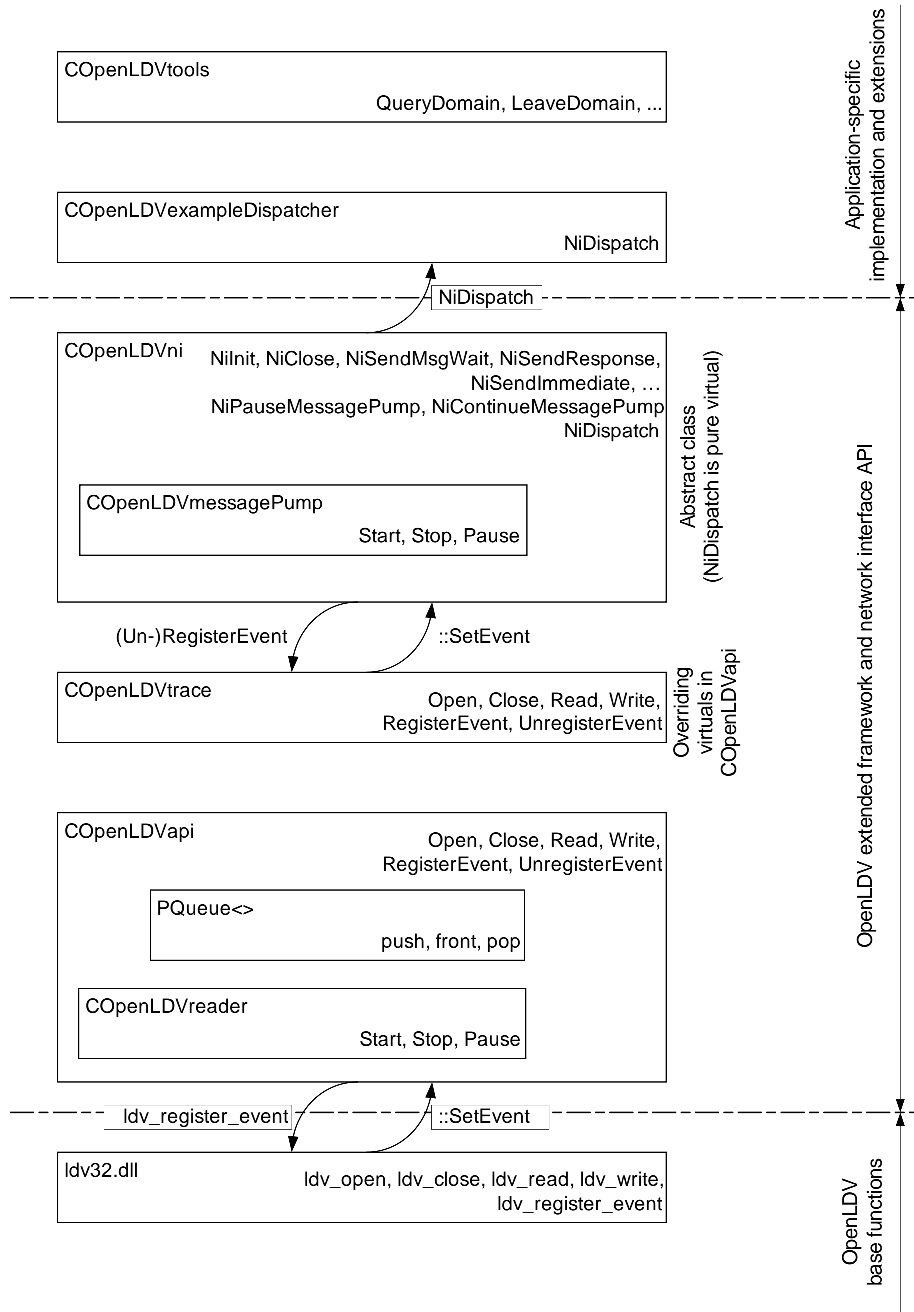


Figure 7. OpenLDV Developer Example Class Hierarchy

5

Using the xDriver Default Profile

This chapter describes how to use the xDriver default profile. It also describes how to use the LONWORKS Interfaces application in the Windows Control Panel to configure an xDriver profile and build an xDriver database into the Windows Registry.

Configuring an xDriver Profile

You can edit an xDriver profile to configure a number of parameters that impact how xDriver handles uplink and downlink sessions, including the automatic reconnection settings. Most applications will use the xDriver default profile.

You can use the automatic reconnection feature to cause xDriver to attempt reconnection when sessions that use the default profile are terminated as a result of power outages, network interface failures, or other communications failures. With automatic reconnection enabled, xDriver attempts reconnection until a failed session has been successfully reestablished, or until a predefined time period expires.

To edit an xDriver profile, perform the following steps:

1. Open the Windows Control Panel and double-click the LONWORKS Interfaces icon to open the LONWORKS Interfaces application, as shown in **Figure 8**. The left-hand device pane shows all of the currently defined devices.

   See the LONWORKS Interfaces online help for a complete description of the user interface.

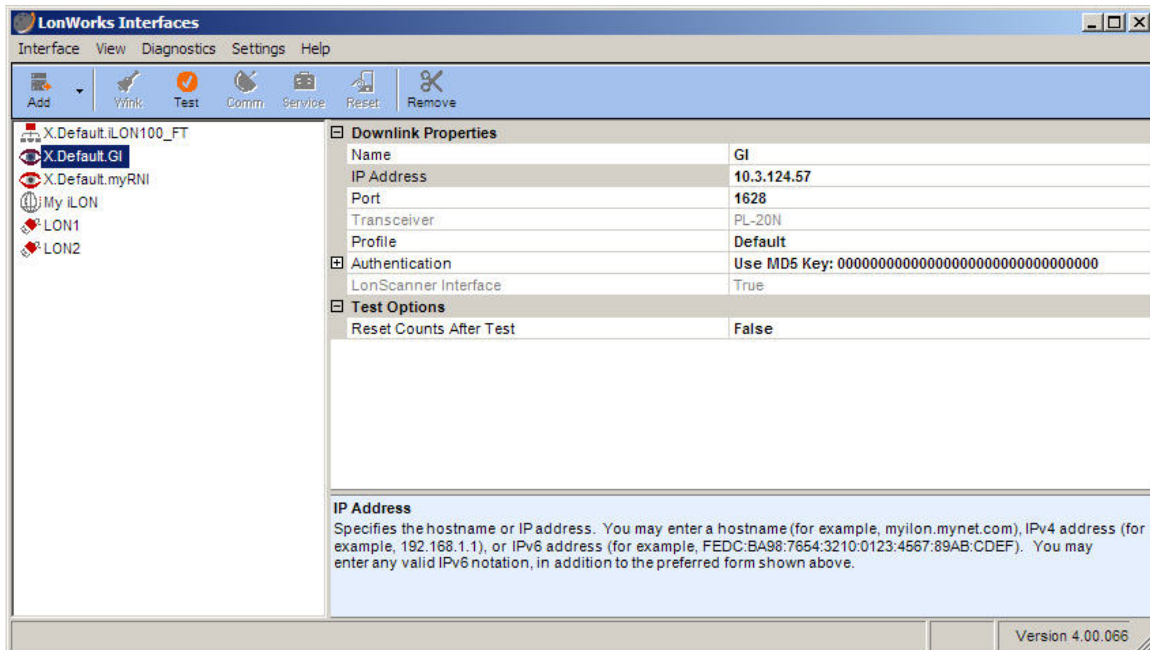


Figure 8. LONWORKS Interfaces Application

2. From the main window, select an RNI Interface, then select **Settings → Edit Profile** to open the Properties dialog, shown in **Figure 9** on page 97.

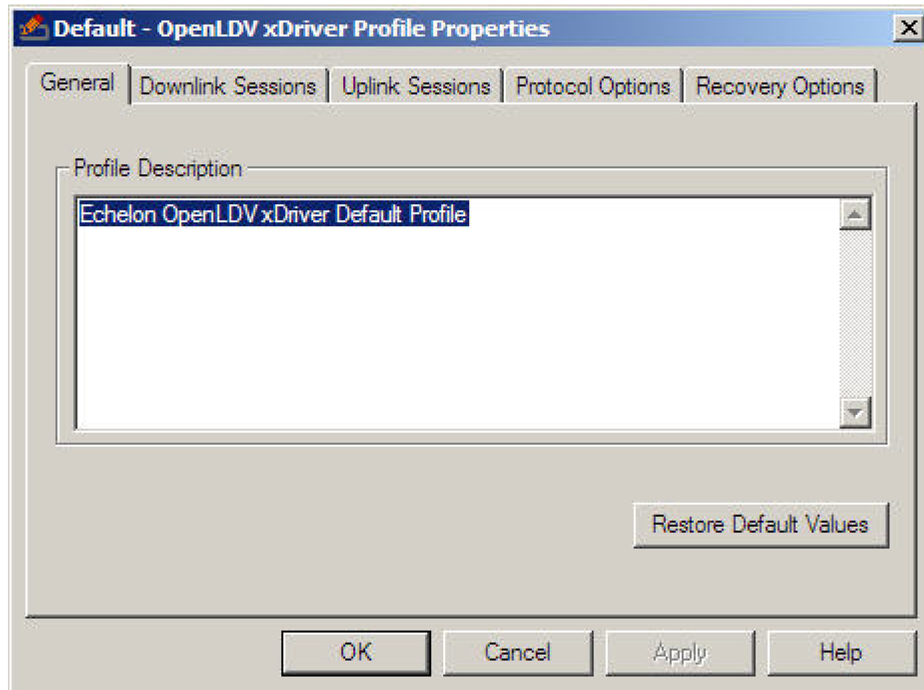


Figure 9. xDriver Profile General Tab

You can edit the description of the xDriver profile by modifying the text in the Profile Description box in the **General** tab. You can also click **Restore Default Values** at any time to restore the default factory settings for the xDriver default profile.

3. Select the **Downlink Sessions** tab to configure how the profile manages downlink sessions, as shown in **Figure 10** on page 98. A downlink session is an xDriver connection that is initiated by an OpenLDV application.

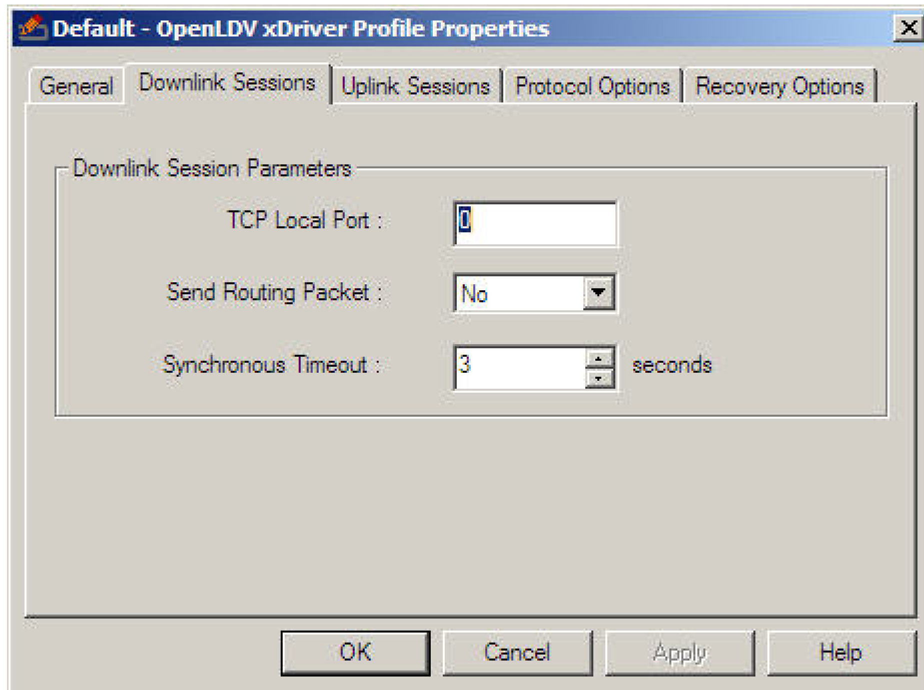


Figure 10. xDriver Profile Downlink Sessions Tab

4. Select the **Uplink Sessions** tab to configure how xDriver manages uplink sessions, as shown in **Figure 11**. An uplink session is an xDriver connection that is initiated when an RNI requests connection to the OpenLDV application.

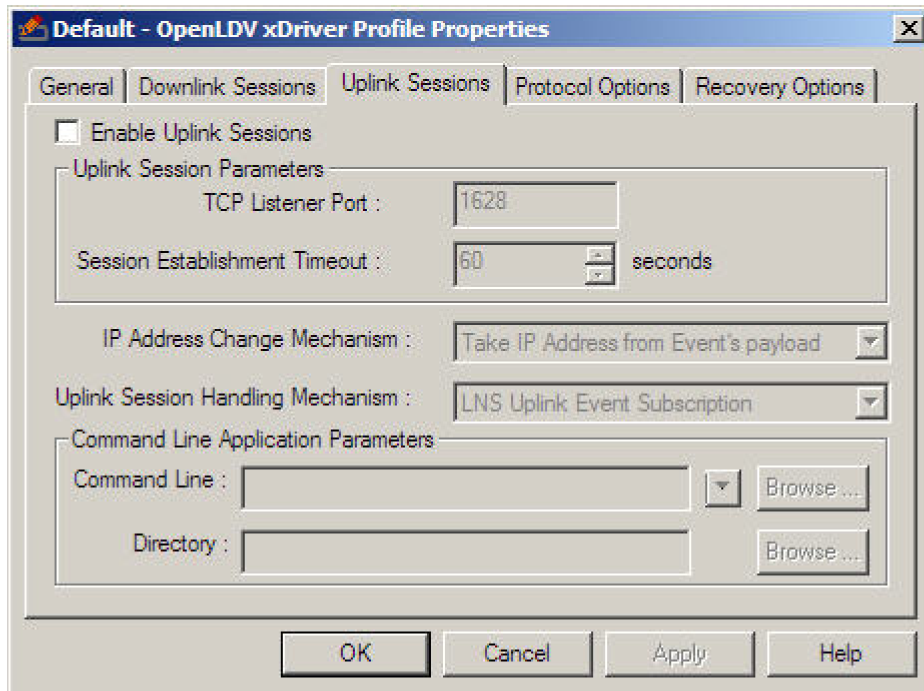


Figure 11. xDriver Profile Uplink Sessions Tab

For xDriver to receive these requests for connection, the xDriver Connection Broker must be running. For information about starting the Connection Broker, see *Starting the Connection Broker* on page 137.

To enable uplink sessions, select the **Enable Uplink Sessions** checkbox, configure the rest of the fields on the tab, and click **OK** to save your changes.

5. Select the **Protocol Options** tab to configure protocol options for xDriver, as shown in **Figure 12**.

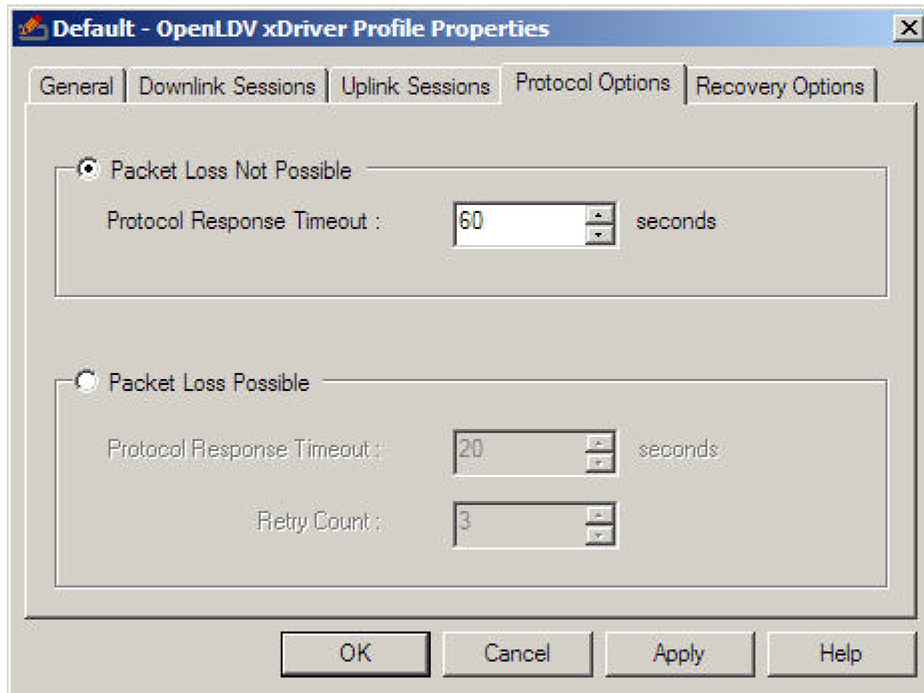


Figure 12. xDriver Profile Protocol Options Tab

6. Select the **Recovery Options** tab to set the recovery options for xDriver, as shown in **Figure 13** on page 100.

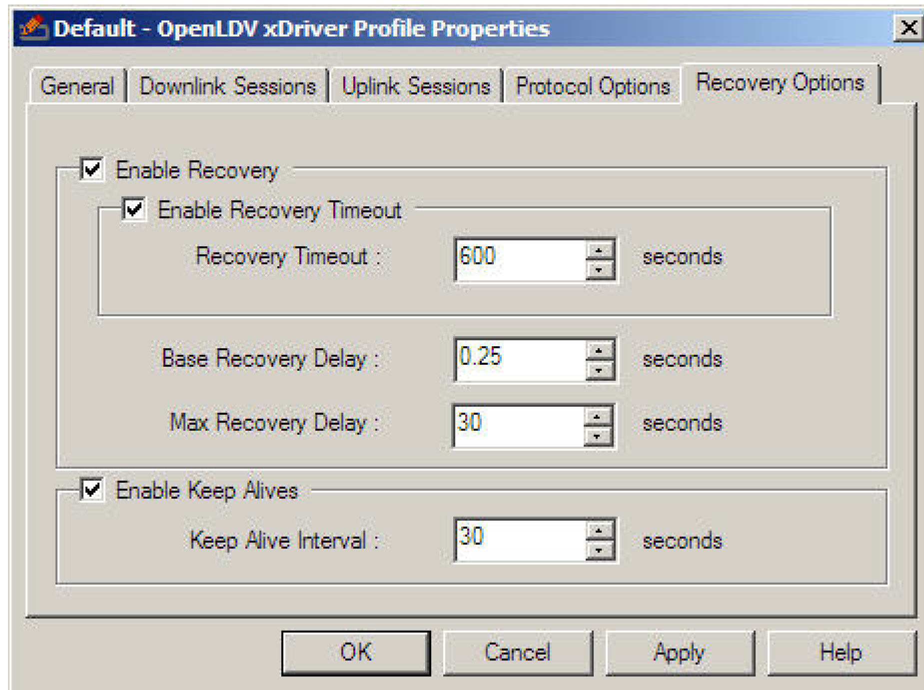


Figure 13. xDriver Profile Recovery Options Tab

You can configure xDriver to automatically attempt reconnection when xDriver sessions that are broken as a result of some unexpected communications failure. Select the **Enable Recovery** checkbox and configure the rest of the fields on the Recovery Options tab to enable the automatic reconnection feature.

7. Click **OK** to save the configuration changes and return to LONWORKS Interfaces application main window.

LNS Applications for xDriver

After you have created Registry entries for your RNIs and configured the default xDriver profile to meet your requirements, you can begin using OpenLDV and LNS applications, such as the LonMaker Integration Tool, to connect the LNS Server to your RNIs. For more information about the LonMaker Integration Tool, see the *LonMaker User's Guide*.

Alternatively, you can begin creating your own OpenLDV or LNS applications for xDriver. Chapter 7, *LNS Programming with xDriver*, on page 139, provides sample programs that can assist you when creating these applications.

6

Extending xDriver

You can extend xDriver by creating custom xDriver lookup extension components, and additional xDriver profiles. This chapter describes why you might need to extend xDriver, and how to extend it.

Most OpenLDV developers will not need to extend xDriver.

Extending xDriver

The OpenLDV driver software includes the LONWORKS Interfaces application, which you can use to create entries in the Windows Registry for each of your RNIs. Each entry stores the lookup information that xDriver requires to connect to one of your RNIs. The default xDriver lookup extension component supplies a COM method that xDriver calls to retrieve this information from the Windows Registry whenever an xDriver connection to an RNI is initiated. The information is then used by xDriver to fully establish the connection.

If you plan to store information for many different RNIs (for example, more than 50), you can improve performance and scalability by using a database management system (DBMS) to store this lookup information, rather than using the Windows Registry. A DBMS provides higher capacity, reliable backup and recovery, faster and more flexible database querying, and security. In addition, a database can be shared by several computers, whereas the Windows Registry is local to a single computer.

If you use a DBMS, you must:

- Replace the default xDriver implementation with a custom lookup extension component. This custom component retrieves the information that xDriver needs to initiate connections from the DBMS.
- Create an xDriver profile to use the custom lookup extension component. An xDriver profile represents a set of configuration parameters that determines how xDriver manages a given connection.

The following gsections describe how the lookup extension component and the xDriver Session Control Object (SCO) interact when an xDriver connection is initiated.

xDriver Sessions

An xDriver session involves a single connection between an RNI and an OpenLDV application. A session begins when a request for connection from the OpenLDV application to an RNI is made (a downlink session), or when a request for connection from an RNI to the OpenLDV application is made (an uplink session). When either request is made, xDriver creates a dedicated SCO for the session. The SCO must be filled in by the xDriver lookup extension component with the information that xDriver needs to establish the connection.

The following sections describe how the SCO is filled in, and how it is used to initiate a connection.

Downlink Sessions

An xDriver session is considered a *downlink* session if the connection is initiated by an OpenLDV application. The OpenLDV application accesses the RNI as though it were opening any other type of network interface.

For an LNS client, if you use a custom lookup extension component, each RNI only appears in the **NetworkInterfaces** collection object during a session that involves that RNI. The network interface name to use is passed to LNS as part

of the downlink lookup key. Chapter 7, *LNS Programming with xDriver*, on page 139, provides programming samples that illustrate this behavior.

Figure 14 shows the application flow for a downlink session.

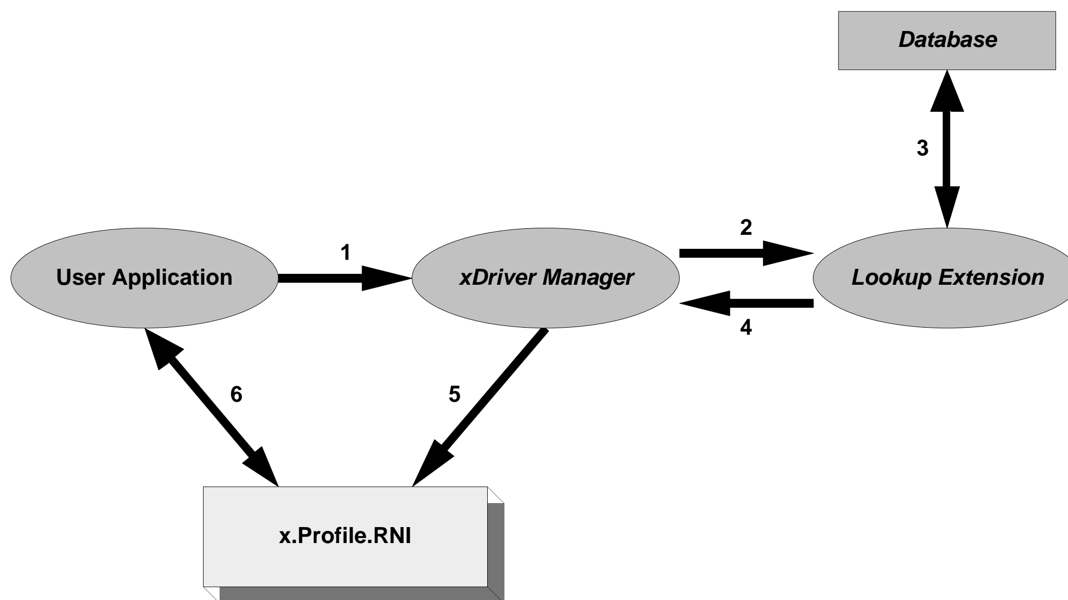


Figure 14. Downlink Session

The application flow for a downlink session includes the following steps (see **Figure 14**):

1. When the OpenLDV application initiates the connection, the xDriver manager initializes an SCO for the session, and adds the downlink lookup key (derived from the network interface name of the RNI) into the SCO.
2. The SCO is then passed to the lookup extension component.
3. The lookup extension component extracts the downlink lookup key from the SCO, and uses it to access the database record for the specified RNI. The lookup extension component then retrieves additional information from the database (such as authentication flag, authentication keys, and IP address and port number of the RNI) to fill in the SCO with the information required to establish the connection.

   For more information about the SCO and the information it stores, see *Session Control Object* on 110. For sample programs that initiate downlink xDriver sessions, see Chapter 7, *LNS Programming with xDriver*, on page 139.
4. The connection is established and the authentication key is used to validate the connection, if authentication is enabled.
5. If the authentication is successful, packets are exchanged in both directions. For more information about authentication, see *Authentication Key Handling* on page 113.
6. The OpenLDV application performs any required network operations.

Figure 15 on page 105 shows the flow of events that occur when a downlink session is initiated within the session-initiating LNS application and the lookup extension component.

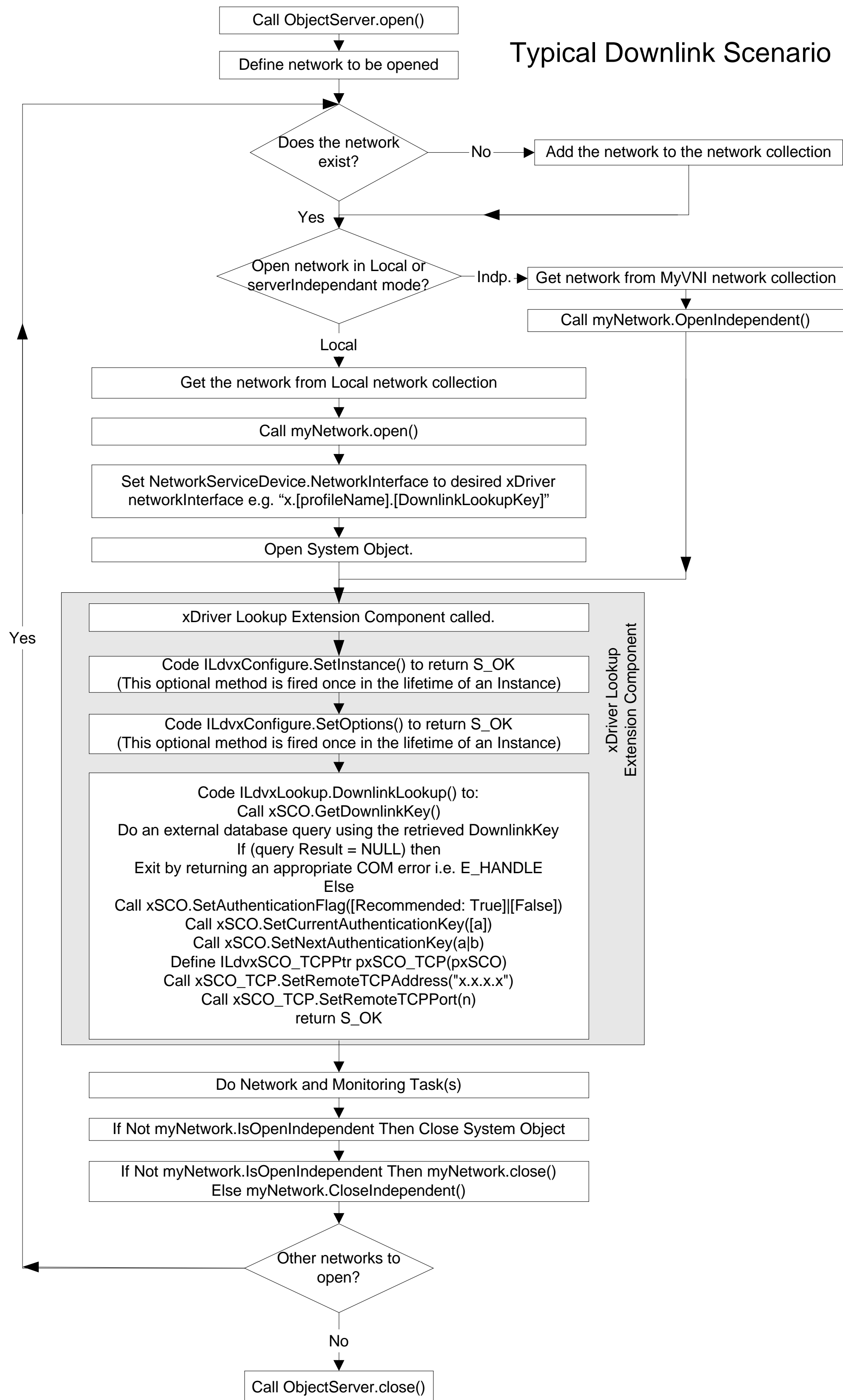


Figure 15. Typical Downlink Session for LNS Server

The events shown in **Figure 15** that occur within the LNS application represent a typical LNS application that opens a downlink session. Your application can vary from these steps.

In addition, the events that occur within the lookup extension component in the flow chart represent the minimal tasks that a lookup extension component must perform during a downlink session. This flow chart refers to the methods that you can use when programming your custom lookup extension component. For more information about these methods, see Appendix C, *Custom Lookup Extension Component Programming*, on page 163.

Uplink Sessions

An xDriver session is considered an *uplink* session if an RNI initiates the session by requesting a connection to an OpenLDV application. This request for connection is usually caused when the RNI receives a message with a qualifying message code.

For xDriver to receive the request, the xDriver Connection Broker must be running. For information about the xDriver Connection Broker, see *Starting the Connection Broker* on page 137.

There must also be at least one xDriver profile with uplink session handling enabled for xDriver to receive the uplink session request. You can use the OpenLDV xDriver Profile Editor to create an xDriver profile that has uplink session handling enabled. Using the Profile Editor, you can assign the profile a port, which the Connection Broker uses to listen for uplink session requests. The profile handles all uplink session requests on that port. For more information about xDriver profiles and the xDriver Profile Editor, see *xDriver Profiles* on page 136.

Figure 16 shows the application flow for a downlink session.

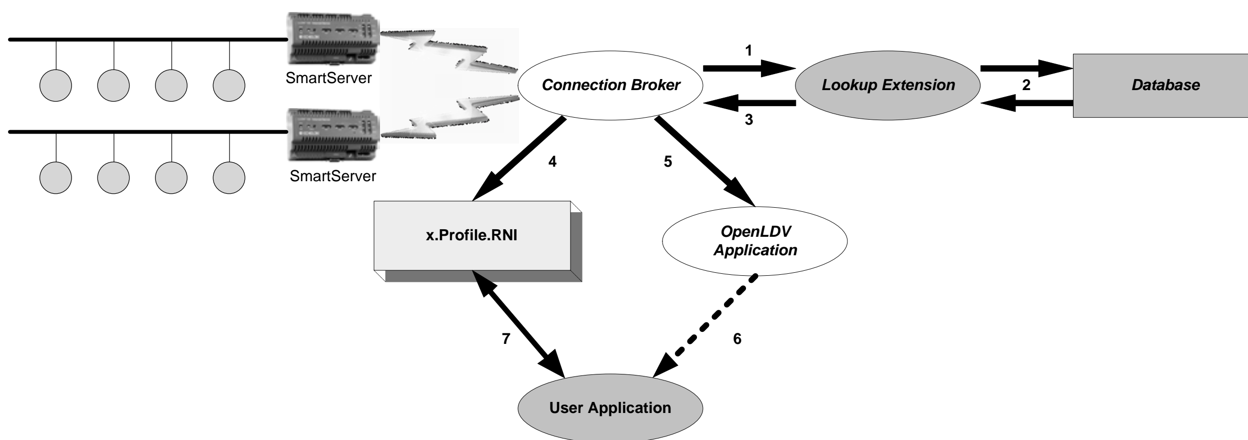


Figure 16. Uplink Session Overview

The application flow for an uplink session includes the following steps (see **Figure 16**):

1. An RNI (for example, a SmartServer) requests an uplink session. When the request for connection is made, the Connection Broker receives an

identification message from the RNI that requested the uplink session. From this message, the Connection Broker obtains the uplink lookup key for the RNI. The SCO for the session is then initialized, and the uplink lookup key is inserted into the SCO.

For RNIs that use a modem to connect to the LNS Server, you can create a listener application that uses the Windows Remote Access Service (RAS) to handle the modem communications with the RNI. This application then passes the uplink request to the Connection Broker. See the Microsoft Developer Network (MSDN) library for more information about Windows RAS programming.

2. The lookup extension component extracts the downlink lookup key from the SCO, and uses it to access the database record for the specified RNI. The lookup extension component then retrieves additional information from the database (such as authentication flag, authentication keys, and IP address and port number of the RNI) to fill in the SCO with the information required to establish the connection.

   For more information about the SCO and the information it stores, see *Session Control Object* on page 110. For sample programs that initiate downlink xDriver sessions, see Chapter 7, *LNS Programming with xDriver*, on page 139.

3. If the authentication flag indicates that authentication is enabled for the session, the xDriver protocol engine uses the authentication keys in the SCO to verify the identity of the request for connection. The xDriver protocol engine handles authentication, and all other message handshaking, when a connection between an LNS Server and an RNI is initiated.

   If authentication fails, the connection is terminated. If authentication succeeds, the following steps occur. For more information about authentication, see *Authentication Key Handling* on page 113.

4. The Connection Broker service creates an entry for the network that requested the uplink session in the **System.NetworkInterfaces** collection.

5. The Connection Broker service sends a message to the OpenLDV application.

6. If the OpenLDV application is an LNS Server, the LNS Server causes the **OnIncomingSessionEvent** event to be fired in an LNS application that is programmed to listen for and manage uplink session requests. The application can then accept or reject the session using the **AcceptIncomingSession** method. These methods are available within LNS for use with xDriver. For more information, see Appendix B, *LNS Methods and Events for xDriver*, on page 155.

   If the application rejects the session, the session is terminated immediately. If it accepts the session, the connection is established, and packets are exchanged in both directions. The LNS application must be running, and must have registered for the uplink session listener event, to receive the uplink session notification. For a sample application that listens for and manages uplink sessions, see *Uplink Sample Application*

on page 144.

You can also use the xDriver Profile Editor to specify a command to run each time that the listener port for that profile receives an uplink session. If you use an LNS Server as the OpenLDV application, the LNS Server provides an enhanced interface for LNS applications.

7. If you are using an LNS Server, after the connection is established, the LNS application can open the remote network interface that requested the connection, enable the monitor set and monitor points for the network, receive the monitor point update event that caused the uplink session request, and handle the event. The monitor set and monitor points can then be closed, followed by closing the network itself.

   The LNS API provides a method to allow the withholding of monitor point update events while an uplink session is started. This method ensures that monitor point update events sent after a network requests an uplink session, but before the network and its monitor set are opened by an LNS application, are not lost, so that the user will receive the monitor point update event that caused the uplink session. For more information about this method, see *ReleasePendingUpdates* on page 160. This feature is only supported by LNS listener applications; it is not supported by command-line initiated uplink event handlers.

   A network interface can reset after receiving and acknowledging (at OSI Layer 2) an alarm event, but before the event has been propagated to the LNS Server, which causes the event to be lost. To prevent this loss, your LNS applications must send monitor point update alarm events for your RNIs to the LNS Server to resend each monitor point update event persistently to the LNS Server until receipt of those events is confirmed. This technique results in reliable performance, and ensures that no monitor point update events are lost before they are processed by the LNS application.

   You must program your LNS application to process uplink request messages, and provide suitable responses to the LONWORKS network, in a timely fashion. Timely responses are particularly critical when you are using a SmartServer and the uplink is delivered over PPP, and the PPP profile (connection) is exclusively set up for your SmartServer. Other SmartServer applications cannot use the PPP link until the LNS application terminates the xDriver. During this time, SmartServer Web connections and alarm notifications that are configured to use a different PPP profile (connection) fail.

Figure 17 on page 109 shows the flow of events that occur during an uplink session within the LNS application that receives the uplink session request and the lookup extension component.

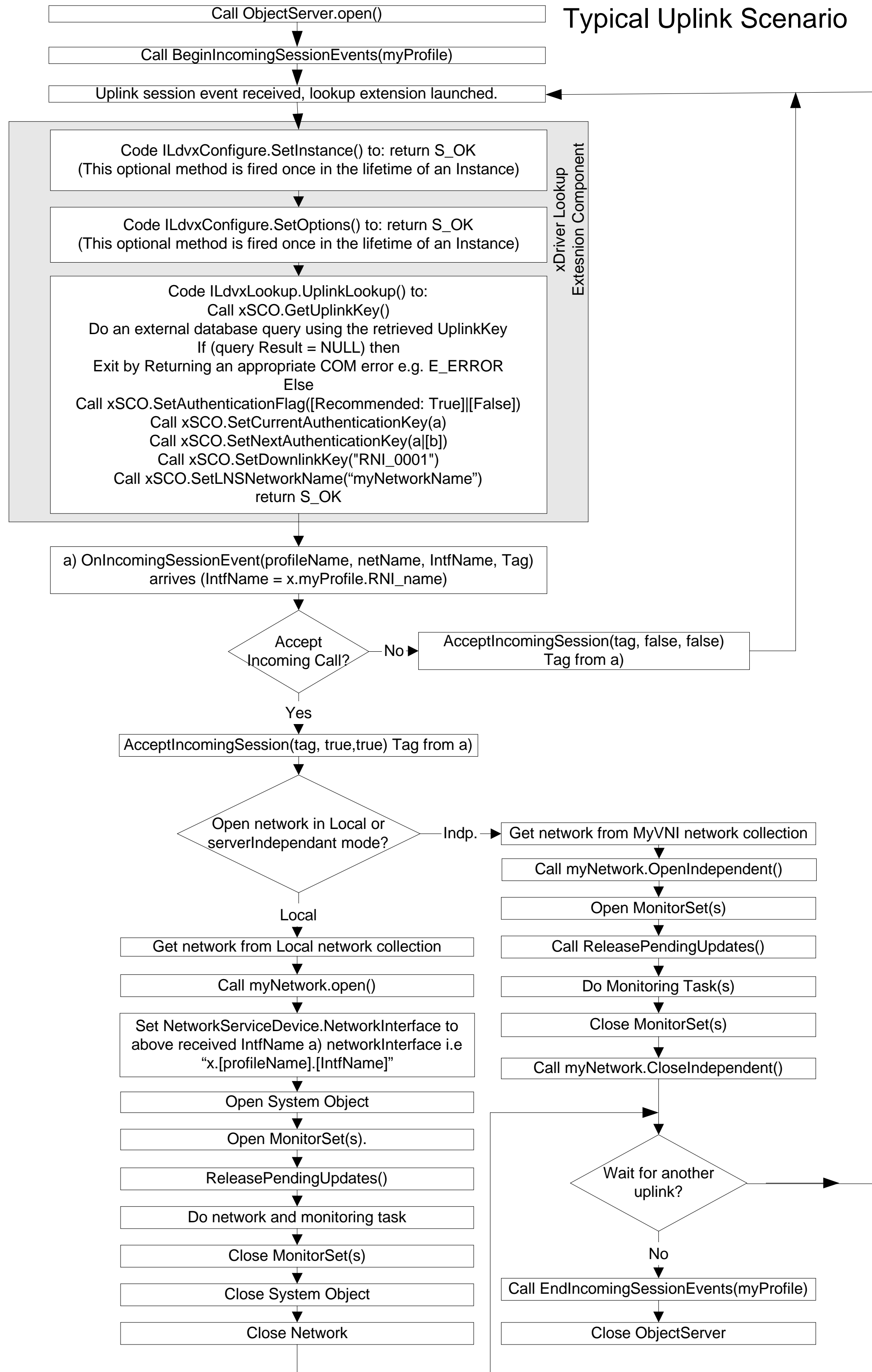


Figure 17. Uplink Session for an LNS Application

The events shown in **Figure 17** that occur within the LNS application represent a typical LNS application that registers for uplink session event handling. Your application can vary from these steps.

In addition, the events that occur within the lookup extension component in the flow chart represent the minimal tasks that a lookup extension component must perform during an uplink session. This flow chart refers to the methods that you can use when programming your custom lookup extension component. For more information about these methods, see Appendix C, *Custom Lookup Extension Component Programming*, on page 163.

Session Control Object

The SCO is created at the beginning of an xDriver session. During its creation, the lookup key of the RNI involved in the session is filled into the SCO. The SCO is then passed to the lookup extension component, which extracts the lookup key from the SCO and uses it to access the xDriver database. The lookup extension component then fills in the rest of the SCO with additional information required to initiate the connection.

Table 35 describes the fields that are filled into the SCO, and how the lookup extension uses them. If you use a database management system as your xDriver database, you must create a custom lookup extension component to access the database and fill in the SCO fields described in **Table 35**.

Creating a Custom Lookup Extension in C++ on page 115 and *Creating a Custom Lookup Extension in Visual Basic* on page 127 describe how to create the framework for a custom lookup extension component using Microsoft Visual Studio 2008. Appendix C, *Custom Lookup Extension Component Programming*, on page 163, describes the xDriver methods that you can use to read and write to each of these SCO fields. The appendix also contains field type constraints to be used when creating a custom database table.

The lookup extension component has read/write access to all of the fields in **Table 35**, and the lookup extension component is required to set each field unless otherwise noted.

Table 35. Session Control Object

Field	**Description**
Session Control Object ID	The SCO ID is a unique, read-only, 32-bit field that is filled in when the SCO is created, before the first call to the lookup extension. It can be used to identify the SCO.
Downlink Lookup Key	The downlink lookup key is an ASCII string (105 characters maximum) that is used by the lookup extension component to access the xDriver database. This field is specified as part of the network interface name for an RNI. For more information about the xDriver network interface naming convention, see *Downlink Sample Applications* on page 140. This field is specified in the network interface name of the RNI during a downlink session, and is read-only during downlinks. This field must be filled in during uplink sessions.

Field	Description
Uplink Lookup Key	The uplink lookup key is an ASCII string (105 characters maximum) passed to the lookup extension component by the RNI during an uplink session. It is used by the lookup extension to access the database record for the RNI that requested the uplink session, so that the lookup extension component can fill in the rest of the fields into the SCO. For a SmartServer or i.LON 600, the uplink lookup key uses the following naming convention: [*Hostname*].[*DNS Suffix*] *[Hostname]* represents the hostname assigned to the SmartServer or i.LON 600 during its configuration. *[DNS Suffix]* is optional, and represents the DNS suffix or domain name assigned to the SmartServer or i.LON 600 during its configuration. For example, if the hostname for an SmartServer or i.LON 600 is “myiLON” and the DNS suffix is “xyz.com”, the uplink key would be “myiLON.xyz.com”. This field is read-only during uplink sessions, and can be optionally filled in during downlink sessions. You can set the downlink key to match the uplink key.
Authentication Flag	This Boolean flag specifies whether authentication between the OpenLDV application and the RNI is enabled for the session. This field is always set to true for the SmartServer and i.LON 600.
Current Authentication Key	If the authentication flag is enabled, this field represents the authentication key to be used for the session. This authentication key must match the MD5 authentication key supplied to the RNI during its configuration. Using an MD5 authentication key prevents the OpenLDV application or the RNI from responding to unauthorized messages during an xDriver session. The authentication key must be entered as a 32-character hexadecimal string representing a 128-bit MD5 key. For example: `0102030405060708090A0B0C0D0E0F10` Setting the authentication key to all 0s causes xDriver to use the pre-defined, default factory authentication key for the RNI. The default factory authentication key is not secure. For more information about how the lookup extension component handles authentication, see *Authentication Key Handling* on page 113.

Field	Description
Next Authentication Key	This field represents the next authentication key to be used by the RNI. The next authentication key is usually the same as the current authentication key. The authentication key must be entered as a 32-character hexadecimal string representing a 128-bit MD5 key. For example: `0102030405060708090A0B0C0D0E0F10` You can initiate a change to the authentication key used by the RNI by changing this field to a value different than the current authentication key. The authentication key configured into the RNI will then be updated to match this field. Setting this field to all 0s causes xDriver to use the default authentication key as the next key for the RNI. The default factory authentication key is not secure. When no change to the current authentication key is desired, this key must be the same as the current authentication key. For more information about how the lookup extension component handles authentication, see *Authentication Key Handling* on page 113.
LNS Network Name	The name of the LNS network to be opened. This field is optional, and only used if an LNS Server is the client and the session is an uplink session, because the network name is specified manually within the LNS application in a downlink session. For an example of such an application, see *Opening a Single Remote Network With xDriver* on page 140. The LNS network name can be a maximum of 85 characters.
Additional Downlink Packet Header Additional Downlink Packet Trailer	These fields are not required under most circumstances, and are only applicable to downlink sessions. They can be used to specify a series of bytes to be prepended or appended to every packet sent during a downlink session if there is an intermediate proxy between the OpenLDV application and the RNI. These bytes can be used to provide routing information that the proxy might require. When you configure an xDriver profile that uses these properties, you might also need to select Send Routing Packet, which causes xDriver to send the proxy a null packet (with the header and trailer bytes specified here) when the connection is established. This selection provides the proxy with the information that it requires to route the connection properly. For more information, see *xDriver Profiles* on page 136.
Encryption Type	The type of encryption used by the RNI. Currently, xDriver supports RC4 encryption only. Selected packets are encrypted using an RC4 encryption algorithm if this option is selected.

Field	Description
Remote TCP Address	The TCP/IP address of the RNI to which to connect. For an uplink session, this field is read-only. The remote TCP address must be specified in the form *x.x.x.x*, where *x* represents an integer between 0 and 255. A DNS-resolvable hostname can also be specified for this field.
Remote TCP Port	The port number that the RNI uses for incoming connections from the OpenLDV application. For an uplink session, this field is read-only. Valid port numbers are 1 to 65535.

Authentication Key Handling

Authentication key handling is an essential part of any lookup extension component. Your custom lookup extension component must fill the authentication key fields into the SCO, and properly handle changes to the authentication key fields. **Figure 18** shows a high-level view of authentication key handling. The figure shows a sample authentication key (ABCD) that does not use the required format.

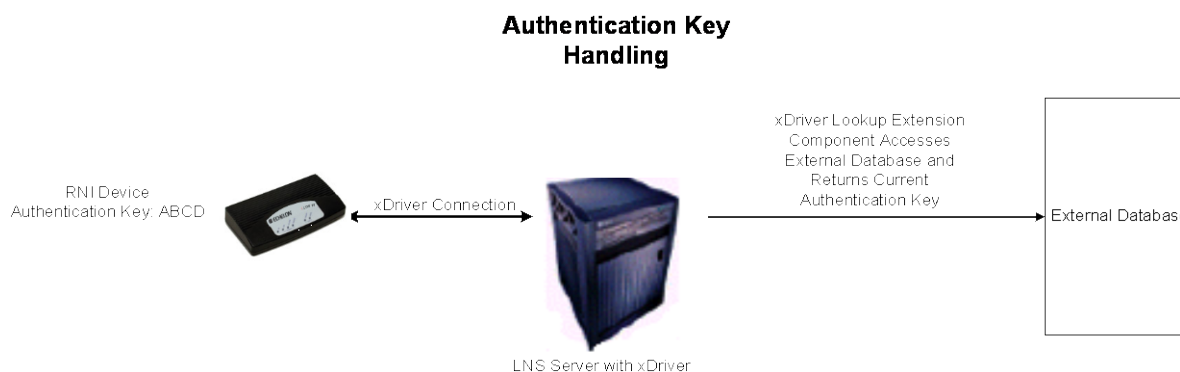


Figure 18. Authentication Key Handling

Note: This section refers to the methods that you can use when programming your custom lookup extension component. For more information about these methods, see Appendix C, *Custom Lookup Extension Component Programming*, on page 163.

Setting the Current Authentication Key

When an xDriver session is initiated, the lookup extension component must access the database, extract the authentication flag for the RNI from the database, and fill it into the SCO using the **SetAuthenticationFlag** method. If the authentication flag is **True**, indicating that authentication is currently enabled, the lookup extension component must also extract the current MD5 authentication key for the RNI from the database, and fill it into the SCO using the **SetCurrentAuthenticationKey** method. The authentication key must be a

unique, 32-character hexadecimal string representing the 128-bit MD5 key that is used by the RNI.

The xDriver lookup interface includes the **SetNextAuthenticationKey()** method, which fills the next authentication key to be used by the RNI into the SCO. If no change the authentication key used by the RNI is desired, the next authentication key must be the same as the current authentication key. This field must be filled in immediately after the current authentication key is filled in.

After these SCO fields are filled in, the xDriver protocol engine generates a 128-bit digest based on the current authentication key; this digest is sent as part of every message to the RNI at the other end of the connection. The digest is extracted by the RNI and compared to a digest produced by the authentication key configured into the RNI. If the two digests match, then the two keys must match and the authentication succeeds.

The current and next authentication keys filled into the SCO must match the authentication key configured into the RNI. You can fill in an authentication key of all 0s to use the pre-defined, default factory authentication key for the RNI as the current authentication key. The default factory authentication key is not secure.

Changing the Current Authentication Key

You can use the **SetNextAuthenticationKey** method from your lookup extension component to change the authentication key within an RNI by filling a next authentication key into the SCO that is different from the current authentication key. This method initiates an incremental change to the authentication key that is configured into the RNI, so that it will end up with the key specified as the Next Authentication Key as its authentication key.

After this change is complete, xDriver calls the **UpdateLookup** method in the lookup extension component to acknowledge the change to the RNI's authentication key. The lookup extension component muts implement an update to the database from the **UpdateLookup** method, so that the new current value of the authentication key is recorded in the database, and the current authentication key in the database matches the key in the RNI. The current and next authentication keys must always be stored in the database, and can only be updated when the **UpdateLookup** method is called.

Table 36 on page 115 describes the flow of events that occurs when the next authentication key field is used to update the authentication key of an RNI. In this example, the lookup extension fills different MD5 authentication keys into the current authentication key and next authentication key fields into the SCO. The table uses a sample authentication key (such as ABCD) that does not use the required format.

Table 36. Changing Authentication Keys

Phase One, Lookup Extension Component Is Called	
Initially, the current authentication key must match the authentication key configured into the RNI for the connection to be established. The authentication key fields start with the following values:	
RNI Authentication Key	ABCD
SCO Current Authentication Key	ABCD
SCO Next Authentication Key	EFGH
Current Authentication Key in the Database	ABCD
Next Authentication Key in the Database	EFGH
Phase Two, RNI's Key Updated	
Because the current and next authentication keys filled into the SCO differ, the authentication key configured into the RNI is incremented so that it matches the next authentication key (EFGH). In addition, the current authentication key stored in the SCO is updated to match the next authentication key. The authentication key fields now have the following values:	
RNI Authentication Key	EFGH
SCO Current Authentication Key	EFGH
SCO Next Authentication Key	EFGH
Current Authentication Key in the Database	ABCD
Next Authentication Key in the Database	EFGH
Phase Three, Update Lookup Method Called	
The **UpdateLookup** method is called after the authentication key configured into the RNI, and the current authentication key in the SCO, have been changed. This method must update the database so that it is updated with the new values of the current authentication key and the next authentication key from the SCO. The current and next authentication keys must always be stored in the database, and can be updated only when the **UpdateLookup** method is called. The next time there is a session with this RNI, the lookup extension will fill in the proper value for the current authentication key. After update lookup has been called, the authentication key fields should have the following values:	
RNI Authentication Key	EFGH
SCO Current Authentication Key	EFGH
SCO Next Authentication Key	EFGH
Current Authentication Key in the Database	EFGH
Next Authentication Key in the Database	EFGH

Creating a Custom Lookup Extension in C++

This section describes the procedure for creating the framework for a custom lookup extension component in C++ using Microsoft Visual Studio 2008. Use a similar procedure for Microsoft Visual Studio 2010, or later releases.

Prerequisite: You must install the OpenLDV 4.0 SDK and Microsoft Visual Studio 2008 SP1 (or later).

Important: If you use the sample project source files in your new project, you must rename the GUIDs in the .IDL and .RGS files.

To create the framework for a custom lookup extension component using Microsoft Visual Studio 2008, perform the following tasks:

1. Start Microsoft Visual Studio
2. Create a new project using the Active Template Library (ATL)
3. Add a COM Object
4. Implement the **ILdvxLookup** interface
5. Add the lookup extension to the xDriver Lookup Component category
6. Build and register the COM server
7. Create a custom xDriver profile
8. Test the lookup extension
9. Rebuild and re-register the COM server
10. Retest the lookup extension
11. Optionally, add implemented **ILdvxLookup** interfaces to the sample component's type library
12. Optionally, add additional private methods or properties to the **ISampleLookupCsv** interface

The following sections describe these tasks in more detail.

Create a New Visual Studio Project

From the Visual Studio main window, select **File → New → Project** to open the New Project dialog, as shown in **Figure 19** on page 117.

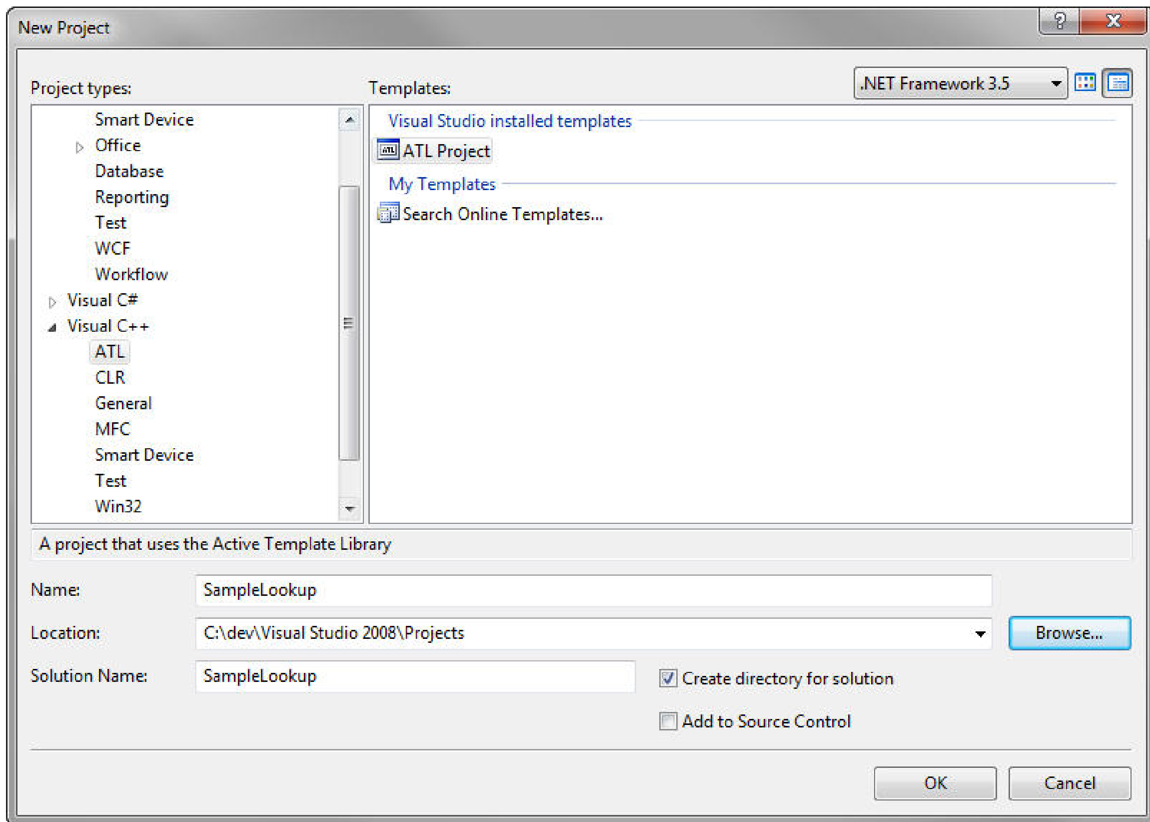


Figure 19. Visual Studio New Project Dialog

In the New Project dialog:

- Expand the C++ category and select **ATL** as the project template
- Specify the version of the .NET Framework to use
- Specify a name and location for the project
- Specify a name and location for the solution

You can use any project location for development. Install the completed custom lookup extension component in the LONWORKS **\xDriver\Components *CompanyName*\Lookup** folder when you distribute the DLL for your application.

Click **OK** to open the ATL Project Wizard.

From the Application Settings page of the ATL Project Wizard, select **Dynamic-link library (DLL)** for the server type, as shown in **Figure 20** on page 118. You can use other server types, but a DLL is likely to provide the best performance.

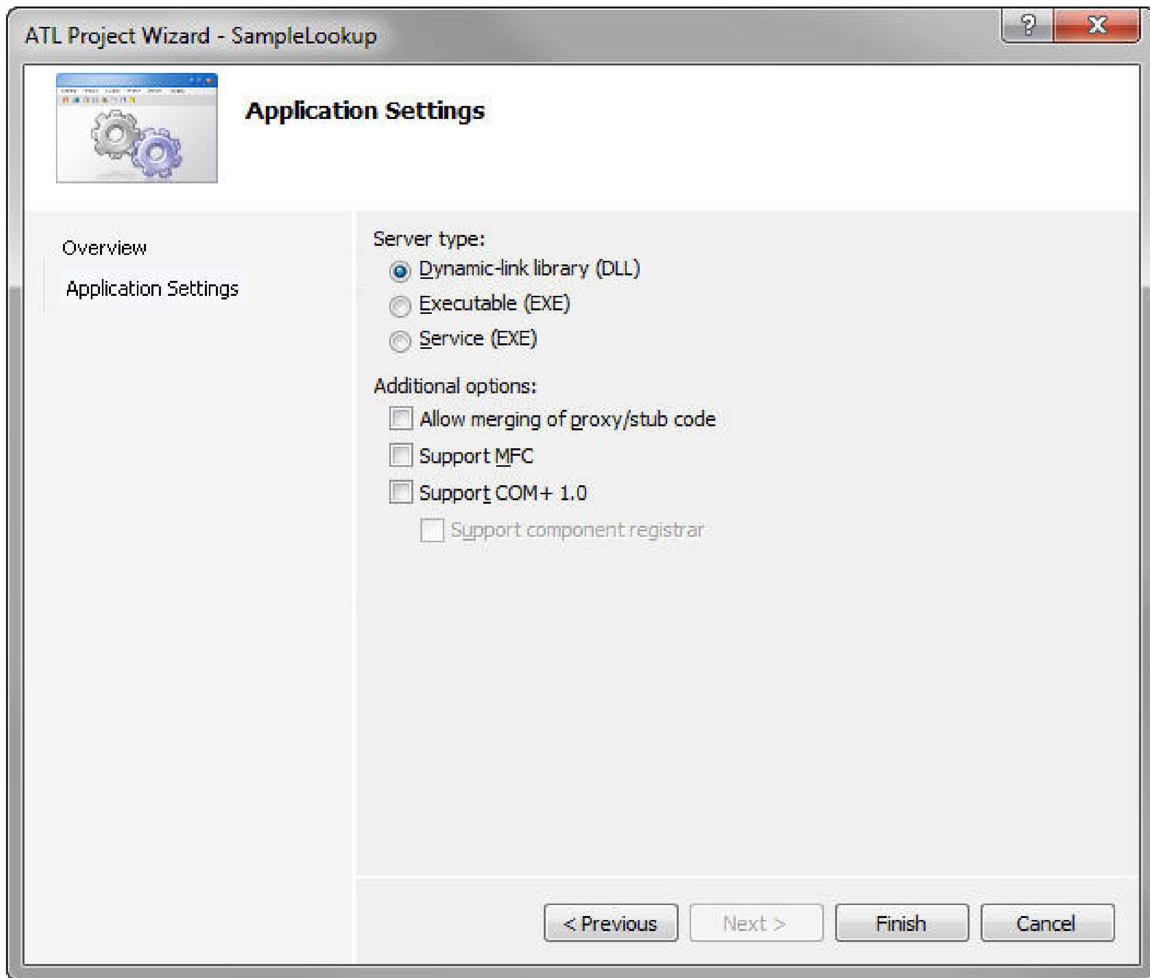


Figure 20. Visual Studio ATL Project Wizard

Click **Finish** to close the ATL Project Wizard and create the project.

Add a COM Object

From the Visual Studio main window, select **Project** → **Add Class** to open the Add Class dialog, as shown in **Figure 21** on page 119.

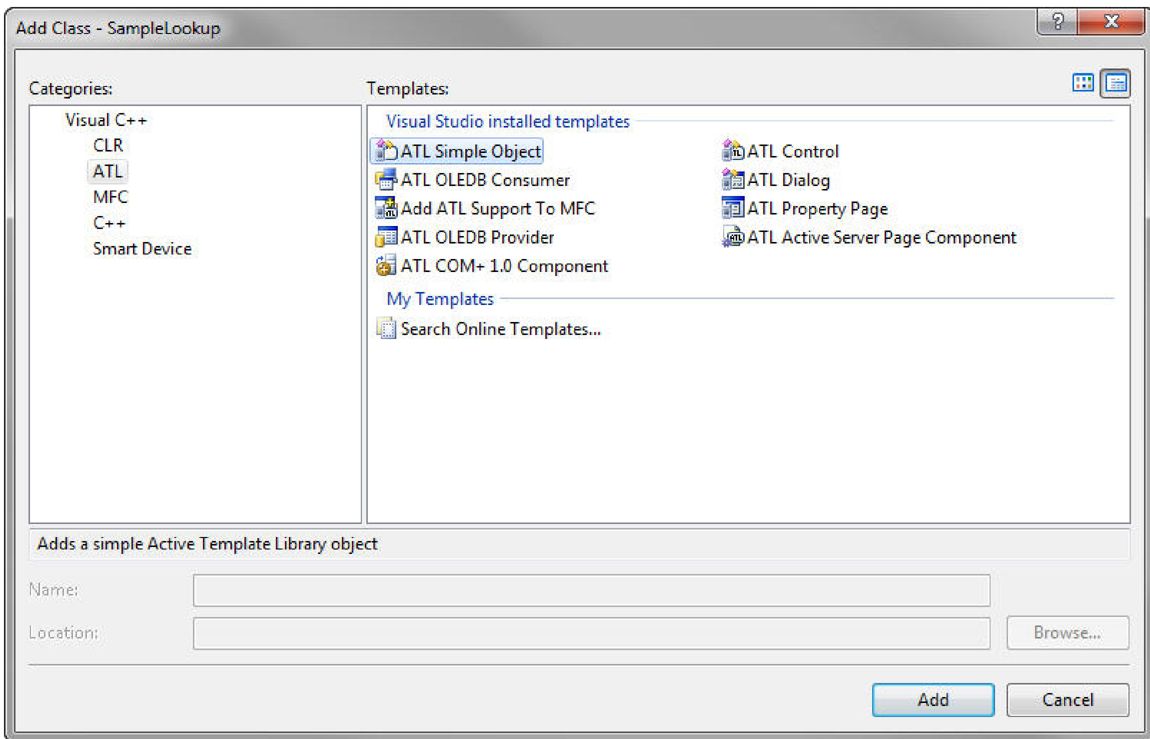


Figure 21. Visual Studio Add Class Dialog

From the Add Class dialog, select the **ATL** category, select the **ATL Simple Object** template, and click **Add** to open the ATL Simple Object Wizard, as shown in **Figure 22** on page 120.

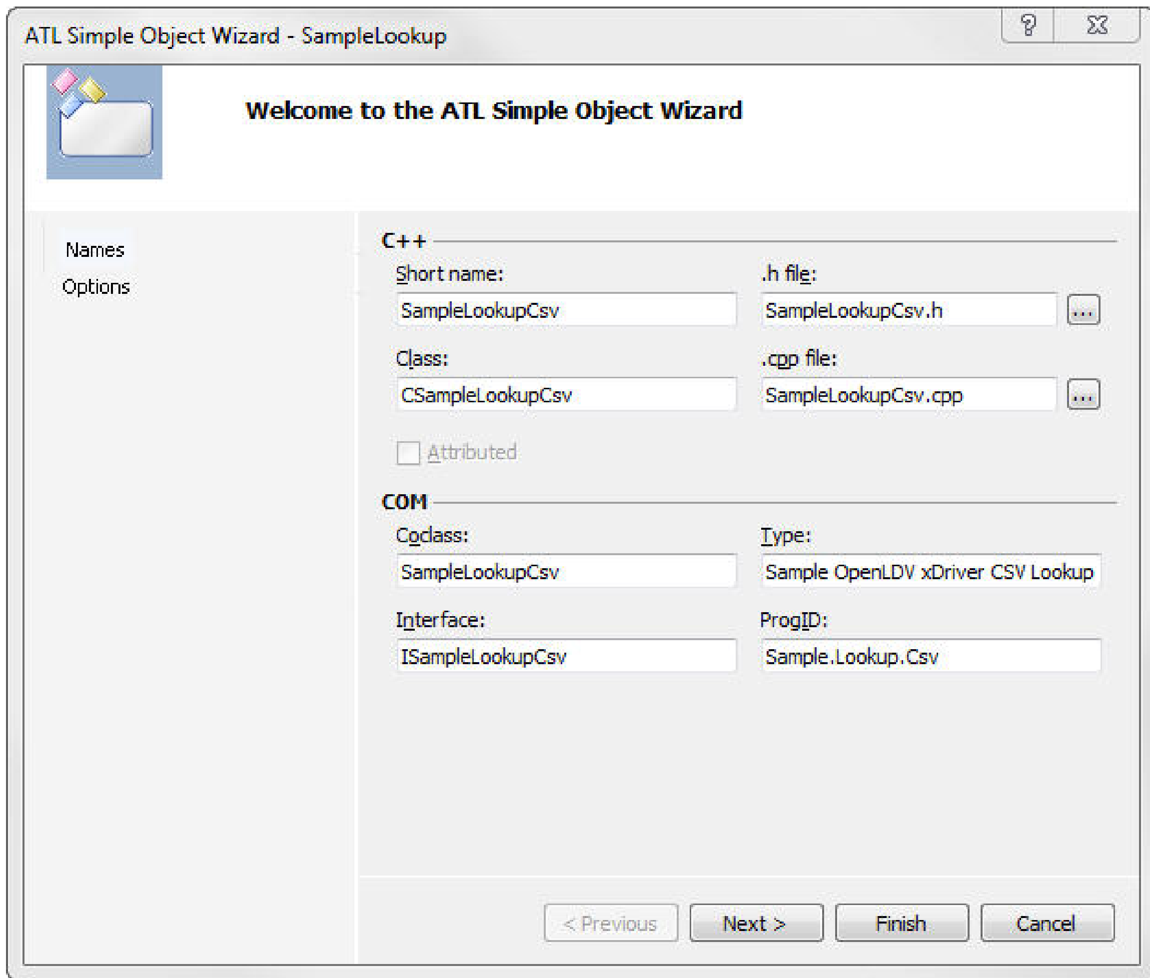


Figure 22. Visual Studio ATL Simple Object Wizard – Names Page

From the ATL Simple Object Wizard Names page, enter a name for the lookup extension component in the **Short Name** text box and fill in the rest of the fields as required. None of these fields should begin with "xDriver." In addition, the short name should not match the project name (see *Create a New Visual Studio Project* on page 116). The COM interface is your automation interface, and is not used by xDriver.

The program ID should not match the program ID for any other lookup or COM component on your computer, or on any computer on which the application will be installed, because it will be used by the Profile Editor to identify the lookup extension component.

Recommendation: Specify the program ID using the following naming convention:

[*Company Name*].**Lookup**.[*Type*]

where [*Company Name*] represents the name of your company and [*Type*] represents the type of database this lookup extension component uses. For example: "MyCompany.Lookup.ODBC".

From the ATL Simple Object Wizard, select the Options page, as shown in **Figure 23**.

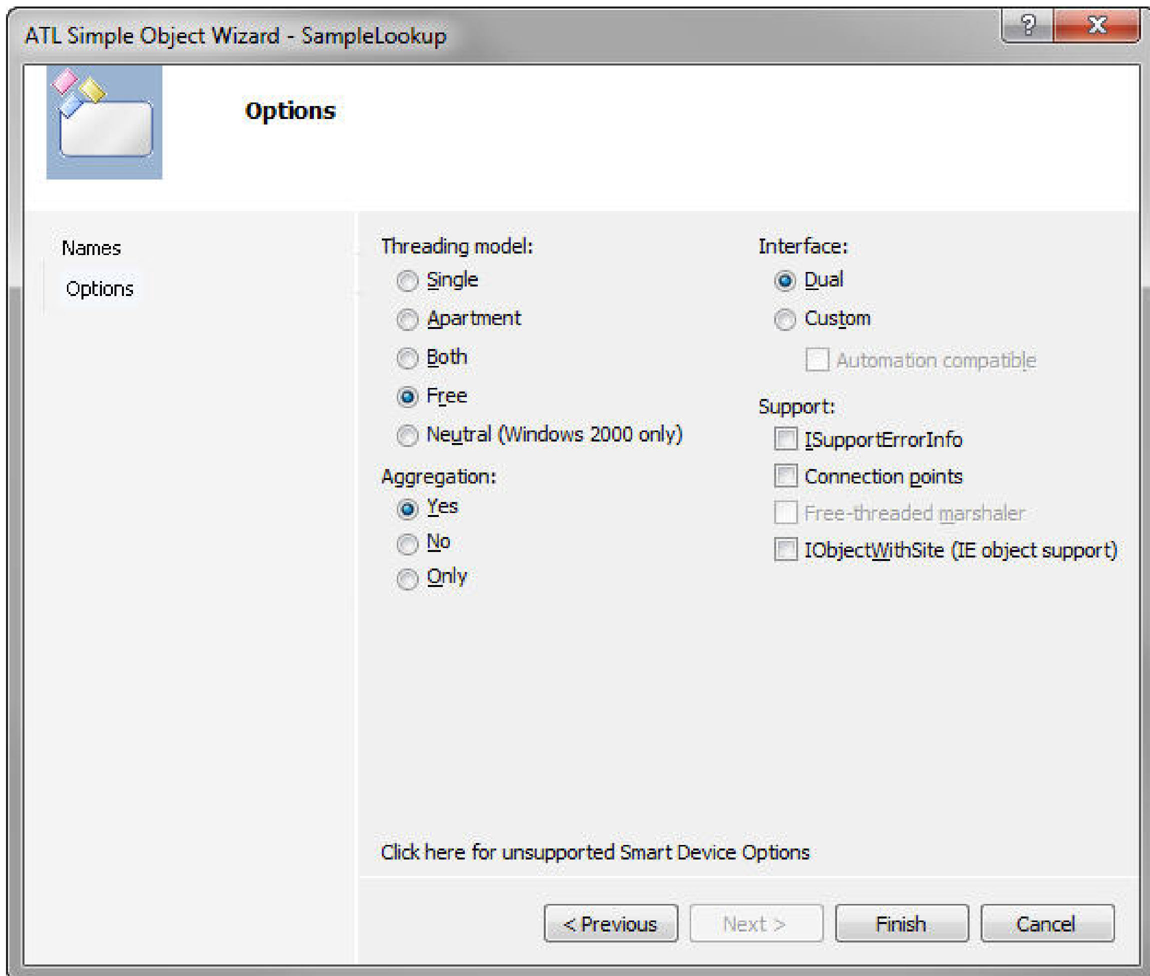


Figure 23. Visual Studio ATL Simple Object Wizard – Options Page

For optimal performance, select **Free** as the threading model. This model requires that your extension be multithread safe, and allows your component to directly access other xDriver extension components.

Click **Finish** to close the ATL Simple Object Wizard and create the object.

Implement the ILdvxLookup Interface

From the Visual Studio main window, open the class view window (**View → Class View**). From the class view, right-click **CSampleLookupCsv** and select **Add → Implement Interface** to open the Implement Interface wizard, as shown in **Figure 24** on page 122.

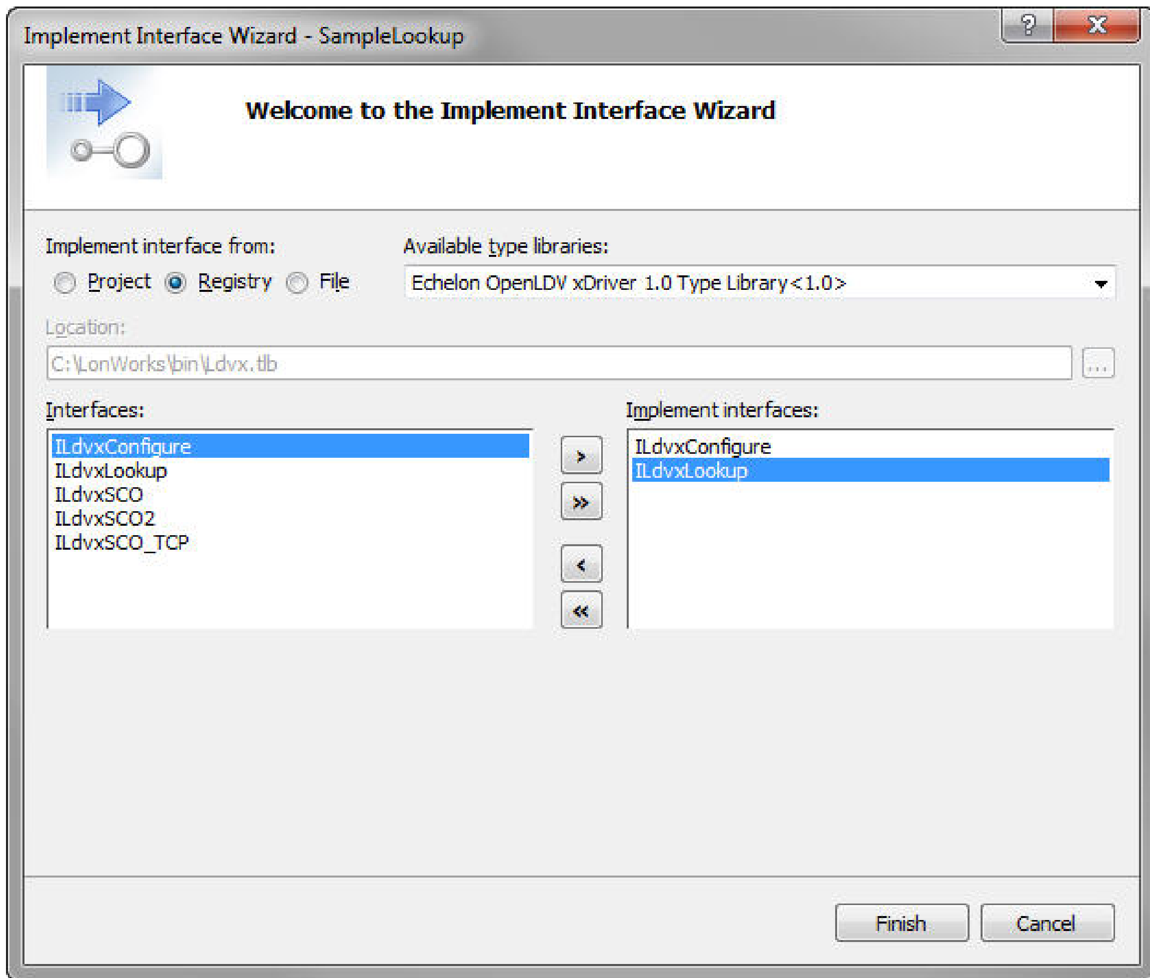


Figure 24. Visual Studio Implement Interface Wizard

Within the Implement Interface wizard, select **Echelon OpenLDV xDriver 1.0 Type Library** from the **Available type libraries** dropdown listbox. For the **Implement interface from** radio button, specify either **Registry** or **File**. If you specify **File**, specify **\LonWorks\bin\Ldvx.tlb** for the **Location**.

For your custom lookup extension component, implement the following interfaces:

- ILdvxLookup
- ILdvxConfigure (optional)

Within the **SampleLookupCsv.h** header file, implement the methods for **ILdvxLookup** and **ILdvxConfigure**. That is, replace the `return E_NOTIMPL` stub code with your own extension-specific code. You code must return COM success or failure codes to xDriver. See the xDriver Lookup Example (**Start → All Programs → Echelon OpenLDV 4.0 SDK → Examples & Tutorials**) for example implementations.

```
    // ILdvxConfigure Methods
public:
    STDMETHOD(SetInstance)(BSTR instance)
    {
        return E_NOTIMPL;
```

```
    }
    STDMETHOD(SetOptions)(BSTR options)
    {
       return E_NOTIMPL;
    }

    // ILdvxLookup Methods
public:
    STDMETHOD(DownlinkLookup)(ILdvxSCO * xSCO)
    {
       return E_NOTIMPL;
    }
    STDMETHOD(UplinkLookup)(ILdvxSCO * xSCO)
    {
       return E_NOTIMPL;
    }
    STDMETHOD(UpdateLookup)(ILdvxSCO * xSCO)
    {
       return E_NOTIMPL;
    }
};
```

Add the Extension to the Component Category

To allow the OpenLDV xDriver Profile Wizard (and other xDriver-related tools) to display your custom lookup extension, you must register the lookup extension as belonging to the xDriver Lookup Component Category:

1. Add the following lines to your component's header file (after `END_COM_MAP()`in **SampleLookupCsv.h**):

   ```
   // COM component category map
   // (CATID_LdvxLookup is defined in LdvxTypes.h)
   BEGIN_CATEGORY_MAP(CSampleLookupCsv)
      IMPLEMENTED_CATEGORY(CATID_LdvxLookup)
   END_CATEGORY_MAP()
   ```

2. Add the following line at the top of your component's header file:

   ```
   #include "LdvxTypes.h"
   ```

3. Add the directory that contains the **LdvxTypes.h** header file (LONWORKS **\OpenLDV SDK\Include**) to your project's include path (select **Project** → ***ProjectName*** **Properties** to open the project's properties dialog, expand **Configuration Properties**, expand **C/C++**, select **General**, then add the path to the **Additional Include Directories** field).

Build and Register the COM Server

Select **Build** → **Build SampleLookup** to build the SampleLookup part of the solution.

Note: For Windows operating systems that include User Account Control (Windows 7 or Windows Vista), if you do not run Visual Studio with elevated permissions (for example, as an Administrator), you will likely see the following error message during registration:

```
Project : error PRJ0050: Failed to register output.  Please
try enabling Per-user Redirection or register the component
from a command prompt with elevated permissions.
```

Start an elevated command shell (right-click the **Visual Studio 2008 Command Prompt** shortcut and select **Run as administrator**), change to the folder that contains your built DLL, and run the following command:

```
REGSVR32 SampleLookup.dll
```

Create a Custom xDriver Profile

Use the OpenLDV xDriver Profile Editor to create a custom xDriver profile for your custom xDriver lookup extension.

1. Select **Start** → **Echelon OpenLDV 4.0 SDK** → **Developer Utilities** → **xDriver Profile Editor** to open the OpenLDV xDriver Profile Editor.
2. Click **Add** to open the New Profile dialog, as shown in **Figure 25**.

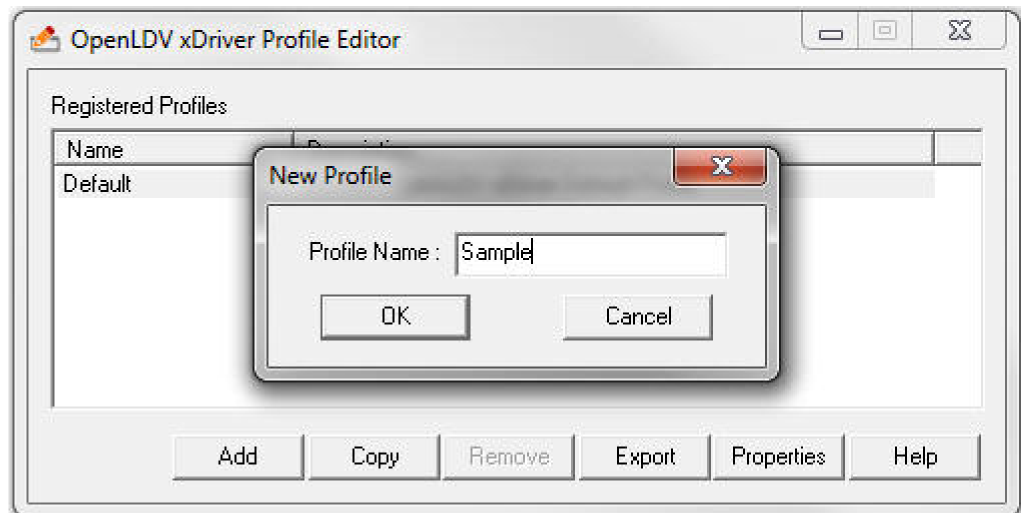


Figure 25. xDriver Profile Editor New Profile Dialog

3. Enter a profile name ("Sample" for this example) in the **Profile Name** field.
4. Click **OK** to close the New Profile dialog.
5. For the General tab of the OpenLDV xDriver Profile Properties dialog for the new profile, enter a description in the Profile Description text box, as shown in **Figure 26** on page 125.

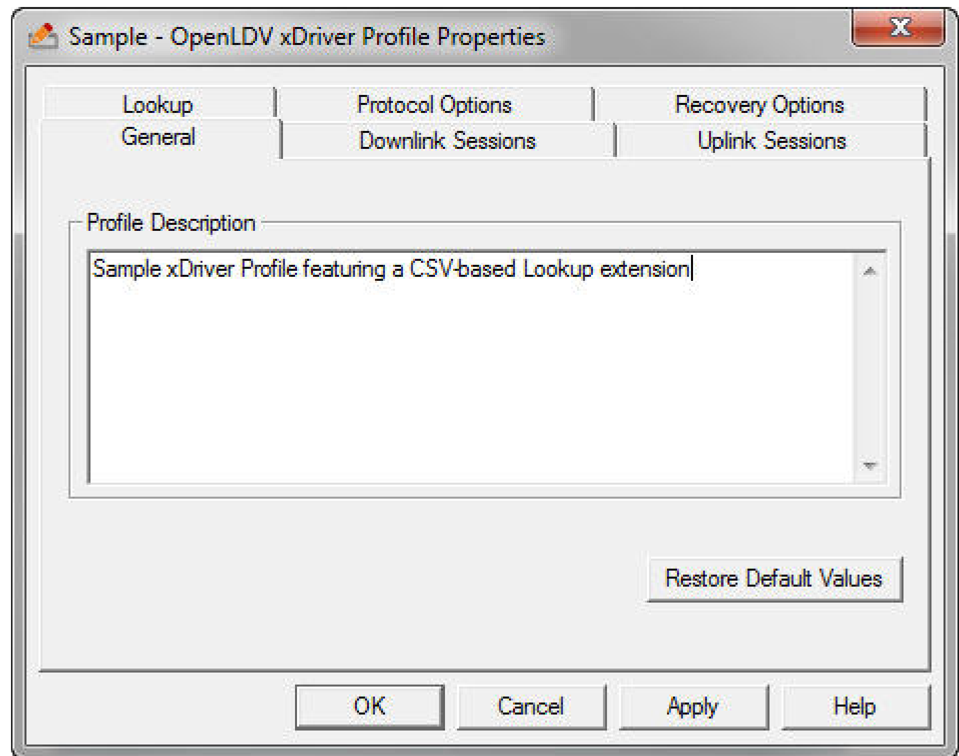


Figure 26. OpenLDV xDriver Profile Properties – General Tab

6. For the Lookup tab of the OpenLDV xDriver Profile Properties dialog for the new profile, select the newly built profile from the **Extension Name** dropdown listbox, as shown in **Figure 27** on page 126.

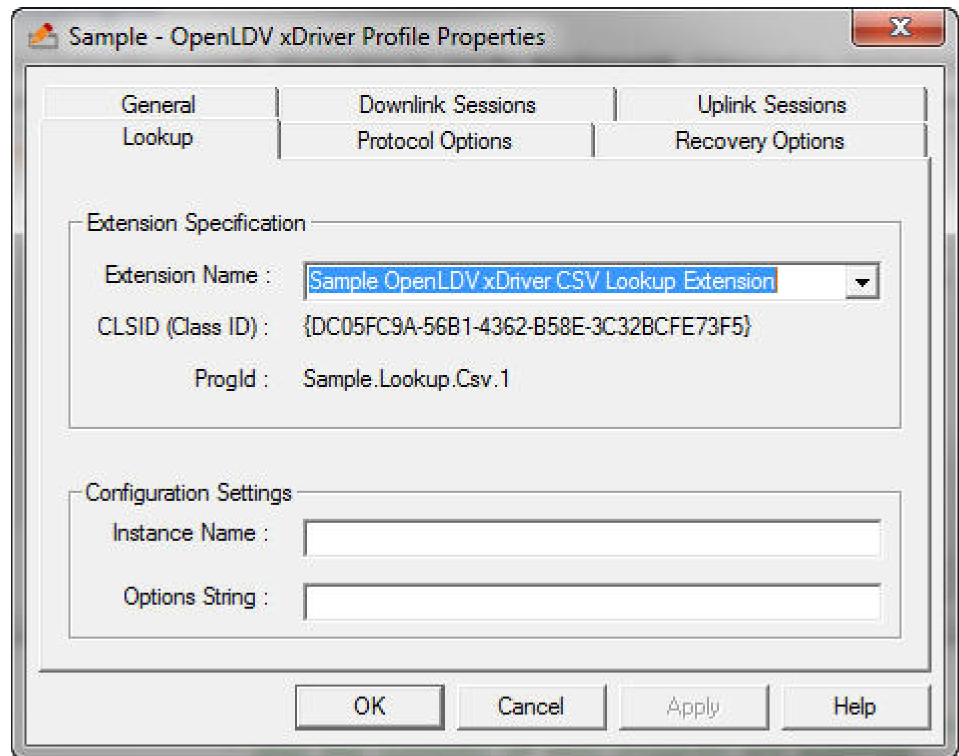


Figure 27. OpenLDV xDriver Profile Properties – Lookup Tab

7. Make any other changes that are appropriate for your custom profile, and click **OK** to accept the changes and close the dialog.

Test the Lookup Extension

Create an RNI for your custom xDriver lookup extension, and give it a name like "**X.Sample.***DownlinkKey*". Use an OpenLDV client program to open the RNI for the custom extension to see calls for the following methods:

- ILdvxConfigure::SetInstance
- ILdvxConfigure::SetOptions
- ILdvxLookup::DownlinkLookup

Note: Until these methods are implemented (see *Implement the ILdvxLookup Interface* on page 121), the default **E_NOTIMPL** return status will cause the startup sequence to fail.

Optional Steps

The following steps are optional:

- Add implemented **ILdvxLookup** interfaces to the sample component's type library; add the following to the library section of **SampleLookup.idl**:

```
importlib("Ldvx.tlb");     // TLB must be in PATH
interface ILdvxLookup;     // add to coclass SampleLookupCsv
```

```
interface ILdvxConfigure; // add to coclass SampleLookupCsv
```

- Add additional private methods and properties to your **ISampleLookupCsv** interface, or remove it if it is not used.

Creating a Custom Lookup Extension in Visual Basic

This section describes the procedure for creating the framework for a custom lookup extension component in Visual Basic using Microsoft Visual Studio 2008. Use a similar procedure for Microsoft Visual Studio 2010, or later releases.

Prerequisite: You must install the OpenLDV 4.0 SDK and Microsoft Visual Studio 2008 SP1 (or later).

To create the framework for a custom lookup extension component using Microsoft Visual Studio 2008, perform the following tasks:

1. Start Microsoft Visual Studio
2. Create a new Class Library project
3. Add a reference to the Echelon OpenLDV xDriver 1.0 Type Library
4. Add a COM class
5. Delete the default project class, **Class1.vb**
6. Import xDriver types to your **System** namespace
7. Implement the **ILdvxLookup** interface
8. Build and register the COM server
9. Create a custom xDriver profile
10. Test the lookup extension
11. Rebuild and re-register the COM Server
12. Retest the lookup extension
13. Optionally, add implemented **ILdvxLookup** interfaces to the sample component's type library
14. Optionally, add additional private methods or properties to the **ISampleLookupCsv** interface

The following sections describe these tasks in more detail.

Create a New Visual Studio Project

From the Visual Studio main window, select **File** → **New** → **Project** to open the New Project dialog, as shown in **Figure 28** on page 128.

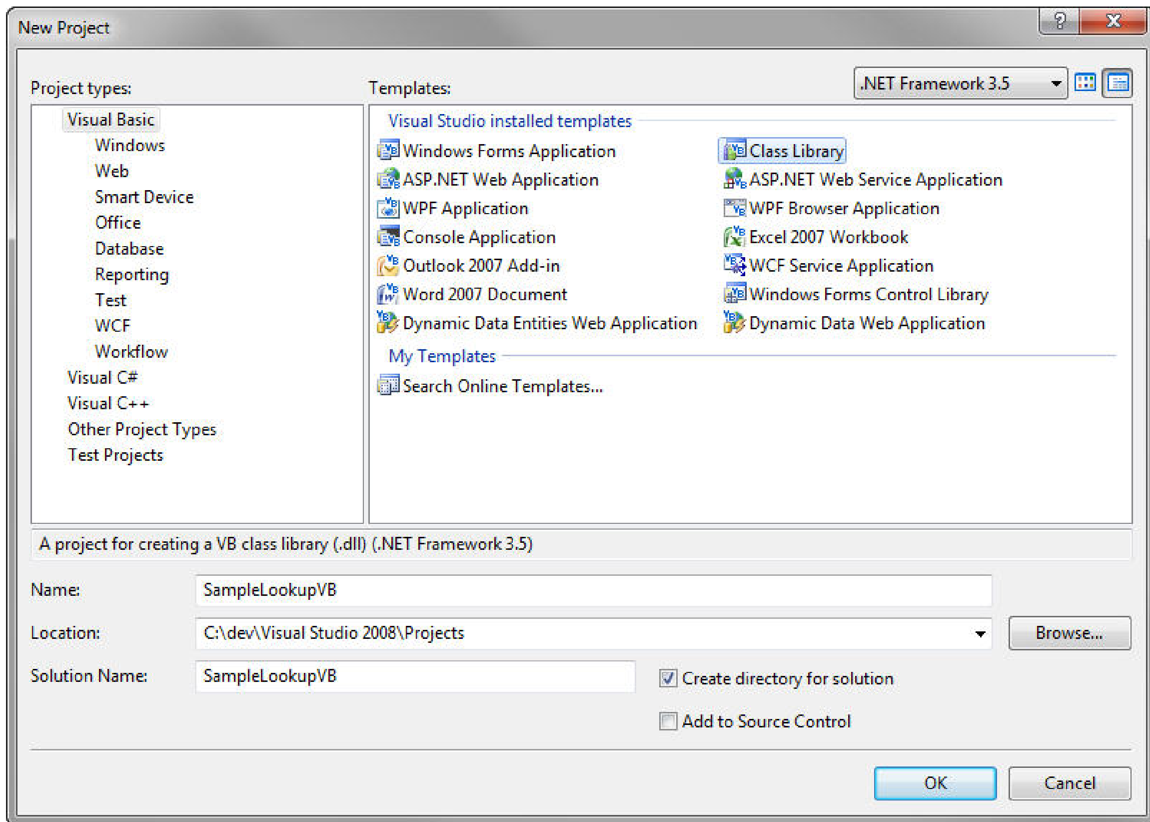


Figure 28. Visual Studio New Project Dialog

In the New Project dialog:

- Expand the Visual Basic category and select **Class Library** as the project template
- Specify the version of the .NET Framework to use
- Specify a name and location for the project
- Specify a name and location for the solution

You can use any project location for development. Install the completed custom lookup extension component in the LONWORKS **\xDriver\Components *CompanyName*\Lookup** folder when you distribute the DLL for your application.

The project name should not match the project name for any other lookup or COM component on your computer, or on any computer on which the application will be installed, because it will be used by the Profile Editor to identify the lookup extension component.

Recommendation: Specify the project name using the following naming convention:

[*Company Name*]**Lookup**[*Type*]

where [*Company Name*] represents the name of your company and [*Type*] represents the type of database this lookup extension component uses. For example: "MyCompanyLookupODBC".

Click **OK** to create the project.

Add a Reference to the xDriver Type Library

From the Visual Studio main window, select **Project** → **Add Reference** to open the Add Reference dialog. Select the COM tab, as shown in **Figure 29**.

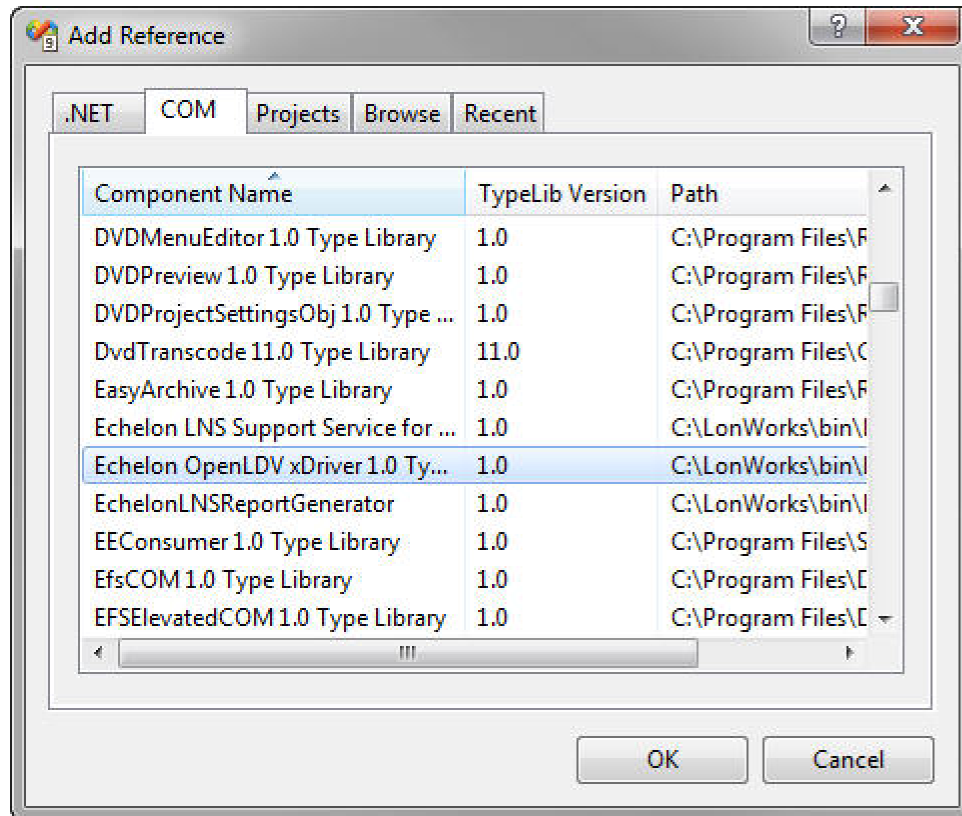


Figure 29. Visual Studio Add Reference Dialog

From the COM tab, select **Echelon OpenLDV xDriver 1.0 Type Library** and click **OK** to add the reference.

Add a COM Class

From the Visual Studio main window, select **Project** → **Add Class** to open the Add New Item dialog, as shown in **Figure 30** on page 130.

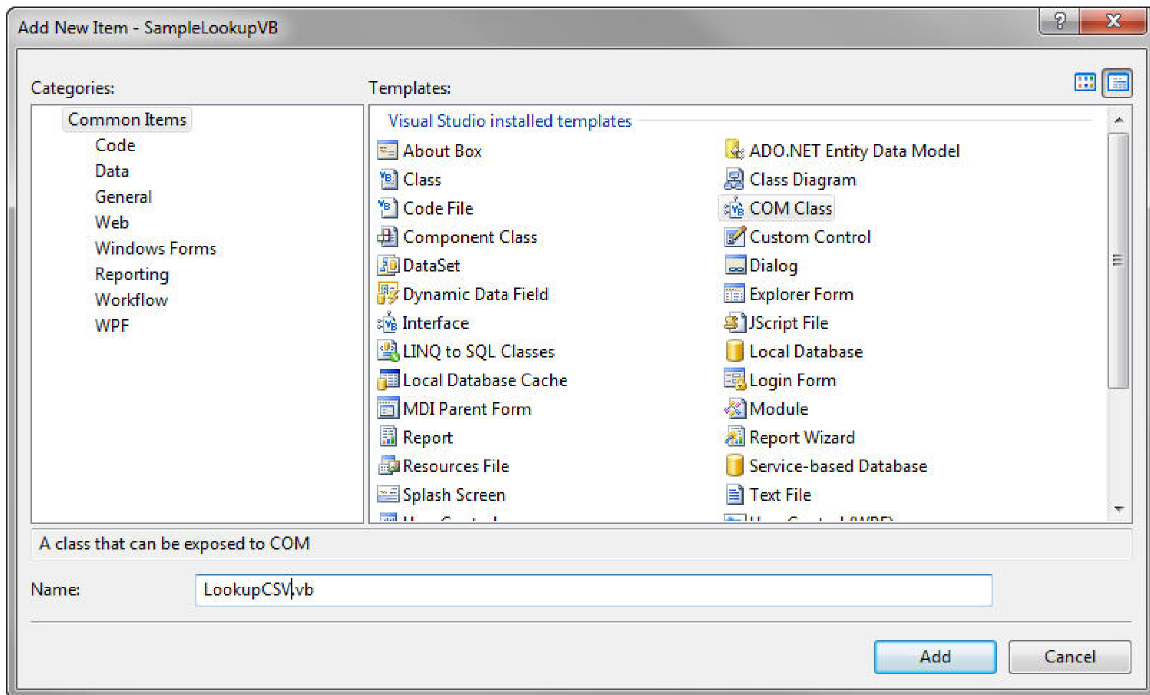


Figure 30. Visual Studio Add New Item Dialog

Select **COM Class** as the template. This template generates the proper GUIDs and the required **New Sub** method for your project. If you copy the sample files to use as a base for your project, you must change the GUIDs.

Recommendation: Use the name **Lookup**[*Database Type*] as the name of the class, where [*Database Type*] represents the type of database management system you are using. You can use this name with the xDriver Profile Editor to identify the lookup extension component.

Delete the Project Default Class

Use the Solution Explorer to delete the **Class1.vb** file, the class that was initially created with the project. Right-click **Class1.vb** and select **Delete**, as shown in **Figure 31** on page 131.

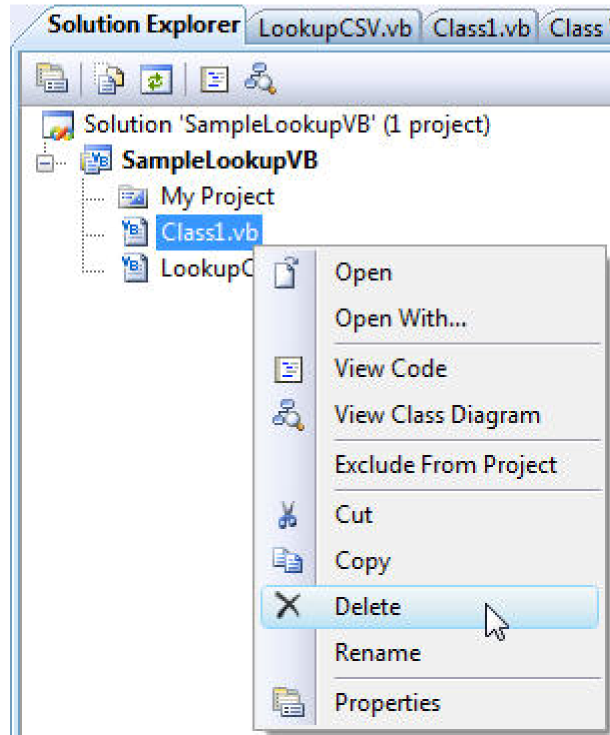


Figure 31. Delete Class1.vb

Import xDriver Types to Your System Namespace

Add the following lines to the beginning of your source code (**LookupCSV.vb**, in this example):

```
Imports LdvxLib
Imports LdvxLib.LdvxEncryption
Imports LdvxLib.LdvxResult
```

These statements add the properties, methods, and types of the Echelon OpenLDV xDriver 1.0 Type Library (see *Add a Reference to the xDriver Type Library* on page 129) to the **System** namespace of your project.

Implement the ILdvxLookup Interface

Add the following lines of code at the beginning of your class:

```
Implements ILdvxConfigure
Implements ILdvxLookup
```

These statements specify the interfaces that will be implemented in your lookup extension. The **ILdvxConfigure** interface is optional.

Select **View** → **Object Browser** to open the Object Browser. Expand **Interop.LdvxLib** to see the methods associated with each RNI, as shown in **Figure 32** on page 132.

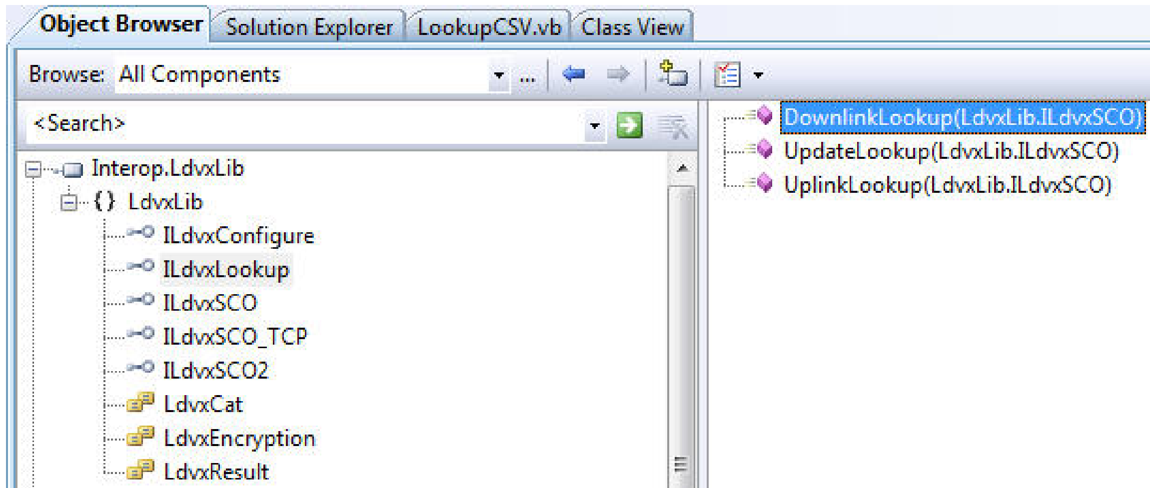


Figure 32. Visual Studio Object Browser View

Your class must implement the **DownlinkLookup**, **UpdateLookup**, and **UplinkLookup** members of the **ILdvxLookup** interface. You can optionally also implement the **SetIntance** and **SetOptions** members of the **ILdvxConfigure** interface.

You can now begin coding your lookup extension component. See Appendix C, *Custom Lookup Extension Component Programming*, on page 163, for more information about custom lookup extension component programming. This appendix contains descriptions of the methods that you can use to program your custom lookup extension component.

Build and Register the Lookup Extension

Select **Build** → **Build SampleLookupVB** to build the SampleLookupVB part of the solution.

Note: For Windows operating systems that include User Account Control (Windows Vista or Windows 7), if you do not run Visual Studio with elevated permissions (for example, as an Administrator), you will likely see the following error message during registration:

```
Project : error PRJ0050: Failed to register output.  Please
try enabling Per-user Redirection or register the component
from a command prompt with elevated permissions.
```

Start an elevated command shell (right-click the **Visual Studio 2008 Command Prompt** shortcut and select **Run as administrator**), change to the folder that contains your built DLL, and run the following command:

```
REGSVR32 SampleLookup.dll
```

Create a Custom xDriver Profile

Use the OpenLDV xDriver Profile Editor to create a custom xDriver profile for your custom xDriver lookup extension.

1. Select **Start** → **Echelon OpenLDV 4.0 SDK** → **Developer Utilities** → **xDriver Profile Editor** to open the OpenLDV xDriver Profile Editor.

2. Click **Add** to open the New Profile dialog, as shown in **Figure 33**.

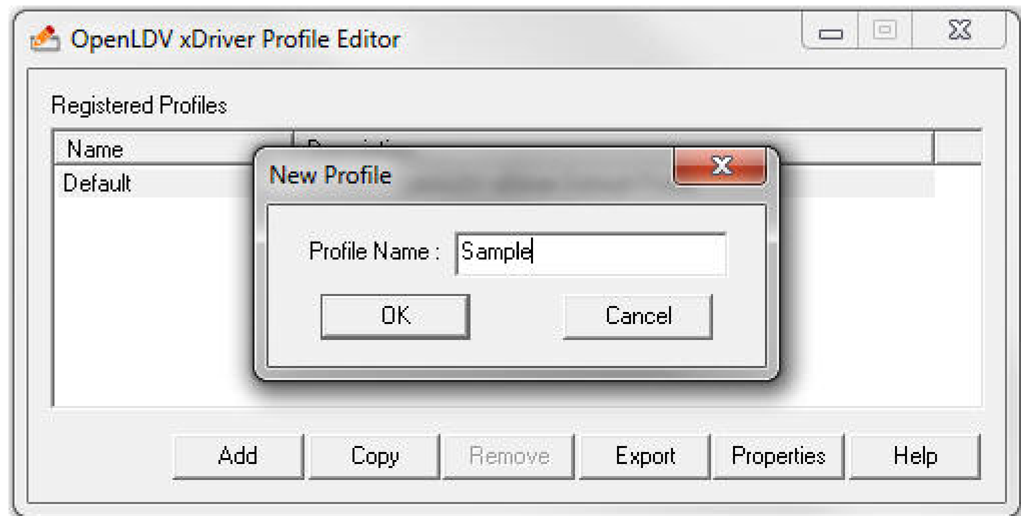


Figure 33. xDriver Profile Editor New Profile Dialog

3. Enter a profile name ("Sample" for this example) in the **Profile Name** field.
4. Click **OK** to close the New Profile dialog.
5. For the General tab of the OpenLDV xDriver Profile Properties dialog for the new profile, enter a description in the Profile Description text box, as shown in **Figure 34**.

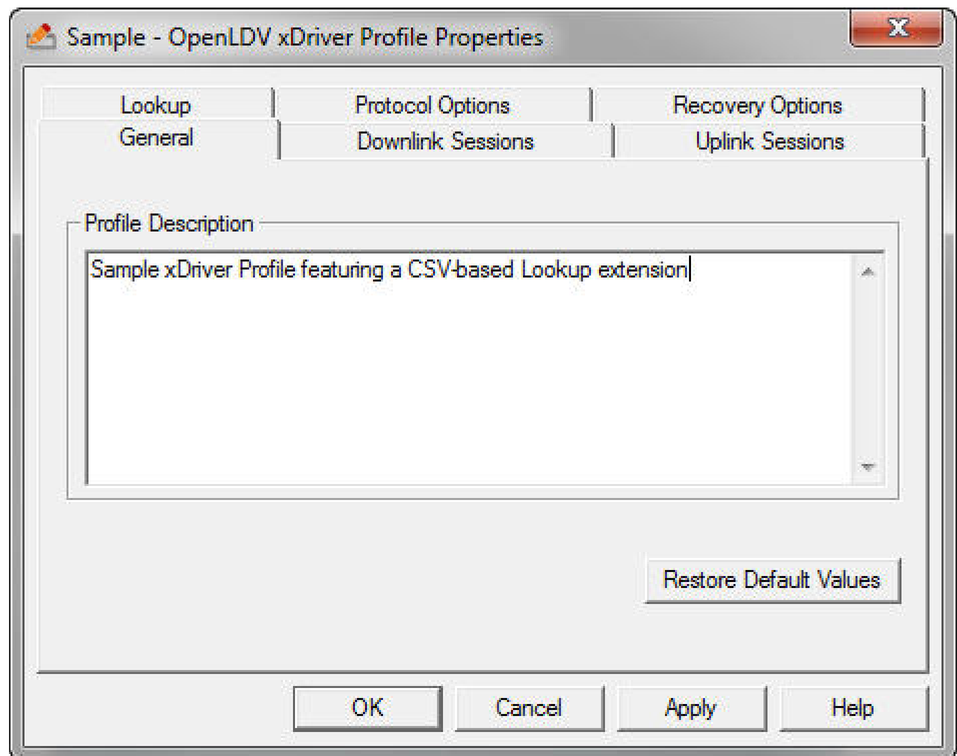


Figure 34. OpenLDV xDriver Profile Properties – General Tab

6. For the Lookup tab of the OpenLDV xDriver Profile Properties dialog for the new profile, select the newly built profile from the **Extension Name** dropdown listbox, as shown in **Figure 35**.

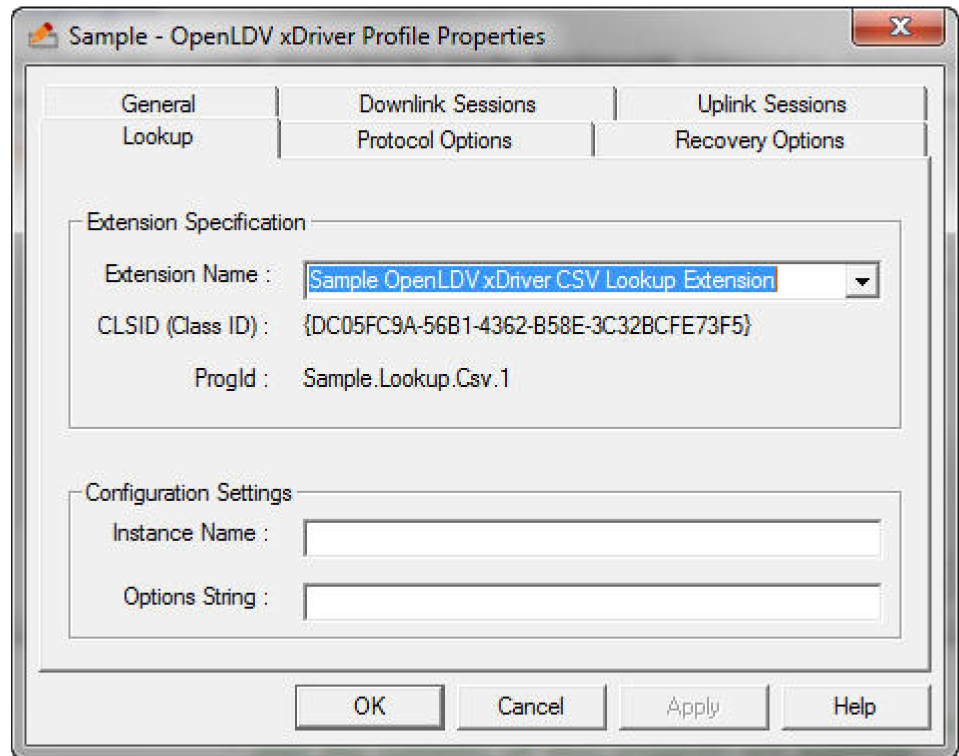


Figure 35. OpenLDV xDriver Profile Properties – Lookup Tab

7. Make any other changes that are appropriate for your custom profile, and click **OK** to accept the changes and close the dialog.

Test the Lookup Extension

Create an RNI for your custom xDriver lookup extension, and give it a name like "**X.Sample.***DownlinkKey*". Use an OpenLDV client program to open the RNI for the custom extension to see calls for the following methods:

- ILdvxConfigure::SetInstance
- ILdvxConfigure::SetOptions
- ILdvxLookup::DownlinkLookup

Note: Until these methods are implemented (see *Implement the ILdvxLookup Interface* on page 131), the default **E_NOTIMPL** return status will cause the startup sequence to fail.

Sample Lookup Extension Component

The **SampleLookupCsv.cpp** program is an xDriver lookup extension written in C++. The **SampleLookupVBNet.vb** program is an xDriver lookup extension Component written in Microsoft Visual Basic .NET. You can find these samples in the LonWorks Data Path folder:

- Windows Vista or Windows 7: **\Users\Public\Documents \LonWorks\OpenLDV SDK\xDriver API Examples**
- Windows XP: **\Documents and Settings\All Users\Documents \LonWorks\OpenLDV SDK\xDriver API Examples**

The example applications are also installed as a ZIP file in the LONWORKS **\OpenLDV SDK\SourceArchive** folder on your computer. The ZIP file is named **xDriverApiExamplesSource_v*n.nn.nnn*.ZIP** file (where *n.nn.nnn* represent the current version and build of the OpenLDV SDK).

The C++ sample program uses a comma-separated variable (CSV) file as its database to store the SCO information for the lookup extension. Because the database is a CSV file, the sample program uses standard file I/O methods for managing the database.

The Visual Basic sample program uses ADO.NET to connect to a database that stores the information that the lookup extension must fill into the SCO each time an xDriver connection is initiated. It retrieves this information from the database using standard ADO techniques, and uses the properties and methods of the xDriver type library to fill this information into the SCO.

ADO.NET can use either SQL or OLE connections to access a database. In general, you use only one connection type to access your database, but this sample is coded to implement either connection type on a per-session basis. Comments are included within the programming sample to provide further guidance.

If you are using an LNS Server, for a downlink session, the sample lookup extension component is launched by an LNS application after the **System** object for the remote network is opened. The **New()** function is called first. The **SetInstance** and the **SetOptions** functions are called next to initialize the SCO with any user-defined parameters. If parameters are not specified, predefined defaults are used. The **DownlinkLookup()** function is then called by the xDriver manager automatically, and the downlink lookup key is extracted from the passed-in SCO. The downlink lookup key is used to locate the database record for the RNI. The **UpdateSCO()** helper function, which extracts the required fields from the located database record and fills them into the SCO, is then called.

If you are using an LNS Server, for an uplink session, the sample lookup extension component is launched when the uplink session request is received by the LNS Server. The **New()** function is called first. The **SetInstance** and the **SetOptions** functions are called next to initialize the SCO with user-defined parameters. If parameters are not specified, predefined defaults are used. The **UplinkLookup()** function is then called by the xDriver automatically, and the uplink lookup key is extracted from the passed-in SCO. The uplink lookup key is used to locate the database record for the RNI that has requested the uplink session. The **UpdateSCO()** helper function, which extracts the required fields from the located database record and fills them into the SCO, is then called.

The **UpdateLookup** function is called after the SCO has been filled in during an xDriver session if the current and next authentication keys filled in differ, which indicates that a change to the authentication key used by the RNI is required. The **UpdateLookup** function must be coded to implement an update to the xDriver database to reflect this change. The xDriver database must always store the correct values of the current and next authentication keys for an RNI, and

these fields should only be updated from the **UpdateLookup** function. The database configuration interface that you create must update these fields in a guaranteed, safe manner. For more information about how the lookup extension component handles authentication key changes, see *Authentication Key Handling* on page 113.

After the SCO has been filled in and the **UpdateLookup()** method has been called (if necessary), the OpenLDV application that launched the lookup extension component can open the network, and perform whatever network operations are desired.

The destructor, **Finalize** function, is called last. It is used to close the database connection if it is still open. If an instance has not been defined, the destructor is called following **DownlinkLookup** function. If an instance has been defined, the destructor is called when the last object using the instance closes.

xDriver Profiles

After you finish programming your custom lookup extension component, you can create an xDriver profile to use it. An xDriver profile represents a set of configuration parameters that determines how xDriver manages sessions, including:

- The port that xDriver uses to listen for uplink connection requests
- The port that xDriver uses to initiate downlink connections
- The lookup extension to use to look up RNIs
- A flag to enable xDriver automatic reconnection

If you enable automatic reconnection, xDriver attempts to reconnect any uplink or downlink session that is broken as a result of some unexpected connection failure. With automatic reconnection enabled, you can configure xDriver to attempt reconnection as soon as it detects a failed session. xDriver attempts reconnection until the session has been successfully reestablished or until a time period that you define has expired.

The profile to be used for each session is determined on a session-by-session basis. For a downlink session, the profile to use is specified in the network interface name of the RNI. The network interface name for an RNI using xDriver must use the following naming convention:

X.[*Profile Name*].[*Downlink Lookup Key*]

where [*Profile Name*] represents the name of the profile to use for the session and [*Downlink Lookup Key*] represents the downlink lookup key that was assigned to the RNI when it was added to the xDriver database. For sample programs that use this naming convention, see *Downlink Sample Applications* on page 140.

For an uplink session, the request for connection arrives on a specific TCP port. The xDriver profile using that port as its listener port handles the uplink session. You can set the listener port that a profile should use with the OpenLDV xDriver Profile Editor.

There are two ways to install profiles when you distribute your application:

- Install the profiles with permanent Registry entries that will never be uninstalled or overwritten. This method has the disadvantage that re-

installing the software does not return the default values to these profiles. However, the xDriver Profile Editor allows you to restore any profile to its default value.

- Install your profiles with temporary Registry entries that are overwritten each time the software is installed. In this case, the default values would be restored each time the software is installed, and all changes that were configured by the user since the last installation would be lost.

After you create a profile to use your custom lookup extension component and (optionally) start the Connection Broker, you can begin using OpenLDV or LNS applications (such as the LonMaker Integration Tool) to access the networks that you will use with xDriver.

Alternatively, you can begin creating LNS applications that use xDriver. See Chapter 7, *LNS Programming with xDriver*, on page 139, for programming samples that can assist you in creating these new applications.

Starting the Connection Broker

The Connection Broker must be running for xDriver to receive uplink session requests. However, it is not required for downlink sessions.

The Connection Broker automatically stops if there are no profiles with uplink session handling enabled. In addition, the Connection Broker must be stopped and restarted each time uplink session handling for a profile is enabled or disabled.

The Connection Broker runs as an interactive service, which allows your applications that use the Connection Broker to display relevant dialogs for user interaction. However, any user dialogs are displayed in a separate Windows desktop from the one the user is logged into, so that a user might not reply to these dialogs. Thus, an uplink application should use an interface that does not require user interaction.

To start the Connection Broker, perform the following tasks:

1. Open the Services administrative control panel applet:
   Windows 7: Control Panel → System and Security → Administrative Tools → Services
   Windows XP: Control Panel → Administrative Tools → Services
2. Locate the entry for the **Echelon xDriver Connection Broker**, as shown in **Figure 36** on page 138.

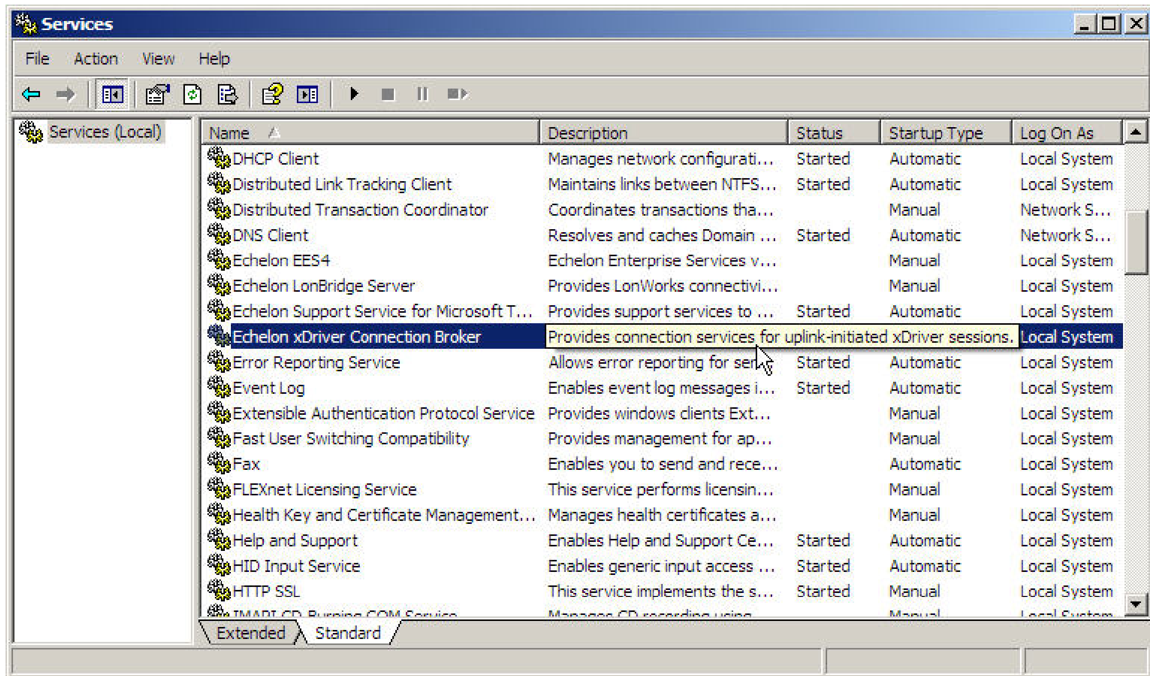


Figure 36. Services Administrative Control Panel Applet

3. To start the Connection Broker, right-click **Echelon xDriver Connection Broker** and select **Start**.

   Important: You must have administrator privileges to start the Connection Broker.

4. To enable the Connection Broker service permanently, right-click **Echelon xDriver Connection Broker** and select **Properties** to open the Echelon xDriver Connection Broker Properties dialog. From this dialog, select **Automatic** from the **Startup type** dropdown listbox. Then, click **Start.**

You can also start or stop the Connection Broker from a Windows Command Prompt:

```
NET START LdvxBroker
```

```
NET STOP LdvxBroker
```

7

LNS Programming with xDriver

This chapter describes sample programs to assist you when creating LNS applications to manage downlink and uplink xDriver sessions.

Because LNS Server includes support for xDriver, you can review these sample programs to understand that support before creating your own LNS applications that use xDriver.

Downlink Sample Applications

LNS applications that manage downlink sessions operate like any other type of network interface. However, the network interface name for an xDriver RNI must use the following naming convention:

X.[*Profile Name*].[*Downlink Lookup Key*]

where [*Profile Name*] represents the name of the xDriver profile to be used in this session and [*Downlink Lookup Key*] represents the downlink lookup key assigned to the RNI when it was added to the xDriver database. xDriver network interface names are not case-sensitive.

Example: If the downlink lookup key assigned to an RNI is "RNI-0001" and this RNI uses the default profile, the network interface name to use for that device would be "X.Default.RNI-0001".

All physical local interfaces appear in the **NetworkInterfaces** collection. However, xDriver RNIs that use custom lookup extension components do not appear in the **NetworkInterfaces** collection until a session with that RNI has been fully established. On the other hand, because the default xDriver lookup extension uses the same part of Windows Registry that describes local network interfaces, all configured RNIs assigned to the default profile appear in the LNS **NetworkInterfaces** collection.

Even if a custom lookup extension is used, and the network interface does not appear in the network interfaces collection, it is possible to open the appropriate **NetworkInterface** object by name from the LNS Network Interfaces collection using the **NetworkInterfaces.Item(*NetworkInterfaces As String*)** method. For example:

```
NetworkInterfaces.Item("X.Default.RNI-0001")
```

See the second downlink programming sample on page 141.

Opening a Single Remote Network With xDriver

The following Visual Basic programming sample is an LNS application that creates, opens, and closes a single remote network with xDriver.

At the completion of the function, the Object Server contains a new network named "Network1" whose database is stored in "c:\Network1", and a new subsystem named "Subsystem1." The network interface name used to access the new network is "X.Default.RNI-0001."

This sample demonstrates the use of the xDriver network interface naming convention. Relevant comments are shown in bold.

```
Dim LcaOs As LcaObjectServer
Dim ActiveNetwork As LcaNetwork
Dim ActiveSystem As LcaSystem
Dim ActiveSubsystem As LcaSubsystem
Dim TempNetworkInterface As LcaNetworkInterface

Private Sub InitializeObjectServer()
    'Initialize the global LNS database.
    Set lcaOs.RemoteFlag = False
    Set lcaOs.SingleUserMode = False
```

```
    lcaOS.Open

    'Add a new network object and open the
    'network database. "Network1" represents the name
    'of the network and "c:\Network1" represents the path to the
    'network database.
    Set ActiveNetwork = lcaOS.Networks.Add _
              ("Network1", "c:\Network1", True)
    ActiveNetwork.Open

    'Fetch the system from the network.
    Set ActiveSystem = ActiveNetwork.Systems.Item(1)

    'Select a network interface. Note the use of the xDriver naming
    'convention in this line: X.[Profile Name].[Downlink Lookup Key]. Default
    'represents the profile name to use. RNI-0001 represents the downlink
    'lookup key for the RNI to be opened.
    Set TempNetworkInterface _
    = lcaOS.NetworkInterfaces.Item("X.Default.RNI-0001")
    Set ActiveSystem.NetworkServiceDevice.NetworkInterface _
              = TempNetworkInterface

    'Set up the LNS Server and open an xDriver downlink session by opening
    'the System object.
    ActiveSystem.Open

    'Set the system into OnNet management mode.
    ActiveSystem.MgmtMode = lcaOnNet

    'Create a subsystem object to hold your AppDevice objects.
    Set ActiveSubsystem = ActiveSystem.Subsystems.Add("Subsystem1")
    'Any other code goes here.
    ActiveSystem.Close
    ActiveNetwork.Close
    LcaOS.Close
End Sub
```

Opening Multiple Remote Networks for Downlink

The following Visual Basic sample program opens multiple remote networks simultaneously. It uses xDriver to connect to two remote LONWORKS networks.

The information required to open each network (network name, xDriver profile name, and the RNI lookup key) for this program is hard coded into the application. **Figure 37** on page 142 shows the form that was created for this application.

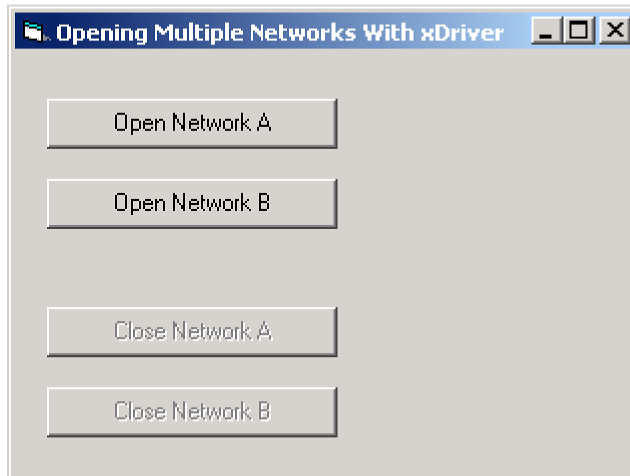


Figure 37. Downlink Application Form

This application can open either of the two remote LONWORKS networks by clicking the **Open Network A** and **Open Network B** buttons. After either network has been opened, the applicable **Close Network** button becomes enabled, so that a user can close the network. When both networks are opened at the same time, each network is assigned its own xDriver network interface.

From this example, you should be able create your own application that uses xDriver to open multiple remote networks simultaneously. For more information on any of the LNS commands used in this sample program, see the *LNS Programmer's Guide.*

```
'Create variables to store the System and Network object for each network to be
'opened with this application. g_NetworkA and g_SystemA serve as the Network
'object and System object variables for Network A. g_NetworkB and g_SystemB
'serve as the Network object and System object variable for Network B.

Dim g_NetworkA As LcaNetwork
Dim g_NetworkB As LcaNetwork
Dim g_SystemA As LcaSystem
Dim g_SystemB As LcaSystem

'Private Sub OpenNetA_Click() is called when the user clicks the Open Network A
'button. This function disables the Open Network A button, so that a user
'cannot try to open Network A while it is already open. It also enables the
'Close Network A button so that the user can close the open network. Following
'this, the function opens the System and Network object for Network A.

Private Sub OpenNetA_Click()
   OpenNetA.Enabled = False       'Disable the Open Network A button.
   CloseNetA.Enabled = True       'Enable the Close Network A button.
   Dim netList As LcaNetworks
   Set netList = g_cOS.Networks   'Set netList as the Network objects
                                  'collection.
   Set g_NetworkA = netList(1)    'Set g_NetworkA as the Network object for the
                                  'first member of the Network objects
                                  'collection. This value must be hard-coded and
                                  'will vary depending on which network you want
                                  'opened.
   g_NetworkA.Open          'Open network A.
   Dim NIs As LcaNetworkInterfaces       'Set NIs as the NetworkInterfaces
   Set NIs = g_cOS.NetworkInterfaces     'collection object.
   Dim curRNI As LcaNetworkInterface
```

```
   Set curRNI = NIs.Item("X.Default.RNI-0001")
                              'Set the variable curRNI as the
                              'network interface you want to open. Note the
                              'use of the xDriver naming convention here:
                              'X.[Profile Name].[Downlink Lookup Key].
   Dim SysList As LcaSystems  'Set Syslist as the System objects collection.
   Set SysList = g_NetworkA.Systems     'Set g_SystemA as the System object for
   Set g_SystemA = SysList(1)           'the network to be opened.
   Dim SysNSD As LcaNetworkServiceDevice
   Set SysNSD = g_SystemA.NetworkServiceDevice
   Set SysNSD.NetworkInterface = curRNI
   g_SystemA.Open
End Sub

'Private Sub CloseNetA_Click() is called when the user clicks the Close Network
'A button. In order for this to happen, the Network object and the System
'object for Network A must be closed. This function also re-enables the Open
'Network A button, so that Network A can be re-opened, and disables the Close
'Network A button, so that the application will not attempt to close a network
'that is not already closed.

Private Sub CloseNetA_Click()
   OpenNetA.Enabled = True    'Re-enable the Open Network A button.
   CloseNetA.Enabled = False  'Disable the Close Network A button.
   g_SystemA.Close            'Close the System object for network A.
   g_NetworkA.Close           'Close the Network object for network A.
End Sub

'Private Sub OpenNetB_Click() is called when the user clicks the Open Network B
'button. This function first disables the Open Network B button, so that a
'user cannot try to open Network B while it is already open. It also enables
'the Close Network B button, so that the user can close the open network.
'Following this, the function opens the System and Network object for Network
'B. Please see the OpenNetA_Click()function for comments that describe each
'line of code in this function.

Private Sub OpenNetB_Click()
   OpenNetB.Enabled = False
   CloseNetB.Enabled = True
   Dim netList As LcaNetworks
   Set netList = g_cOS.Networks
   Set g_NetworkB = netList(2)
   g_NetworkB.Open
   Dim NIs As LcaNetworkInterfaces
   Set NIs = g_cOS.NetworkInterfaces
   Dim curRNI As LcaNetworkInterface
   Set curRNI = NIs.Item("X.Default.RNI-0002")
   Dim SysList As LcaSystems
   Set SysList = g_NetworkB.Systems
   Set g_SystemB = SysList(1)
   Dim SysNSD As LcaNetworkServiceDevice
   Set SysNSD = g_SystemB.NetworkServiceDevice
   Set SysNSD.NetworkInterface = curRNI
   g_SystemB.Open
End Sub

'Private Sub CloseNetB_Click() is called when the user clicks the Close
'Network B button. In order for this to happen, the Network object and the
'System object for Network B must be closed. This function also re-enables
'the Open Network B button, so that Network B can be re-opened, and disables
'the Close Network B button, so that the application will not attempt to close
'a network that is already closed. Please see the CloseNetA_Click()
'function for comments that describe each line of code in this section.
```

```
Private Sub CloseNetB_Click()
   OpenNetB.Enabled = True
   CloseNetB.Enabled = False
   g_SystemB.Close
   g_NetworkB.Close
End Sub

'Private Sub Form_Load() is called when the form Load event occurs.
Private Sub Form_Load()
   g_cOS.RemoteFlag = False       'Set the application access mode to local.
   g_cOS.SingleUserMode = False   'Allow multiple applications to access LNS
                                  'server.
   g_cOS.Open                     'Open the Object Server.
   CloseNetA.Enabled = False      'Disable the Close Network buttons, so that a
   CloseNetB.Enabled = False      'cannot try to close a network before it has
                                  'been opened.
End Sub

Private Sub Form_Unload(Cancel As Integer)
   G_cOS.Close                    'Close the object server.
End Sub
```

Uplink Sample Application

The following Visual Basic sample program is a listener application that manages uplink sessions between the LNS Server and multiple RNIs. This application first registers for uplink session event handling, and uses a timer control to check when uplink session requests are received. When the timer control discovers that a request for uplink session has been received, the application opens the network that has requested the session. Thus, the application can run without user interaction, other than starting and stopping the application.

This sample program uses several events and methods within the LNS Server for use with xDriver. For additional information, see Appendix B, *LNS Methods and Events for xDriver*, on page 155.

In the **Form_Initialize()** function, the application registers for uplink session event handling by calling the **BeginIncomingSessionEvent** function. The **OnIncomingSessionEvent** event is fired each time an uplink session event is received. This event handler is called **m_cOS_OnIncomingSessionEvent** in this application, because the instance of the LNS Server for this application is named **m_cOS**.

The uplink session is accepted or rejected by calling the **AcceptIncomingSession** method from the **m_cOS_OnIncomingSessionEvent** event handler. If the session is accepted, the name of the network that requested the uplink session is stored in a global variable. This variable is used to open the network. This application also sets the **DoPostponeUpdates** flag to **True** when it calls the **AcceptIncomingSession** method. All monitor-point updates are then withheld until the **ReleasePendingUpdates** method is called. Your application can receive the monitor-point update event that caused the uplink session request.

The network should not be opened from the event handler, so the **OnIncomingSession** event sets a flag to **True** to indicate that an uplink session request has been received. This flag causes the **StartButton_Click()** function to

be called from the **Timer1_Timer()** function after the next timer control interval expires. The **StartButton_Click()** function opens the network that requested connection in independent mode, and enables monitoring of the network variable monitor set for that network. It also calls the **ReleasePendingUpdates** method to release the monitor-point update events that have been withheld since the session began.

The application then receives the monitor-point update event that caused the uplink session request (for example, an alarm). Upon receiving a monitor-point update event, the **m_cOS_OnNvMonitorPointUpdateEvent()** function is called automatically. This function saves all information associated with the monitor-point update event publicly, and sets a flag to **True** to indicate that a monitor-point update has been received. This flag causes the **DisplayMP_Click()** function to be called from the **Timer1_Timer()** function after the next timer control interval expires, eliminating the need for any calls from within the event handler. The **DisplayMP_Click()** function displays all information saved for the monitor-point update event. It also sets a flag to **True** to end the current uplink session.

When this flag is **True**, the **StopButton_Click()** function is called from the **Timer1_Timer()** function after the next timer control interval expires. The **StopButton_Click()** function disables the monitor point and monitor set for the open network, and the network is closed. The application continues to listen for requests for connection, and handles them in this fashion, until the application is closed.

Important: A network interface could reset after receiving an alarm event, but before the event has been propagated to the LNS Server, causing the event to be lost. To prevent this loss, your applications must resend each monitor-point update at short intervals until receipt of that event is confirmed, especially for alarm applications. This technique results in reliable performance, and ensures that no monitor point update events are lost before they are processed by the LNS application.

For more information about any of the LNS commands used in this programming sample, see the *LNS Programmer's Guide*.

```
Dim m_cCurNet As LcaNetwork
Dim m_cMyVni As LcaAppDevice
Dim m_cMS As LcaMonitorSet
Dim m_gDP As LcaDataPoint
Dim m_gbInMP As Boolean                'Use the monitor point update event
                                       'flag to track whether a monitor point
                                       'update event has been received.
Dim m_gbInUplinkOpen As Boolean        'Use the uplink open flag to track
                                       'whether an uplink session is currently
                                       'being handled by the application.
Dim m_gbInUplinkClose As Boolean       'Use the uplink close flag to track
                                       'whether the most recent uplink session
                                       'handled by the application has been
                                       'closed.
Dim m_szIncomingNetName As String      'This variable will be used to store
                                       'the network name of a network
                                       'requesting connection to the LNS
                                       'server.

Private Sub Form_Initialize()
     ExitButton.Enabled = True         'Enable the Exit button.
     m_cOS.RemoteFlag = False          'Set the application access mode to
```

```
                                        'local.
     m_cOS.SingleUserMode = False       'Allow multiple applications to access
                                        'the LNS Server.
     m_cOS.Open                         'Open the LNS Server.
     m_cOS.BeginIncomingSessionEvents ("Default")     'Register the application
                                        'for uplink session event handling.
                                        'After this, each time the listener
                                        'port assigned to default profile
                                        'receives an uplink session request,
                                        'the m_cOS_OnIncomingSessionEvent event
                                        'will be fired.
     m_gbInMP = False                   'Set the monitor point update event
                                        'flag to False until a monitor point
                                        'update event is received.
     m_gbInUplinkOpen = False           'Set the uplink open and uplink close
     m_gbInUplinkClose = False          'flags to False until an uplink
                                        'session begins.

     Timer1.Interval = 100              'Set the interval for invocation of the
                                        'timer control function to 100
                                        'milliseconds, or an interval of your
                                        'choice.
End Sub

'The m_cOS_OnIncomingSessionEvent function is invoked each time an uplink
'session request is received, as long as the application has registered for
'uplink session handling by invoking the BeginIncomingSessionEvents method.
'This function stores the name of the calling network in the variable
'm_szIncomingNetName and sets the uplink 'open flag to True.

Private Sub m_cOS_OnIncomingSessionEvent(ByVal XDriverProfileName As String, _
 ByVal NetName As String, ByVal IntfName As String, ByVal Tag As Long)
    If m_gbInUplinkOpen = False & XDriverProfileName = "Default" Then
        Timer1.Enabled = False    'Disable the timer while processing the
                                  'uplink. Start of critical section. The
                                  'network cannot be opened from the
                                  'event handler.
m_cOS.AcceptIncomingSession Tag, True, True    'Accept the uplink. The
                                  'DoPostponeUpdates flag is set to True, which
                                  'means that all monitor point updates for the
                                  'network will be withheld until the
                                  'ReleasePendingUpdates method is called.
m_szIncomingNetName = NetName     'Store the name of the network that has
                                  'requested connection in the variable
                                  'm_szIncomingNetName. This will be used to
                                  'open the network later.
m_gbInUplinkOpen = True           'Set the uplink open flag to True. This will
                                  'cause the Timer1_Timer() function to invoke
                                  'the StartButton_Click() function after the
                                  'next timer interval expires.
        Timer1.Enabled = True     'Re-enable the timer. End of critical section.
    Else
        m_cOS.AcceptIncomingSession Tag, False, False 'Reject the uplink session
                                  'if there is another uplink session open.
    End If
End Sub

'The m_cOS_OnNvMonitorPointUpdateEvent function is called when a monitor
'point update for an open network has been received. This function saves all
'information associated with the update so that it can be displayed by the
'DisplayMP_Click() function.

Private Sub m_cOS_OnNvMonitorPointUpdateEvent(ByVal MonitorPoint As Object, _
```

```
ByVal DataPoint As Object, ByVal srcaddr As Object)
    Dim src_addr As LcaSourceAddress
    Set src_addr = srcaddr       'Store the calling network address source in
                                 'the variable src_addr.
     X1 = src_addr.NodeId        'Store the node ID reporting the event in the
                                 'variable X1.
     X2 = src_addr.SubnetId      'Store the subnet ID in the variable X2.
     Set m_gMP = MonitorPoint    'Store the monitor point for the event in
                                 'variable m_gMP.
     Set m_gDP = DataPoint       'Store the data point for the event in
                                 'variable m_gDP.
     TextStatus = TextStatus + vbCrLf + "OnNvMonitorPointUpdateEvent"     'The
                                 'monitor point update event has been received
                                 'and all data has been saved.
     m_gbInMP = True             'Set the monitor point update event flag to
                                 'True. This will cause the Timer1_Timer()
                                 'function to invoke the DisplayMP_Click()
                                 'function after the next timer interval,
                                 'which will display the information saved for
                                 'this monitor point update event.
End Sub


'The StartButton_Click() function is invoked by the Timer1_Timer() function
'when the uplink session open flag is true (this flag is set True by the
'm_cOS_OnIncomingSession function whenever an uplink session is started). 'This
function opens the network in independent mode, and enables the 'monitoring of
the network variable monitor set for the network.

Private Sub StartButton_Click()
    On Error GoTo do_err1
    Dim m_cNets As LcaNetworks
    Set m_cNets = m_cOS.VNINetworks
    Set m_cCurNet = m_cNets.Item(m_szIncomingNetName)  'Use m_cCurNet as the
                                         'Network Object.
    m_cCurNet.OpenIndependent            'Open the network in independent mode.
    Set m_cMyVni = m_cCurNet.MyVNI
    Dim m_cMSs As LcaMonitorSets
    Set m_cMSs = m_cMyVni.MonitorSets
    Set m_cMS = m_cMSs.Item(1)    'Enable the monitor set for the network.
    m_cMS.Open True, True
    m_cMyVni.ReleasePendingUpdates()    'Release all pending monitor point
                                 'update events that occurred after the
                                 'session began and before the monitor
                                 'set was opened.
    ExitButton.Enabled = False    'Disable the Exit button while the network and
                                 'monitor set are open.
    GoTo do_done
do_err1:
    Debug.Print Err.Description
do_done:
End Sub

'The DisplayMP_Click() function is invoked by the Timer1_Timer() function 'when
the monitor point update event flag is true (this flag is set True by the
'm_cOS_OnNvMonitorPointUpdateEvent function whenever a monitor point update
'event is received). It displays some of the information saved for the monitor
'point update received in the text box of the form for this project.
Private Sub DisplayMP_Click()
    TextStatus = TextStatus + vbCrLf + ":: '" + m_gMP.Name _
             + "'" + vbCrLf + ":: tag := " + CStr(m_gMP.Tag) + vbCrLf _
             + ":: " + m_gDP.FormattedValue   'The application used all of the
                                 'data saved from the monitor point update
```

```
                                 'event and will now close the monitor points
                                 'and the network.
    m_gbInUplinkClose = True     'Set the uplink close flag to True.
End Sub

'Private Sub StopButton_Click() closes all monitor points, monitor sets, and
'networks. It also re-enables the Exit button so that the user can close the
'application when he wants.

Private Sub StopButton_Click()
    m_cMS.Close                  'Close monitor set.
    m_cCurNet.CloseIndependent   'Close the open network.
    ExitButton.Enabled = True    'Re-enable the Exit button since the network
                                 'is closed.
End Sub

'The ExitButton_Click() function closes the application. Before doing so, it
'invokes the EndIncomingSessionEvents method so that it will stop receiving
'incoming session events, and closes the object server.
Private Sub ExitButton_Click()
     m_cOS.EndIncomingSessionEvents ("Default")      'Stop receiving incoming
                          'session events.
     m_cOS.Close          'Close the Object Server.
     End                  'End the program.
End Sub

'The Timer1_Timer() function is invoked automatically by the function each
'time the timer control interval expires. This function uses flags to
'determine if an uplink session is open, or if a monitor point update event has
'been received, and acts accordingly.

Private Sub Timer1_Timer()
    If m_gbInMP Then             'If monitor point update event flag is True:
        m_gbInMP = False         'First, set the flag to False.
        DisplayMP_Click          'Then, invoke DisplayMP_Click(). This
    End If                       'displays all data for the update event.

    If m_gbInUplinkOpen Then     'If the uplink open flag is True:
        m_gbInUplinkOpen = False 'Set the flag back to False.
        StartButton_Click        'Then, invoke StartButton_Click()
    End If                       'to process the request for uplink.

    If m_gbInUplinkClose Then    'If the uplink close flag is True:
        m_gbInUplinkClose = False'Set the flag back to False.
        StopButton_Click         'Then, invoke StopButton_Click() to
    End If                       'close the open network.
End Sub
```

A

Custom Network Interfaces

This appendix provides high-level guidance for working with a custom network interface that can work with the OpenLDV driver.

Overview

Echelon and other manufacturers provide a wide selection of network interfaces for different LONWORKS channel types and for various computer requirements. You can also create a custom OpenLDV compatible network interface.

Figure 38 shows a simplified view of how a custom network interface communicates with an application:

- When a user uses a custom network interface for the first time or installs custom network interface software, Windows installs the device driver for the network interface and updates the Windows registry. In addition, a custom network interface might use additional software for device configuration (for example, you can use the LONWORKS Interfaces application in the Windows Control Panel to configure Echelon devices).
- The application uses the OpenLDV API to communicate with both the network interface and the LONWORKS network. For example, the application calls the **ldv_open()** function to open the custom network interface for communications.
- The OpenLDV API uses the entries in the Windows registry to map the custom network interface's logical name (what is presented to the user or the application) to the physical device name (what is presented to the device driver). This mapping is done whenever the custom network interface is opened for communications.

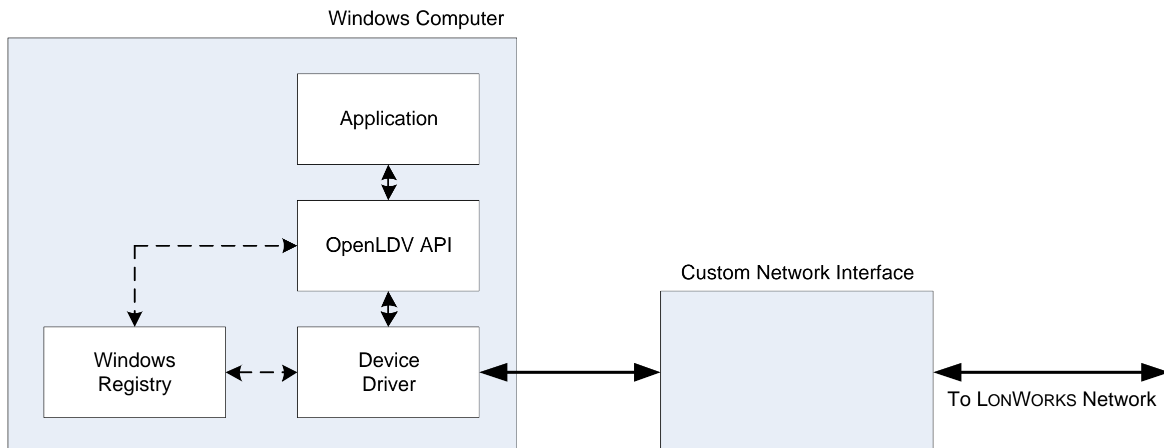


Figure 38. A Custom Network Interface Communicates with an Application

This appendix describes how you can integrate a custom network interface driver with the OpenLDV driver so that it is accessible from any OpenLDV application. It does not describe how to develop the custom network interface hardware or how to write the Windows device driver for the custom network interface.

Working with a Custom Network Interface

To make a custom Windows device driver accessible through the OpenLDV interface, the driver must be a kernel-mode Windows Driver Foundation (or Windows Driver Model) driver that provides standard create, read, write, and

close calls. The OpenLDV driver calls these driver functions to interact with the custom network interface.

You can create an OpenLDV application that manages the custom network interfaces supported by your custom Windows driver, similar to the Echelon LONWORKS Interfaces application. The OpenLDV application should perform the following basic tasks:

1. Create an **LDVDriverInfo** object:
    a. Set the `size` equal to the struct size.
    b. Set the `id` to any unused value > 127. Values less than 127 are reserved for Echelon use.
    c. Set the `type` to LDV_DRIVER_TYPE_LNI. This value specifies a Windows device driver.
    d. Set the `name` to a suitable name for your driver.
    e. Set the `desc` to a suitable description for your driver.

    ```
    LDVDriverInfo myDriver =
    { (DWORD)sizeof(LDVDriverInfo),
      (LDVDriverID)myDriverID,
      (LDVDriverType)LDV_DRIVER_TYPE_LNI,
      (LPCSTR)myDriverName,
      (LPCSTR)myDriverDesc
    };
    ```
2. Call the **ldv_set_driver_info()** function:

    ```
    LDVCode rc = ldv_set_driver_info(
                    myDriver.id,
                    *myDriver);
    ```
3. Create devices that use this driver.
4. Create an **LDVDeviceInfo** object:
    a. Set the `size` equal to the struct size.
    b. Set the `driver` to **NULL**. This parameter is ignored for the **ldv_set_device_info()** function.
    c. Set the `name` to a suitable name for your device. The name must be unique for the computer. The name must not begin with “X.” (that naming convention is reserved for xDriver devices). The name can follow the “LON1” naming convention, but you must ensure that no naming conflicts arise.
    d. Set `physName` to a suitable physical name (matching the name specified in your Windows driver) for the device. This name must follow the Windows `\\.\`*`name`*`.0` format. You can match the `physName` parameter with the `name` parameter.
    e. Set the `desc` to a suitable description for your device.
    f. Set the `caps` to suitable capabilities for your device. For example, your custom network interface might operate as a Layer 5 device and use the SICB data format, so you specify a logical

OR of the **LDV_DEVCAP_L5** and **LDV_DEVCAP_SICB** enumeration values.

g. Set the `capsMask` to suitable current capabilities of the device. When creating the device, you generally set this parameter to the same values as the `caps` parameter.

h. Set the `transId` to the transceiver ID of the custom network interface, as appropriate.

i. Set the `driverId` to the driver specified in the **ldv_set_driver_info()** function in step 2 on page 151.

```
LDVDeviceInfo myDevice =
{ (DWORD)sizeof(LDVDeviceInfo),
  (LDVDriverInfo)NULL,
  (LPCSTR)myDeviceName,
  (LPCSTR)myDevicePhysName,
  (LDVDeviceCaps)LDV_DEVCAP_L5 | LDV_DEVCAP_SICB,
  (LDVDeviceCaps)LDV_DEVCAP_L5 | LDV_DEVCAP_SICB,
  (BYTE)myTransID,
  (LDVDriverID)myDriver
};
```

5. Call the **ldv_set_device_info()** function:

```
LDVCode rc = ldv_set_device_info(
                  myDevice.name,
                  *myDevice);
```

After completing these steps, OpenLDV applications can use the other OpenLDV API functions to communicate with the custom network interface.

Windows Registry Entries

The installation program for the custom network interface must create a subkey for the device driver within the **\HKEY_LOCAL_MACHINE \SOFTWARE\LonWorks\DeviceDrivers** Windows registry path. The name of this subkey must be the logical name for the custom network interface; this logical name is what is displayed to the user (for example, within the LONWORKS Interfaces Control Panel application) or to the application. You can use any name that identifies the custom network interface, or you can follow the legacy "LON1" naming convention.

Within the custom network interface's subkey, you must create the following registry entry:

- Value name — **device name**
- Data type — **REG_SZ** (a string value)
- Value — The custom network interface's physical device name, as set by Windows for the device driver, with the format `\\.\`*name*`.0`

You must ensure that the value for the device name matches the logical name that you assign for the device.

Example: If the installed device driver for the custom network interface is assigned the physical device name `\\.\MyCustomLON1.0`, then the logical name

for the custom network interface should be “MyCustomLON1”. The subkey for the device within the registry should also have the name “MyCustomLON1”. **Figure 39** and **Figure 40** show these example registry entries.

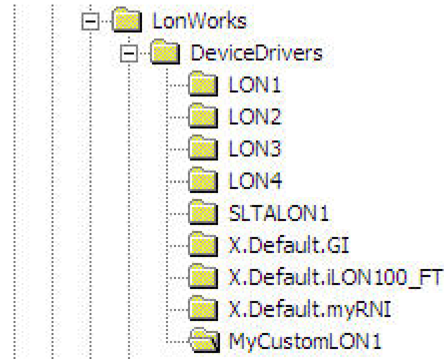


Figure 39. The MyCustomLON1 Registry Key

Figure 40. The device name Registry Value

B

LNS Methods and Events for xDriver Support

This appendix describes the methods and events that are included with LNS Server for use with xDriver. You use these methods and events when creating LNS applications to initiate and manage xDriver sessions.

xDriver Methods and Events

This appendix describes the LNS methods and events you use when creating an application to manage xDriver sessions. To use these methods and events, you must install the LNS Application Developer's Kit. Within a COM application, add a reference to the LNS Object Server. See Chapter 4, "Programming an LNS Application" in the *LNS® Programmer's Guide* or the LNS Application Developer's Kit help for more information.

For a sample program that uses these methods and events, see *Uplink Sample Application* on page 144.

AcceptIncomingSession

Applies to: Object Server object

Call this method to accept or reject an incoming uplink session request.

Syntax

objServer.**AcceptIncomingSession** *tag*, *acceptUplink*, *postponeUpdates*

Table 37. AcceptIncomingSession Parameters

Element	Description
tag	The *Tag* element is passed to the **OnIncomingSessionEvent** event when the uplink session is received. It should then be used by the **AcceptIncomingSession** method to identify the xDriver session when it is accepted or rejected.
acceptUplink	A **True** or **False** value. Use **True** to accept the session, or **False** to reject it.
postponeUpdates	A **True** or **False** value. Use **True** to hold off all monitor-point update events while the uplink session is being opened. In this case, withheld monitor-point updates must be released by calling the **ReleasePendingUpdates** method. For more information, see *ReleasePendingUpdates* on page 160. If **False**, all monitor-point update events that occur while the session is being initialized are lost. This feature is only supported by LNS listener applications. It is not supported by command-line initiated uplink event handlers.

Remarks

Use the **AcceptIncomingSession** method to accept or reject an uplink session request after the **OnIncomingSessionEvent** event has fired.

If an uplink session request is rejected, the session is terminated. If the uplink session request is neither accepted nor rejected before the session establishment time-out period for the profile handling the session expires, it is rejected

automatically. The session establishment time for a profile can be configured using the xDriver Profile Editor. For more information, see *xDriver Profiles* on page 136.

BeginIncomingSessionEvents

Applies to: Object Server object

This method is used to register for incoming session event handling. The application is then notified of incoming uplink session requests to the LNS Server.

Syntax

objServer.**BeginIncomingSessionEvents** *xDriverProfileName*

Table 38. BeginIncomingSessionEvents Parameters

Element	Description
xDriverProfileName	xDriver profile name as a String (20 characters max). The application is informed of incoming session requests that come in on the TCP listener port assigned to the specified profile. You can use the OpenLDV xDriver Profile Editor to enable a profile for incoming session handling and assign it a listener port. For more information, see *xDriver Profiles* on page 136.

Remarks

This method can be invoked multiple times per application if you want to use multiple profiles to listen for incoming session requests in a single application. However, multiple applications cannot register for uplink session event handling with the same profile simultaneously.

After an application has registered for incoming session handling with this method, the **OnIncomingSessionEvent** event is fired each time a request for connection is received. The application is then responsible for accepting or rejecting all incoming uplink sessions. For more information on this event, see *OnIncomingSessionEvent* on page 159.

EndIncomingSessionEvents

Applies to: Object Server object

This method is used to end uplink session event handling within an application. It must be called before closing an application that has registered for uplink session handling with the **BeginIncomingSessionEvents** Method, or when the application should no longer be responsible for handling incoming sessions.

Syntax

objServer.**EndIncomingSessionEvents** *xDriverProfileName*

Table 39. EndIncomingSessionEvents Parameters

Element	Description
xDriverProfileName	The name of the xDriver profile used in the call to **BeginIncomingSessionEvents**.

Remarks

Call this method for each profile for which the **BeginIncomingSessionsEvent** method was called before closing an application.

NetworkInterfaces.Item( )

Applies to: NetworkInterfaces collection object

The *Item* property of the **NetworkInterfaces** collection object behaves differently when the network interface being accessed is an xDriver network interface.

Syntax

retrievedObject = *collObject*.**Item**(*index*)

retrievedObject = *collObject*.**Item**(*stringExpression*)

Table 40. NetworkInterfaces.Item Parameters

Element	Description
retrievedObject	Object variable that stores the **NetworkInterface** item retrieved from the **NetworkInterfaces** collection.
collObject	The collection object to be acted on.
index	A Long type specifying the ordinal index of the object to retrieve.

Element	Description
stringExpression	A string type specifying the name of the object to retrieve. For xDriver network interfaces, the network interface name of the RNI can be a maximum of 128 characters long, and must be specified using the following naming convention: **X**.[*ProfileName*].[*Downlink Lookup Key*] where [*ProfileName*] represents the name of the xDriver profile that manages the connection to the RNI and [*Downlink Lookup Key*] represents the downlink lookup key assigned to the RNI in the xDriver database. For example, if the xDriver profile name is *myProfile* and the downlink lookup key is *RNI-0001*, the network interface name would be: **X.myProfile.RNI-0001** For information about using this method with non-xDriver network interfaces, see the *LNS Object Server Reference* online help.

Remarks

All local interfaces appear in the **NetworkInterfaces** collection. However, xDriver RNIs that use custom lookup extension components do not appear in the **NetworkInterfaces** collection until a session with that RNI has been fully established. On the other hand, because the default xDriver lookup extension uses the Windows Registry, all configured RNIs appear in the LNS **NetworkInterfaces** collection. For other lookup extension implementations, it is possible to create the appropriate **NetworkInterface** object by name from the LNS **NetworkInterfaces** collection object using the **NetworkInterfaces.Item** method.

For sample programs that use this method with xDriver networks, see *Downlink Sample Applications* on page 140.

OnIncomingSessionEvent

Applies to: Object Server object

This event is fired whenever a request for connection to the LNS Server is received, as long as the application has registered for uplink session event handling with the **BeginIncomingSessionEvents** method.

Syntax

OnIncomingSessionEvent(*xDriverProfileName*, *netName*, *intfName*, *tag*)

Table 41. OnIncomingSessionEvent Parameters

Element	Description
xDriverProfileName	This string identifies the profile that is using the TCP listener port for this session. This name can be useful in an application that registers for uplink session event handling with multiple xDriver profiles.

Element	Description
netName	A string that represents the LNS network name of the network that requested the session.
intfName	A string that represents the network interface name of the network that requested the session.
tag	This value must be used when the **AcceptIncomingSession** method is called to accept or reject the session.

Remarks

After an uplink session request has been received and this event has fired, use the **AcceptIncomingSession** method to accept or reject the request. The *Tag* element passed to this event is used by the **AcceptIncomingSession** method to identify the xDriver session. The other elements can be used to open the network if the incoming session is accepted.

Important: Do not open the network within the event handler. Instead, signal your main thread to open the network by posting a message or using a timer.

ReleasePendingUpdates

Applies to: Application device object

Call this method to release monitor-point update events withheld after the *PostponeUpdates* field in the **AcceptIncomingSession** method is set to **True**.

Syntax

appDevice.**ReleasePendingUpdates**

Table 42. ReleasePendingUpdates Parameters

Element	Description
appDevice	The AppDevice object being acted upon.

Remarks

The **AcceptIncomingSession** method includes a *PostponeUpdates* parameter. If this parameter is set to **True** when a session is accepted, monitor-point updates for the network involved in this session are withheld until this method is called. Withholding the updates ensures that no monitor-point update events are lost before the network that requested the uplink session is opened, and that the application receives the monitor-point update event that caused the uplink session request.

The **ReleasePendingUpdates** method must be called after the monitor set for the remote network involved in the session is enabled. For an example of this, see *Uplink Sample Application* on page 144.

Recommendation: Open the network in server-independent mode when you plan to use this method, because using this method in server-dependent mode could disrupt network management operations. If you are not operating in server-independent mode and you call this method, an exception is thrown. However, the monitor-point update events are released.

C

Custom Lookup Extension Component Programming

This appendix describes the interfaces and methods that your custom lookup extension component can use or must implement.

Overview

This appendix describes the interfaces and methods that your custom lookup extension component can use or must implement:

- Implement
  - ILdvxConfigure (optional)
  - ILdvxLookup (required)
- Use
  - ILdvxSCO
  - ILdvxSCO2
  - ILdvxSCO_TCP

A lookup extension component must implement **ILdvxLookup**, and can optionally implement **ILdvxConfigure**. During its operation, the lookup extension component calls methods of the **ILdvxSCO**, **ILdvxSCO2**, and **ILdvxSCO_TCP** interfaces. Types and error codes used by the RNIs are declared in the **LdvxTypes.h** and **LdvxResult.h** header files, or in the **Ldvx.tlb** type library.

The **SampleLookupCsv.cpp** and **SampleLookupVBNet.vb** sample lookup extension components use the methods described in this chapter. See *Sample Lookup Extension Component* on page 134 for more information about these examples.

ILdvxConfigure Interface

This configuration interface is an optional interface implemented by an xDriver lookup extension component. It defines methods that are used to pass configuration information (instance name and options) to the lookup extension component at instantiation.

SetInstance Method

Applies to: xDriver Lookup Extension Component

This method is passed the instance name of the lookup extension component for a session. Typically, each profile has its own instance of a lookup extension component.

Profiles can be configured to share an instance of an xDriver lookup extension component by using the same instance name, which allows a single xDriver lookup extension component to be shared by multiple profiles. The instance name itself can be used as a key internally. You can set the lookup instance to be used by a given xDriver profile with the OpenLDV xDriver Profile Editor. For more information, see *xDriver Profiles* on page 136.

Syntax

C++	STDMETHOD(**SetInstance**)(BSTR *instance*)
Visual Basic	Sub **SetInstance**(ByVal *instance* As **String**)

Table 43. SetInstance Parameters

Parameter	Description
instance	Name of the lookup extension instance. Defaults to the profile name, if not configured in the profile.

Returns

Standard xDriver LdvxResult (**HRESULT**) describing the result of the call.

SetOptions Method

Applies to: xDriver Lookup Extension Component

This method passes an arbitrary options string from the profile to the xDriver lookup extension component. For example, this string can specify a database path or a dial-up-networking prefix, depending on the needs of a custom extension.

You can set the lookup options string to be used by a given xDriver profile with the OpenLDV xDriver Profile Editor. For more information, see see *xDriver Profiles* on page 136.

Syntax

C++	STDMETHOD(**SetOptions**)(BSTR *options*)
Visual Basic	Sub **SetOptions**(ByVal *options* As **String**)

Table 44. SetOptions Parameters

Parameter	Description
options	Arbitrary options string (defaults to empty string).

Returns

Standard xDriver LdvxResult (**HRESULT**) describing the result of the call.

ILdvxLookup Interface

The lookup interface is the primary interface implemented by an xDriver lookup extension component. It defines the methods that are used to look up session parameters in your xDriver database.

DownlinkLookup Method

Applies to: xDriver Lookup Extension Component

This method is called by xDriver when a downlink session is initiated. It is passed a pointer to the xDriver Session Control Object (SCO) for the session. The SCO contains the downlink lookup key to be looked up. This lookup key comes from the network interface name of the RNI being opened. For example, in the "X.Custom.Location-123" NetworkInterface name, the downlink key is "Location-123".

It is the responsibility of the xDriver lookup extension to extract the downlink lookup key from the SCO, use it to access its xDriver database, and retrieve the authentication and TCP parameters to be used by the rest of the xDriver framework. Then, the lookup extension component must fill in the corresponding fields of the SCO, including the Authentication Flag, Current Authentication Key, Next Authentication Key, Additional Downlink Packet Header (optional), Additional Downlink Packet Trailer (optional), Encryption Type, Remote TCP Address, and Remote TCP Port. This information is used by xDriver to complete the session establishment.

The **ILdvxSCO** interface provides methods that you can use to obtain the downlink lookup key and fill in the SCO fields. For more information, see *ILdvxSCO Interface* on page 168.

Syntax

C++	STDMETHOD(**DownlinkLookup**)(**ILdvxSCO *** *xSCO*)
Visual Basic	Sub **DownlinkLookup**(ByVal *xSCO* As **LdvxLib.ILdvxSCO**)

Table 45. DownlinkLookup Parameters

Parameter	**Description**
xSCO	Pointer to the SCO object that contains the downlink key to be looked up. The **DownlinkLookup** implementation should fill the required fields into the SCO.

Returns

Standard xDriver LdvxResult (**HRESULT**) describing the result of the call. The result is typically **LDVX_S_OK** (see **LdvxResult**). If the specified downlink key is not recognized, the lookup extension component must return the error code

E_HANDLE, **LDVX_E_INVALID_DOWNLINK_KEY**, or **LDVX_E_LOOKUP_FAILURE**.

UpdateLookup Method

Applies to: xDriver Lookup Extension Component

This method is called by xDriver upon the completion of a change to a session authentication key by xDriver. The lookup extension component must implement an update to its database from this method, so that it stores the new values of the current authentication key and the next authentication key from the SCO. These fields can only be updated from the **UpdateLookup** method.

For more information about authentication key handling, see *Authentication Key Handling* on page 113.

Syntax

C++	STDMETHOD(**UpdateLookup**)(**ILdvxSCO** * *xSCO*)
Visual Basic	Sub **UpdateLookup**(ByVal *xSCO* As **LdvxLib**.**ILdvxSCO**)

Table 46. UpdateLookup Parameters

Parameter	Description
xSCO	Pointer to the Session Control Object that contains the new authentication keys.

Returns

Standard xDriver LdvxResult (**HRESULT**) describing the result of the call.

UplinkLookup Method

Applies to: xDriver Lookup Extension Component

This method is called by xDriver when an uplink session is initiated. It is passed a pointer to the xDriver Session Control Object (SCO) for the session. The SCO contains the uplink lookup key passed in by the RNI.

It is the responsibility of the xDriver lookup extension to extract this lookup key from the SCO, use it to access its xDriver database, and retrieve the authentication and network parameters to be used by the rest of the xDriver framework. Then, the lookup extension component must fill in the corresponding fields of the SCO, including the Authentication Flag, Current Authentication Key, Next Authentication Key, LNS Network Name, Downlink Key, and Encryption Type. This information is then used by xDriver to complete the session establishment.

The **ILdvxSCO** interface provides methods that you can use to obtain the uplink lookup key and fill in the SCO fields. For more information, see *ILdvxSCO Interface* on page 168.

Syntax

C++	STDMETHOD(**UplinkLookup**)(**ILdvxSCO** * *xSCO*)
Visual Basic	Sub **UplinkLookup**(ByVal *xSCO* As **LdvxLib.ILdvxSCO**)

Table 47. UplinkLookup Parameters

Parameter	Description
xSCO	Interface to the SCO object. This element contains the uplink key to be looked up. The **UplinkLookup** implementation must fill in the required SCO fields.

Returns

Standard xDriver LdvxResult (**HRESULT**) describing the result of the call. The result is typically **LDVX_S_OK** (see **LdvxResult**). If the specified uplink key is not recognized, the lookup extension component must return the error code **E_HANDLE**, **LDVX_E_INVALID_UPLINK_KEY**, or **LDVX_E_LOOKUP_FAILURE**.

ILdvxSCO Interface

The Session Control Object interface is one of the main interfaces to the xDriver Session Control Object (SCO). It provides methods that are used by user extensions to access and assign values to the common fields of the SCO. For a description of these fields, see *Session Control Object* on page 110.

These methods are called from the **UplinkLookup**, **DownlinkLookup**, and **UpdateLookup** methods. **Table 48** lists the access that the lookup extension component has to each SCO field from these methods.

Table 48. SCO Fields

	Called From		
Field Name	**DownlinkLookup**	**UplinkLookup**	**UpdateLookup**
Session Control Object ID	Read Only	Read Only	Read Only
Authentication Flag	Read/Write	Read/Write	Read Only
LNS Network Name	Read/Write	Read/Write	Read Only
Downlink Key	Read Only	Read/Write	Read Only

Field Name	Called From		
	DownlinkLookup	**UplinkLookup**	**UpdateLookup**
Uplink Key	Read/Write	Read Only	Read Only
Current Authentication Key	Read/Write	Read/Write	Read Only
Next Authentication Key	Read/Write	Read/Write	Read Only
Additional Downlink Packet Header	Read/Write	Read/Write	Read Only
Additional Downlink Packet Trailer	Read/Write	Read/Write	Read Only

GetAdditionalDownlinkPacketHeader Method

Applies to: Session Control Object

This method obtains any additional bytes that are pre-pended to the packet headers sent during a downlink session.

Syntax

C++	STDMETHOD(**GetAdditionalDownlinkPacketHeader**)(BSTR * *hexBytes*)
Visual Basic	Function **GetAdditionalDownlinkPacketHeader**() As **String**

Table 49. GetAdditionalDownlinkPacketHeader Parameters

Parameter	Description
hexBytes	String to contain returned bytes.

Returns

Hexadecimal string containing the bytes to be pre-pended, two characters per byte.

GetAdditionalDownlinkPacketTrailer Method

Applies to: Session Control Object

This method obtains any additional bytes that are being appended to the packet trailers sent during a downlink session.

Syntax

C++	STDMETHOD(**GetAdditionalDownlinkPacketTrailer**)(BSTR * *hexBytes*)
Visual Basic	Function **GetAdditionalDownlinkPacketTrailer**() As **String**

Table 50. GetAdditionalDownlinkPacketHeader Parameters

Parameter	Description
hexBytes	String to contain returned bytes.

Returns

Hexadecimal string containing the bytes to be appended, two characters per byte.

GetAuthenticationFlag Method

Applies to: Session Control Object

This method obtains the flag that determines whether the xDriver protocol engine is to use link authentication with an MD5 per-packet digest. This field is always true for a SmartServer or i.LON 600.

Syntax

C++	STDMETHOD(**GetAuthenticationFlag**)(VARIANT_BOOL * *bAuth*)
Visual Basic	Function **GetAuthenticationFlag**() As **Boolean**

Table 51. GetAuthenticationFlag Parameters

Parameter	Description
bAuth	Boolean variable that stores the return data.

Returns

xDriver authentication state as Boolean. If **True**, the xDriver protocol engine generates and validates link-level authentication. If **False**, the xDriver protocol engine neither generates nor validates link-level authentication.

GetCurrentAuthenticationKey Method

Applies to: Session Control Object

This method obtains the current xDriver authentication key.

Syntax

C++	STDMETHOD(**GetCurrentAuthenticationKey**)(BSTR * *authKey*)
Visual Basic	Function **GetCurrentAuthenticationKey**() As **String**

Table 52. GetCurrentAuthenticationKey Parameters

Parameter	**Description**
authKey	String variable that stores the return data.

Returns

Current authentication key for the RNI, as a 32-character hexadecimal string representing a 128-bit MD5 authentication key.

GetDownlinkKey Method

Applies to: Session Control Object

This method obtains the downlink lookup key. This key comes from the network interface name that is specified in the LNS application. For example, in the "X.Default.Location-123" network interface name, the downlink lookup key is "Location-123". It is the responsibility of the lookup extension component to map this key to the database, and then fill the authentication and TCP parameters to be used by the rest of the xDriver framework into the SCO in the DownlinkLookup method.

For more information about the DownlinkLookup method, see *DownlinkLookup* on page 166.

Syntax

C++	STDMETHOD(**GetDownlinkKey**)(BSTR * *dnKey*)
Visual Basic	Function **GetDownlinkKey**() As **String**

Table 53. GetDownlinkKey Parameters

Parameter	**Description**
dnKey	Variable (String) that stores the return value.

Returns

The downlink lookup key of the RNI as a string.

GetEncryptionType Method

Applies to: Session Control Object

This method obtains the type of encryption that the xDriver protocol engine is using when sending encrypted data packets for this session.

Syntax

C++	STDMETHOD(**GetEncryptionType**)(**LdvxEncryption** * *nType*)
Visual Basic	Function **GetEncryptionType**() As **LdvxLib**.**LdvxEncryption**

Table 54. GetEncryptionType Parameters

Parameter	Description
nType	String variable that stores the return data.

Returns

Encryption type being used for the session.

GetLNSNetworkName Method

Applies to: Session Control Object

This method obtains the LNS network name. It is only used when the OpenLDV application is an LNS Server. This name is only available if the LNS network name has been set using the **SetLNSNetworkName** method.

Syntax

C++	STDMETHOD(**GetLNSNetworkName**)(BSTR * *lnsNetwork*)
Visual Basic	Function **GetLNSNetworkName**() As **String**

Table 55. GetLNSNetworkName Parameters

Parameter	Description
lnsNetwork	Variable (String) that stores the return value.

Returns

The name of the LNS network associated with the RNI as a String.

GetNextAuthenticationKey Method

Applies to: Session Control Object

This method obtains the next xDriver authentication key.

Syntax

C++	STDMETHOD(**GetNextAuthenticationKey**)(BSTR * *authKey*)
Visual Basic	Function **GetNextAuthenticationKey**() As **String**

Table 56. GetNextAuthenticationKey Parameters

Parameter	Description
authKey	String variable that stores the return data.

Returns

The next authentication key for the RNI, as a 32-character hexadecimal string representing a 128-bit MD5 authentication key.

GetSessionControlObjectID Method

Applies to: Session Control Object

This method obtains the SCO ID. The SCO ID can be used as a key to store information related to an xDriver session in external memory. You could allocate a block of memory for these fields, and tag that block of memory with the SCO ID. You would then program your lookup extension component to find and retrieve this memory block using the SCO ID.

Syntax

C++	STDMETHOD(**GetSessionControlObjectID**)(long * *nSCOID*)
Visual Basic	Function **GetSessionControlObjectID**() As **Integer**

Table 57. GetSessionControlObjectID Parameters

Parameter	Description
nSCOID	Variable (Long) that stores the SCO ID.

Returns

32-bit Session Control Object ID as Long.

GetUplinkKey Method

Applies to: Session Control Object

This method obtains the xDriver uplink lookup key. This key comes from the RNI identifier that is passed to xDriver during an uplink session, and is filled into the SCO automatically. It is the responsibility of the lookup extension component to map this key to the authentication and LNS network parameters to be used by the rest of the xDriver framework, and fill them into the SCO from the UplinkLookup function.

For more information on the UplinkLookup function, see *UplinkLookup* on page 167.

Syntax

C++	STDMETHOD(**GetUplinkKey**)(BSTR * *upKey*)
Visual Basic	Function **GetUplinkKey**() As **String**

Table 58. GetUplinkKey Parameters

Parameter	**Description**
upKey	Variable (String) that stores the return value.

Returns

The uplink lookup key of the RNI as a string.

SetAdditionalDownlinkPacketHeader Method

Applies to: Session Control Object

This method sets any additional bytes to pre-pend to the packet header sent during a downlink session. Normally this field is an empty string (the default). However, it can be used to specify a series of bytes that are pre-pended to every packet used in a downlink session if there is an intermediate proxy between the OpenLDV application and the RNI. These bytes can be used to provide routing information the proxy might require.

Syntax

C++	STDMETHOD(**SetAdditionalDownlinkPacketHeader**)(BSTR *hexBytes*)
Visual Basic	Sub **SetAdditionalDownlinkPacketHeader**(ByVal *hexBytes* As **String**)

Table 59. SetAdditionalDownlinkPacketHeader Parameters

Parameter	Description
hexBytes	Hexadecimal string containing the bytes to be pre-pended, as pairs of hexadecimal digits. For example: `0B0C0D0E0F10`.

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if field is presently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

SetAdditionalDownlinkPacketTrailer Method

Applies to: Session Control Object

This method sets any additional bytes to append to every packet trailer sent during a downlink session. Normally this field is an empty string (the default). Developers of lookup extensions can set this field to specify a series of bytes that are to be appended to the end of every packet used in a downlink session if there is an intermediate proxy between the OpenLDV application and the RNI. These bytes can be used to provide routing information the proxy might require.

Syntax

C++	STDMETHOD(**SetAdditionalDownlinkPacketTrailer**)(BSTR *hexBytes*)
Visual Basic	Sub **SetAdditionalDownlinkPacketTrailer**(ByVal *hexBytes* As **String**)

Table 60. SetAdditionalDownlinkPacketTrailer Parameters

Parameter	Description
hexBytes	Hexadecimal string containing the bytes to be appended, as pairs of hexadecimal digits. For example: `0B0C0D0E0F10`.

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if the field is presently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

SetAuthenticationFlag Method

Applies to: Session Control Object

This method sets the flag that determines whether xDriver should use link authentication with an MD5 per-packet digest for the session. The lookup extension component sets this value depending on the RNI used.

For more information about how the xDriver lookup extension component handles authentication, see *Authentication Key Handling* on page 113.

Syntax

C++	STDMETHOD(**SetAuthenticationFlag**)(VARIANT_BOOL *bAuth*)
Visual Basic	Sub **SetAuthenticationFlag**(ByVal *bAuth* As **Boolean**)

Table 61. SetAuthenticationFlag Parameters

Parameter	Description
bAuth	Authentication state as Boolean. If **True**, the xDriver protocol engine generates and validates link-level authentication. If **False**, the xDriver protocol engine neither generates nor validates link-level authentication. Always **True** for a SmartServer or i.LON 600.

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if the field is currently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

SetCurrentAuthenticationKey Method

Applies to: Session Control Object

This method sets the current xDriver authentication key. It is the responsibility of the lookup extension component to map the downlink or uplink lookup key to this authentication key, which is then used by xDriver to validate the connection. The current authentication key must match the MD5 authentication key configured into the RNI. For more information about how the lookup extension component should handle authentication, see *Authentication Key Handling* on page 113.

Setting an invalid current authentication key causes loss of contact with the RNI, because an xDriver session cannot be started with an invalid authentication key.

Syntax

C++	STDMETHOD(**SetCurrentAuthenticationKey**)(BSTR *authKey*)
Visual Basic	Sub **SetCurrentAuthenticationKey**(ByVal *authKey* As **String**)

Table 62. SetCurrentAuthenticationKey Parameters

Parameter	Description
authKey	xDriver authentication key for this session, as a 32-character hexadecimal string representing a 128-bit authentication key. The authentication key must be entered as a 32-character hexadecimal string representing a 128-bit MD5 key. For example: `0102030405060708090A0B0C0D0E0F10`

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if field is presently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

SetDownlinkKey Method

Applies to: Session Control Object

This method sets the downlink lookup key. xDriver sets this value automatically during a downlink session, and the lookup extension component should be programmed to fill it into the SCO during an uplink session.

Syntax

C++	STDMETHOD(**SetDownlinkKey**)(BSTR *dnKey*)
Visual Basic	Sub **SetDownlinkKey**(ByVal *dnKey* As **String**)

Table 63. SetDownlinkKey Parameters

Parameter	Description
dnKey	The downlink lookup key of the RNI, as a String (a maximum of 105 characters).

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if the field is currently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

SetEncryptionType Method

Applies to: Session Control Object

This method sets the encryption type that xDriver should use when sending encrypted data packets to an RNI. The xDriver lookup extension component

must set this value appropriately depending on the RNI used. The default value is **LDVX_ENCRYPTION_AUTO**.

Syntax

C++	STDMETHOD(**SetEncryptionType**)(**LdvxEncryption** *nType*)
Visual Basic	Sub **SetEncryptionType**(ByVal *nType* As **LdvxLib**.**LdvxEncryption**)

Table 64. SetEncryptionType Parameters

Parameter	Description
nType	Encryption type. The xDriver type library includes the following encryption identifiers: • LDVX_ENCRYPTION_AUTO • LDVX_ENCRYPTION_BEST • LDVX_ENCRYPTION_NONE • LDVX_ENCRYPTION_RC4 It is currently required that you use the **LDVX_ENCRYPTION_AUTO** identifier.

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if the field is currently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

SetLNSNetworkName Method

Applies to: Session Control Object

This method sets the LNS network name. It is only used when the OpenLDV application is an LNS Server. The lookup extension component must map the uplink key to the appropriate record in its xDriver database, extract the LNS network name for that RNI from the database, and fill it into the SCO using this method. For a downlink session, the LNS network name is specified manually within the LNS application and is not required.

Syntax

C++	STDMETHOD(**SetLNSNetworkName**)(BSTR *lnsNetwork*)
Visual Basic	Sub **SetLNSNetworkName**(ByVal *lnsNetwork* As **String**)

Table 65. SetLNSNetworkName Parameters

Parameter	Description
lnsNetwork	LNS network name as String (a maximum 85 characters).

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if field is currently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

SetNextAuthenticationKey Method

Applies to: Session Control Object

This method specifies the next xDriver authentication key to be used for the session. When no change to the current authentication key is desired, this key must be set the same as the current authentication key.

Changing this field causes the authentication key of the RNI to be incrementally updated with the new value of this field. After this change is complete, the **UpdateLookup** method is called. For more information about the **UpdateLookup** method, see *UpdateLookup* on page 167. For more information about how the lookup extension component should handle authentication, see *Authentication Key Handling* on page 113.

Syntax

C++	STDMETHOD(**SetNextAuthenticationKey**)(BSTR *authKey*)
Visual Basic	Sub **SetNextAuthenticationKey**(ByVal *authKey* As **String**)

Table 66. SetNextAuthenticationKey Parameters

Parameter	Description
authKey	The next authentication key for this session, as a 32-character hexadecimal string representing a 128-bit authentication key. The authentication key must be entered as a 32-character hexadecimal string representing a 128-bit MD5 key. For example: `0102030405060708090A0B0C0D0E0F10`

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if the field is presently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

SetUplinkKey Method

Applies to: Session Control Object

This method sets the uplink lookup key. This unique key comes from the ASCII RNI identifier passed to xDriver during an uplink session and is filled into the SCO automatically. Therefore it is not required that you fill the uplink key into the SCO.

Syntax

C++	STDMETHOD(**SetUplinkKey**)(BSTR *upKey*)
Visual Basic	Sub **SetUplinkKey**(ByVal *upKey* As **String**)

Table 67. SetUplinkKey Parameters

Parameter	Description
upKey	Uplink lookup key of the RNI as a String (105 characters maximum).

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if the field is currently read-only. **Table 48** on page 168 lists each SCO field, along with when these fields are read-only.

ILdvxSCO_TCP Interface

The **SCO_TCP** interface is one of the interfaces to the Session Control Object. It provides properties and methods used by user extensions to access and assign values to the TCP-based fields of the xDriver Session Control Object.

This interface applies to the same object as the **ILdvxSCO** interface, and can be accessed using standard COM techniques.

The methods included in this interface can be called from the **UplinkLookup**, **DownlinkLookup**, and **UpdateLookup** methods. **Table 68** lists the access that the lookup extension component has to each **SCO_TCP** field from each of these methods.

Table 68. SCO_TCP Fields

	Called From		
Field Name	**DownlinkLookup**	**UplinkLookup**	**UpdateLookup**
Remote TCP Address	Read/Write	Read Only	Read Only
Remote TCP Port	Read/Write	Read Only	Read Only

GetRemoteTCPAddress Method

Applies to: Session Control Object

This method obtains the remote TCP address of the RNI for the xDriver session. For an uplink session, the remote TCP address is filled into the SCO automatically. You could use this method to check that the remote TCP address filled into the SCO is valid by comparing the address that this method returns against the address stored in the database.

Syntax

C++	STDMETHOD(**GetRemoteTCPAddress**)(BSTR * *tcpAddress*)
Visual Basic	Function **GetRemoteTCPAddress**() As **String**

Table 69. GetRemoteTCPAddress Parameters

Parameter	Description
tcpAddress	String variable that stores the return data.

Returns

The remote TCP address that the RNI is using, as a dotted decimal string or as a hostname.

GetRemoteTCPPort Method

Applies to: Session Control Object

This method obtains the remote TCP port that the RNIs involved in the xDriver session use to receive packets. For an uplink session, the remote TCP port number is filled into the SCO automatically. You could use this method to check that the remote TCP port filled into the SCO is valid by comparing it against the port expected to be stored in the database if static outbound RNI addresses are used.

Syntax

C++	STDMETHOD(**GetRemoteTCPPort**)(short * *tcpPort*)
Visual Basic	Function **GetRemoteTCPPort**() As **Short**

Table 70. GetRemoteTCPPort Parameters

Parameter	**Description**
tcpPort	Integer variable that stores the return data.

Returns

The remote TCP port number of the RNI as an Integer.

SetRemoteTCPAddress Method

Applies to: Session Control Object

This method sets the remote TCP address of the RNI. For a downlink session, it is the responsibility of the xDriver lookup extension to map the downlink lookup key to the database, extract the TCP address from the database, and fill it into the SCO using this method.

For an uplink session, the remote TCP address is filled in to the SCO automatically by the Connection Broker.

Syntax

C++	STDMETHOD(**SetRemoteTCPAddress**)(BSTR *tcpAddress*)
Visual Basic	Sub **SetRemoteTCPAddress**(ByVal *tcpAddress* As **String**)

Table 71. SetRemoteTCPAddress Parameters

Parameter	**Description**
tcpAddress	The remote TCP Address used to connect to the RNI as a dotted decimal IP address, or as a hostname. If an IP address is used, it must be entered in the form *x.x.x.x*, where *x* represents an integer between 0 and 255.

Returns

Standard COM **HRESULT** describing the result of the call. Returns **LdvxLib.LdvxResult.E_ACCESSDENIED** if the field is currently read-only. **Table 68** on page 180 lists each **SCO_TCP** field, along with when these fields are read-only.

SetRemoteTCPPort Method

Applies to: Session Control Object

This method sets the remote TCP port that the RNI at the other end of the connection uses to receive connections from the LNS Server. For a downlink session, it is the responsibility of the xDriver lookup extension to map the downlink lookup key to the database, extract the remote TCP port from the database, and fill it into the SCO using this method.

Syntax

C++	STDMETHOD(**SetRemoteTCPPort**)(short *tcpPort*)
Visual Basic	Function **GetRemoteTCPPort**() As **Short**

Table 72. SetRemoteTCPPort Parameters

Parameter	**Description**
tcpPort	Remote TCP port number that the RNI uses to receive connections from the LNS Server on as an integer. Must be in the range 1 - 65535. **Recommendation**: Use port 1024 or higher.

Returns

Standard COM **HRESULT** describing the result of the call. Returns **E_ACCESS_DENIED** if the field is currently read-only. **Table 68** on page 180 lists each **SCO_TCP** field, along with when these fields are read-only.

ILdvxSCO2 Interface

The **SCO2** interface is one of the interfaces to the Session Control Object. It provides additional properties and methods for user extensions.

GetNeuronID Method

Applies to: Session Control Object

This method gets the Neuron ID of the RNI.

Syntax

C++	STDMETHOD(**GetNeuronID**)(BSTR * *nNeuronID*)
Visual Basic	Function **GetNeuronID**() As **String**

Table 73. GetNeuronID Parameters

Parameter	**Description**
nNeuronID	String variable that stores the return data.

Returns

String representation of the 48-bit hexadecimal Neuron ID.

ECHELON®