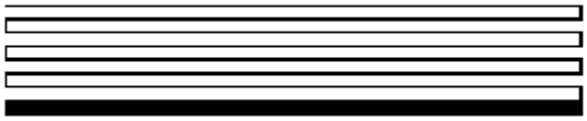


LNS Programmer's Guide

Turbo Edition

ECHELON
Corporation

078-0177-01F

Echelon Corporation
www.echelon.com

Preface

The LNS® network operating system provides a comprehensive set of software tools that allows multiple network applications to perform a broad range of services over LONWORKS® and IP networks. These services include network management (network installation, configuration, maintenance, and repair) and system-wide monitoring and control. This guide describes how to use the LNS Object Server ActiveX Control to write LNS applications on a Microsoft Windows® Server 2003, Windows XP, or Windows 2000 host PC.

Purpose

This guide describes how to use the LNS Object Server ActiveX Control to develop an LNS application on a Microsoft Windows Server 2003, Windows XP, or Windows 2000 host PC.

Audience

This guide is intended for software developers creating LNS applications. LNS applications may be written in any language that supports COM Components or ActiveX controls, including Microsoft® Visual C++ (versions 6.0-7.1) and Microsoft Visual Basic 6.0. Readers of this guide should have programming experience in such a language, and familiarity with LONWORKS® technology and COM/ActiveX control concepts. An introduction to LONWORKS technology can be found in the *Introduction to the* LONWORKS *System* document available at www.echelon.com.

Examples

Throughout this guide, Visual Basic code examples are used to illustrate concepts. To make the examples more easily understood, they have been simplified. Error checking has been removed, and in some cases, the examples are only fragments that will not compile.

For complete examples, see the example applications provided with the LNS Application Developer's Kit. This includes example network management, monitor and control, plug-in and director, and xDriver applications. Appendix C of this document describes the example applications in more detail.

Technical Support

If you have technical questions that are not answered within this document or the online help files provided with the *LNS Application Developer's Kit,* you can contact Echelon for technical support. Your LNS Application Developer's Kit distributor may also provide technical support. In addition, you can enroll in training classes at Echelon to learn more about LNS.

You can find out more about Echelon's technical support and training services on the Echelon Support home page at http://www.echelon.com/support.

System Requirements

System requirements and recommendations for the PC on which the LNS Application Developer's Kit and LNS Redistribution Kit, as well as the LNS Server or LNS Remote Client redistributions, will run are listed below.

Development System

The following requirements are for the PC on which the LNS Application Developer's Kit or LNS Redistribution Kit will be run:

- Windows Server 2003, Windows XP, or Windows 2000
- Pentium III 600 MHz or faster
- 256 MB RAM (512 MB RAM recommended)
- 50 MB or more of free disk space. For highest performance during development, Echelon recommends using a high-performance hard disk.
- Any Windows application development tool that supports the use of COM components or ActiveX controls. Echelon has tested and offers technical assistance on the following development environments only:
  - Microsoft Visual Basic 6.0, Service Pack 6 or higher
  - Microsoft Visual Studio .NET 2003 (C++, used with ATL or MFC)
- CD-ROM drive
- Mouse or compatible pointing device
- LNS Network Interface card. Chapter 11 of this document lists and describes the compatible LNS Network Interface cards. Note that an Ethernet card can function as an LNS Network Interface if you are using Echelon's *i*.LON 1000 Internet Server or the *i*.LON 600 LONWORKS/IP Server, and can also be used to access network interfaces such as the *i*.LON 100 Internet Server or the *i*.LON 10 Ethernet Adapter.
- The minimum screen size required to use all LNS utilities is SVGA 800x600.

LNS Server PC for a Smaller Network

The following requirements are for a PC on which an LNS Server for a smaller network will be run.

- Windows Server 2003, Windows XP, or Windows 2000
- Pentium 100MHz or faster
- 50 MB or more of free disk space, not taking into account the size of the LNS application and LNS network databases
- 256 MB RAM or more, depending upon the requirements of the LNS applications that will be running in addition to the LNS Server
- LNS Network Interface card. Chapter 11 of this document lists and describes the compatible LNS Network Interface cards. Note that an Ethernet card can function as an LNS Network Interface if you are using Echelon's *i*.LON 1000 Internet Server or the *i*.LON 600 LONWORKS/IP Server, and can also be used to access network interfaces such as the *i*.LON 100 Internet Server or the *i*.LON 10 Ethernet Adapter.
- The minimum screen size required to use all LNS utilities is SVGA 800x600.

LNS Server PC for a Larger, Busier Network

These requirements and recommendations are also valid for a PC on which a large network database is engineered before the LNS Server for that network becomes operational.

- Windows Server 2003, Windows XP, or Windows 2000
- Pentium 4 1GHz or faster
- 50 MB or more of free disk space, not taking into account the size of the LNS application and LNS network databases. For highest performance during development, Echelon recommends using a high-performance hard disk.
- 512 GB RAM or more, depending upon the requirements of the LNS applications that will be running in addition to the LNS Server. Echelon recommends that you have between 500 MB and 1 GB available.
- LNS Network Interface card. Chapter 11 of this document lists and describes the compatible LNS Network Interface cards. Note that an Ethernet card can function as an LNS Network Interface if you are using Echelon's *i*.LON 1000 Internet Server or the *i*.LON 600 LONWORKS/IP Server, and can be used to access network interfaces such as the *i*.LON 100 Internet Server or the *i*.LON 10 Ethernet Adapter.
- The minimum screen size required to use all LNS utilities is SVGA 800x600.

LNS Remote Client PC

The following requirements are for a PC on which the LNS Remote Client redistribution will be run.

- Windows Server 2003, Windows XP, or Windows 2000
- Pentium 100MHz or faster
- 50 MB or more of free disk space, not taking into account the size of the LNS application and LNS network databases
- 128 MB RAM or more, depending upon the requirements of the LNS applications that will be running in addition to the LNS Server
- LNS Network Interface card. Chapter 11 of this document lists and describes the compatible LNS Network Interface cards. Note that an Ethernet card can function as an LNS Network Interface if you are using Echelon's *i*.LON 1000 Internet Server or the *i*.LON 600 LONWORKS/IP Server, and can also be used to access network interfaces such as the *i*.LON 100 Internet Server or the *i*.LON 10 Ethernet Adapter.
- The minimum screen size required to use all LNS utilities is SVGA 800x600.

Table of Contents

Chapter 1 - Installing the LNS Software

This chapter describes how to install the LNS Turbo Edition software.

System Requirements

System requirements and recommendations for the PC on which the LNS Application Developer's Kit, LNS Server or LNS Remote Client redistributions will run are listed in the *System Requirements* section on page ii of this document. Before installing any LNS software, you should make sure the installation PC meets these requirements.

3rd Party Software

Before installing the LNS Application Developer's Kit, you should note that LNS installs the following 3rd party software:

- FastObjects 9.5. LNS uses FastObjects 9.5 as its object database engine.
- CrypKey 5.7. LNS uses CrypKey 5.7 as part of its licensing software.
- Microsoft XML Parser 3.0

You should be aware of this if you are using other versions of these products on your target PC. In addition, you should be aware that when service packs and fixes for these products are released, it may have ramifications for the LNS runtime software.

Installing the LNS Application Developer's Kit

To install the LNS Application Developer's Kit and begin developing an LNS application, follow these steps:

1. Install the LNS Application Developer's Kit.
2. If you are planning to redistribute your LNS applications, install the LNS Redistribution Kit.
3. Start developing your LNS application.

Installing the LNS Application Developers Kit Software

The LNS Application Developer's Kit uses an automated, Windows-based installation program called `SETUP`. Because the LNS Application Developer's Kit CD contains compressed files, you cannot install the software by copying the files directly to your hard disk. This section describes how you can install the LNS Application Developer's Kit.

You should stop all LNS-based applications, utilities and services before performing this installation. This includes the LNS Server application, the OpenLDV xDriver connection broker, and any LNS client applications running as Windows services. You will need to manually restart these services after you complete the installation.

In addition, you must log in as a member of the Administrators user group when installing the LNS Application Developer's Kit. To install the LNS software, perform the following steps:

1. Insert the LNS Application Developer's Kit CD into a CD-ROM drive. If the start-up window does not automatically start after a few seconds, start the setup program manually. You can start the start-up program

like any other Windows program, e.g. by selecting the **Run...** item from the **Start** menu of the Windows task bar, browsing to the Setup application, and clicking **Open**.

2. This opens the LNS Application Developer's Kit screen. Select **Install Products**, and then select **LNS Application Developer's Kit** to begin installing the LNS Application Developer's Kit software.

3. This opens the Welcome window. Click **Next** to continue.

4. This opens the License Agreement window. Read the license agreement, and if you agree to the terms of the license agreement, click **I accept the terms in the license agreement.** Then, click **Next.**

5. This opens the Customer Information dialog. Enter your name in the **User Name** text box, and enter your company name in the **Organization** text box. If you want the LNS Application Developer's Kit to be available to anyone who logs onto your PC, select **Anyone who uses this computer**. Or, select **Only for me** if only you should have access to the LNS Application Developer's Kit. Then, click **Next** to continue.

6. This opens the **Ready to Install the Program** window. Click **Install** to begin the installation.

7. When LNS has completed the installation, a confirmation dialog will appear. Click **Finish** to exit the installation and return to the main LNS Application Developer's Kit screen.

Installing the LNS Redistribution Kit

If you are using the LNS Application Developer's Kit, you may distribute your LNS applications, but may not distribute any of the LNS runtime files. If you want to redistribute LNS runtimes files, you must use the LNS Redistribution Kit. See Chapter 14, *Distributing LNS Applications,* for more information on redistributing LNS applications.

In LNS Turbo Edition, the LNS Server and LNS Remote Client installations, which are created and installed with the LNS Redistribution Kit, have been recreated as Windows Installer installations. This is to make LNS installations more compatible with recent versions of Windows. For more information on this change, see *New LNS Runtime Installations* on page 18.

You must log in as a member of the Administrators user group when installing the LNS Redistribution Kit. To install the LNS Redistribution Kit, follow these steps:

1. Insert the LNS Application Developer's Kit CD into a CD-ROM drive. If the start-up window does not automatically start after a few seconds, start the setup program manually. You can start the start-up program like any other Windows program, e.g. by selecting the **Run...** item from the **Start** menu of the Windows task bar, browsing to the Setup application, and clicking **Open**.

2. This opens the LNS Redistribution Kit screen. Select **Install Products**, and then select **LNS Redistribution Kit** to begin installing the LNS Redistribution Kit software.

3. This opens the Welcome window. Click **Next** to continue.

4. This opens the License Agreement window. Read the license agreement, and if you agree to the terms of the license agreement, click **I accept the terms in the license agreement.** Then, click **Next.**

5. This opens the Customer Information dialog. Enter your name, company name, and product serial number in the appropriate text boxes. If you want the LNS Redistribution Kit to be available to anyone who logs onto your PC, select **Anyone who uses this computer**. Or, select **Only for me** if only you should have access to the LNS Redistribution Kit. Then, click **Next** to continue.

6. This opens the **Ready to Install the Program** window. Click **Install** to begin the installation.

7. When LNS has completed the installation, a confirmation dialog will appear. Click **Finish** to exit the installation and return to the main LNS Redistribution Kit screen.

Developing Your LNS Application

Your applications will access the network services provided by LNS using the LNS Object Server ActiveX Control. This control gives you access to the LNS object hierarchy, where each object corresponds to physical and logical objects in your network. Your LNS applications will retrieve information from the LNS network operating system via the object hierarchy. You will also use the hierarchy to access device information directly. In short, everything will be done through the hierarchy.

You do not need to understand or use every object in the hierarchy to write a useful LNS application. However you should be familiar with most of the objects in the hierarchy. You should read the remainder of the *LNS Programmer's Guide* for information on the LNS programming model, and detailed instructions on how you should write your LNS applications.

The next chapter describes the new features that have been added to LNS for Turbo Edition. The following chapter (Chapter 3, *LNS Overview*) provides an overview of the LNS object hierarchy, and a roadmap you can follow through the rest of this document.

Chapter 2 - What's New in Turbo Edition

This chapter introduces the new features that have been added to LNS for Turbo Edition, and describes how LNS performance has improved in Turbo Edition. It also provides guidelines to follow when upgrading your applications to use Turbo Edition.

Performance Enhancements

For Turbo Edition, numerous internal changes have been made that will significantly improve the speed, performance and stability of most LNS applications. This includes improved internal use of transactions, locking and indexing for faster database operations.

Internal indexing in some of the major LNS collections has been vastly improved, so the time required to access items from these collections has been reduced from a linear-time operation (varying with respect to the size of the collection) to a relatively small, constant-time operation. For large networks with large databases, this will make for a noticeable performance improvement when you use the `Item` property, the `ItemByHandle()` method, the `ItemByIndex()` method, or the `ItemByProgrammaticName()` method to retrieve an object from the following collections:

- `AppDevices` Collection (when accessed through `Subsystem` object)
- `NetworkVariables` Collection
- `ConfigProperties` Collection
- `LonMarkObjects` Collection
- `MessageTags` Collection
- `Routers` Collection

The introduction of temporary monitor sets in Turbo Edition will improve the performance of your monitor and control applications. Temporary monitor sets serve the same purpose as the permanent monitor sets that were introduced in Release 3.0. However, temporary monitor sets are not stored in the LNS database. As a result, it takes considerably less time to create temporary monitor sets than it does to create permanent monitor sets. Temporary monitor sets can only be used in a single client session by the client application that created them. In contrast, permanent monitor sets can be used by multiple clients in multiple client sessions. If you are creating a monitor and control application that does not need to re-use monitor sets and monitor points, you can maximize your application's performance by using temporary monitor sets. For more information on temporary monitor sets, see the next section, *New Features*.

New Features

This section describes the major features that have been added to LNS for Turbo Edition. This includes the following topics:

- *Enhanced Data Formatting*
- *Changeable Network Variable Types*
- *Improved Support for Dynamic Interfaces*
- *Improved Monitoring Performance*
- *Availability of Network Resource Information*

- *Enhanced LonMark Interoperability*
- *Improved Device Commissioning Performance*
- *System Management Mode Enhancements*
- *Enhanced Configuration Property Management*
- *Online Database Validation and Backup*

Enhanced Data Formatting

LNS 3.0 featured the introduction of the `DataPoint` object, which you can use to read and write to the values of monitor points. Formatting of `DataPoint` objects is handled locally, in the client process. As a result, formatting changes made to the value of a monitor point through a data point by a given application do not affect other applications that are reading the value of the same monitor point. This eliminates any confusion caused when separate client applications need to format data independently.

In LNS Turbo Edition, you can create `DataPoint` objects to read and write the values of network variables and configuration properties, as well as to the values of monitor points. LNS Turbo Edition also features several modifications and enhancements to the `DataPoint` object that will allow each client application to better format and display the data it accesses through `DataPoint` objects. This section describes these changes.

GetDataPoint Method

The `ConfigProperty` and `NetworkVariable` objects now include an additional method, the `GetDataPoint()` method. This method returns a `DataPoint` object you can use to read and write to the value of the configuration property or network variable. Most of the properties and methods you can use with each `DataPoint` you access through a configuration property or a network variable are the same as those you could use with the `DataPoint` objects accessed through monitor points in LNS 3.0. For more information on using `DataPoint` objects to access network variables, see *Directly Reading and Writing Network Variables* on page 231. For more information on using `DataPoint` objects to access configuration properties, see *Using Configuration Properties In a Monitor and Control Application* on page 233.

In addition, several properties have been added to the `DataPoint` object in this release: the `SourceOptions`, `SourceIndex`, `MinValue` and `MaxValue` properties. You can use the `SourceOptions` and `SourceIndex` properties to identify the type of source object used to create a data point. You can use the `MinValue` and `MaxValue` properties to identify or set the minimum and maximum values that should be assigned to a scalar data field. For more information on these properties, see the *LNS Object Server Reference* help file.

FormatLocales Collection

The new `FormatLocales` collection is another feature that you can take advantage of when using `DataPoint` objects. Each `FormatLocale` object contains a series of properties that reflect a particular geographical area's conventions for data display. These conventions affect how data should be displayed in that area, including factors

such as language, measurement system (U.S. or Systeme Internationale), date formats, time formats, and decimal number formats. The settings of a `FormatLocale` object determine how data accessed through the `FormattedValue` properties of all `DataPoint` objects will be displayed when your application uses that `FormatLocale` object.

Each client application can select the `FormatLocale` object it will use by passing that `FormatLocale` to the `CurrentFormatLocale` property of the `ObjectServer` object before opening any networks and formatting any data. As a result, client applications in different regions can use their own sets of local formats when displaying data, without affecting the data formatting used by other clients. This greatly reduces the risk of your application returning information that is confusing or misleading due to formatting changes made by another application.

You can access the `FormatLocales` collection through the `FormatLocales` property of the `ObjectServer` object. When initialized, the `FormatLocales` collection contains 4 pre-defined `FormatLocale` objects:

1. `UserDefaultRegionalSettings`. This is the default value for the `CurrentFormatLocale` property. When you use this `FormatLocale` object, all the properties will be set based on the user-defined Windows regional settings for the user currently logged onto the PC running your application. You can change the regional settings on a PC using the Windows control panel Regional Options applet. Consult the Microsoft Developer's Network (MSDN) documentation of the Win32 `GetLocaleInfo()` function for more information on the Windows regional settings.

2. `SystemDefaultRegionalSettings`. When you use this `FormatLocale` object, all the properties will be set based on the default Windows regional settings on the PC running the application. The default settings may vary, depending on which operating system is installed on the PC. Consult the MSDN documentation of the Win32 `GetLocaleInfo()` function for more information on the Windows Regional settings.

3. `LonMarkCompatibility`. When you use this `FormatLocale` object, all properties will be set so that formatted data will be displayed based on the LonMark standards used prior to LNS 3.0, when localized formatting was not available. In this case, Systeme Internationale measurement units, LonMark-defined time and date formats, and U.S. options for everything else, will be used to display all formatted data.

4. `ISO8601DateAndTime`. When you use this `FormatLocale` object, the settings will be the same as the `LonMarkCompatibility` settings, except for the localized time and date formats, which will be based on the ISO 8601 standard. This standard helps avoid confusion that may be caused by the different national notations used for dates and times, and increases the portability of computer user interfaces.

The pre-defined `FormatLocale` objects are read-only, but you can create custom `FormatLocale` objects to suit the specific needs of your application with the `Add` method of the `FormatLocales` collection. Note that all custom `FormatLocale` objects are instantiated with the same initial settings as the pre-defined `UserDefaultRegionalSettings FormatLocale` object described above.

For more information on the `FormatLocales` collection, see *Data Formatting* on page 241.

Changeable Network Variable Types

Each `NetworkVariable` object contains a new property called the `ChangeableTypeSupport` property, which indicates whether or not you can use LNS to change the network variable's type. If the `ChangeableTypeSupport` property is set to `lcaNvChangeableTypeSdOnly` or `lcaNvChangeableTypeSCPT`, you can change the network variable's type. You can do so by modifying the `NetworkVariable` object's new `TypeSpec` property.

The `TypeSpec` property returns a `TypeSpec` object. If a network variable supports changeable types, you can change the various properties of the `TypeSpec` object (`ProgramId`, `Scope`, `TypeName`), and then pass it back to the `TypeSpec` property to change the network variable's type. Alternatively, you can write the `TypeSpec` property of one network variable to the `TypeSpec` property of another network variable. By doing so, you can change the type of a network variable to any type contained in your local set of resource files, provided that the device containing the network variable supports the selected type.

If a network variable supports changeable types via a SCPTnvType configuration property (as specified in version 3.3 of the *LonMark Application-Layer Interoperability Guidelines*), LNS will automatically set the value of the SCPTnvType configuration property when the network variable's `TypeSpec` or `SnvtId` property is changed.

For more information on LNS support for changeable network variable types, see *Changeable Network Variable Types* on page 185.

Improved Support for Dynamic Interfaces

A typical device in a LONWORKS network uses a static interface, consisting of a static set of LonMark Functional Blocks, member network variables and configuration properties that define the device's functionality in the system. In most cases, this functionality is dependent upon the device's hardware and defined by the device manufacturer.

Sometimes, there is a need to modify this functionality. For example, controller devices may be used to control other devices. As a result, the number of components required for a controller device's interface is often an attribute of the network configuration (e.g. how many devices it is controlling), rather than of the device's hardware. Ideally, the resources on these controllers would be allocated dynamically, in order to fit the changing requirements of a given network as devices are added to it.

In Release 3.0, LNS supported dynamic network variables, which solved one aspect of the dynamic allocation of resources on a controller. However, dynamic network variables do not convey semantic information to a device or an LNS application in the same way that LonMark Functional Blocks and LonMark Functional Block membership do. In order to support the dynamic definition of a controller's interface in Turbo Edition, LNS has been enhanced to support additional dynamic interface components.

Dynamic LonMark Functional Blocks and dynamic network variables can only be added to application devices that support them. For example, Network Services Devices support up to 4096 dynamic network variables, but do not support dynamic LonMark Functional Blocks. The number of LonMark Functional Blocks and dynamic network variables that a device supports is documented in the device's external interface file and self-documentation. LNS reads this information from the external interface file when it is

imported, or from the device's self-documentation when it uploads the device's device template.

In LNS, an application device's interface is represented by an `Interface` object. The `Interface` objects contained by an application device may be the device's main, static interface, or they may be custom interfaces that have been added to the device dynamically, using the new features described in this section. You can access the main interface of a device through the `AppDevice` object's `Interface` property. Main interfaces are static interfaces that cannot be modified.

You can access the collection of custom interfaces that have been added to a device through the device's `Interfaces` property. If supported by the device, you can modify these custom interfaces by adding or removing objects from their `NetworkVariables`, `LonMarkObjects`, and `MessageTags` collections.

Or, you can add custom `Interface` objects to a device by accessing the `Interfaces` collection through the device's `Interfaces` property, and then calling the `Add()` method. Once you have created a custom `Interface` object, you can modify it to suit your exact needs.

You can access the collection of LonMark Functional Blocks defined on the custom `Interface`. A LonMark Functional Block represents a collection of network variables and configuration properties on a device that perform a related function. For example, a digital input device with four switches could contain one functional block for each switch.

In LNS, LonMark Functional Blocks are represented by `LonMarkObject` objects. Each `Interface` object contains a `LonMarkObjects` property, which you can use to access the collection of LonMark Functional Blocks defined on that `Interface`. If the interface supports dynamic LonMark Functional Blocks, you can add and remove `LonMarkObject` objects from the collection using the `Add()` and `Remove()` methods. In addition, you can assign and un-assign existing network variables to and from the `LonMarkObject` objects using the `AssignNetworkVariable()` and `UnassignNetworkVariable()` methods. You can check if an `Interface` object supports the addition of dynamic `LonMarkObject` objects by reading the `Interface` object's `DynamicLonMarkObjectCapacity` property. You can check if an `Interface` object supports the addition of dynamic network variables by reading the `Interface` object's `MaxNvSupported` and `StaticNvCount` properties.

You can also access the collection of network variables defined on an `Interface` or on a `LonMarkObject` through the object's `NetworkVariables` property. This property contains the entire set of `NetworkVariable` objects defined on that `Interface`. You can use the `Add()` and `Remove()` methods to modify this collection on your custom `Interface` objects as you desire.

In LNS Turbo Edition, each custom `Interface` object also contains a `DynamicMessageTags` collection. You can use this collection to add dynamic `MessageTag` objects to the interface. This allows you to add message tags to any device that supports monitor sets, and use those message tags to send explicit messages from that device to a group of devices, as with static message tags. For example, consider the case of a Network Service Device. Network Service Devices do not contain static message tags. However, you can now add dynamic message tags to the `AppDevice` object associated with a `NetworkServiceDevice`. Once you have added to a message tag to a Network Service Device, you can connect the message tag to the devices you want to send the messages to. Following that, you can create a message monitor point on the Network

Service Device that specifies the new dynamic message tag as the monitor target. Then, open the monitor set and use the message monitor point to send messages from the Network Service Device to the devices bound to the message tag. Note that you can still use message monitor points to send messages to individual application device, as described in Chapter 9 of this document.

To help you keep track of changes related to these new dynamic interface capabilities, the `OnNodeIntfChangeEvent`, which is fired each time a device's external interface is changed, has been modified for Turbo Edition. The `OnNodeIntfChangeEvent` event now returns several new values indicating that interface changes related to these new features have been made. For example, once an application registers for this event, it will be fired each time a network variable is added or removed from a `LonMarkObject`, or each time a `LonMarkObject` is added to an interface.

For a general overview of device interfaces, see *Device Interfaces* on page 104. For details on how you can use the dynamic device interface features described in this section, see *Using Dynamic Device Interfaces* on page 179.

Improved Monitoring Performance

As of LNS Turbo Edition, there are two separate types of `MonitorSet` objects: permanent `MonitorSet` objects, which can be used in multiple client sessions, and temporary `MonitorSet` objects, which can only be used in a single client session by the application that created them.

If you need monitor points that will only be used in a single client session, you should use temporary `MonitorSet` objects, as it takes less time and database resources to create them. If you are creating a group of monitor points that you need to use in multiple client sessions, you must use the permanent `MonitorSet` objects.

This section describes the major differences between permanent and temporary monitor sets.

Using Permanent Monitor Sets

Each `Network` object contains a `MyVNI` property, which returns an `AppDevice` object representing your client's Network Service Device on the network. You can use this `AppDevice` to access all the permanent `MonitorSet` objects that are stored in the LNS database for your client's Network Service Device. Echelon recommends that you only use the `MyVni` property to access `MonitorSet` objects when creating permanent `MonitorSet` objects, or when modifying the configuration of those `MonitorSet` objects. For actual monitor and control operations with permanent monitor sets, use the `CurrentMonitorSets` property of the `Network` object.

The `CurrentMonitorSets` property returns a collection of all the permanent `MonitorSet` objects on a network that are currently stored in your client's Network Service Device. This is useful if you have created monitor sets while the system management mode is set to `lcaMgmtModeDeferConfigUpdates` (note that prior to LNS Turbo Edition, this was `lcaOffNet`). Although those monitor sets exist in the LNS database and can be accessed through the `MyVni` collection mentioned in the previous paragraph, they will not be commissioned into the Network Service Device, and cannot be used for monitoring operations, until the system management mode is set to

lcaMgmtModePropagateConfigUpdates (note that prior to LNS Turbo Edition, this was lcaOnNet) and the Network Service Device is updated. The collection accessed through the CurrentMonitorSets property allows access to all the monitor sets in your Network Service Device that you can currently open and enable on a network (the collection accessed through the MyVni property allows access to these monitor sets, as well as those in the LNS database that have not yet been commissioned into your client PC's Network Service Device). You can use all the monitor sets obtained through the CurrentMonitorSets property as runtime monitor sets, meaning that you can enable them and use them for monitoring operations. However, persistent changes to their configuration are not allowed when accessed through this collection. As noted previously, you should use the collection obtained through the MyVni property when you need to change the persistent configuration of your client's local MonitorSet objects.

NOTE: You can access the CurrentMonitorSets collection when running in independent mode (i.e. without connection to the LNS Server PC).

Using Temporary Monitor Sets

If you need monitor points that will only be used in a single client session, you should use temporary MonitorSet objects. As of LNS Turbo Edition, you can create a temporary monitor set with the CreateTemporaryMonitorSet() method of the Network object.

The properties and methods that can be used on a temporary MonitorSet object and the monitor points it contains are generally the same as those that can be used on a permanent MonitorSet object and its monitor points. However, if you have been using permanent MonitorSet objects with LNS 3.0, you should note a few exceptions to this rule.

Temporary MonitorSet objects cannot be created or used while in independent mode. The Open() and Close() methods have no effect on temporary MonitorSet objects, because temporary MonitorSet objects are opened as soon as they are created, and closed when they are released, or when the client session in which they were created ends. You should also note that temporary monitor sets are not enabled as they are opened. You must explicitly enable temporary monitor sets and temporary monitor points with your application using the applicable Enable method. For this purpose, the MsgMonitorPoint object now includes an Enable method.

In addition, the DefaultOptions properties stored in MsgMonitorPoint and NvMonitorPoint objects in temporary monitor sets are read-only. The values applied to these properties are taken from the temporary monitor set's MsgOptions or NvOptions properties.

Monitor points in temporary monitor sets do not support the use of connection description templates to define certain monitoring options, as monitor points in permanent monitor sets do. As a result, you must set the connDesc element to NULL when you use the Add() method to add a message monitor point or network variable monitor point to a temporary monitor set.

There is one other variance to note when you use temporary MonitorSet objects. Network variable monitor points in temporary monitor sets cannot be automatically bound to the monitoring node. As a result, the UseBoundUpdates property of all temporary monitor sets and monitor points can only be set to False, meaning that you cannot use LNS to implicitly connect network variables monitored by temporary monitor

points to the host PC. However, you can still connect network variables to your Network Service Device using explicit connections, and utilize these with the `SuppressPollingIfBound` property.

For more information on monitor and control in this document, see Chapter 9.

Availability of Network Resource Information

The `NetworkResources` property has been added to the `System` object for Turbo Edition. This property returns a `NetworkResources` object that provides access to important network resource information for a system, including the number of `AppDevices` and `Routers` that have been installed on the system, the number of exclusive and sharable network variable selectors available on the system, and the number of subnets and group IDs allocated on the system.

This information is useful if you are managing a large system. For example, if you are writing an application that creates large numbers of multicast connections on a system, you will need to know how many groups and exclusive selector are available on the system. Or, if you are merging two LNS databases, you will need to know how many subnets, groups, and exclusive selectors have been assigned in each database, to make sure that the merged database will not exceed the limits for each resource.

For more information on the `NetworkResources` object and its properties, see the *LNS Object Server Reference* section of the LNS Application Developer's Kit help file.

Enhanced LonMark Interoperability

Several changes have been made to LNS Turbo Edition to comply with version 3.3 of the *LonMark Application-Layer Interoperability Guidelines*, and to better support the latest versions of the LonMark standard resource files.

When importing a device interface from an external interface file, previous versions of LNS would set the `Mode` property of all `LonMarkObject` objects defined in the device interface to one of two values. It would set the `Mode` property of a `LonMarkObject` object to 0 if the `LonMarkObject` object's `TypeIndex` property was in the range of standard Functional Profile Template (FPT) indices, or to 3 if the `LonMarkObject` object's `TypeIndex` property was in the range of user-defined FPT indices. Turbo Edition features automatic scope determination, which means LNS will now search the set of installed and cataloged resource files to find the most device-specific match for the FPT, and set the `LonMarkObject` object's `Mode` property based on this determination when the device interface is imported. If no match is found, LNS will set the `LonMarkObject` object's `Mode` property to `lcaResourceScopeUnknown`.

In addition, the `ResyncToResources()` method has been added to the `DeviceTemplate` object. You can use this method to re-synchronize a device template with modified or newly accessible device resource file information. This may be necessary if you are upgrading to the current version of the LonMark resource files, or if you imported a device's external interface file before the resource files for that device were available in the resource file catalog.

To complement this change, the `ResyncToTemplate()` method has been added to the `AppDevice` object. You can use this method at any time to re-synchronize the configuration of an `AppDevice` with the `DeviceTemplate` it is using. This may be

necessary if you have recently re-imported a device's external interface file, or if you have used the `ResyncToResources()` method to update the `DeviceTemplate` that the device is using.

There are several other changes that you may find useful when using the LonMark resource files. For example, the `TypeInherits` property has been added to the `ConfigProperty` object. This property indicates whether or not the configuration property inherited its type from the network variable that the configuration property applies to. If a configuration property inherits its type, you may need to program your application to account for changes to the configuration property's type. For example, consider a configuration property with the `TypeInherits` property set to `True`. If the configuration property applies to a network variable, and an application changes the network variable's type, then LNS would change the data display format of the configuration property automatically, since the `TypeInherits` property is set to `True`. You would need to know about this change when reading the value of the configuration property, and you can use this property to keep track of which configuration properties could be modified by LNS in this fashion.

In addition, the `IndexToSNVT` property has been added to the `LonMarkAlarm` object. The `IndexToSNVT` property contains the device index number of the network variable on the `LonMarkObject` that caused the current alarm condition (i.e. the condition summarized by the `LonMarkAlarm` object). This will allow you to quickly identify all alarm conditions that occur on your network. Note that the `LonMarkAlarm` object supports alarming on devices that implement their alarms through `SNVT_alarm` network variables, but not through the more recent `SNVT_alarm2` type. However, `SNVT_alarm2` type network variables can be monitored and controlled using standard monitor and control techniques.

Finally, the `OnLonMarkObjectStatusChangeEvent` event has been added to LNS to allow you to track when an LNS application on your system changes the status of a `LonMarkObject` object.

Consult the *LNS Object Server Reference* help file for more information on the properties and methods introduced in this section. For general information on device resource files and device interfaces, see *Device Interfaces* on page 104

Improved Device Commissioning Performance

When you commission a device using an LNS version prior to Turbo Edition, LNS validates that the device is on the intended channel, and that it is using a program interface consistent with the definition in the LNS database. This validation is provided to catch simple errors that might otherwise cause serious system problems. However, performing this validation requires the transmission of several messages, which may significantly increase the time required to commission a slow network with simple nodes. For systems whose content is extremely well controlled, the benefits of validation may not be worth the extra time the validation process incurs.

For this reason, the `DeviceValidation` property has been added to the `DeviceTemplate` object. You can use this property to determine what validation steps LNS will perform when commissioning devices that use that `DeviceTemplate`. This may be useful if you are commissioning a large number of devices, and are confident that the devices contain the correct program information and are installed on the correct

channel. In this case, you can modify the DeviceValidation property to bypass parts of the validation process, and reduce the time it takes to commission those devices.

You should be aware that if parts of the validation process are disabled, the risk of network configuration problems due to inconsistent device settings will increase. If you disable the validation process, you should be sure that the Channel and ProgramId properties of the AppDevice objects using the device template have valid settings before commissioning, upgrading or replacing those devices. Skipping this validation, and then commissioning a device with the wrong interface may make subsequent communication with the device impossible. Note that you can also use this property to force a validation check of a device, by setting the DeviceValidation property to perform all validation steps, and then re-commissioning the device.

In addition, the SelfDocConsistency property has been added to the DeviceTemplate object. This property determines how much LNS will assume about the self-documentation of devices that are using the device template. This affects how LNS will read the self-documentation data of those devices, and what level of program interface validation LNS will perform when commissioning those devices. Some settings of this property allow LNS to assume higher degrees of self-documentation consistency among devices using this template, and cause LNS to operate more efficiently when updating those devices during the commissioning process. However, these settings may cause problems if the device developer has produced multiple devices that have the same program ID, but use different self-documentation strings or formats, as version 3.3 of the *LonMark Application-Layer Interoperability Guidelines* allows.

For more information on device commissioning, and on the features described in this section, see *Commissioning Devices* on page 121.

System Management Mode Enhancements

Several changes have been made to enhance the LNS operation when the system management mode (MgmtMode property) is set to lcaMgmtModeDeferConfigUpdates. Before considering these changes, you should note that new names for the system management mode settings have been provided in Turbo Edition. The old names still exist in LNS for compatibility reasons, but the documentation refers only to the new names. The new lcaMgmtModeDeferConfigUpdates value maps to the old lcaOffNet value, and the new lcaMgmtModePropagateConfigUpdates value maps to the old lcaOnNet value.

When the system management mode is set to lcaMgmtModeDeferConfigUpdates, all network configuration changes caused by your application will be applied to the LNS database, but not to the physical devices on the network. Prior to LNS Turbo Edition, these changes would be queued and then applied to all the physical devices as soon as the network management was set back to lcaMgmtModePropagateConfigUpdates. However, as of LNS Turbo Edition, you can call the new PropagateDeviceConfigUpdates() method on a device to apply configuration changes that affect only that device (such as configuration property values) to the device without changing the system management mode back to lcaMgmtModePropagateConfigUpdates. This may be useful if you have configuration changes pending for a large number of devices, and only want to apply them to a subset of those devices. This method will not propagate updates that affect multiple devices, such as changes to addresses or connections.

LNS Turbo Edition also includes a new option (`lcaReplaceFlagPropagateUpdates`) for the `ReplaceEx()` method, which you can use to replace an `AppDevice` or `Router`. If you call the `Replace()` method, or if you call the `ReplaceEx()` method without specifying the `lcaReplaceFlagPropagateUpdates` option, on a device while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, the physical devices will not be updated with configuration changes caused by the replace process until the system management mode is set back to `lcaMgmtModePropagateConfigUpdates`. You can now use the `ReplaceEx()` method with the `lcaReplaceFlagPropagateUpdates` option to replace devices while the system management mode is still set to `lcaMgmtModeDeferConfigUpdates`. This may be useful if there are configuration changes pending for a large number of devices, and you want to replace a device without waiting for all of those changes to be applied. A similar option is also provided for use with the new `CommissionEx()` method, which you can use to re-commission a device while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`.

Finally the `OnSystemMgmtModeChangeEvent` event has been added to the `ObjectServer` object. Once an application registers for this event, it will be fired each time the system management mode changes.

For more information on the features described in this section, see Chapter 6, *Defining, Commissioning and Connecting Devices*.

Enhanced Configuration Property Management

Several properties have been added to the `ConfigProperty` object that will allow your application to better identify the source and configuration of a `ConfigProperty` object. For example, in some cases the value stored in the LNS database for a configuration property may not match the actual value of the configuration property in the physical device on the network. To help you make this determination, the `ValueStatus` has been added to the `ConfigProperty` object. The `ValueStatus` property indicates whether or not the value currently stored in the LNS database for a configuration property matches the value stored in the physical device on the network.

Two additional properties, the `DeviceSpecificAttribute` and `ConstantAttribute` properties, have been added to the `ConfigProperty` object. These properties allow you to read and write the device-specific and constant attributes of the configuration property with your LNS application.

For more information on these configuration property attributes, see the *Configuration Property Flags* section of version 3.3 of the *LonMark Application-Layer Interoperability Guidelines*. For more information on configuration properties in this document, and for information on how you might want to use configuration properties within an LNS application, see *Using Configuration Properties In a Monitor and Control Application* on page 233.

Online Database Validation and Backup

LNS Turbo Edition features two new ways to help you maintain your LNS network databases: online backup and database validation. You can perform database validations using the LNS Database Validation Tool, or by calling the `Validate()` method on a `Network` object. When you perform a database validation, LNS will process the contents

of an entire network database, and report any errors or inconsistencies it discovers. It can optionally repair some of these errors during the validation process. Inconsistencies and errors that may be discovered during the database validation procedure include orphan objects (inaccessible objects in the database), broken interfaces, and duplicate objects.

You can access the LNS Database Validation Tool by selecting **LNS Database Validator** from the Echelon LNS Utilities group in the Windows Programs menu. Consult the online help for the LNS Database Validation Tool for more detailed information on how to use the utility.

You can also perform a database validation with your own LNS application by calling the `Validate()` method on a network. LNS will then start a transaction and validate the network database. When the validation is complete, the method returns a `DatabaseValidationReport` object containing information summarizing the results of the validation. This is the same set of information that would be provided by the LNS Database Validation Tool if you used it to perform a validation on the same database, including descriptions of all the errors discovered, all the errors that were repaired, and all the errors that were not repaired.

While the ability of the new LNS database validation feature can be used to repair many of the errors that might occur in your network databases, you should not consider it a replacement for consistent, regular back-ups of your network databases. To facilitate database back-ups, the `Backup()` method has been added to the `Network` object. You can call this method on any local network to make a backup copy of the network database, and export the backup copy to a local folder of your choice. This method is quite useful, as you can use it to back up a network database while the network and system are open, and while clients are attached. Echelon recommends that you use this feature in conjunction with the database validation feature to perform regular, validated backups of your network databases.

For more information on the database validation feature, see *Validating Network Databases* on page 252. For more information on the online backup feature, see *Backing Up Network Databases* on page 251.

Miscellaneous

LNS Turbo Edition features several other changes you may find useful as you develop your applications. Consult the *LNS Object Server Reference* help file for more information on these features.

- The `RemoveEx()` method has been added to the `Networks` collections. The `RemoveEx()` method allows you to specify whether the network database is to be deleted when the network is removed from the collection. If you do not delete the network database, you can then restore the network later, without having to re-create the database. This may be useful if you want to store network databases on a central file server and add them to (or remove them from) any LNS Server PC on demand.
- The `CopyWithParent` property has been added to the `Extension` object. An `Extension` object represents user-defined data. You can use the `CopyWithParent` property to indicate whether or not the `Extension` object should be copied when its parent object is copied. This may be useful if you are writing an application that is copying an object containing a large `Extensions`

collection. You may not want the application to copy every extension record stored in the object. You can use this property to mark which extension records the application should copy.

- The `ConstNodeConnChangeEvent` event now returns additional values. Previously, the possible values that could be returned for the event's `ObjectChangeType` element were contained in the `ConstObjectChangeTypes` constant. Those values are now stored in the `ConstNodeConnChangeType` constant, which includes more detailed information. The values of the `ConstNodeConnChangeTypes` constant now map to the values of the `ConstObjectChangeTypes` constant, so this change will not affect applications that were written using pre-Turbo Edition versions of LNS.
- The `ItemByProgrammaticName()` method has been added to the `LonMarkObjects` and `NetworkVariables` collections. This allows you to retrieve a `LonMarkObject` or `NetworkVariable` from its respective collection by its programmatic name.
- The `AliasUseCount` and `AliasCapacity` properties have been added to the `AppDevice` object. These properties indicate how many aliases are being used on a device, and how many are available, respectively.
- You can now use the `OnSessionChangeEvent` event to keep track of when the Network Service Device is offline. You will need to know this because polling is suspended and monitor and control events will not be delivered to your application while the Network Service Device is offline. Once your application registers for this event, is will be fired each time the state of your client's Network Service Device changes from the online state to the offline state, or vice versa.

New LNS Runtime Installations

The LNS Server and LNS Remote Client installations, which are created and installed with the LNS Redistribution Kit, have been re-created as Windows Installer (a.k.a. Microsoft Installer) installations in order to make LNS Turbo Edition installations more compatible with recent versions of Windows. This has important implications for how the LNS redistribution installations should be installed. For more information, please see Chapter 14, *Distributing LNS Applications*.

Compatibility

Echelon's goal is to allow easy integration of new LNS releases into your LNS applications. COM interface compatibility allows for the possibility of "drop in" replacement of one LNS release with another, and Echelon strives to design new features and fix defects in a way that supports this. The following sections provide details on several LNS compatibility issues you should consider when upgrading to LNS Turbo Edition. Some of these issues are applicable to all LNS releases, and some apply specifically to LNS Turbo Edition.

Interface Compatibility

LNS follows the Microsoft COM interface guidelines for maintaining interface compatibility. Methods and properties can be added to the LNS OCX interface, but existing interfaces cannot be modified. Automated comparison tools are run on the LNS interface at each release to verify that the COM interface rules are followed.

Because COM methods and properties may not be modified or removed from an interface, an established interface can become cluttered with obsolete and duplicate methods and properties. As a result, Echelon has introduced the concept of a deprecated interface to provide better guidance for optimal use of LNS Turbo Edition. Deprecated properties, methods and objects are marked as such in the *LNS Object Server Reference* help file. This help file also lists which version each property, method and object was introduced in, to assist the development of applications that work with multiple versions of LNS. Note that in the *LNS Object Server Reference* help file, Turbo Edition is referred to as Release 3.20, and all properties, methods and objects marked as added to LNS in Release 3.20 are new in Turbo Edition.

For a list of deprecated interfaces in Turbo Edition, see Appendix A, *Deprecated API and Obsolete Files*. Remember that deprecated interfaces are generally deprecated because they do not provide the complete functionality that alternative interfaces provide. However, most of these interfaces are still included in the LNS Object Server for backward compatibility reasons. Deprecated features that are no longer implemented or supported are marked as such in Appendix A.

Database

LNS uses the FastObjects database engine to store its objects. When new features are added, the database schema must change to support them. The result is that LNS databases are not usually backward compatible. An LNS Turbo Edition database will not be accessible by any application using LNS 3.0, or versions of LNS prior to that.

However, LNS databases are always forward compatible. When an application using LNS Turbo Edition opens an LNS 3.0 database, the database will be automatically converted to LNS Turbo Edition format in a potentially time-consuming process. Since database conversion is a one-way process, Echelon recommends that backup databases be kept as long as there is any question about rolling back to a previous LNS Server runtime. Note that you cannot use the new `Backup()` method to do so, as this will upgrade the database to Turbo Edition format.

LNS Service Pack releases, starting with LNS 3.04, have not modified the database schema, and Echelon will continue to follow this discipline in future Service Packs. If a Service Pack must modify the schema in order to fix a serious problem, the Service Pack read-me file will note that fact prominently.

Runtime Component Updates

In general, LNS Service Packs may not be uninstalled. Patching technology, including Windows Installer patches, does not generally support uninstallation of patch updates. Since LNS databases are usually not backward compatible, the updated LNS Turbo Edition installation will update the LNS global database and individual network databases as they are opened, further complicating any attempt to return to a previous version of LNS.

LNS Turbo Edition servers and clients do not interoperate with clients and servers of other LNS versions. When updating an LNS site with a central server and remote clients, the LNS Server PC must be updated first. LNS server-independent mode, introduced in LNS 3.0, will allow any remote clients that run only in server-independent mode to be stopped, updated, and restarted some time after the server is updated. Remote clients that do not operate in server-independent mode should be updated when the LNS Server PC is. Live updates of LNS servers or clients are not supported, and server-independent mode is the only LNS client mode that does not require a constant client-server connection, so this is the only distributed update scenario that Echelon supports. For more information on server-independent mode, see *Independent Clients* on page 41.

Application Developer's Kit Include Files

The LNS Turbo Edition Application Developer's Kit has changed from previous LNS versions in that it does not install any include files or "class wrapper" files for using the OCX with C++ projects. There were two problems with these files:

- Only C++ was supported. None of the other supported development environments, such as Visual Basic 6.0, could use the C++ include files.
- With the same definitions in two places (the include files and on the OCX), the definitions sometimes became inconsistent.

The recommended solution in this and all subsequent releases is to provide all of the necessary constants on the OCX interface, accessible to C++ users through the `#import "lcaobjsv.ocx" named_guids rename_namespace("lca")` directive. For more information on this, see Chapter 4. As an alternative, you could generate the old class wrapper files within your project. For more information on this, see Appendix B, *LNS, MFC and ATL*.

Remember that if you are not using C++ to develop your LNS application, you can continue accessing the OCX as in your previous version of LNS.

Exception Codes

As new features are added or defects fixed, some existing LNS features will add new exception codes to provide more useful information about new or modified failure cases.

For example, prior to Turbo Edition, LNS threw the SRSTS_NEURON_VERSION_MISMATCH exception (from the obsolete include file ns_srsts.h) when an attempt to load an application image into a device failed because the Neuron model number or the system image version number of the device did not match that of the linked application. LNS will now distinguish these two exception cases. The following new exception codes have been defined:

`lcaErrNsNeuronModelMismatch` "Incompatible Neuron model number."

`lcaErrNsFirmwareVersionMismatch` "Incompatible firmware version number."

The `lcaErrNsFirmwareVersionMismatch` value maps to the (now obsolete) SRSTS_NEURON_VERSION_MISMATCH code, since it is more likely that the system image version number would differ than the neuron model number.

Common exception codes for particular methods can and should be handled specially to provide more guidance to your particular end-user. However, because the complete set of exception codes that can be thrown for a given method can be large, and because all exception codes must be handled by your application, you should also use a generic error handler in your application that will handle all errors returned by LNS.

New Features

When adding new features to LNS, Echelon must make compatibility tradeoffs in order to improve functionality and performance. Some of the new features described earlier in this chapter may introduce compatibility issues into your application, so you should consider the following information carefully when integrating these new features.

Propagating Device Changes While Offnet

The commission and replace operations behave identically in LNS Turbo Edition as in prior versions of LNS, unless the new "propagate" flags provided with the `ReplaceEx()` and `CommissionEx()` methods are specified by the LNS application.

The new `AllowPropagateModeDuringRemoteOpen` property is simply an alias for the `RemoteIgnorePendingUpdate` property. The `RemoteIgnorePendingUpdate` property is marked as hidden, but reading and writing the property is still supported by LNS Turbo Edition.

Opening a remote Full client application with LNS Turbo Edition may succeed while the system management mode is set `lcaMgmtModeDeferConfigUpdates`, even if there are pending updates for other devices, and the `RemoteIgnorePendingUpdate` flag is `False`. The Network Service Device will be re-commissioned in this case. In prior versions of LNS, the open would fail, and return an error.

Opening a remote Full client application with LNS Turbo Edition while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, and the `AllowPropagateModeDuringRemoteOpen` property is set to `True`, may result in commissioning the remote Network Service Device without propagating pending updates to other devices. In prior versions of LNS, this would always result in propagating all pending changes.

Dynamic Functional Blocks

Your application might currently assume that it will find all defined `LonMarkObject` objects for a device by accessing the `DeviceTemplate` object's interface. However, the dynamic `LonMarkObject` objects on a device do not appear in the interface of the device's `DeviceTemplate` object. Another possible problem is if your application assumes that the collection index assigned to a `LonMarkObject` is one greater than the value of the `LonMarkObject` object's `Index` property. This relationship happens to hold for static `LonMarkObject` objects, but may not for dynamic `LonMarkObject` objects.

DataPoint Object Improvements

The `DataPoint` object was introduced in LNS 3.0 to allow convenient access to the data contained in monitor points. The feature allowing access to individual `DataPoint` fields contained several defects that may not allow a current implementation of the feature to

be compatible with a previous implementation. However, in this release, the `DataPoint` object provides new functionality – including consistent field-by-field formatting between network variables and configuration properties, and client-specific formatting – that is the new standard for LNS data formatting.

Formatting Enhancements

The default Formatting Locale has been changed in this release from the System Default Locale to the User Default Locale. In recent versions of Windows, it has become more and more difficult to set System Default Locale settings, and Echelon made this change in order to make current applications easier to use. In most systems, the System and User Default Locales are the same, and this change should not cause a noticeable difference in operation. Consult the MSDN documentation of the Win32 `GetLocaleInfo()` function for more information on the System and User Default Locales, as LNS uses this function to retrieve the Windows locale settings.

Enhanced LonMark Interoperability

Under some conditions, LNS may use the new automatic scope determination feature to set the `Mode` property of a `LonMarkObject` object to values that were not previously set automatically. In previous versions of LNS, the `Mode` property of each `LonMarkObject` object would only be automatically set to 0 or 3. In Turbo Edition, the `Mode` property of each `LonMarkObject` object may be automatically set to any of the values of the `ConstResourceScope` constant when an external interface file is imported. For more information on this, see the help pages for the `LonMarkObject` object's `Mode` property and the `ConstResourceScope` constant in the *LNS Object Server Reference* help file.

LonWorks Interfaces Control Panel

The LONWORKS Interfaces Control Panel Applet was introduced in LNS 3.07 to support the *i*.LON 10 Ethernet Adapter and the *i*.LON 100 Internet Server. In this release, the LONWORKS /IP Channels Control Panel application introduced in LNS 3.0 has been merged with the LONWORKS Interfaces Control Panel Applet to provide this similar functionality through a single utility. If your user documentation provided directions for using the LONWORKS /IP Channels Control Panel, it should be updated to direct them to the new, merged LONWORKS Interfaces application in the Windows Control Panel.

Support for *i*.LON 1000, *i*.LON 600 and ANSI/CEA-852 Channels

LONWORKS /IP channels in LNS Turbo Edition require the use of the new Echelon LONWORKS/IP Configuration Server. This means that if a site using an *i*.LON 1000 Internet Server upgrades from LNS 3.0 to LNS Turbo Edition, they must also upgrade the *i*.LON 1000 Configuration Server. To mitigate this requirement, the LNS Server Turbo Edition installation installs the Echelon LONWORKS/IP Configuration Server. LNS Turbo Edition and the new Echelon LONWORKS/IP Configuration Server support backward compatibility with the *i*.LON 1000 Internet Server by allowing the creation of, and connection to, LonWorks/IP channels that are compatible with the *i*.LON 1000 Internet Server. If the Echelon LONWORKS/IP Configuration Server is not running on the same PC as the LNS Server installation, the Echelon LONWORKS/IP Configuration

Server installer will be available to LNS customers on the Echelon web-site, and on the LNS Application Developer's Kit CD for redistribution by licensed LNS developers.

For more information on the *i*.LON 1000 Internet Server and the *i*.LON 600 LONWORKS/IP Server in this document, see Chapter 11, *LNS Network Interfaces*.

Flexible Program ID

By default, LonMark devices with a given program ID are expected to have the same self-documentation data format. LNS has never verified that the number and type of LonMark Functional Blocks on two devices sharing a program ID match. However, such inconsistencies would normally be detected in previous releases because LNS checked the self-documentation string length, and did a spot check on the node self-documentation null terminator. In LNS Turbo Edition, LonMark devices are allowed to have different self-documentation strings by default. Therefore, LNS will no longer detect many inconsistencies in LonMark self-documentation data by default.

Modifiable Device-Specific Configuration Properties

There are two main behavioral differences an LNS application might observe when running LNS Turbo Edition instead of a previous LNS version, due to the changes made to support modifiable device-specific configuration properties.

The way LNS reads a modifiable device-specific configuration property with the `ConfigProperty` object's `GetElement()` or `GetRawValues()` methods, or the `Value` or `RawValue` properties, has changed. Prior versions of LNS did not recognize modifiable device-specific configuration properties as any different than "normal" configuration properties, and these properties and methods would return data from the LNS database (if the value in the database was known). Therefore no contact with the device was necessary.

In LNS Turbo Edition, these methods and properties read the data directly from the device. As a result, reading these values in LNS Turbo Edition is likely to take longer than in previous releases, and may return different data values. If the value cannot be read from the device, LNS will return an error, whereas in previous releases it would not. This is consistent with the treatment of constant device-specific configuration properties, and is the same as if the value in the database was not present. If the value is not known, and the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, prior versions of LNS would return an exception, whereas LNS Turbo Edition will attempt to read the value from the device.

In prior versions of LNS, constant device-specific configuration properties would never be set in the database. As of LNS Turbo Edition, operations such as calling the `DownloadConfigurationProperties()` method with the `lcaConfigPropOptSetDefaults` option set will cause constant device-specific configuration property values to be stored in the database, as well as other configuration property values.

Likewise, calling the `UploadConfigurationProperties()` method may store a constant device-specific configuration property value in the database. Using the new operation to read configuration property values from the database will allow an application to see the database copy of the constant device-specific configuration property. However, none of the methods or properties available in previous releases will read the database version of a constant device-specific configuration property. Under no

circumstances will LNS write a constant device-specific configuration property to a device unless the configuration property's `DeviceSpecificAttribute` property is set to `False`.

Changeable Network Variable Types

This feature has been supported to some extent for several LNS releases. Version 3.3 of the *LonMark Application-Layer Interoperability Guidelines* introduced a new, standard mechanism for supporting network variables with changeable types and sizes. The version 11 LONMARK standard resource files include a new SCPTnvType configuration property type that a network manager can use to inform an application device of a change to a network variable type and size. The version 11 resource files also include a new SCPTmaxNVLength configuration property type, to be implemented as a constant configuration property, to document the maximum legal length of a given network variable.

Compatibility Case 1 – Writing Compatible Network Management Tools and Plug-Ins

Some network management tools use the SCPtnvType configuration property to change the type of a network variable. They set the value of the configuration property to match the new type they want the network variable to use, and they also set the value of the `SnvtId` property in the LNS database. This use-case demonstrates how a plug-in that was designed to manage devices using SCPTnvType configuration properties with previous versions of LNS will work using LNS Turbo Edition.

A) Tools that set the SCPTnvType configuration property first should follow these steps:

1. Get the SCPTnvType configuration property for the network variable.
2. Change its value. When using LNS Turbo Edition this will automatically set the network variable's `SnvtId` property to the correct value (0 for UNVTs, >0 for SNVTs).
3. Set the value of the `SnvtId` property in the LNS database to match the type chosen in step 2 (0 for UNVTs, >0 for SNVTs). When using LNS Turbo Edition this step has no effect, since the `SnvtId` property was already modified by the previous step.

B) Tools that set the `SnvtId` property first should follow these steps:

1. Set the `SnvtId` property in the LNS database. When using LNS Turbo Edition, if the `SnvtId` represents a standard type (non-zero), LNS will automatically set the SCPTnvType configuration property for the network variable.
2. Get the SCPTnvType configuration property for the network variable.
3. Change its value. When using LNS Turbo Edition, this step has no effect if the `SnvtId` property was set to a standard type in step 1, since the configuration property value would have already been set at that point.

For either scenario, the same end results will apply. The network variable's type will be successfully changed in the LNS database, an event will be generated to indicate this,

and the value of the SCPTnvType configuration property will be updated to inform the device of the network variable type.

For ideal performance, an application using versions of LNS prior to LNS Turbo Edition should modify the `SnvtId` property first, and then the SCPTnvType configuration property. The operations should be performed in the same transaction. However, applications using LNS Turbo Edition and later should use the `TypeSpec` property for all network variable type changes, since this method supports user-defined types and standard types, is more efficient, and does not require the application to explicitly manage the SCPTnvType configuration properties.

Compatibility Case 2 – Devices That Support Old and New Style NV Type Management

Consider a device that supports both the old SI/SD method of network variable type modification, and can read the SCPTnvType configuration property. Such an application should:

1. Define the default value of the network variable with its type_category set to NVT_CAT_NUL.
2. Honor the SI/SD value if the configuration property's type_category is NVT_CAT_NUL (i.e. use the SNVT_ID stored on the device). Otherwise, honor the value of the configuration property.

First, consider the device being managed by an application using an LNS version prior to Turbo Edition that does not update the SCPTnvType configuration property. Changes to the network variable type will result in updating the SI/SD data on the device. The application device will then use the SI/SD information because the value of the SCPTnvType configuration property value will remain invalid.

Now consider how an upgrade to Turbo Edition will affect this application. LNS will recognize that the configuration property value is invalid (its type_category set to NVT_CAT_NUL) or unknown, so LNS will continue to use the SI/SD to inform the device of the network variable type. A replace or recommission operation on the device will set the SI/SD information to be updated, based on the type assigned to the network variable in the LNS database. Changing the network variable type to a standard type in the LNS database will cause LNS to update the SCPTnvType configuration property automatically and to stop updating the SI/SD data for the network variable. The application will honor the configuration property. Changing the network variable type to 0 (indicating a non-standard type) will cause LNS to update the SCPTnvType configuration property with a category of NVT_CAT_NUL, and set the SI/SD value in the physical device to 0.

Next, consider a device being managed by an application that uses the SCPTnvType configuration property properly, using an LNS version prior to Turbo Edition. In this case, the application device honors the value of the SCPTnvType configuration property, since it is valid. When upgrading to LNS Turbo Edition, the LNS database conversion will recognize that the configuration property is valid, and will update the network variable type based on that value. A subsequent replace operation will not update the SI/SD data for that network variable.

Security

New security features have been added to LNS since Release 3.0 to protect against unwanted access to sensitive data. In LNS 3.07, Echelon introduced the OpenLDV xDriver to support a new class of remote network interfaces, including the *i*.LON 10 Ethernet Adapter and the *i*.LON 100 Internet Server. The TCP/IP link to those new interfaces includes MD5 authentication and sequence numbering to prevent unauthorized access or replay attacks on the link. In LNS Turbo Edition, RC4 encryption is also used when passing LonWorks authentication keys over the link.

For more information on the OpenLDV xDriver, see the *OpenLDV Programmer's Guide, xDriver Supplement*. This document can be downloaded from Echelon's website at:

http://www.echelon.com/support/documentation/default.htm

Chapter 3 - LNS Overview

This chapter provides an overview of the LNS network operating system. It introduces the various LNS components, and describes the fundamental network services provided by LNS. For an introduction to LONWORKS networks, see the *Introduction to the* LONWORKS *System* document, which is available for download from Echelon's web site at http://www.echelon.com/.

Introduction to LNS

LONWORKS Network Services (LNS) is the control networking industry's first multi-client network operating system. Much like a standard operating system, which implements the fundamental operating tasks of a computer, the LNS network operating system encapsulates common LONWORKS network operations, providing the essential directory, installation, management, monitoring, and controlling services required by most network applications. In addition, LNS provides a standard interface that enables multiple network applications from multiple vendors to interoperate.

The LNS architecture combines the power of client-server architecture with object-oriented, component-based software design. LNS also incorporates Internet Protocol (IP) support for remote applications, and it works with the most common development platforms, including rapid application development (RAD) tools, thus offering the fastest way to bring network control on-line with all your other information systems. With LNS, multiple system integrators, managers, and maintenance personnel can simultaneously access network and application management services and data from separate client tools.

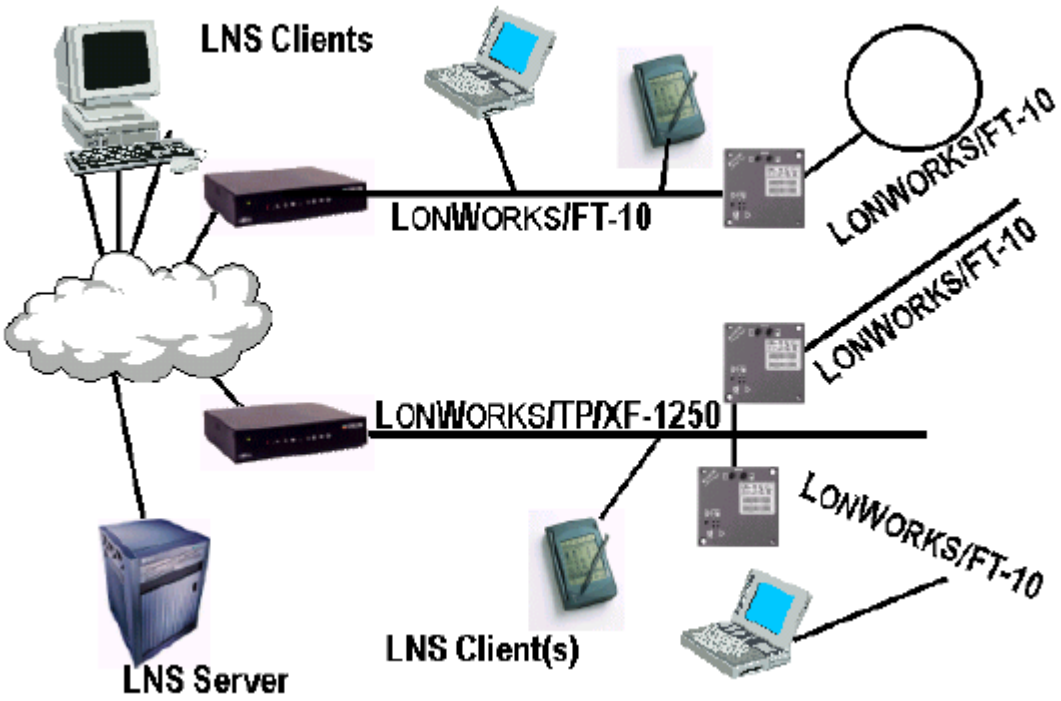


Figure 3.1 Sample Network Managed by LNS

The LNS network operating system provides the following benefits:

- *Reduced commissioning time and cost.* With LNS, multiple installers can work on the same system at the same time, without conflicts. Each tool is a client to the LNS Server, allowing multiple installers to work in parallel without database synchronization problems.

  Since remote client tools do not need to contain a network database, they can

be anything from a remote monitoring station, to an installation laptop PC using wireless networking. Each client tool can have a different user interface, optimized to the particular network being managed (e.g. material handling, access control, gas analysis, or HVAC) or to the skill set of the user. By building application-specific knowledge into network tools, all or part of the commissioning process can be automated, further reducing commissioning time and training cost.

- *Simplified system integration*. By defining the basic object framework as well as the higher-level component specifications, the LNS network operating system provides a basis for tools to interact and communicate with each other. Interoperable tools greatly simplify system integration, and the use of plug-in applications allows system integrators to add new features to their LONWORKS systems quickly and at low cost.
- *Easy customization.* By allowing tools to interoperate, LNS allows developers to create custom system-level or device-level tools that complement their systems or devices. For OEMs, this provides another way to add value to their systems by embedding application-specific knowledge into their tools. For integrators, system-to-system communication reduces the need to understand the implementation details within a given system. For end-users, this results in disparate systems that work together, leading to more efficient operation, higher yield, and greater comfort.
- *Greater access to data*. LNS provides users with the ability to deploy Human Machine Interface (HMI), Supervisory Control and Data Acquisition (SCADA), and data logging stations. Because of its client-server architecture, there are no databases to copy or redundant updates to make. Users no longer need to worry about tools losing synchronization with the network's configuration. LNS tracks the requirements of each tool, and automatically informs them of configuration changes.
- *Increased system up-time*. With LNS, repair technicians can plug tools into the network at any point, and access all network services and data. Since multiple tools can interoperate on the same network, multiple technicians can diagnose problems and make repairs simultaneously, with no need to coordinate their actions or even to be aware of one another. By building application-specific knowledge into their tools, OEMs can further reduce system downtime by automating fault detection, isolation, reporting, and repair.
- *Transparent IP network communication*. LNS allows tools to access LONWORKS networks over TCP/IP links. Any workstation connected to an LNS Server can run LNS applications that operate like local tools. This allows users to easily integrate LNS-based networks with Internet-based applications to create powerful enterprise-wide solutions, as well as allowing for a high-speed connection using existing LAN infrastructure.

The LNS Programming Model

The LNS network operating system provides a compact, object-oriented programming model that reduces development time, host code space, and host processing requirements. LNS uses a hierarchy of objects that correspond to physical network devices and logical objects to represent a LONWORKS network. Each object provides a set of methods, properties, and events that implement the functionality and configuration of

the network device represented by the object. This is described in more detail later in the chapter.

LNS leverages Microsoft's COM and ActiveX technologies, the Windows standard for component-based software, to simplify the development of network applications for Windows hosts. Using LNS, developers are free to take advantage of the support for ActiveX and COM components built into Windows development tools.

COM interfaces are 32-bit, language-independent programmable objects that can be used with a variety of development tools, such as Microsoft Visual Basic and Visual C++. These tools make COM interface calls and property assignments look like native language calls and assignments. LNS applications can therefore be implemented without knowing the underlying ActiveX and COM mechanisms.

LNS Components

The LNS Programming Model is comprised of the following major components. These components are described in the following sections.

- *LNS Databases and the LNS Server*
- *LNS Object Server*
- *Network Service Devices*
- *Network Interfaces*

LNS Databases and the LNS Server

LNS uses two types of databases to store and maintain the configurations of your LONWORKS networks: the *LNS global database,* and a set of *LNS network databases*. These are high-performance disk-based databases with in-memory caching to optimize repeated access to data.

The PC containing the LNS global database acts as the LNS Server. The location of the global database is maintained in the Windows Registry, and can be accessed by reading the `DatabasePath` property of the LNS ActiveX Object Server Control. By default, the global database is stored in the `ObjectServer\GlobalDb` subfolder of the LONWORKS folder. The location of the global database will be set when LNS is installed and should never be changed, as LNS applications must access the same global database if they are to interoperate.

The global database contains the `Networks` collection, which is the group of LONWORKS networks that are being managed with the LNS Server. Each of these networks has its own network database. The network database contains the network and device configuration information for that particular LONWORKS network. The location of each network database is specified when a network is created, and stored in the global database. Each LNS network database also contains extension records, which are user-defined records for storing application data. For more information on this, see the online help for the `Extensions` object in the *LNS Object Server Reference* help file.

LNS applications can run as local client applications, meaning that they run on the same PC as the LNS Server and the global database. Or, they can run as remote client applications, meaning that they do not run on the same PC as the LNS Server. This is shown in Figure 3.2.

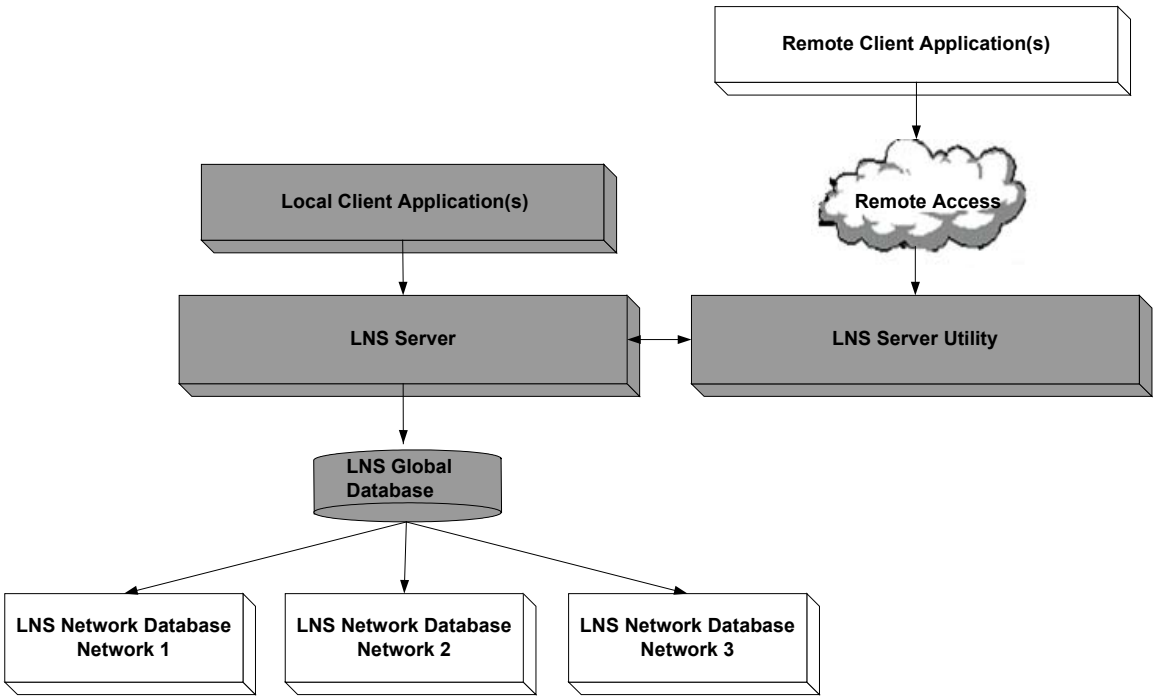


Figure 3.2 LNS Architecture

The gray-shaded blocks in Figure 3.2 represent the LNS Server PC, and items on the LNS Server PC. This includes the LNS global database, any local LNS applications running on the LNS Server PC, and the LNS Server utility.

In Figure 3.2, the global database is connected to three network databases, indicating that there are three networks defined in the global database. Note that the network databases are not necessarily stored on the same PC as the global database and the LNS Server.

As shown in Figure 3.2, the LNS Server utility also runs on the LNS Server PC. **You can use the LNS Server utility to enable remote LNS applications to access the LNS databases and the LNS Server. You must enable the LNS Server with this utility to perform network management applications with remote LNS applications.** The LNS Server utility, the different remote client types, and the media they use to connect to the LNS Server, are described in detail later in this chapter.

LNS Object Server

All local and remote LNS applications must reference the LNS Object Server to access the LNS Server and the LNS databases. Once an application has referenced the LNS Object Server, it can read and write LNS object properties, and call LNS object methods. The LNS Object Server routes each request to the appropriate LNS database.

You can reference the LNS Object Server within your application by importing the LNS ActiveX control into your development environment, as described in the *Importing the LNS ActiveX Control* section on page 46. The LNS Object Server provides all the network services you can use with LNS. It maintains the network databases that store the configuration of your LONWORKS networks, and enables and coordinates multiple points of access to its services and data.

Applications access the services provided by the LNS Server using the LNS Object Server. The LNS Object Server is a COM server that provides an interface, independent of any specific programming language, to the LNS Server and the LNS database.

The LNS Object Server provides the resources to manage and record the configuration information for any LONWORKS network within the following constraints:

- Up to 100 open networks per application when operating in server-independent mode, and up to 50 open networks per application when operating in server-dependent mode. The differences between the server-independent and server-dependent modes are described later in this chapter in the *LNS Clients* section.
- Up to 10 remote LNS client applications, and an unlimited number of local applications, per network. These applications may invoke services on the LNS Server, provide their own subsystem-specific services, properties and events, as well as act as application nodes in the network. Some applications can open more than one network at the same time.
- Up to 1,000 channels.
- Up to 1,000 routers.
- Up to 32,385 application devices.
- There are up to 12,288 network variable selectors in each network. A network variable's selector is a 14-bit number used to identify connected network variables. All network variables in a given connection use the same network variable selector. The LNS Server shares a network variable selector among connections if the connections share one or more network variables. LNS can intelligently reuse network variable selectors; thus, an LNS-managed network is not limited to 12,288 connections.
- Up to 4,096 network variables on each host-based device, and up to 62 network variables on each device hosted on a Neuron Chip or Smart Transceiver.

LNS Object Server Hierarchy

The LNS Object Server defines a set of objects, properties, methods and events that represent the physical attributes of your network and their configurations. The objects are grouped together in a hierarchical fashion, such that the `ObjectServer` object (i.e. the object representing the LNS Object Server) is at the top of the hierarchy.

The `ObjectServer` object contains a collection of `Network` objects, each of which represents a network defined in the global database that the LNS Object Server can administer. Each `Network` object contains a `System` object representing the network's system, and each `System` object contains a set of `Subsystem` objects that represent logical or physical partitions of that particular network. Each `Subsystem` object contains a collection of `AppDevice` objects, which represent the application devices defined in that subsystem, and `Router` objects, which represent the routers defined in that subsystem.

For example, you could set up a network to control a building with 3 floors. You could define 3 subsystems within the network, so that you could logically group the devices and routers on each floor separately. You could also set up additional subsystems representing different rooms on each floor of the building, as your network design requires.

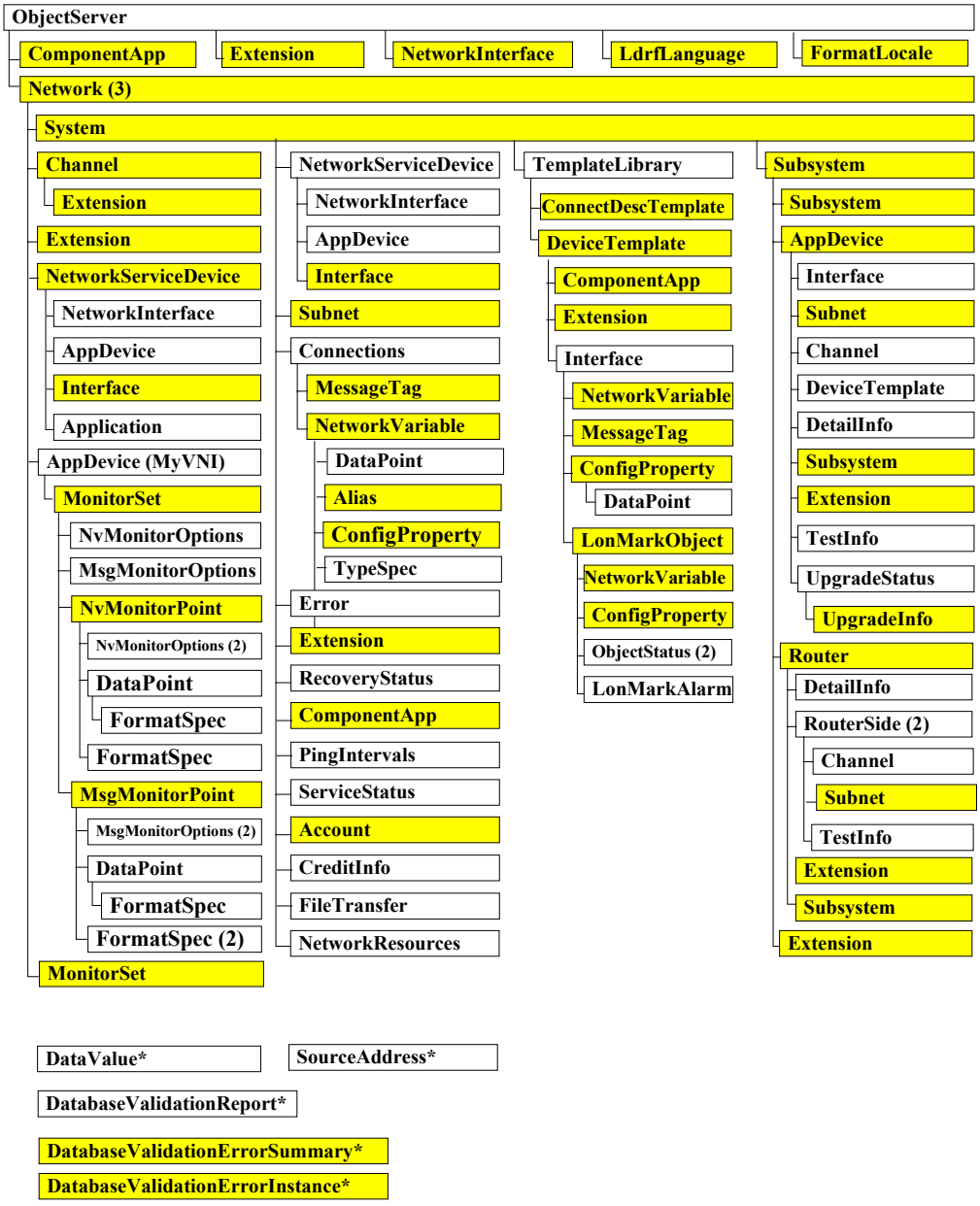
Because subsystems are logical divisions of a network, and devices can belong to multiple logical divisions, you can create multiple subsystems that cross-reference a network for different purposes with your application, and add individual devices to each one. In this manner, devices could appear in multiple subsystems. For example, you could create one subsystem to represent the physical layout of your network, and another to represent the functional layout, separating the HVAC, lighting and security systems in this way.

Each of the object types defined in the LNS Object Server includes its own set of properties and methods. Properties contain information defining the current configuration and operational behavior of an object. Methods provide a mechanism to perform various operations on each object. The LNS Object Server also defines a set of events that you can use to keep your application informed when the configurations of certain objects are modified, or when certain operations are performed.

The LNS Object Server supports all valid COM data types. In addition, the LNS Object Server contains a series of defined exceptions, which are error messages that will be generated if you write invalid data to any of the properties or objects defined in the LNS Object Server, or if an operation caused by the invocation of a method fails for any reason.

The complete LNS Object Server Hierarchy is displayed below. The various objects included in this diagram are described in this manual, and in the *LNS Object Server Reference* help file.

NOTE: You can use the LNS Object Browser to browse the contents of the LNS Object Hierarchy when developing your application. The LNS Object Browser application offers point-and-click access to almost every object, method, and property in the LNS Network Operating System. This makes the Object Browser the ideal tool to investigate an existing LNS-based environment during LNS application development, or to explore LNS for training purposes. It can also be used to manipulate an LNS-based system by modifying properties and invoking methods. Consult the online help provided with the LNS Object Browser for more information on the utility.



*The objects marked with the * character are not referenced directly by any other object, but are created by events and methods. The DataValue and SourceAddress objects are created by network variable and monitor point update events, respectively. The DatabaseValidationReport, DatabaseValidationErrorSummary, and DatabaseValidationErrorInstance objects are created when you call the Validate() method.

Figure 3.3 LNS Object Server Hierarchy

Network Service Devices

You will notice in Figure 3.3 that each `Network` object includes a collection of `NetworkServiceDevice` objects. Each `NetworkServiceDevice` object represents a LONWORKS device implemented by LNS for use with that particular network.

Each LNS application uses a Network Service Device to communicate with the application devices and routers on a LONWORKS network when installing, connecting, managing, and monitoring and controlling the network. Each Network Service Device contains an LNS network interface. Note that the PC running the LNS Server and containing the LNS network database is included in the `NetworkServiceDevices` collection. The Network Service Device for the LNS Server PC sends all network management commands required when any application connected to a network installs, configures and maintains the devices on that network. This is handled transparently by LNS.

As noted earlier in this chapter, LNS applications can run on the LNS Server PC, or on a remote PC. Depending on the way a remote application connects to the LNS Server, the client might use the same Network Service Device as the LNS Server PC within a network, or it may use its own Network Service Device. For more information on this, and for an overview the various client types you can use with LNS, and how each one connects to the LNS Server, see *LNS Clients* on page 37.

Separate Network Service Devices are defined for each network, even if they are both associated with the same application. For example, if a single application has two networks open, each network will contain a separate `NetworkServiceDevice` object representing the application's Network Service Device on that network. All LNS applications running on the LNS Server PC will use the same `NetworkServiceDevice` object within each separate network. Each Network Service Device has its own LonTalk address, and may have network variables and monitor sets defined on it to support monitoring and control operations.

Network Interfaces

Each Network Service Device uses a network interface. A network interface (also called a network services interface, or NSI) is the device that connects the Network Service Device to the LONWORKS network. LNS network interfaces also provide the physical connection to the network and the messaging connection to the LNS Server. They route service requests to the LNS Server and other LNS network interfaces (if multiple Network Service Devices are connected to the network) and coordinate with the LNS Server to manage transactions.

For more information on network interfaces, and for a list of the various network interface devices that are commonly used with LNS, see Chapter 11, *LNS Network Interface Devices*.

LNS Network Services

The primary purpose of the LNS network operating system is to simplify the performance of *network services*. Network services are operations that fall into the following three major categories:

- Network installation and configuration
- Network maintenance and repair
- System monitoring and control

The first two categories are collectively called *network management*. The third category is called *monitor and control*.

A simple network service typically results in the transmission of multiple LonTalk messages across the network. When the application requests a service, e.g. to connect a set of network variables, the LNS Object Server translates the request into LNS service calls which are then expanded into the required network actions — allocating network resources; building, sending, and processing network messages; performing error checking and recovery. The result of the request is routed back to the LNS Object Server, which notifies the application if an error has occurred.

By selecting the appropriate object and invoking its methods, an LNS application can quickly accomplish each network management task with a minimum of overhead. The LNS Object Server manages the device and network resources for the application, and sends and processes the required LonTalk messages.

The LNS Object Server automates network-related tasks whenever possible. You can configure it to automatically discover the presence of newly attached devices on the network. Where automation is not possible, simplification is provided. For example, the LNS Object Server provides connection description objects that allow the LNS application to specify all the attributes of a connection in a single object. As a result, an application can choose a pre-existing connection description when connecting two devices, instead of manually specifying each attribute of the connection. By adding application knowledge (for example, the set of devices that it may encounter and the types of configurations that make sense), an application can further automate tasks and simplify the user interface.

The LNS Object Server provides these core network services to LNS applications. See the sections following this for a more detailed overview of network services that can be applied to network management tasks, and services that can be applied to monitor and control tasks.

Network Management

LNS provides all the network services you need to install, configure, and manage a network with your application. This includes the following:

- Device discovery, installation, configuration, removal, replacement, resetting, testing, and management
- Router installation, configuration, removal, replacement, testing, and management
- Importing device self-documentation and self-identification information ad hoc or through external interface files
- Connection and disconnection of network variables and message tags

- LonMark object access
- Receiving service pin messages
- Data formatting based on standard and user-defined resource files
- Addition and removal of network variables, message tags and LonMark Functional Blocks on host-based devices that support dynamic interface components.
- Copying of configuration property values from one device to another
- Querying and setting of device properties, such as locations, priority slots, self documentation, and network variable attributes
- Generation of network configuration change events
- Subnet and channel creation
- Modification of network variable types of unbound network variables
- Creating connections between devices
- Database recovery from the network

Monitor and Control

Monitoring is the process of fetching data from devices on the network, and control is the process of writing data to network devices. Both involve subordinate tasks such as data formatting and address change tracking (to ensure that data is not lost due to address changes). In a LONWORKS network, data may be retrieved from application devices using bound connections, polling messages at regular intervals, or explicit one-time polls. Formatting is accomplished automatically by processing network variable types according to predefined formatting files. Additional formatting or processing may also be performed by the LNS application.

The first step to take when programming an LNS application is to select a client type and initialize the LNS Object Server. Following that, you can use any of the network services provided by LNS to perform the monitor and control (or network management) tasks listed in these sections. Before doing so, you should review the rest of this chapter, which describes the various client types you can use with LNS, and provides a roadmap you can use when reading this document.

LNS Clients

LNS applications are capable of running locally or remotely. Multiple applications, some running locally and some running remotely, can access a single network simultaneously. A local application is one that is operating on the same PC as the LNS Server and the LNS global database.

A remote client application is one that is running on a PC that does not contain the LNS database or run the LNS Server. Remember that you need to use the LNS Server utility on the LNS Server PC to enable access to the LNS Server by remote clients. If you are using an LNS high performance (Layer 2) network interface, an application running on your client can simultaneously access one or more local networks and one or more remote networks using a single physical network interface (i.e. access a network database on the application's PC as well as a network database on another PC).

An application that accesses the LNS Server remotely can do so as a *Lightweight Client* or as a *Full Client*. A Lightweight Client is an application that communicates with the physical network through the LNS Server PC, via a TCP/IP connection. A Full Client is as application that has its own network interface, and thus communicates with the LNS Server PC through the network, as well as directly with devices on the network. A Full Client is as application that has its own Network Service Device connecting to the network and to the LNS Server PC. The Network Services Device uses an LNS network interface, and either a regular LONWORKS channel or a LONWORKS /IP channel, to do so. A remote Full client communicates with the LNS Server PC through the network, as well as directly with devices on the network.

Note that some applications, known as *Independent Clients,* can access a network without connecting to the LNS Server or to the global database. Independent clients can only be used for monitor and control operations.

When enabling remote client access with the LNS Server application, the LNS Server starts listening at TCP port 2540 by default. Any Lightweight client can then access your LNS Server, if they are aware of your LNS Server's IP address and TCP base port numbers. If you wish to completely disable access by Lightweight clients, see the available options that are detailed in the help file for the LNS Server application. If you wish to restrict access by remote LNS applications to specific IP addresses or IP address ranges, use the `PermissionString` property. For more information on the use the `PermissionString` property, see the LNS Object Server Reference help file.

The following sections detail the various ways local and remote applications communicate with the LNS Server and the LONWORKS network.

Local Client Applications

Local Client applications run on the LNS Server PC – the same PC that contains the LNS global database. This is shown in Figure 3.4. In this configuration, the PC running the application contains an LNS network interface that is used to communicate directly with the physical network. For descriptions of the network interfaces you can use with LNS, see Chapter 11, *LNS Network Interfaces.*

Each network contains a `NetworkServiceDevice` object that is shared by the LNS Server PC, and all local applications that have that network open.

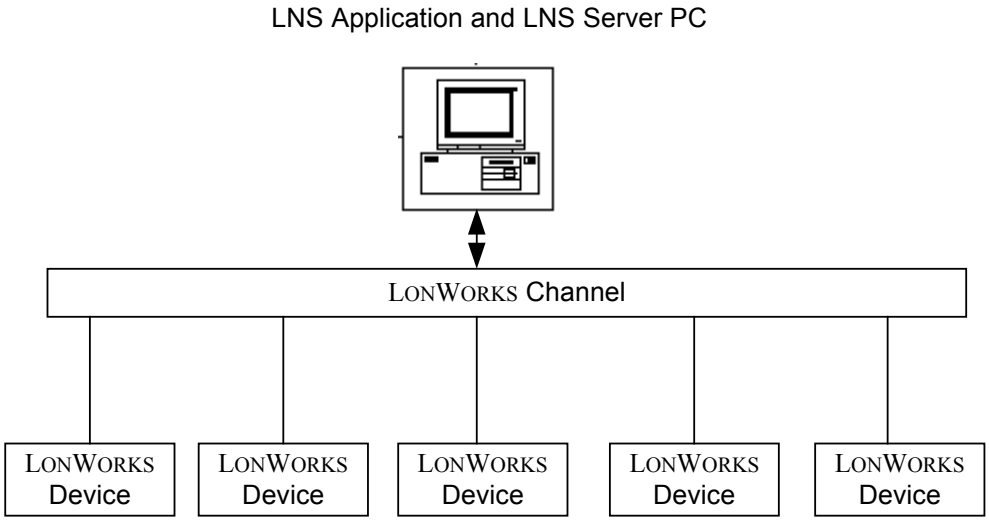


Figure 3.4 Network Communication as a Local Client

Lightweight Client Applications

Lightweight client applications run on a different PC than the LNS Server and the LNS database. The remote PC is connected to the LNS Server PC via TCP/IP sockets, as shown in figure 3.5.

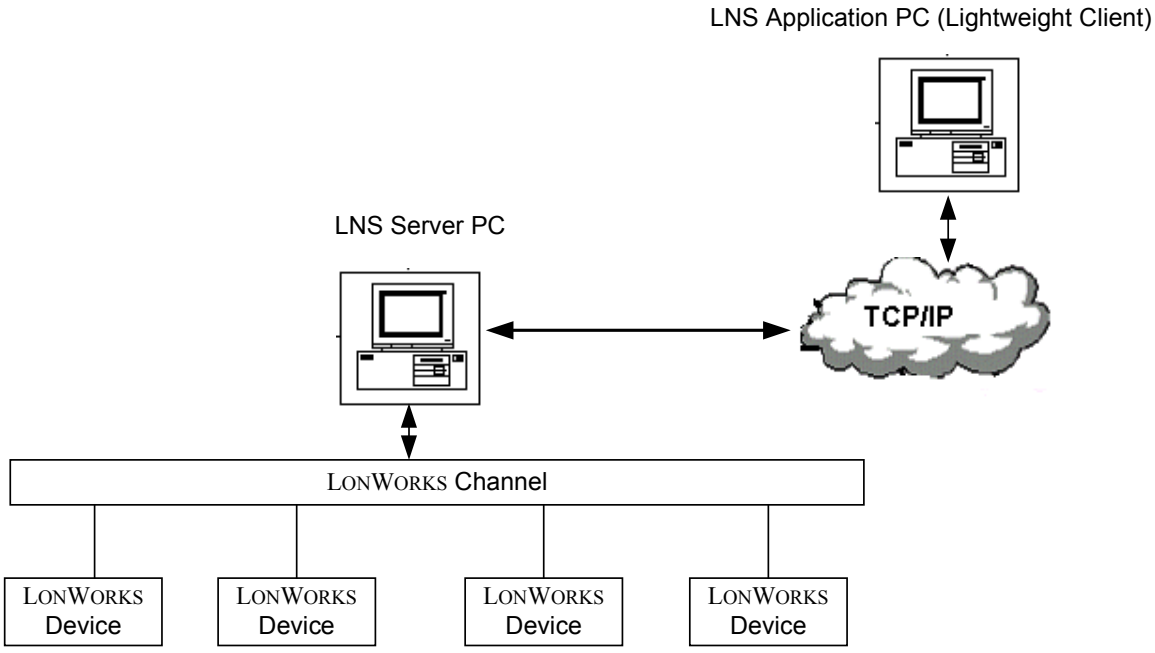


Figure 3.5 Network Communication as a Remote Lightweight Client

Remote Lightweight client applications must have regular contact with the LNS Server. A Lightweight client application will be disconnected from the LNS Server if the TCP/IP connection is broken. A Lightweight client application cannot perform monitor and control operations if the connection to the LNS Server is lost, because both the

connection to the LNS Server and the data monitoring and control are through the TCP/IP connection.

On some PCs, if an established TCP/IP connection is idle for some time, the PC enters a power-save mode that causes TCP/IP disconnection. The LNS Server treats this as it would any other disconnection. Once the LNS Server detects the disconnection, the client application must reestablish its connection to the LNS Server by closing the `System` and `Network`, and then reacquiring and reopening them. Similarly, all other LNS object references that the LNS application holds in scope must be released and reacquired from the LNS Server. To avoid going into power save mode unintentionally, disable the power saver mode on the LNS Server PC and the PC running your Lightweight client application.

All remote Lightweight client applications use the same `NetworkServiceDevice` object that the LNS Server PC uses, and that the local client applications use for a given network. Therefore, any network variables, connections, and monitor sets defined on that Network Service Device are available to both Local and Lightweight clients.

Full Client Applications

Full Client applications run on a different PC than the LNS Server and the LNS database, as shown in Figure 3.6.

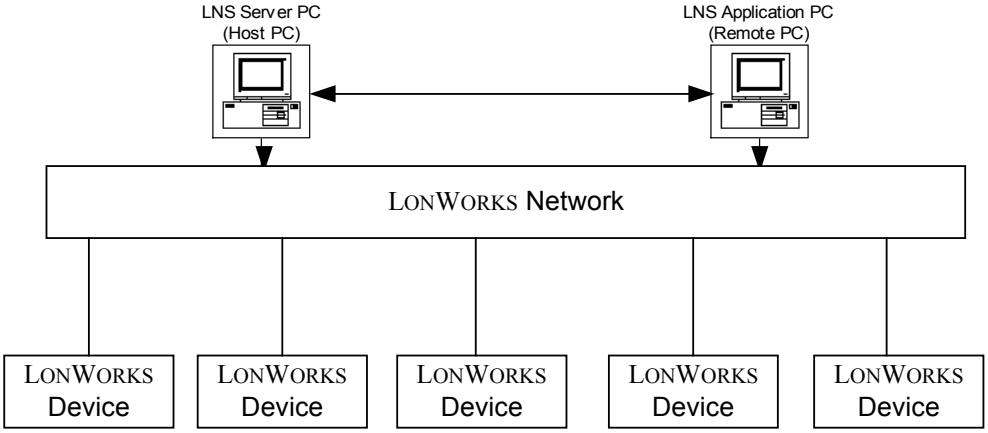


Figure 3.6 Network Communication as a Full Client

Each Full client PC contains a network interface it uses to connect to the LONWORKS channel, and communicate with the LONWORKS network and LNS Server. For descriptions of the network interfaces you can use with LNS, see Chapter 11, *LNS Network Interfaces*.

In the case of a communication problem between the LNS Server and a Full Client application, the LNS Server will retry the connection for more than a minute before timing out and determining that the client is disconnected. From the point of view of the Full Client application, a communication problem that results in disconnection will appear to be an unusually long LNS service that ultimately returns the `LCA:#120 lcaErrNoConnectionToServer` exception.

Each remote Full Clients use a `NetworkServiceDevice` object defined for shared use by it and all the other remote Full Clients on the same PC connected to the network. They do not share the Network Service Device used by Local Client applications,

Lightweight Client applications, or the LNS Server PC. Therefore network variables, connections and monitor sets defined on the Network Service Device used by a remote Full client application are shared only by other remote Full clients on the same PC that are connected to the same network.

Independent Clients

If an application does nothing but perform monitor and control services, it does not need to access the LNS network database, and thus does not need to connect to the LNS Server. For this reason, LNS provides a *server-independent* mode. An application running in server-independent mode is considered an Independent client application.

Independent client applications do not need the LNS Server to be running on the PC containing the LNS global database, as shown in Figure 3.7. However, only applications which can directly access the LONWORKS network via an LNS network interface or LONWORKS/IP channel can operate as Independent client applications. Note that in Figure 3.7, the LNS Server PC is not attached to the network.

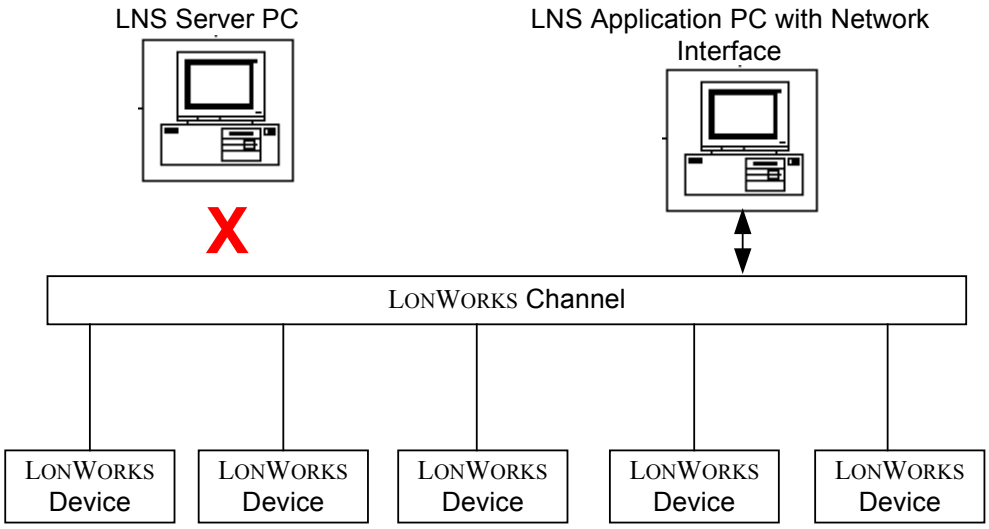


Figure 3.7 Network Communication As An Independent Client

Independent client applications can monitor and control monitor points that have been previously created by a Full or Local client application, and automatically receive network addressing updates from the network. They cannot create monitor sets, or add monitor points to a monitor set, because Independent client applications cannot access the LNS database. No objects can be added or removed by an Independent client application, and most methods and properties unrelated to monitor and control are unavailable. As noted, Independent client applications can only be used for monitor and control operations.

For more information on monitor and control, see Chapter 9 of this document.

Getting Started

The first step to take when programming an LNS application is to select a client type and initialize the LNS Object Server. Following that, you can use the network services provided by LNS. Before doing so, you should review the rest of this document. For your

convenience, Table 3.1 lists the remaining chapters of this document, and summarizes their contents.

You should note that the remaining chapters of this document describe LNS methods, objects and properties as they pertain to each network management or monitor and control task. In many cases, you can find more extensive details about an object, property, method or event in the *LNS Object Server Reference* file. The *LNS Object Server Reference* file can be accessed through the main Echelon LNS Application Developer's Kit help file via the Echelon LNS Application Developer's Kit menu in the Windows Programs group.

Table 3.1 LNS Programmer's Guide Document Roadmap

Chapter	Description
Chapter 4, *Programming an LNS Application*	This chapter describes the steps you need to take when initializing an LNS application, whether you plan to use that application for network management, or for monitor and control. This includes all the steps you need to take to open the LNS Object Server, and initially access a LONWORKS network. Separate instructions are provided for each of the client types introduced in this chapter.
Chapter 5, *Network Management : Installation Scenarios*	This chapter begins the discussion of the network management operations you can perform with LNS. It starts with an overview of the three basic installation scenarios you can use when creating a LONWORKS network with LNS, and follows with step-by-step instructions you can follow when performing each installation scenario. The tasks included in each step are described in more detail in Chapters 5, 6 and 7 of this document.
Chapter 6, *Network Management : Defining, Commissioning, and Connecting Devices*	Chapter 5 introduces the tasks you need to take when installing and defining a network, and explains when you should perform each task. Chapter 6 provides additional details on these tasks. This includes topics such as installing, configuring, commissioning, and connecting devices.
Chapter 7, *Network Management : Optimizing Connection Resources*	This chapter contains guidelines you may find useful when planning connections between devices on large networks.
Chapter 8, *Network Management : Advanced Topics*	This chapter provides information on advanced topics that may be of use to you when writing a network management application. This includes how to properly manage a Network Service Device, how to manage a network with multiple channels, and how you can take advantage of the dynamic device interface features added to LNS for Turbo Edition.
Chapter 9, *Monitor and Control*	This chapter describes how to write applications to monitor and control the devices on your LONWORKS network.
Chapter 10, *LNS Database Management*	This chapter provides information about how you should manage your LNS databases, and describes the LNS features you can use to do so. This includes a description of the LNS database validation and online database backup features that have been added to LNS for Turbo Edition, how to move LNS databases, and how to perform a network database recovery with LNS.

Chapter	Description
Chapter 11, *LNS Network Interface Devices*	This chapter describes the various network interfaces you can use with LNS.
Chapter 12, *Director Applications and Plug-Ins*	This chapter discusses the standards and development methodology for creating interoperable LNS director and plug-in applications.
Chapter 13, *LNS Licensing*	This chapter describes the LNS licensing model. This will be of interest to developers writing a network management application, as you will need to consider some licensing issues when installing and commissioning the devices on your network.
Chapter 14, *Distributing LNS Applications*	This chapter describes how you can redistribute your LNS applications, including how to use the LNS Redistributable Maker utility and how to install your LNS application.
Chapter 15, *Advanced Topics*	This chapter describes advanced topics you may find useful when using LNS. This includes subjects like file transfer, developing remote and portable tools, multi-threading, avoiding memory leaks, and writing interoperable LNS applications.
Appendix A, *Deprecated Methods and Obsolete Files*	This appendix will be of interest mainly to those who are upgrading to LNS Turbo Edition from a previous version of LNS. It may also be of interest if you are developing a plug-in application that must work with older versions of LNS. It lists LNS methods, properties and objects that you should avoid using with Turbo Edition. It also lists files installed with previous versions of LNS that are no longer installed or used.
Appendix B, *LNS, MFC and ATL*	This appendix provides additional information you will need when developing an LNS application with MFC and ATL.
Appendix C, *LNS Turbo Edition Example Application Suite*	This appendix summarizes the example applications included with the LNS Turbo Edition software.

Chapter 4 - Programming an LNS Application

This chapter describes the steps you will need to follow when initializing an LNS application, whether it is an application for network management, or for monitor and control. The tasks described in this chapter include how to initialize the LNS Object Server, how to open a network, and how to terminate an LNS application.

Programming an LNS Application

This chapter describes the tasks you need to follow when initializing an LNS application. This includes the following steps:

1. Import the LNS ActiveX Control into your development environment. For more information on this, see the next section, *Importing the LNS ActiveX Control*.
2. Initialize the LNS Object Server, and open the network you plan to use. For more information on these tasks, see *Initializing an LNS Application* on page 48.
3. Open the system and set the system parameters. For more information on these tasks, see *Opening a System* on page 61.
4. Once you have completed the tasks described in steps 1-3, you can use the network management and monitor and control services provided by LNS. Chapters 5, 6, 7 and 8 of this document describe the network management tasks you can perform with LNS. Chapter 9 describes the monitor and control services you can take advantage of.

   Before moving to these chapters, you should note that this chapter contains information on other LNS features that may be vital to any LNS application. This includes guidelines on how to use transactions and sessions within an LNS application, how to handle errors, guidelines on handling events within an LNS application, and instructions to follow when terminating your LNS application.

Importing the LNS ActiveX Control

To begin developing your LNS application, you first need to import the LNS Object Server ActiveX Control into your development environment, and add a reference to the control to your project. This will allow you to access the objects, methods and properties included in the LNS object hierarchy with your application.

Importing the Control into Visual Basic 6.0

The Visual Basic environment is especially suited for rapid application development. Before you use the LNS Object Server, you must add the LNS Object Server ActiveX Control to the Visual Basic tool palette using the **Custom Controls** menu item. To do so, follow these steps:

1. Open Visual Basic.
2. From the **Project** menu, select the **Components...** command.
3. In the list box scroll down to the **LNS Object Server ActiveX Control 3.20** box. Click on the box to the left so that an **X** appears.
4. Click the **OK** button to close the **Custom Controls** dialog box. The Object Server control should now appear in the Visual Basic Tool Palette.

Once the tool is part of the Visual Basic tool palette, you can add the LNS Object Server to an application by selecting the tool and dragging it onto one of the application's forms. To do so, follow these steps:

1. Select the LNS Object Server icon on the Visual Basic tool palette.
2. Drag the Object Server control onto one of your application's forms. By default, the name of the control will be `[formName].LcaObjectServer1`, where `[formName]` is the name of the form where you placed the control.
3. Optionally, change the name of the Object Server control by setting the `Name` property for the control.

Importing the Control into Visual C++

To import the LNS Object Server control into Visual C++ and Visual C++ .NET, make sure that the LNS Application Developer's Kit is installed on your development PC, and then add the following statement to a global header file for your project, such as `StdAfx.h`:

```
"#import "lcaobjsv.ocx" named_guids rename_namespace("lca")"
```

This statement will direct the preprocessor to locate the LNS type library, and create header files containing wrappers that expose the available interfaces. Once the wrappers are created, the `#import` statement will also serve as an include statement for the wrapper header files. For more information on this, see Appendix B, *LNS, MFC and ATL*.

For Visual C++ .NET, adding the #import statement shown above is the only step you need to take to import the LNS Object Server control. If you are using Visual C++ 6.0 and want to use the MFC Class Wizard for event handlers, perform these steps as well:

1. From the **Project** menu, select the **Add to Project** command. Then, select the **Components and Controls Item** in the sub-menu. This activates the Component and Controls Gallery.
2. Open the **Registered ActiveX Controls** folder.
3. Select the **LONWORKS ObjectServer** control icon, and click **Insert**.
4. A confirmation dialog appears with the name of the wrapper class to be created for the Object Server.
5. Click the **OK** button to finish adding the control. The LNS Object Server control now appears as a tool icon in the resource editor's control toolbar.
6. Click the **Close** button to close the component gallery dialog.

Once the control is part of the resource editor's toolbox, you can add the control to an application by dragging the control from the toolbox onto a dialog resource, and then using the MFC Class Wizard to create the member variable for the control and its LNS Object class definitions. When creating a Visual C++ application, Echelon recommends that you do not reduce the default Visual C++ stack size.

To create an instance of the Object Server control in a Visual C++ application, follow these steps:

1. Select the LNS Object Server icon on the resource editor toolbox.

2. Drag the Object Server control onto one of your application's dialogs.
3. Change the control's resource ID, as necessary, by right clicking on the newly added control and selecting Properties. This opens a dialog box containing the current resource ID and other properties of the control.
4. From the **View** menu, select **Class Wizard**. This opens the Visual C++ Class Wizard.
5. Select the **Member Variables** tab.
6. Select the Object Server's control ID (defined in step 3) from the list of control IDs, and click the **Add Variable** button.
7. Enter the variable name, verify that the Category box contains the value "Control", then click **OK**. The member variable is then created within the class for the dialog containing the control.

Initializing an LNS Application

After you have imported the LNS Object Server Active X control, you can begin using LNS services to perform network management or monitor and control operations. You first need to perform some initialization tasks.

The first initialization task is to configure and open the LNS Object Server. To do so, your must select the network access mode and licensing mode for your application. Then, your application can open the Object Server and begin accessing your LONWORKS networks.

The steps you need to follow vary for Local, Full, Lightweight and Independent client applications. The following sections describe the steps you need to take for each client type. For a description of each client type, and reasons why you might want to use each client type, see *LNS Clients* on page 37.

Initializing a Local Client Application

This section describes how to initialize a Local client application. This includes the following steps:

1. Selecting the Local Access Mode
2. Specifying the License Mode
3. Opening the Object Server
4. Selecting a Network Interface
5. Opening a Network

Table 4.1 includes sample code that you could use when performing each of these tasks. See the sections following Table 4.1 for more information on each task.

Table 4.1 Initializing a Local Client Application

Task	Sample Code	For More Information, See...
Selecting the Local Access Mode	`ObjectServer.RemoteFlag = False`	*Selecting the Access Mode* on page 49
Specifying the License Mode	`ObjectServer.SetCustomerInfo(CustomerID,CustomerKey)`	*Specifying the Licensing Mode* on page 49
Opening the Object Server	`ObjectServer.Open()`	*Opening the Object Server* on page 50

Selecting a Network Interface	`Dim NICollection as LcaNetworkInterfaces`
	`Dim SelectedNI as LcaNetworkInterface`
	
	`Set NICollection = ObjectServer.NetworkInterfaces`
	`Set SelectedNI = NICollection.Item("LON1")`
		Selecting a Network Interface on page 50

Opening a Network	`Dim MyNetworks as LcaNetworks`
	`Dim MyNetwork as LcaNetwork`
	
	`Set NetworksCollection = ObjectServer.Networks`
	`Set MyNetwork=NetworksCollection.Item("Building 76")`
	`MyNetwork.Open()`
		Opening a Network on page 51

Selecting the Access Mode

Each application must have some way of determining whether it will run as a remote or local client. Sometimes, the access mode is user-determined, meaning that a user selects a remote or local option as part of your application's start-up process. In other cases, you might design your applications specifically for one form of operation.

To specify the LNS application access mode as local, set the `ObjectServer` object's `RemoteFlag` property to `False`, as shown below:

```
ObjectServer.RemoteFlag = False
```

Specifying the Licensing Mode

The last step you should take before opening the LNS Object Server is to set the licensing mode. The licensing mode determines how the LNS Object Server will track the addition of devices to a network by your application. You can set the licensing mode to either Demonstration Mode, or Standard Mode. Demonstration Mode is used by default, but you must use the Standard Mode once your application begins normal operation.

To enable the Standard Mode, set your customer identification information by calling the `SetCustomerInfo()` method on the `ObjectServer`, as shown below:

```
ObjectServer.SetCustomerInfo(CustomerID, CustomerKey)
```

The CustomerID and CustomerKey parameter supplied to the function refer to the customer identification and key numbers that are printed on the back cover of the LNS Application Developer's Kit CD-ROM jewel case. For more information on LNS licensing, and for more details on the differences between Standard Mode and Demonstration Mode, see Chapter 13, *LNS Licensing*.

Opening the Object Server

Once you have set the network access mode and licensing mode, you can open the Object Server, as shown below:

```
ObjectServer.Open()
```

This opens the LNS global database. An application can determine or change the location of the global database, which is stored in the Windows System Registry, by accessing the ObjectServer object's DatabasePath property. If the value of this property is changed, the Windows Registry will be updated automatically. By default, the path value is set to the following by the LNS software setup program:

```
[Windows Drive]\LONWORKS\ObjectServer\Globaldb
```

Note that interoperable LNS applications should not modify the global database path, since all LNS applications running on the PC must share the global database, and changing the path may cause other applications to be unable to access the global database.

NOTE: If you will be opening any networks with an LNS application that is running as a Windows service, then the first application to open the LNS Object Server must also be running as a Windows service. In addition, if a network is to be opened by an LNS application that is running as Windows service, then that network and system must be opened by an LNS application that is running as Windows service before it is opened with an LNS application running as a user process. For more information on this, consult the help pages for the Open() methods of the Network and ObjectServer objects in the *LNS Object Server Reference* help file.

Selecting a Network Interface

When operating locally, an LNS application may run while physically attached to a network, or while detached from the network. To run attached, the LNS application must select the network interface it will use to communicate with the network. To run detached, the LNS application must explicitly de-select the previously selected network interface.

Note that before you use a network interface, you may need to configure the network interface with the LONWORKS Interfaces application in the Windows control panel. For more information on this, and for general information on the various network interfaces you can use with LNS, see Chapter 11, *LNS Network Interfaces*.

The engineered mode installation scenario described in Chapter 5 of this document splits the process of installing LONWORKS devices on a network into two stages — a definition stage and a commissioning stage. During the definition stage, a network interface is not required. So, when using the engineered mode installation scenario, your application does not need to specify a network interface. This is referred to as *engineered mode*. If the network interface has never been assigned, the system will start in engineered mode. If a network interface has been previously assigned, the application must explicitly deselect that network interface to start in engineered mode (by setting the NetworkInterface

property to NOTHING for Visual Basic, or NULL for Visual C++). This is described in more detail in Chapter 5.

If multiple client applications have the same network open, they must either use the same network interface, or they must all be in engineered mode. If an LNS application opens a network without explicitly selecting a network interface, the network will be opened using the same network interface that was used the last time the network was opened, or it will be opened in engineered mode if the network was last opened in engineered mode. Newly created networks will be opened in engineered mode if the network interface is not explicitly specified.

To select a network interface for local operation, follow these steps:

1. Fetch the ObjectServer object's NetworkInterfaces collection. All network interfaces registered in the Windows System Registry on the PC running your application are automatically detected by the LNS Object Server and included in this collection.

   ```
   Dim NICollection as LcaNetworkInterfaces
   Set NICollection = ObjectServer.NetworkInterfaces
   ```

2. Select the desired network interface. The following code selects a network interface named "LON1" which will be used to open the system.

   ```
   Dim SelectedNI as LcaNetworkInterface
   Set SelectedNI = NICollection.Item("LON1")
   ```

Opening a Network

Once you have opened the Object Server, you can begin using your Local client application to access LONWORKS networks. This section describes how to open an existing network, and how to create a new network. To open a network, or to create a new network, follow these steps:

1. Retrieve the Networks collection from the ObjectServer object's Networks property. The Networks collection contains all local networks. For Local client applications, the Object Server retrieves the Networks collection from the LNS global database.

   ```
   Dim MyNetworks as LcaNetworks
   Set MyNetworks = ObjectServer.Networks
   ```

2. To create a new network, call the Add() method on the Networks collection. Networks can only be created when running locally. The following code creates a network named "Building 75" with database path "c:\bldg75". The final parameter supplied to the Add() method forces the creation of a new network database in the specified database path if set to True, and imports an existing database from the specified database path if set to False.

   ```
   Dim MyNetwork as LcaNetwork
   Set MyNetwork = MyNetworks.Add("Building75","c:\bldg75",True)
   ```

   To fetch an existing network from the collection, obtain the desired Network object from the Networks collection. You can iterate through the Networks collection to list all available networks, or you can use the Item property to open a specific network by its name or by its index

within the Networks collection. The name to use is specified by the Network object's Name property. The following code fetches an existing network called "Building76" from the global database.

```
Dim MyNetwork as LcaNetwork
Set MyNetwork = MyNetworks.Item("Building 76")
```

3. Call the Open() method on the Network retrieved in step 2 to open its LNS network database.

```
MyNetwork.Open()
```

4. Optionally repeat steps 2 and 3 to open more networks. Once you have opened a network, you should follow the tasks described in the *Opening a System* section later in this chapter to configure and open the system.

Initializing a Remote Full Client Application

This section describes how to initialize a Full client application. This includes the following steps:

1. Selecting the Remote Access Mode
2. Specifying the License Mode
3. Opening the Object Server
4. Selecting a Network Interface
5. Opening a Network

Table 4.2 includes sample code that you could use to incorporate each of these tasks. See the sections following Table 4.2 for more information on each task.

Table 4.2 Initializing a Full Client Application

Task	Sample Code	For More Information, See..
Selecting the Access Mode	`ObjectServer.RemoteFlag = True` `ObjectServer.Flags = lcaFlagsUseNSI`	*Selecting the Access Mode* on page 53
Specifying the License Mode	`ObjectServer.SetCustomerInfo(CustomerID,CustomerKey)`	*Specifying the License Mode* on page 53
Opening the Object Server	`ObjectServer.Open()`	*Opening the Object Server* on page 53

Selecting a Network Interface	`Dim NICollection as LcaNetworkInterfaces`
	`Dim SelectedNI as LcaNetworkInterface`
	
	`Set NICollection = ObjectServer.NetworkInterfaces`
	`Set SelectedNI = NICollection.Item("LON1")`
	`Set ObjectServer.ActiveRemoteNI = SelectedNI`
		Selecting a Network Interface on page 54

Task	Sample Code	For More Information, See..

Opening a Network	`Dim MyNetworks as LcaNetworks`
	`Dim MyNetwork as LcaNetwork`
	
	`ObjectServer.RemoteFlag = True`
	`Set MyNetworks = ObjectServer.Networks`
	`Set MyNetwork=NetworksCollection.Item("Building 75")`
	`MyNetwork.Open()`
		Opening a Network on page 55

Selecting the Access Mode

As described in *Selecting the Access Mode* on page 49, each application must have some means of determining whether it will run as a remote or local client. To specify the access mode for a Full client, follow these steps:

1. If the PC running your application has not previously opened the network you plan to use, or if the Network Service Device your application is using is on a different channel than it was last time the network was opened, select remote operation by setting the `ObjectServer` object's `RemoteFlag` property to `True`.

   ```
   ObjectServer.RemoteFlag = True
   ```

2. Set the `ObjectServer` object's `Flags` property to `lcaFlagsUseNSI`. This indicates that the Full client application will use a network interface to access the LNS Server:

   ```
   ObjectServer.Flags = lcaFlagsUseNSI
   ```

   NOTE: The `lcaFlagsUseNSI` and `lcaFlagsUseTCP` flags are mutually exclusive.

NOTE: The first time you open a network with a remote Full client application, the LNS Object Server will need to query the network to determine what remote networks are currently opened in all connected LNS Servers. This may take a long time, since the LNS Object Server must search the network for all accessible servers. After the first time you have opened a network, you can generally bypass this process by changing the way you initialize your Full client application. For instructions on this, see *Opening a Network* on page 55.

Specifying the License Mode

The last step you should take before opening the LNS Object Server is to set the licensing mode. The licensing mode determines how the LNS Object Server will track the addition of devices to a network by your application. You can set the licensing mode to either Demonstration Mode, or Standard Mode. Demonstration Mode is used by default, but you must use the Standard Mode once your application begins normal operation.

To enable the Standard Mode, set your customer identification information by calling the `SetCustomerInfo()` method on the `ObjectServer`, as shown below:

```
ObjectServer.SetCustomerInfo(CustomerID, CustomerKey)
```

The `CustomerID` and `CustomerKey` parameter supplied to the function refer to the customer identification and key numbers that are printed on the back cover of the LNS

Application Developer's Kit CD-ROM jewel case. For more information on LNS licensing, and for more details on the differences between Standard Mode and Demonstration Mode, see Chapter 13, *LNS Licensing*.

Opening the Object Server

Once you have set the network access and license modes, you can open the Object Server, as shown below:

```
ObjectServer.Open()
```

NOTE: If you will be opening any networks with an LNS application that is running as a Windows service, then the first application to open the LNS Object Server must also be running as a Windows service. In addition, if a network is to be opened by an LNS application that is running as Windows service, then that network and system must be opened by an LNS application that is running as Windows service before it is opened with an LNS application running as a user process. For more information on this, consult the help pages for the `Open()` methods of the `Network` and `ObjectServer` objects in the *LNS Object Server Reference* help file.

Selecting a Network Interface

Local clients can access a record of the possible networks and servers from the LNS global database. In contrast, each Full client application must register with the LNS Server in order to interact with a network. Each Full client does so by querying the LONWORKS network it's attached to for available servers.

Each Full client applications must specify its network interface prior to having access to the available servers and networks. You can do so by following these steps:

1. Fetch the `NetworkInterfaces` collection from the `ObjectServer` object. All network interfaces registered in the Windows System Registry on your application PC are automatically read by the Object Server, and are included in this collection.

   ```
   Dim NICollection as LcaNetworkInterfaces
   Set NICollection = ObjectServer.NetworkInterfaces
   ```

2. Select the desired network interface. Remember that the `NetworkInterfaces` collection contains all registered network interfaces, including those that are not suitable for use with LNS.

   NOTE: In LNS 3.0 and all subsequent releases, if multiple Full Client applications on the same PC attempt to open the same network, they must use the same network interface. Consider a case where you have two PCLTA-20 network interface cards called LON1 and LON2 on a PC that is running several Full client applications. If an application on that PC opens a network using LON1, and then another application on the PC attempts to open the same network using LON2, the second application will receive the `NS#149 lcaErrNsConflictWithCurrentNetwork` exception when it attempts to open the network. However, the second application will be able to successfully open the network using LON1.

   The following code selects a network interface named "LON1."

```
Dim SelectedNI as LcaNetworkInterface
Set SelectedNI = NICollection.Item("LON1")
```

3. Set the remote network interface for the LNS application to the network interface selected in step 2.

   ```
   ObjectServer.ActiveRemoteNI = SelectedNI
   ```

   Note that before you use a network interface, you may need to configure the network interface with the LONWORKS Interfaces application in the Windows control panel. For more information on this, and for general information on the various network interfaces you can use with LNS, see Chapter 11, *LNS Network Interfaces*.

Opening a Network

This section describes how to connect to a network with a Full client application. You cannot create new networks with Full client applications. The LNS Server utility must be running on the PC containing the LNS databases (i.e. the LNS Server PC) to open a network as a Full client.

The steps required to open a network with a remote Full client application vary, depending on whether a remote Full client application using your Network Service Device has opened the network before. If a remote Full client application has not previously accessed the network to be opened with your Network Service Device, follow these steps:

1. Get the `ObjectServer` object's `Networks` collection. The LNS Object Server will need to query the network to determine what remote networks are currently opened in all connected LNS Servers. This may take a few moments, since the LNS Object Server must search for all accessible servers.

   ```
   Dim MyNetworks as LcaNetworks
   Set MyNetworks = ObjectServer.Networks
   ```

2. Get the network to be opened. You need to specify the network to be opened by the `Network` object's `Name` property, which was specified when the network was created, or by using the object's numerical index within the `Networks` collection.

   ```
   Dim MyNetwork as LcaNetwork
   Set MyNetwork = MyNetworks.Item("Building 75")
   ```

3. Call the `Network` object's `Open()` method to open its LNS network database. The LNS Server must be running on the PC containing the LNS network database, or the operation will fail.

   ```
   MyNetwork.Open()
   ```

4. Optionally repeat steps 2 and 3 to open more networks. Note that a remote Full client application can access multiple networks simultaneously, but must use the same network interface to open each one.

   Once you have opened a network, you should follow the tasks described in the *Opening a System* section later in this chapter to configure and open the system.

If a remote Full client application using your Network Service Device has accessed the network before, and the `LcaNsdType` property of the Network Service Device is set to `lcaNsdTypePermanent`, you can use the `RemoteNetworks` collection to access the network. This allows you to skip the time consuming process of querying the network for available servers. The `RemoteNetworks` collection contains all the networks that have previously been opened by Full client applications running on your PC.

When using the `RemoteNetworks` collection, the `Flags` property should not be set to `lcaFlagsUseNSI`, and your application should not specify a network interface, as described earlier in this section. When a network is opened via the `RemoteNetworks` collection, LNS will use the last network interface used to open the network. If you want to use a different network interface to open a network with a remote Full client application, you need to access the network through the `Networks` collection, as described previously.

To open a network via the `RemoteNetworks` collection, follow these steps:

1. Before opening the Object Server, make sure that the `Flags` property is not set to `lcaFlagsUseNSI`, and that your application has not specified a remote network interface. In addition, set the `RemoteFlag` property to `False`.

   ```
   ObjectServer.ActiveRemoteNI = nothing
   ObjectServer.RemoteFlag = False
   ObjectServer.Flags = ObjectServer.Flags And Not _
        (lcaFlagsUseNSI Or lcaFlagsUseTCP)
   ```

2. Access the `RemoteNetworks` collection.

   ```
   Dim MyNetworks as LcaNetworks
   Set MyNetworks = ObjectServer.RemoteNetworks
   ```

   You should be aware that changes made on the LNS Server PC can result in inconsistencies between the LNS Server and the `RemoteNetworks` collection. This includes changing the local network interface (if this results in a new Neuron ID on the LNS Server PC), or removing the Network Service Device for a Full client. These changes may prevent your remote Full client application from successfully using an entry in the `RemoteNetworks` collection. In addition, if the `LcaNsdType` of the client application's Network Service Device is set to `lcaNsdTypeTransient` or (in some cases) `lcaNsdTypeStandard`, the Network Service Device will be removed when the client application closes, and will not exist in the `RemoteNetworks` collection. Finally, if your application's Network Service Device, or if the Network Service Device used by the LNS Server has changed channels, it may not be possible to successfully use an entry in the `RemoteNetworks` collection.

In any of these cases, your client application should resort to using the `Networks` collection. Once opened, the entry for the network in the `RemoteNetworks` collection will be repaired.

3. Get the network to be opened. In this case, you need to specify the network to be opened by the `Network` object's `RemoteNetworkName` property. The default value for the `RemoteNetworkName` property is r_<Network Name>, where <NetworkName> represents the value assigned to the `Network` object's `Name` property. For example, if the `Name` property is set to HVAC, the name in the `RemoteNetworks` collections will be r_HVAC.

   ```
   Dim MyNetwork as LcaNetwork
   Set MyNetwork = NetworksCollection.Item("r_Building 75")
   ```

4. Call the `Network` object's `Open()` method to open its LNS network database. The LNS Server utility must be running on the PC containing the LNS network database, or the operation will fail.

   ```
   MyNetwork.Open()
   ```

5. Once you have opened a network, you should follow the tasks described in the *Opening a System* section later in this chapter to configure and open the system.

Initializing a Remote Lightweight Client Application

This section describes how to initialize a Lightweight client application. This includes the following steps:

1. Selecting the Remote Access Mode
2. Specifying the License Mode
3. Opening the Object Server
4. Opening a Network

Table 4.3 includes sample code that you could use to incorporate each of these tasks. See the sections following Table 4.3 for more information on each task.

Table 4.3 Initializing a Lightweight Client Application

Task	Sample Code	For More Information, See..
Selecting the Remote Access Mode	`ObjectServer.RemoteFlag = True` `ObjectServer.Flags = lcaFlagsUseTCP`	*Selecting the Remote Access Mode* on page 58
Specifying the License Mode	`ObjectServer.SetCustomerInfo(CustomerID,CustomerKey)`	*Specifying the License Mode* on page 58

Task	Sample Code	For More Information, See..
Opening the Object Server	`ObjectServer.Open()`	*Opening the Object Server* on page 58

Opening a Network	`Dim Networks as LcaNetworks`
	`Dim MyNetwork as LcaNetwork`
	
	`Set MyNetworks = ObjectServer.Networks`
	`Set MyNetwork=MyNetworks.Item("Building 75")`
	`MyNetwork.Open()`
		Opening a Network on page 59

Selecting the Remote Access Mode

As described in *Selecting the Access Mode* on page 49, each application must have some means of determining whether it will run as a remote or local client. To specify the access mode for a Lightweight client, follow these steps:

1. Set the `ObjectServer` object's `Flags` property to `lcaFlagsUseTCP`. This sets the network transport mode, and indicates that the Lightweight client will access the LNS Server via a TCP/IP connection:

   ```
   ObjectServer.Flags = lcaFlagsUseTCP
   ```

   NOTE: The `lcaFlagsUseNSI` and `lcaFlagsUseTCP` flags are mutually exclusive.

2. Select remote operation by setting the `ObjectServer` object's `RemoteFlag` property to `True`. This will cause the `Networks` collection to contain all remote networks have been entered into the Windows Registry on the PC running your application after the Object Server has been opened:

   ```
   ObjectServer.RemoteFlag = True
   ```

Specifying the License Mode

The last step you should take before opening the LNS Object Server is to set the licensing mode. The licensing mode determines how the LNS Object Server will track the addition of devices to a network by your application. Remote Lightweight client applications must always operate in Standard Mode. To enable the Standard Mode, set your customer identification information by calling the `SetCustomerInfo()` method on the `ObjectServer`, as shown below:

```
ObjectServer.SetCustomerInfo(CustomerID, CustomerKey)
```

The `CustomerID` and `CustomerKey` parameter supplied to the function refer to the customer identification and key numbers that are printed on the back cover of the LNS Application Developer's Kit CD-ROM jewel case. For more information on LNS licensing, and for more details on the differences between Standard Mode and Demonstration Mode, see Chapter 13, *LNS Licensing*.

Opening the Object Server

Once you have set the network access mode, and licensing mode for your Lightweight client application, you can open the Object Server, as shown below:

```
ObjectServer.Open()
```

NOTE: If you will be opening any networks with an LNS application that is running as a Windows service, then the first application to open the LNS Object Server must also be running as a Windows service. In addition, if a network is to be opened by an LNS application that is running as Windows service, then that network and system must be opened by an LNS application that is running as Windows service before it is opened with an LNS application running as a user process. For more information on this, consult the help pages for the `Open()` methods of the `Network` and `ObjectServer` objects in the *LNS Object Server Reference* help file.

Opening a Network

This section describes how to connect to a network with a Lightweight client application. Lightweight client applications cannot create new networks. The LNS Server utility must be running on the PC containing the LNS databases (i.e. the LNS Server PC) to open a network with a Lightweight client application. To connect to an existing network with a Lightweight client application, follow these steps:

1. Retrieve the `Networks` collection from the `ObjectServer` object's `Networks` property. The `Networks` collection will contain all Lightweight client networks that have been entered into the Windows Registry on the PC running your application (i.e. all the networks that have previously been opened on your PC).

   ```
   Dim MyNetworks as LcaNetworks
   Set MyNetworks = ObjectServer.Networks
   ```

2. If this is the first time the network will be opened on the PC running your application, use the `Networks` collection object's `Add()` method to save the network's name, IP address, and port (identified by the database path) into the Windows Registry on the remote PC. In this case, the final `createDatabase` parameter you supply to the `Add` method must be set to `True`.

   ```
   Dim MyNetwork as LcaNetwork
   Set MyNetwork = MyNetworks.Add("Building 75", _
            "lns://bldgServer.acme.com:2540", True)
   ```

   Note that the database path supplied to the `Add()` method contains a URL containing the LNS Lightweight client protocol name ("lns://"), the server's IP address ("bldgServer.acme.com", or "10.1.2.3" for a fixed IP address), and the port on which the LNS Server listens for Lightweight clients (":2540").

   If the network to be opened has already been opened on the PC running your application, get the `Network` object from the `Networks` collection. You can iterate through the `Networks` collection to list all available networks, or you can specify the network to be opened by its name.

   ```
   Dim MyNetwork as LcaNetwork
   Set MyNetwork = MyNetworks.Item("Building 75")
   ```

3. Call the `Network` object's `Open()` method to open the LNS network database. The LNS Server utility must be running on the PC containing the LNS network database, or the operation will fail.

   ```
   MyNetwork.Open()
   ```

4. Once you have opened a network, you should follow the tasks described in the *Opening a System* section later in this chapter to configure and open the system.

Initializing an Independent Client

As described in Chapter 3, you can use Independent client applications to open networks in server-independent mode. You should note that only networks that have been previously opened by a Local or Full client application using your application's Network Service Device can be opened in server-independent mode. To open a network in server-independent mode, follow these steps:

1. Set the access and licensing modes for the Independent client application and open the `ObjectServer` object as you would with a Local or Full client application. For more information on these tasks, see *Initializing a Local Client Application* on page 48.

   Note that when opening a network with an Independent client application, you cannot select a new network interface, as you can with other client types. LNS will open the network using the same network interface as the last Local or Full client application that opened the network.

2. Select a network from the `ObjectServer` object's `VNINetworks` collection. The `VNINetworks` collection contains all networks that have

been previously opened by Local or Full client applications with the Network Service Device your application is using:

```
Dim IndependentNetworks as LcaNetworks
Dim MyNetwork as LcaNetwork
Set IndependentNetworks = ObjectServer.VNINetworks
Set MyNetwork= IndependentNetworks.Item("Building 75")
```

3. Call the `OpenIndependent()` method on the selected network:

   ```
   MyNetwork.OpenIndependent()
   ```

4. Once you have opened a network with an Independent client application, you can begin performing monitor and control tasks with the application, as described in Chapter 9, *Monitor and Control.*

Opening a System

Once you have completed the tasks described in the previous sections and opened a network or group of networks, you need to open the `System` object for each of your open networks, and set the `System` object's parameters. If an application is running locally, this allows your application to attach to the network and make modifications to the network database. If your application is running remotely, this registers your application with the LNS Server and attaches it to the network, which allows the application to make database modifications. Note that you cannot open the system when operating an Independent client application.

To open the system for Full, Lightweight and Local client applications, follow these steps:

1. Get the `System` object from the active network's `Systems` collection. There is only one system per network, so select the first system in the `Systems` collection.

   ```
   Dim MySystems as LcaSystems
   Dim MySystem as LcaSystem
   Set MySystems = MyNetwork.Systems
   Set MySystem = MySystems.Item(1)
   ```

2. If you are running a Full or Lightweight client application and authentication is enabled for this `System`, set the authentication key. The authentication key is a 12-character string representing a 12-digit hexadecimal value.

   ```
   MySystem.AuthenticationKey = "01FE23DC45BA"
   ```

   Local client applications do not need to set the authentication key for client/server communication. However, Full and Lightweight client applications must, or they will be unable to communicate with the LNS Server. You can provide additional IP security for Lightweight clients by setting IP permissions on the LNS Server PC via the `System` object's `PermissionString` property.

3. If the `System` is being opened for the first time and can share media with other independently managed systems, configure the system to use shared media by setting the `InstallOptions` property to `lcaSharedMedia`. The default for this property is private media (`lcaPrivateMedia`). The shared media setting is typically required for systems with power line, RF, or LONWORKS/IP channels.

   Setting the `InstallOptions` property after you have initially opened a system has no effect. You can only set this property with a Local client application before opening the system for the first time. You cannot set this property with a remote client application.

   When the `lcaSharedMedia` option is specified, the LNS ObjectServer assigns a unique 6-byte value to the `DomainId` property that is based on the Neuron ID of the Network Service Device that the LNS Server is using. It also disables background discovery by setting the `DiscoveryInterval` property to zero, disables background device-pinging, and disables automatic registration of devices when service pin messages are received. This is to avoid registering devices from other systems. For more information on shared media, see *Using Shared Media* on page 167.

   ```
   MySystem.InstallOptions = lcaPrivateMedia
   ```

4. If the system is being opened for the first time and you are not using shared media, set the system's domain ID. Domain IDs can be 1, 3, or 6 bytes in length and are represented using 2 hexadecimal characters per byte. If shared media was not selected and no domain ID is specified, the LNS Object Server will set the domain ID length to 1 byte and the value to 01. The following code shows a 3-byte domain ID.

   ```
   MySystem.DomainId = "32A0CF"
   ```

   NOTE: You can change the domain ID after opening the system. If you plan on using the engineered mode installation scenario described in Chapter 5 of this document, you do not need to set the domain ID until the commissioning stage. This allows a single database to be defined that is commissioned with many systems, each with a different domain ID.

5. For Full client applications, set the `System` object's `RemoteChannel` property to the `Channel` object corresponding to the remote PCs channel.

   You do not need to perform this step if you are running a Local or Lightweight client application, if the system only contains one channel, or if the system contains multiple channels and only uses configured routers, bridges or physical repeaters. This step is only necessary for remote Full client applications attached to multiple channel networks containing store and forward repeaters.

   For those client applications that do require this step, it is important to note that the `Network` object's `Channels` property is not valid until the `System` has been opened. As a result, since the `System` object's `RemoteChannel` property must be set before the `System` is opened, you must pre-open the `System` to obtain the desired channel, set the `RemoteChannel` property, and then re-open the `System`. This is demonstrated in the following example:

```
Dim MyChannels as LcaChannels

' Pre-open the system:
MySystem.Open()

' Fetch and assign the desired channel
Set MyChannels = MyNetwork.Channels

' Set the RemoteChannel property
Set MySystem.RemoteChannel = MyChannels.Item("C2")
```

6. Invoke the `System` object's `Open()` method to open the `System` (or re-open the system, if you set the `RemoteChannel` property in step 5).

```
MySystem.Open()
```

Setting System Parameters

System parameters are represented as properties of the `System` object. You should set some of these properties before opening the `System` object, as described in the previous section. You should set other key properties of the `System` object after you have opened it, as follows:

1. `MgmtMode` Property. This property sets the system management mode, which determines whether device configuration changes are propagated to the devices (`lcaMgmtModePropagateConfigUpdates`) on the network as they are applied to the LNS database, or saved for later processing (`lcaMgmtModeDeferConfigUpdates`). For a new network, the default setting is `lcaMgmtModeDeferConfigUpdates`. If you are opening an existing network, this setting may have been changed by another LNS application.

   While the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, changes are stored in the LNS database. They are automatically propagated to the network as soon as the system management mode is changed back to `lcaMgmtModePropagateConfigUpdates`. While the system

management mode is set to `lcaMgmtModePropagateConfigUpdates`, all device configuration changes are immediately propagated to the network.

Note that the system management mode is global, and affects all clients currently attached to the system. For more information on the system management mode, see *System Management Mode Considerations* on page 97.

```
MySystem.MgmtMode = lcaMgmtModePropagateConfigUpdates
```

2. `DiscoveryInterval` Property. If the shared media option was not selected when the system was opened, set the discovery interval. The discovery interval specifies the rate, in seconds, at which the LNS Object Server will scan the network for unconfigured devices that have been attached to the network. The default setting for this property is 180 if the `InstallOptions` property is set to `lcaPrivateMedia`, and 0 if the `InstallOptions` property is set to `lcaSharedMedia`. The setting may have been changed by another LNS application if you are opening an existing network.

   Setting the interval to 0 disables discovery. You should set this property to 0 unless the control network is expected to be highly dynamic, and your LNS application is programmed to respond to the automatic registration of new devices.

   Because this property affects all current and future clients accessing this network, Echelon recommends that you specify an appropriate discovery interval only when you create the network and open the `System` for the first time.

   ```
   ActiveSystem.DiscoveryInterval = 600
   ```

3. `UpdateInterval` Property. This property determines how often the LNS Object Server will try to complete operations such as device commissioning that occurred as a result of a device update failure. A device update failure occurs when a transaction has been completed and committed to the LNS database, but LNS is unable to load the information into the physical device due to some error. The default value for this property is 120 seconds. Note that you can force the LNS Object Server to complete such operations at any time by calling the `RetryUpdates()` method on the `System` object. Setting this property to 0 disables automatic retries, and requires explicit calls to the `RetryUpdates()` method.

   ```
   MySystem.UpdateInterval = 240
   ```

   Because this property affects all current and future clients accessing this network, Echelon recommends that you specify an appropriate update interval only when you create the network and open the `System` the first time. It is recommended to set this property to 0 only when you require explicit control of bandwidth usage in low-bandwidth networks.

Steps 1 through 3 list several key properties of the `System` object that you need to set when initializing your application, but there are many others you should consider as you program your application. You should also note that many properties of the `System`

object are not available until the system has been opened. See the *LNS Object Server Reference* help file for a complete list of the properties of the `System` object, and descriptions of those properties.

Once you have opened the `System` object and set its parameters as you desire, you can begin programming your application to perform monitor and control operations, or to perform network management tasks. Echelon recommends that you review the rest of this chapter before proceeding to these tasks.

Using Transactions and Sessions

LNS provides two ways to group sets of database modifications: transactions and sessions.

Transactions allow a set of database and network modifications to be treated as an atomic operation. This allows sequences of changes to be canceled on the network and in the LNS databases if a failure occurs during any part of the sequence. You can also use transactions to substantially speed up operations if you group multiple changes within a single transaction. Unless operations must be kept separate or are not allowed within a transaction, transactions should be used to optimize performance and allow for atomic roll back of the operation sequence.

Sessions allow certain types of database and network modifications to be grouped and executed together, without requiring verification that each property write and method invocation was valid until all operations have completed. This allows your application to create connections more efficiently, and to avoid failure scenarios that can occur when devices or routers are moved or changed one step at a time.

Managing Transactions

Transactions can be explicitly managed by an LNS application using the transaction methods provided by the LNS Object Server, or implicitly managed by the LNS Object Server. If an application does not explicitly start a transaction, the LNS Object Server automatically starts one when the LNS application invokes a method that causes a database modification to two or more objects, and commits the transaction when the service completes. This is called an *implicit transaction*.

Implicit transactions are easy to use because they are transparent to the application. Explicit transactions are useful when you want to group multiple actions into a single operation. For example, you may want to treat installing a device and connecting that device to a set of other devices as one indivisible action. Or, you may want exclusive access to the services you are performing. Only one client can use a transaction at a time. Thus, if the user cancels part of a transaction prior to completion, the application can cancel the entire transaction, returning the database to the same condition it was in prior to the invocation of any of the services. Grouping several related actions as a transaction improves performance in many cases, since device updates are not sent out until the transaction is committed. The performance improvements caused by using explicit transactions can be significant, so use them whenever possible, particularly if you are creating multiple objects of any type or if you are iterating through a collection. For more information on using transactions with collections, see the *Using Transactions With Collections* section later in this chapter.

To explicitly start a transaction, call the `System` object's `StartTransaction()` method. Once a transaction is started for an application, all network modifications invoked by that application will be considered part of the transaction. To end the transaction and save the changes, call the `CommitTransaction()` method.

To abort the transaction and cause the LNS database to revert to its state prior to the start of the transaction, the application can use the `CancelTransaction()` method. Explicit transactions and some implicit transactions can be canceled while they are in progress from within the `OnSystemNssIdle` event handler. For more information on the `OnSystemNssIdle` event handler, see *Using the OnSystemNssIdleEvent* on page 316.

If a transaction is interrupted prior to completion for any reason, the LNS databases and the network are returned to their states prior to the start of the transaction. If an application starts a transaction, then it should commit or cancel the transaction within a reasonable time to avoid hanging other applications. No application can start a new transaction until the current one is committed or cancelled. While a transaction is in progress, LNS automatically queues requests to start either additional implicit or explicit transactions. If an application shuts down while it has an outstanding transaction, and another application attempts to start a transaction, the LNS Object Server will abort the abandoned transaction within 30 seconds. Once a transaction is committed or canceled, other transactions on the system can begin.

This behavior can cause problems when debugging your application. For example, LNS may cancel a transaction because your application has started a transaction and is halted in a breakpoint. You can disable this behavior by setting the following Windows Registry DWORD entry to a non-zero value:

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\NSS\Configuration\Transaction
Debugging
```

Set its value to 0 (zero), or remove the entry, to disable the transaction debugging mode. You should only modify this Registry entry for debugging purposes, since disabling this feature may cause all LNS clients to lock up if the value is changed, and an application running a transaction is improperly terminated.

Monitoring and Transactions

Transactions may be used when working with monitor and control-related methods and properties, just as with other LNS operations. However, it is important to realize that some properties used for monitor and control operations are not stored in the LNS database persistently. As a result, it is not possible for the LNS Object Server to rollback changes to these properties if a transaction is cancelled.

For more information on monitor and control, see Chapter 9 of this document.

Using Transactions With Collections

In many cases, you will need to iterate through the objects in a collection. LNS provides several methods and properties you can use to do so for most collections, such as the `Item` property and the `ItemByHandle()` method. The `Item` property allows you to retrieve an object by specifying the object's name or collection index, and the `ItemByHandle()` method allows you to retrieve an object by specifying its handle.

Echelon recommends that you use the handle or name assigned to an object to retrieve it from a collection. The handle may be most desirable, since it is a static, unique value. However, if you need to iterate through a collection using index numbers to retrieve each object, you should do so within a transaction. If another client application adds or removes an object from a collection while your application is iterating through it, you may encounter invalid or repeated data as LNS adjusts the indices assigned to other objects in the collection. Exceptions you might encounter in this situation are the `LCA:#96 lcaErrObjectDeleted`, `LCA:#15 lcaErrInvalidCollectionIndex`, and `LCA:#6 lcaErrObjectNotFound` exceptions. If you iterate through the collection within a transaction, your application will maintain a consistent view of the collection during the transaction, and it will not be affected if other client applications modify the collection. In addition, using a transaction will result in better performance.

Managing Sessions

A given client can have at most one session in progress at a time. A session must be part of an explicit transaction, and each transaction can contain more than one session. Changes made in a session will not be committed until the transaction that contains the session is committed. Sessions can be used to group operations that are within the same transaction.

To begin a session within an explicit transaction, call the `System` object's `BeginSession()` method. This method takes one argument, `sessionClass`, which must be set to `lcaSessionMove`. Once a session is started, certain methods and properties may be invoked, and these methods and properties will be considered part of the session until the session is ended.

Consider a case where you are moving a large number of devices with subnet broadcast connections from one channel to another, using the unacknowledged repeat messaging service. If only some of these devices are moved, your application would detect that not all of the devices are on the same subnet and the move would fail (since unacknowledged repeat service for domain wide broadcast is not allowed). By grouping the moves in a session, the actual connecting and validation does not take place until the session ends, and so the network configuration is not affected if the operation fails.

The only network operations you should perform within sessions are those related to changes in the physical topology of your network. This includes moving devices and routers, adding and removing routers, and setting router classes. As a result, the methods you can use within a session include the following methods of the `AppDevices` and `Routers` collections: `PreMove()`, `MoveEx()`, `PostMove()`, `Add()`, and `Remove()`. You can also write to the `Class` property of a `Router` object within a session.

To end a session, call the `EndSession()` method. This method also takes the `sessionClass` argument, which must be set to `lcaSessionMove`. When this method is called, LNS checks for any error conditions that may have resulted from all method and property calls that were made since the `BeginSession()` method was called, and applies the changes caused by those calls to the LNS database.

Event Handling

LNS uses *events* to inform the application of a variety of network occurrences, such as the arrival of service pin messages, or changes to the network's configuration. You can

subscribe your application to most events by invoking the `Begin<event>` method (where `<event>` represents the name of the event) for the desired event. Each separate application must subscribe to an event to receive that event (i.e. one client application subscribing to an event will not cause other client applications to receive that event). The source of a stream of events is called the *event generator*. An application that subscribes to a stream of events is called an *event subscriber*.

LNS implements a core set of events for installation and for monitoring and control tools. These events are described below. The rest of this document describes when certain events might be useful. You can also find out more about these events in the *LNS Object Server Reference* help file.

- *Service pin events*. The `OnSystemServicePin` event is generated when a service pin message is received. This event is used in several of the installation scenarios described in Chapter 5.
- *Change events*. These events are generated whenever devices, routers, channels, or subnets are added, deleted, moved or renamed, or when device interfaces are modified.

  Examples of change events include the `OnChangeEvent`, which is fired any time an object in the LNS database is modified, and the `OnNodeIntfChangeEvent`, which is fired every time a device's external interface is modified.

  Note that some external interface change events are fired when the change is made to the database and others are fired as the changes are propagated to the devices. See the online help for the `OnNodeIntfChangeEvent` event for more information on this.
- *Update and update failure events*. These events are generated when a monitored network variable or message point is updated, or when an LNS application fails to update a network variable or message point. Examples of update events include the `OnNvMonitorPointUpdateEvent`, which is fired every time a network variable monitor point update is received, and the `OnMsgMonitorPointUpdateEvent`, which is fired every time a message monitor point update is received. Update failure events include the `OnNvUpdateErrorEvent` and `OnNvMonitorPointErrorEvent` events.
- *Commissioning status change events for devices and routers*. The `OnCommission` event is generated whenever the commission status of an application device or router changes. Whenever a change that affects the configuration of a device or router is made to the LNS database and that device or router's `CommissionStatus` property is set to `lcaCommissionUpdatesCurrent` (indicating that the device has no pending configuration changes), LNS will generate an `OnCommission` event to indicate that the device or router now has pending updates (and the `CommissionStatus` property will be set to `lcaCommissionUpdatesPending`).

  The `OnCommission` event will also be generated whenever the LNS Object Server propagates, or fails to propagate changes to a device or router.

  This event is useful if you want to initiate operations as soon as a device's network image is up-to-date. For example, you might use it to determine when to set a device online after its network image has been updated by

an automated installation tool. You could also use it to provide feedback to a user about the configuration state of devices or routers on your network. For example, when building a connection, you might want to be informed of devices that could not be updated. Or, upon receiving a commission status change event with an "update failed" code, your application could indicate this by drawing a red circle around the device's icon. Later, when the device is reattached to the network and the Object Server refreshes the device's network image, your application can remove the red circle upon receiving a commissioning status change event with an "update succeeded" code.

- *Attach/detach events*. The `OnAttachment` event is generated when the attachment status of an `AppDevice` or `Router` changes. Once an application device or router has been commissioned, it can be monitored via periodic pinging to ensure it is attached. A change in the attachment status results in this event being generated. Note that the attachment/detachment events can also be used to monitor the presence of other remote Full client applications on the network.
- *Licensing events*. You can use the `OnLicenseEvent` event to monitor when the licensing status of the LNS Server changes. This event is generated upon device crediting, device debiting, deficit credit usage, and license expiration.
- *Missed-event events*. The `OnMissedEvent` event is generated for Full client applications when one or more events were generated, but not received by subscribers. For example, missed events could occur while your application's Network Service Device is being updated. This event will contain information about how many events were missed and whether the missed events are recoverable. You can use the `SetEventSyncMode` method to determine whether missed events can be recovered or not, and you can use the `DoEventSync` method to maintain event synchronization in the case of missed events. Consult the *LNS Object Server Reference* help file for more information on these methods.

When implementing an event handler for any of the LNS-generated events discussed in this document, the developer should take into account the following guidelines:

- Each event handler must be implemented as an `IDispatch()` method with a valid DISPID. The valid DISPIDs you can use with LNS are included in the `ConstEventIds` constant. For a complete list, see the help page for the `ConstEventIds` constant in the *LNS Object Server Reference* help file. LNS does not query your event sink object for DISPIDs by name. This is usually handled automatically by the application framework (e.g. ATL, MFC, Visual Basic) you are using.
- When not using direct callbacks, your application must service its Windows Message Queue in order to receive events. In addition to not receiving events, your application will appear to leak memory while the queue is not serviced. This is usually only a concern for console or Windows service applications. For more information on direct callbacks, see *Multi-Threading and LNS Applications* on page 318.
- Each event handler should process the event, and return quickly. New events cannot be processed by a client until the current event handler returns.

- Event handlers should not try to release any event object parameters, per COM rules. LNS will handle this itself on return from the event handler. If a client needs to make a copy of an object parameter, it must `AddRef()` the object to ensure it remains valid. For more information on this, see *Avoiding Memory Leaks with LNS* on page 319.
- Where possible, event handlers should avoid making new calls into LNS (especially modifying calls), except to extract information from any passed-in object parameters. This should not cause direct problems, but no further events will be processed until the event handler returns when using direct callbacks. As an alternative, you can post a Windows message to your main thread for further processing.
- Transactions within event handlers are treated the same as outside event handlers. In particular, any operations performed within in an event handler will become a part of the current transaction.
- Event handlers should not start or terminate any transactions, especially when using direct callbacks, except for cancellation of explicit transactions from within an `OnNssIdleEvent` event handler.
- Event handlers will be executed from one of several non-client threads when using direct callbacks. You may need to keep this in mind for thread safety, e.g. if your application makes use of thread local storage. For more information on direct callbacks and LNS, see *Multi-Threading and LNS Applications* on page 318.
- Applications should not subscribe to an event unless the application is prepared to withdraw useful information from an event, as receiving but not responding to events causes undesirable performance degradation. This can also consume a large amount of network bandwidth, particularly for Full client applications.

Exception Handling

As mentioned in the *LNS Components* section in Chapter 3 of this document, LNS defines a set of exceptions that will be returned when an operation fails for any reason. With the enhanced support for ATL in LNS Turbo Edition, there are now several ways to handle runtime errors and warnings. Most applications have a try/catch clause (or some similar construct) for general exceptions or COM errors, such as 'out of memory' or 'access violation'.

Every property and method in LNS has at least two public interfaces: one that returns an error code, and one that does not. For example, `ILcaAppDevices.get_Item()` returns HRESULT error codes and does not throw exceptions for LNS-specific error conditions, and `ILcaAppDevices.GetItem()` uses exceptions to notify the application of error conditions. If the developer chooses the interface that does not return an error code, then their catch clause will catch and handle the error. When using this type of interface, both errors and update warnings will be thrown as COM exceptions.

If the developer chooses the interface that returns an error code, then the catch clause will not be invoked upon an LNS error or warning for that invocation. Instead, the returned exception will contain an error code, 0x8004yyyy, where yyyy represents an LNS error code (i.e. 0x4269) or an LNS warning code (i.e. 0x0FBE). For a complete list of the exceptions LNS may return, see the *LNS Errors Online Reference* section of the *LNS Object Server Reference.*

Some properties and methods normally return an object, e.g. the `Add()` method of the `AppDevices` collection, and some return a value. However, if an exception is generated, the return value has no meaning and is not used, and the returned object (were it to be non-zero) is not valid. Likewise, when you use the interface that returns HRESULT codes instead of producing exceptions, the accessors return data results via pointers (output parameters). In case the HRESULT code indicates failure, the result pointer does not refer to valid data.

Note that when making changes inside of an explicit transaction, the updates are deferred until the transaction is committed. Therefore, update warnings are never thrown or returned when methods are called from within a transaction, but may be thrown or returned when the `CommitTransaction()` method is called. However, warnings or errors related to parameters provided to a method within the transaction may be reported before you call the `CommitTransaction()` method. For example, calling the `AddTarget()` method with a bad target reference parameter may result in an immediate error condition.

You also need to consider the pairing requirements for open and close statements, and `StartTransaction()` and `CommitTransaction()` statements. If an open operation was successful, you should always make sure to invoke the matching close operation. However, if the open failed, it may not make sense to perform the matching close operation (although the close operation should not cause any problems in this case). Transactions have a different requirement. After the `StartTransaction()` method has been called, your application will eventually need to invoke the `CommitTransaction()` method or the `CancelTransaction()` method. You should note that if the call to `CommitTransaction()` fails, you still need to invoke the `CancelTransaction()` method to cancel the transaction.

Terminating an LNS Application

You should terminate any LNS application, regardless of client type, by following these steps:

1. Stop all monitoring and control tasks. If using permanent monitor sets, those `MonitorSet` objects should be closed. For more information on monitor and control, see Chapter 9, *Monitor and Control.*
2. If you are operating in server-dependent mode (i.e. not as an Independent client), close each open `System` object by invoking its `Close()` method. If the application is a Local or Full client, this detaches the LNS Object Server from the network. If the application is a Local client, and the LNS Server application was automatically launched by an LNS application, this also shuts down the LNS Server application. For Lightweight clients, this method de-registers the application from the LNS Server.

   ```
   MySystem.Close()
   ```
3. Close each open `Network` with the `Close()` method. If you are running an independent client, use the `CloseIndependent()` method.

   ```
   MyNetwork.Close()
   ```
4. Close the ObjectServer control by invoking its `Close()` method.

   ```
   ObjectServer.Close()
   ```

Chapter 5 - Network Management : Installing a Network

This chapter describes how you can use LNS to install and configure a LONWORKS network. This includes descriptions of three installation scenarios you can use: automatic installation, ad hoc installation, and engineered mode installation.

LNS Network Installation Scenarios

To understand what is required to install LONWORKS devices on a LONWORKS network, you should consider the types of control systems that LONWORKS networks replace. Many conventional control systems use wiring harnesses or point-to-point wires. In these systems, the wiring between devices serves two purposes when the devices are installed. It physically interconnects the devices, and it determines which control signals should be sent to which device. Once attached to the wire, the behavior and interaction among the devices is completely defined.

Other control systems use a master-slave architecture, and require DIP switches or dials on each device to specify the device's address. The device addresses are predefined and based on the control algorithm in the master. When these devices are installed, the master polls each address, and the appropriate device responds. Such systems are usually limited to a small number of devices, and changes to system behavior usually require resetting the DIP switches on each device, and modifying the master control software.

A LONWORKS network consists of intelligent devices called *nodes* or *application devices* that are connected by one or more communications media. Application devices communicate with one another using the LONWORKS protocol (also referred to as the LonTalk protocol). Each intelligent device on the network, e.g. a programmable thermostat in a building control system, is a LONWORKS application device. The devices communicate with one another across a shared communications medium, such as a twisted pair cable, a power line circuit, or an RF link. Figure 5.1 shows the wiring difference between a conventional system and a LONWORKS network.

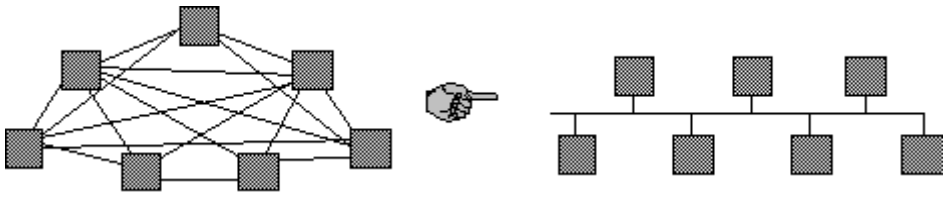


Figure 5.1 Wire Reduction in a LONWORKS Network

The devices on a LONWORKS network contain objects that respond to a variety of inputs, and produce desired outputs. Although the function of a given device may be quite simple, the interaction among devices allows LONWORKS networks to perform complex tasks. A benefit of LONWORKS networks is that a small number of common device types can perform a broad spectrum of different functions, depending on how they are configured and logically connected.

In a LONWORKS network, devices share their physical media (e.g. twisted-pair wire or a power line circuit), which eliminates the redundant point-to-point wiring found in conventional control systems. Without point-to-point wiring (e.g. a light switch wired to a lamp) the physical attachment no longer uniquely identifies a device. In a LONWORKS network, the physical attachment only provides a path for devices to send and receive messages. It does not tell the devices which other devices they should send data to. Therefore, in addition to physically attaching the devices to the network, you also need to perform the following tasks when installing a LONWORKS network:

- Assign a network address to each device. A network address identifies which application device a LONWORKS messages should be sent to, just as a postal address identifies which house a letter should be delivered to. A

device's network address consists of three components — the device's domain, the device's subnet, and the device's subnet ID. The LNS Object Server is responsible for assigning each device a unique network address when the device is installed.

- Define the information that devices share with one another. Devices communicate with one another using high-level objects called network variables, or low-level messages. Interoperable network devices send messages using implicit addressing for network variable updates and application messages. When using implicit addressing, the Neuron Chip firmware on the application device builds and sends network variable and application messages using information contained in tables in its EEPROM. In order to send application messages in this fashion, the device application specifies a message tag when sending the message. The message tag is associated with an address table entry stored in the device's EEPROM.

  When an LNS application requests that a device share information with another device, an address table entry is allocated and configured on the device sending the information. This address table entry associates the output defined by the device application (either a network variable or a message tag) with the domain/subnet/node address, group address or broadcast address of the device or devices receiving the information. The process of creating and configuring these tables is called binding or connecting. The addressing established during this process is called a connection. The LNS Object Server is responsible for allocating the network resources used by connections.
- Set site-specific parameters. LONWORKS technology provides the flexibility to customize and tune network behavior and response characteristics, if required by the system. For example, network performance can be fine-tuned by assigning devices to priority slots on a channel. You can assign these priority slots with LNS. You can use LNS to further customize devices by setting application-specific information such as location, temperature set points, and calibration tables.

Installation Scenarios

The first step in writing an LNS application to install a LONWORKS network is to select the installation scenario or scenarios that the application will support. Based on your knowledge of the network and the capabilities of the installation personnel, you must decide what steps the installer will go through to add devices and build connections, how much flexibility will be required, and what tasks can be automated by your LNS application. Once you choose an installation scenario, you can map the scenario to the required objects in the LNS database, and add intelligence to the application to automate tasks as appropriate.

The installation scenario you use to install your network determines the "look and feel" of the network as viewed by the person responsible for network installation. The best scenario for any given network depends on many factors, including the skill level of the installer, the amount of flexibility desired, and the requirements of the end-user. In all cases, the installation process should be automated as much as possible. Automation both simplifies and speeds network installation.

The three installation scenarios are automatic installation, engineered mode installation, and ad hoc installation. Note that you can install a network using a mix of these

scenarios. For example, you could begin defining a network's devices and connections using the engineered system scenario. Once the network is commissioned, you could add additional devices to it using the ad hoc scenario.

Engineered Mode Installation

In the engineered mode installation, installation is a two-step process consisting of a definition phase and a commissioning phase. In the definition phase, the application defines all of the network configuration information in the LNS database, without modifying the physical network. In the commissioning phase, the application loads the configuration information defined during the definition stage into the physical network.

The advantage of the engineered mode installation scenario is that the network installation on-site is quick, easy, and error free, since most of the time-consuming data entry and processing is done off-site. This scenario is often used when installing systems that require preplanning, when building multiple clones of a single network design, or when installing systems that are built in response to a bid.

For more information on engineered mode installation, see *Engineered Mode* on page 77.

Ad Hoc Installation

In the ad hoc installation scenario, installation is a one-step process. In this scenario, a network tool loads the network configuration information into each device as the devices are physically installed on the network, and then creates connections between them. The LNS application provides a user interface that controls the amount of information and the sequencing of the information required.

This is different from the engineered system scenario in that information is loaded incrementally. It is different from the automatic installation scenario in that the installer makes decisions about the network configuration instead of the tool (although the tool may provide assistance). The goal of ad hoc installation is to integrate all installation activities into a single step. Ad hoc installation offers the most flexibility, since it allows the installer to make decisions on-site. Since this scenario can be time consuming, it is typically used in conjunction with the engineered system scenario for large installations. Ad hoc is the installation scenario typically used when servicing an existing network.

Automatic Installation

Automatic installation is usually accomplished by an onsite network tool with a minimal user interface. The network tool automates all installation tasks, so it must discover when new devices have been attached to the network, and form network variable and message tag connections between the discovered devices in a way that makes sense in the context of the network design. The network tool must also be able to determine when devices have been removed from the network, and reconfigure the network accordingly.

To be able to automate installation and eliminate or minimize end-user interaction, the network tool's application program needs to be very knowledgeable about the system it is managing. For this reason, automatic installation is most commonly used in single vendor systems, or in systems that are dedicated to a single function.

For more information on automatic installation, see *Automatic Installation* on page 89.

Engineered Mode

In the engineered mode installation scenario, the network installation consists of two phases:

1. A *definition phase* in which the user defines the configuration of the devices and connections on the network in the LNS database without physically modifying the network.
2. A *commissioning phase* in which the application loads the network configuration created during the definition phase from the LNS database into the physical devices on the network. Application images may also be loaded during this phase, and some configuration properties may be adjusted based on testing in order to properly calibrate the system.

Definition Phase

In the definition phase, the network configuration is defined off-site and loaded into the LNS database, without modifying the physical network. The LNS application defines the user interface that is used to enter the information. The interface can be anything from a simple keypad, to a graphic display. The amount of information that the user must enter is under the control of the application. In the most general case, the user could enter all the information related to the network, including what devices will be installed on the network, and how they will be connected. Alternatively, the application could automate certain parts of the definition process. For example, the user may only need to pick a configuration from a list of options, or the user may only need to enter device configuration information while the tool automates connections.

The steps you need to follow to accomplish the tasks involved in the definition phase are shown in figure 5.2. They are described in more detail in the section following figure 5.2.

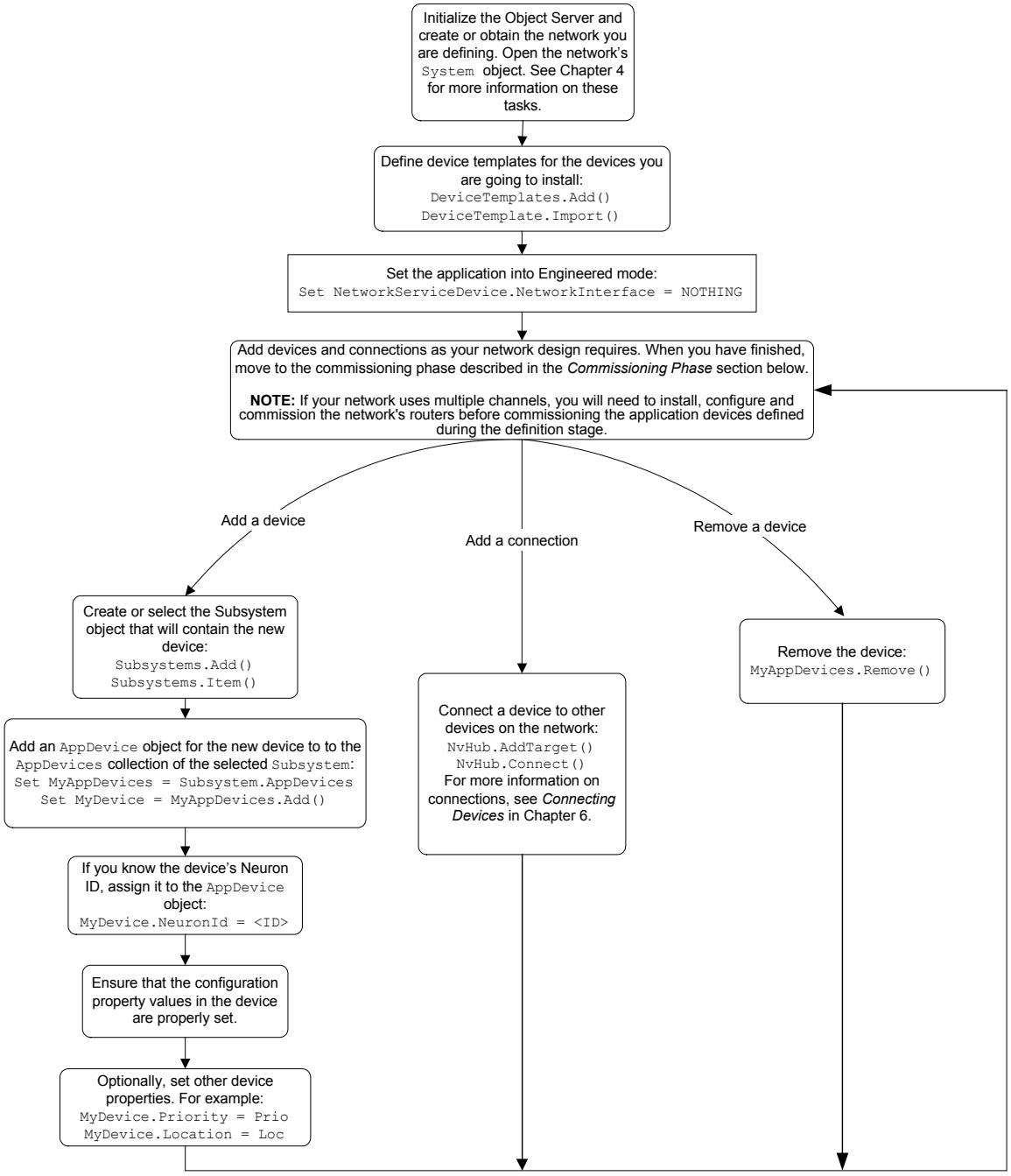


Figure 5.2 Engineered Mode Installation Tasks - Definition Phase

The following section describes the steps listed in figure 5.2 in more detail:

1. Initialize your application, create the network that is to be installed, and open the system. These tasks are described in Chapter 4, *Programming an LNS Application.*

2. The definition phase can be performed while your application is in engineered mode, meaning that it is not attached to the network. In this case, your application does not need to specify a network interface.

Chapter 4 describes how to specify a network interface for each client type. If your application has already done so, you can switch the application to engineered mode by setting the `NetworkInterface` property of the system's `NetworkServiceDevice` object to `NOTHING`.

```
Dim MyNetworkServiceDevice as LcaNetworkServiceDevice
Set MyNetworkServiceDevice = System.NetworkServiceDevice
Set MyNetworkServiceDevice.NetworkInterface = NOTHING
```

3. Begin defining the devices and connections on your system. To do so, fetch the `Subsystems` collection from the `System` object, and use the collection's `Item` property to obtain the `Subsystem` you plan to add the devices to. Alternatively, you can use the `Add()` method to create a new `Subsystem`.

   The returned `Subsystem` object contains an `AppDevices` collection you can add the new devices to. You can add new devices using the collection's `Add()` method.

   When using the engineered mode scenario, you must specify a `DeviceTemplate` and `Channel` object when you call the `Add()` method. If you do not specify a `DeviceTemplate`, you will not be able to pre-define the device's connections, and or pre-load its configuration properties. You must specify the `Channel` object so the LNS Object Server can properly assign network addresses and compute transaction timers for the device when it is added to connections. For more information on creating `AppDevice` objects, see *Creating AppDevice Objects* on page 113.

   NOTE: If your network uses multiple channels, you will also need to define the network's routers and channels during the definition stage. For more information on this, and on other considerations you will need to make when managing a network with multiple channels, *see Managing Networks with Multiple Channels* on page 169.

4. If you know the Neuron ID that the device will use, enter it into the database by writing the Neuron ID to the `AppDevice` object's `NeuronId` property.

   ```
   MyAppDevice.NeuronId = DeviceNeuronId
   ```

   It is common to obtain the devices' Neuron ID from a sticker on the device that contains the ID in barcode and/or readable form. In a typical scenario, the on-site installation staff mounts and wires the physical devices. While doing so, each device's Neuron ID sticker is taken from the device, and attached to a floor plan or similar document. That document is then used during the definition phase to provide Neuron IDs to the LNS database.

   Note that the Neuron ID may also be acquired via the device's service pin and assigned during the commissioning phase, as described in the next section.

5. Ensure that all configuration properties are properly set, and that they will be downloaded to the device when it is commissioned. You can do this by calling the `DownloadConfigProperties()` method on the

device with the `lcaConfigPropOptLoadValues` and `lcaConfigPropOptSetDefaults` options set.

Then, set individual configuration properties as desired using configuration property data points. Note that some configuration properties may need to be adjusted (or calibrated) during the commissioning phase as portions of the system are brought online and tested. For information on setting configuration property values, see *Writing Configuration Property Values* on page 128.

6. Set the `Priority` and `Location` properties of the `AppDevice` object to their desired values. Optionally, set any other properties of the `AppDevice` object, as described in *Configuring Devices* section on page 125.

7. Repeat steps 3, 4, 5 and 6 until you have defined all the devices you plan on adding to the network. Then, create connections between the network variables on those devices as your network design requires. For detailed information on creating connections and connection management, see *Connecting Devices* on page 138.

8. Once you have completed these tasks, you are ready to move onto the commissioning phase of the network installation. For more information on this, see the next section, *Commissioning Phase*.

9. Figure 5.2 references additional tasks you may need to perform when managing a network installed with this installation scenario, such as removing devices and connections from the network. For details on how to perform these tasks, see *Other Device Management Operations* on page 129.

NOTE: The following methods cannot be invoked while the LNS Object Server is in engineered mode:

- `AppDevice.Load()`
- `AppDevice.LoadEx()`
- `AppDevice.Test()`
- `AppDevice.UploadConfigProperties()`
- `AppDevice.Wink()`
- `System.DeconfigNetwork()`

The following properties cannot be read while the LNS Object Server is in engineered mode:

- `AppDevice.SelfDocumentation`
- `AppDevice.State`
- `AppDevice.DetailInfo`
- `NetworkVariable.SelfDocumentation`
- `Router.State`
- `RouterSide.State`
- `RouterSide.DetailInfo`
- `NetworkVariable.Value`

Commissioning Phase

During the commissioning phase, the configuration information defined during the definition phase is loaded into the devices on the network. The LNS Object Server must be attached to the network during this phase, meaning that the application must specify a network interface. The application used for this phase can be different than the application used for the definition phase. For example, the application used for the definition phase may provide a more functional user interface than the application used for the commissioning phase. In fact, the application running the commissioning phase could be completely automated. If a different PC is used for the commissioning phase, the network database must be transferred to the new PC and imported into its LNS Object Server. For instructions on this, see *Moving Network Databases* on page 256.

It is common, but not required, for the commissioning phase to be performed locally. A local commissioning tool allows for easy on-site trouble-shooting. However, the commissioning phase can be performed remotely when using network interfaces such as the SLTA-10, the *i*.LON 10 Ethernet Adapter or the *i*.LON 100 Internet Server. Alternatively, you can connect locally to a network containing distant devices using an Internet router such as the *i*.LON 600 LONWORKS/IP Server or the *i*.LON 1000 Internet Server. If you plan to commission your network remotely, it is recommended that you assign each device its Neuron ID during the definition phase.

The methods and tasks you need to perform to accomplish each task in the commissioning phase are shown in figure 5.3.

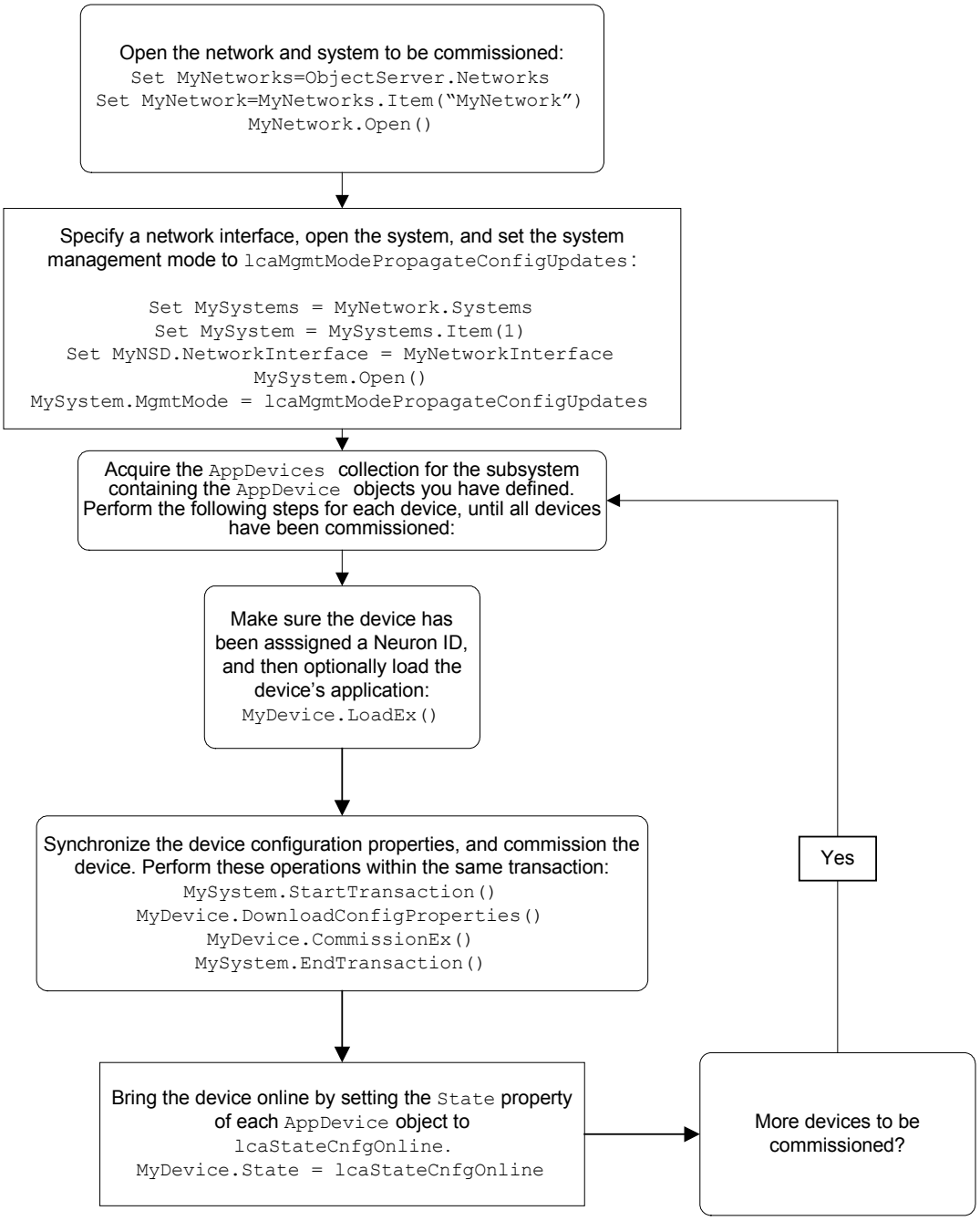


Figure 5.3 Engineered Mode Installation Tasks - Commissioning Phase

The following section describes the steps depicted in figure 5.3 in more detail:

1. If the database was defined on a different PC, copy the database and import it into the PC being used for the commissioning phase. For details on how you can perform these tasks, see Chapter 10, *LNS Database Management.*

2. Open the network and the system, and then set the system management mode to `lcaMgmtModePropagateConfigUpdates`. Before opening the system, make sure that your application has specified a network interface.

```
Set MyNetworks = ObjectServer.Networks
Set MyNetwork = MyNetworks.Item("Broadcasting Center")
Set MySystems = MyNetwork.Systems
Set MySystem = MySystems.Item(1)
Set MySystem.NetworkInterface = MyNetworkInterface
MySystem.Open()
MySystem.MgmtMode = lcaMgmtModePropagateConfigUpdates
```

   NOTE: Generally, there are many factors you need to consider when writing to the `MgmtMode` property. For more information on this, see *System Management Mode Considerations* on page 97.

3. Acquire the `AppDevices` collection for the subsystem containing the `AppDevice` objects you have defined. Then, follow steps 4-8 of this procedure for each device in the collection.

4. If a device was not assigned a Neuron ID during the definition phase, you need to explicitly set the `AppDevice` object's `NeuronId` property at this point. For information on how you can use LNS to determine a device's Neuron ID, see *Neuron ID Assignment* on page 115.

5. If a device does not have the correct application image defined, you should load the device's application image at this point. For instructions on loading device application images, see *Loading Device Application Images* on page 119.

6. Start a transaction, and commission the device with the `Commission()` or `CommissionEx()` methods. Before committing the transaction, you should use the `DownloadConfigProperties()` method to synchronize the configuration property values in the physical device with those stored in the LNS database.

   Alternatively, you could commission the device and then call the `UploadConfigProperties()` method after committing the transaction to synchronize the configuration property values in the database with those stored in the LNS database. For more details on how you can synchronize configuration property values, and for information on other considerations you should make when commissioning devices, see *Commissioning Devices* on page 121.

   NOTE: If your network uses multiple channels, you will need to commission the network's routers as you commission the devices. The *Managing Networks with Multiple Channels* section on page 169 provides details on this, including guidelines on the order you should follow when commissioning routers and application devices at the same time.

7. Bring each device online by setting the `State` property of each `AppDevice` object to `lcaStateCnfgOnline`.

   You may wish to commission all the devices in the network first, and then set all the devices online. This prevents superfluous network traffic generated by those devices that are already online, network delivery errors caused by network variable updates targeted to devices that are not yet online, and generally helps the commissioning process proceed more efficiently.

8. As you manage the network you have created, you may need to perform other maintenance tasks, such as the removal of devices and connections, and the replacement of devices. For details on how you can use LNS to perform these tasks, see *Other Device Management Operations* and page 129.

Commissioning Phase, Multiple Networks

When commissioning multiple networks that have similar configurations, it may be most efficient to use a "cookie cutter" approach. To do so, follow the steps described in the *Definition Phase* section to create a "prototype" LNS database that contains the basic configuration you want to apply to each network. You can then customize individual networks from this proto-type by adding additional information on-site during the commissioning phase, as described below:

1. Initialize your application in engineered mode, meaning that the application explicitly resets the network interface and detaches from the network. Chapter 4 describes how to specify a network interface for each client type.

   ```
   Set MyNetworkServiceDevice.NetworkInterface = NOTHING
   ```

2. Create device templates for the devices on that network by importing external interface files. Then, define each device, adjust its configuration property values, and create connections between the devices.

   For detailed information on how to define devices in engineered mode, see the *Definition Phase* section earlier in this chapter.

3. Create your common reference database by making a backup copy of the network folder. You can use the `Backup()` method to do so. See Chapter 10, *LNS Database Management,* for more information on the `Backup()` method.

4. Rename the database copy created in step 3 as desired, or create a copy of the backup using another directory with a name of your choice.

5. To load the reference database into a new `Network` object, access the `ObjectServer` object's `Networks` collection, and invoke the collection's `Add()` method. Specify the `createDatabase` parameter as `False`, and the `databasePath` parameter as the path to the copy of the reference database.

6. Assign a suitable network interface to the `NetworkServiceDevice` object's `NetworkInterface` property to exit engineered mode and attach the application to the physical network.

7. Open the system by accessing the `System` object for the newly created network, and calling the `Open()` method.

8. Set the `DomainId` property to the domain ID for the site. For shared media, Echelon recommends that you use the Neuron ID of the on-site PC's network interface as the domain ID. For standard network interfaces, this ensures uniqueness as long as the same network interface is not used for multiple installations on the same shared media.

   If the same network interface is used for multiple installations, Echelon recommends using a random domain ID. For networks containing shared media such as powerline or RF channels or shared twisted-pair channels, Echelon recommends using a 6-byte domain ID to ensure uniqueness. For most private

networks, a 1-byte domain ID is sufficient, and allows for optimum performance. The domain ID is automatically selected when the shared media installation option is specified before the system is opened for the first time.

9. Set the `System` object's `MgmtMode` property to `lcaMgmtModePropagateConfigUpdates`.

10. If any of the devices' Neuron IDs have not been specified, acquire and assign them using the methods described in the *Neuron ID Assignment* section on page 115.

11. If a device does not have the correct application image defined, you should load the device's application image at this point. For instructions on loading device application images, see *Loading Device Application Images* on page 119.

12. For each device, start a transaction and commission the device with the `Commission()` or `CommissionEx()` methods. Before committing the transaction, you should you should use the `DownloadConfigProperties()` method to synchronize the configuration property values in the physical device with those stored in the LNS database. For details on how you can do so, and for information on other considerations you should make when commissioning devices, see *Commissioning Devices* on page 121.

13. Add and connect any additional devices that are not included in the basic network configuration defined by the prototype database.

Ad Hoc Installation

In the ad hoc installation scenario, the LNS application loads the network configuration information into the physical devices on the network as the installer defines devices and connections in the LNS database. The LNS application can provide a user interface that controls the amount of information and sequencing of the information required, or it could automate as much of the installation process as desired. The basic tasks you will need to perform during an ad hoc installation are shown in figure 5.4. These are described in more detail in the following sections.

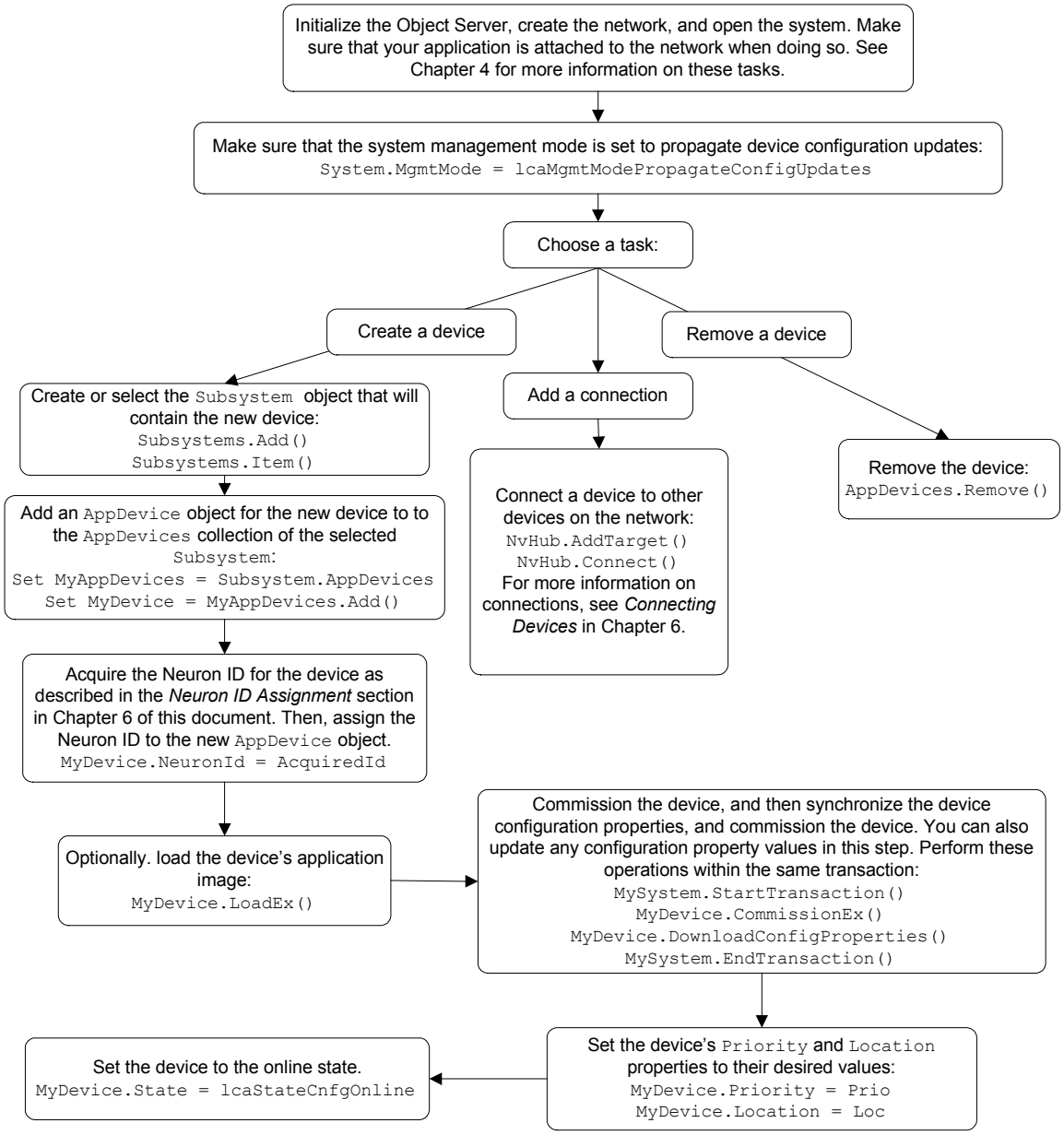


Figure 5.4 Ad Hoc Installation Tasks

The following section describes the tasks depicted in figure 5.4 in more detail. Note that if your network uses multiple channels, you will need to define the network's channels and install and configure the network's routers as you define your devices. For more information on this, and on other considerations you will need to make when managing a network with multiple channels, see *Managing Networks with Multiple Channels* on page 169.

1. Initialize your application, create the network that is to be installed, and open the system. Make sure that your application is attached to the network at this point (meaning that is has specified a network interface). These tasks are described in Chapter 4, *Programming an LNS Application.*

2. You can now begin defining the devices and connections on your network. Fetch the `Subsystems` collection from the `System` object, and then use the collection's `Item` property to obtain the `Subsystem` you want to add the devices to. Alternatively, you can use the `Add()` method to create a new `Subsystem`.

   The returned `Subsystem` contains an `AppDevices` collection. You can add new devices using the AppDevices collection's `Add()` method. When creating an `AppDevice` object with the ad hoc installation scenario, you are not required to specify the device's `DeviceTemplate` and channel. When you commission the device later, the LNS Object Server will automatically detect the device's external interface, create a `DeviceTemplate` object as needed, and query the device across the network for all relevant details.

   However, Echelon recommends that you specify the device template whenever possible, since doing so is more efficient and provides more complete information than creating the device template by reading the information from the device. The LNS Object Server will also automatically detect the channel the device is connected to, and assign a subnet suitable for that channel.

   For more information on creating `AppDevice` objects, see *Creating AppDevice Objects* on page 113.

3. Associate a physical device with the newly created `AppDevice` by assigning its Neuron ID. You can supply the Neuron ID using any of the three methods described in the *Neuron ID Assignment* section on page 115.

4. Load the application images for any devices that require application image loading. To do so, invoke the `Load()` or `LoadEx()` methods on each `AppDevice`. For more information on loading device application images, see *Loading Device Application Images* on page 119.

5. Start a transaction, and commission each device with the `Commission()` or `CommissionEx()` methods. After you commission each device, you should synchronize the configuration property values in the physical device with those stored in the LNS database before committing the transaction.

   If the application's configuration properties definitions are known (meaning that the `ConfigPropertiesAvailable` property of the device's `Interface` is set to `True`), set all values to their defaults by calling the `DownloadConfigProperties()` method with the `lcaConfigPropOptLoadValues` and `lcaConfigPropOptSetDefaults` options set. Then, use data points to update any configuration property values and commit the transaction.

   You can bypass this step if you want to preserve the configuration property values currently stored in the physical device. If you do so, you will need to upload the values from the device after you commission it, as described in step 6. For more information on these tasks, see *Application-specific Configuration Data* on page 125.

   Note that if the `ConfigPropertiesAvailable` property is set to `False`, you cannot update the configuration property values until after this transaction has been committed, and the configuration property definitions have been loaded. This condition occurs if the device's template was uploaded from the device (rather than imported from an external interface file), and the configuration

properties for that template were not uploaded after another device with the same template was commissioned. The next step of this procedure describes how you can upload the configuration properties for the template.

```
MySystem.StartTransaction
MyDevice.Commission()
Set MyInterface = MyDevice.Interface
If MyInterface.ConfigPropertiesAvailable Then
  MyDevice.DownloadConfigProperties _
   (lcaConfigPropOptLoadValues OR lcaConfigPropOptSetDefaults)
End If
MySystem.CommitTransaction()
```

For more information on other considerations you should make when commissioning devices, see *Commissioning Devices* on page 121.

6. If the `ConfigPropertiesAvailable` property was set to `False` in Step 5, or if you bypassed calling the `DownloadConfigProperties()` method in Step 5 to preserve the configuration property values stored in the physical device, upload the configuration property definitions into the LNS database with the `UploadConfigProperties()` method. This will change the `ConfigPropertiesAvailable` property on all devices using this template to `True` (or `False`, depending on the device's implementation).

   Typically, you will also want to upload the values from the device and set the defaults from the device by specifying the `lcaConfigPropOptLoadValues` and `lcaConfigPropOptSetDefaults` options when you call the `UploadConfigProperties()` method. Alternatively, you could update some or all of the configuration property values using data points, and then call `UploadConfigurationProperties()` with the `lcaConfigPropOptLoadValues`, `lcaConfigPropOptSetDefaults` and `lcaConfigPropOptLoadUnknown` options set to upload all values not specifically set, and then set the defaults to the values defined for this device. Note that for optimal performances, you should set all configuration property values in a single transaction. For more information on these tasks, see *Application-specific Configuration Data* on page 125.

7. Set the `Priority` and `Location` properties of the `AppDevice` object to their desired values. Optionally, set any other properties of the `AppDevice` object as described in *Generic Configuration Data* on page 125. You can perform this step in the same transaction as when you commission the device.

8. Bring all the devices online by setting the `State` property of each `AppDevice` object to `lcaStateCnfgOnline`.

9. Figure 5.4 references additional tasks you may need to perform when managing the network you create with the ad hoc installation scenario, such as removing devices, creating connections, and replacing devices. For details on how to perform these tasks, see *Other Device Management Operations* on page 129.

NOTE: Depending on the system management mode, the changes you make to each device may or may not be propagated to the physical device on the network as you invoke each method. For more information on this, see the *System Management Mode Considerations* section later in this chapter.

Automatic Installation

In the automatic installation scenario, an embedded network tool automates installation tasks so that little or no user interaction is required. The steps you will need to take when performing an automatic installation are shown in figures 5.5. The sections following figure 5.5 describe these tasks in more detail.

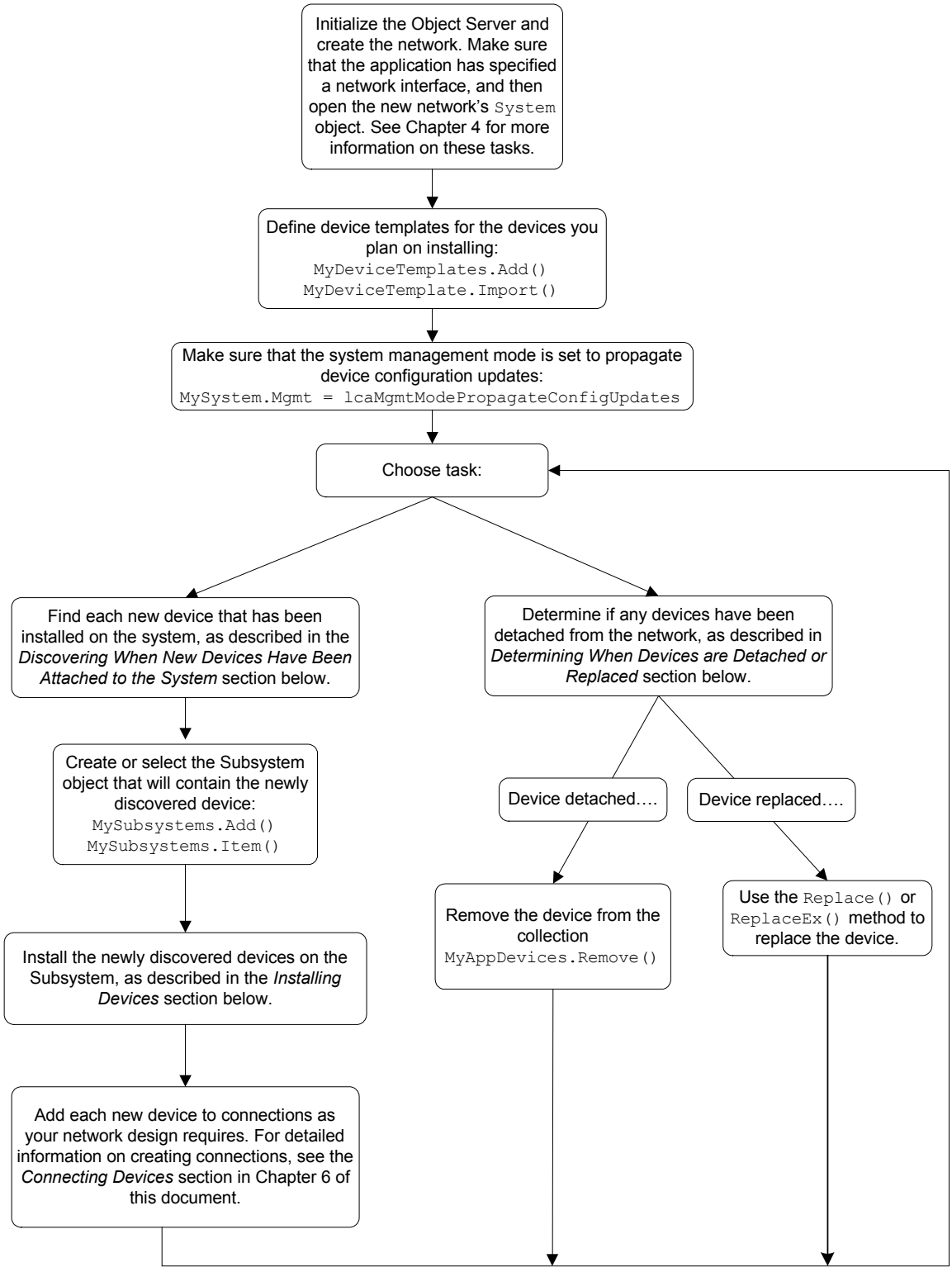


Figure 5.5 Automatic Installation Tasks

There are several main phases involved in the automatic installation scenario depicted in Figure 5.5. These phases are listed below:

1. The first step is to discover the devices that have been added to the network. There are several ways to discover new devices. For more information, see *Discovering When New Devices are Attached to the Network* on page 91.
2. Once you have discovered a new device, you need to define and configure the device in the LNS database, and connect it to other devices on your network. For more information on these tasks, see *Installing Devices* on page 93.
3. You will also need to program your application to determine when devices are removed or replaced on the network, and update the LNS database accordingly. For more information on this, see *Discovering When Devices are Detached or Replaced* on page 96.

Discovering When New Devices are Attached to the Network

The first major step in the automatic installation scenario is for the application to discover the devices that are attached to the network. There are three ways your LNS application can do so:

- Allow the LNS Object Server to discover the devices using the background discovery process. Devices discovered by this process will be represented as `AppDevice` objects in the `Discovered.Uninstalled` subsystem. The `Discovered.Uninstalled` subsystem is managed by LNS. The background discovery interval determines the time it will take for the LNS Object Server to find a device. The default for this interval is 180 seconds, and you can change this value by writing to the `System` object's `DiscoveryInterval` property. If a device is configured, the background discovery process will ignore it. Therefore, the background discovery process will not discover configured devices that you move from one network to another. For this reason, device designers usually provide a mechanism within an application that will deconfigure a device when it is moved. For example, if the application detects an asserted service pin for 10 seconds following a power cycle, it could call the Neuron C `go_unconfigured()` function.

  When using this approach, you should use the `System` object's `BeginChangeEvent()` method to subscribe your application to the `OnChangeEvent` events. Once you do so, LNS will generate an `OnChangeEvent` event each time it discovers a new device or router on the system. See the *LNS Object Server Reference* help file for more information on the `OnChangeEvent` event.

  Note that the `OnChangeEvent` event reports changes to the LNS network database. Thus, by the time the `OnChangeEvent` reports the detection of a new physical device, the `AppDevice` object associated with that device exists in the network database and is ready for use by your application. Subscribing to the related `OnAttachment` event provides similar information, but relates to network operations. Thus, an `AppDevice` object associated with a physical device may not have been added to the

network database when your application receives the `OnAttachment` event.

- Force the LNS Object Server to discover configured devices on a given domain, and all unconfigured devices on the network, by calling the `System` object's `DiscoverDevices()` method. Unconfigured devices discovered by this method will be placed in the `Discovered.Uninstalled` subsystem. Configured devices will be placed in the `Discovered.Installed` subsystem.

  This method cannot be used to discover devices configured on a domain other than the system domain if those devices are on the far side of a router whose class is `lcaConfiguredRouter, lcaBridge` or `lcaPermananentBridge`.

- Cause the device to inform the LNS Object Server that it has been attached to the network by sending a service pin message. The device's application program or hardware could be designed to send a service pin message when the device is attached to the network, or the installation staff could simply press the device's service pin. To discover a device via the service pin message, your application should call the `BeginServicePinEvent()` method to subscribe to the `OnSystemServicePin` event, and process the service pin events as they are generated.

  Alternatively, your application can set the `System` object's `RegisterServicePin` property to `True`. The LNS Object Server will then add an `AppDevice` object to the `Discovered.Uninstalled` subsystem (and generate an `OnChangeEvent` event) when it receives a service pin message from any unconfigured device. Subscription to service pin message events is not required in this case. If you use this approach, pressing the device's service pin at the completion of physical installation is not required, but it allows for instant recognition of the newly added device.

Once the LNS Object Server has discovered a device, you should perform the tasks described in the *Installing Devices* section on page 93 to configure the device in the LNS database. Following that, you can also begin adding the device to connections on your network.

Determining a Device's Location

Before using LNS to discover the devices attached to a network, you should note that most LNS applications will need to know the physical location of a device in order to automate installation in some scenarios. For example, suppose a building is composed of a group of rooms, each of which contains a group of light devices and a switch device. All of the light devices in a room must be connected to the switch device in that particular room. The application can identify the type of each device by its program ID, which is stored in the LNS database, but it needs another to way to determine which light devices are in the same room as a given switch device.

In order for an application to determine the location of a device, a user needs to supply the application with the information, or the device itself must provide the information. One way for a device to communicate its physical location is by using the location field in its Neuron Chip. The location field is a 6-byte field used for storing installation-related information.

For a device to provide location information to the application, there must be some way for the device to know its location. For example, each device's location field can be programmed with a location code, and each device can be sent to the field with instructions for where the device is to be attached to the network. Alternately, a hand-held tool could be used to set the location string of each device at installation time. If the LNS application contains a table that maps codes to physical locations, the application can fetch the location code using the `AppDevice` object's `LocationInNeuron` property, and determine where the device is located.

Another way for a device to determine its location is to provide a way for it to read its location using I/O pins. You could achieve this by embedding location information at each attachment site. The network connector could contain a 6-bit connector ID that allows the device to tell which of 64 connectors it has been attached to. Alternately, a serial ID device such as a Dallas touch memory could be used to provide a unique ID for each location while using fewer I/O pins on each application device. The use of I/O pins may require that device's program is running, so that it can read the I/O pins. If such behavior is required, the device must be delivered in the configured state or the device's application must use the Neuron C pragma `run_unconfigured` compiler directive.

There is another method to determine a device's approximate location. This method is less precise, as it provides a device's approximate location, but it is fully automatic. When using automatic installation, LNS will automatically detect the channel each application device is attached to, as long as the network only uses configured routers. The channel is an indicator for the device's approximate location and may assist with automatic location detection. Further, if pinging is enabled on the system, the LNS pinging process will verify the device's presence on the channel it was previously attached to. See the *Discovering When New Devices are Attached to the Network* section later in this chapter for more details on this.

If a device can determine its own location, it may also be able determine when it has been moved. This information can be used as part of the installation scenario. For example, if a temperature sensor is moved from room 1 to room 5, the sensor can tell that it has moved and, under application control, activate its service pin to issue a "request for service". When the LNS application receives the service pin event, it can check the database and see that the device was already installed in the system as part of another room, and then reconnect the network to make the sensor part of the control scheme for room 5.

Installing Devices

Once your application has discovered a newly discovered device, you need to define the device in the LNS database, and connect it to other devices on the network. Depending on how the device was discovered, the steps required to do so may vary slightly. These tasks are described below.

Note that if your network uses multiple channels, you will need to install and configure the network's routers, and create channels, as you define your devices. For more information on routers, and on special considerations you may need to make when installing devices on a network with multiple channels, see *Installing Devices With Multiple Channels* on page 175.

1. In several of the device discovery methods described in this section, an `AppDevice` object is added to the `Discovered.Uninstalled` subsystem's `AppDevices` collection each time a new device is discovered. You need to move each device to

another subsystem before installing the device. Iterate through the AppDevices collection to find the device you want to move, and then call the AddReference() method on the destination Subsystem to move the device to the destination subsystem.

If your application discovered a newly attached device via the OnSystemServicePin event, the new device's handle would typically have been returned with the event. You can use that to locate the AppDevice in the Discovered.Uninstalled subsystem by calling the ItemByHandle() method. Then, call the AddReference() method on the destination Subsystem for that device to move it to its desired location. However, if the device has not yet been added to the Discovered.Uninstalled subsystem's AppDevices collection, the device handle returned in the OnSystemServicePin event will be 0. In this case, your application could set the System object's RegisterServicePin property to True and evaluate the OnChangeEvent event instead of the OnSystemServicePin event, as the OnChangeEvent event notifies the application on the completion of a database change. Thus, by the time the application receives the OnChangeEvent event, the related AppDevice object will be ready for use and could be acquired using any appropriate means.

NOTE: The DeviceTemplate object assigned to each AppDevice specifies that device's external interface definition. You should check to make sure that the AppDevice is using the correct DeviceTemplate object before proceeding. For more information on DeviceTemplate objects and external interfaces, see *Device Interfaces* on page 104.

2. Associate the new AppDevice object with the appropriate physical device on the network by setting its NeuronId property. LNS will have already set the NeuronId property for any devices added to the Discovered.Uninstalled subsystem. There are several ways to use LNS to determine a device's Neuron ID. For more information, see *Neuron ID Assignment* on page 115.

   ```
   MyAppDevice.NeuronId = AcquiredNeuronId
   ```

3. If you need to update the device's application image, you should do so now. For more information on loading device applications, see *Loading Device Application Images* on page 119.

4. For each device, start a transaction, and commission the device with the `Commission()` or `CommissionEx()` methods. Before committing the transaction, you should use the `DownloadConfigProperties()` method to synchronize the configuration property values in the physical device with those stored in the LNS database. Note that the `ConfigPropertiesAvailable` property of the device's `Interface` must be set to `True` for you to call `DownloadConfigProperties()` at this point.

   Alternatively, you could commission the device and then call the `UploadConfigProperties()` method after committing the transaction to synchronize the configuration property values in the database with those stored in the LNS database. For more details on how you can synchronize configuration property values, and for information on other considerations you should make when commissioning devices, see *Commissioning Devices* on page 121.

   ```
   MySystem.StartTransaction()
   If MyInterface.ConfigPropertiesAvailable Then
       MyAppDevice.DownloadConfigProperties_
           (lcaConfigPropOptLoadValues OR lcaConfigPropOptSetDefaults)
   End If
   MyAppDevice.CommissionEx(lcaCommissionFlagNone)
   MySystem.EndTransaction()
   ```

5. If the `ConfigPropertiesAvailable` property was set to `False` in Step 4, or if you bypassed calling the `DownloadConfigProperties()` method in Step 4 to preserve the configuration property values stored in the physical device, upload the configuration property definitions into the LNS database with the `UploadConfigProperties()` method. This will set the `ConfigPropertiesAvailable` property on all devices using this template to `True` (or `False`, depending on the device's implementation).

   Typically, you will also want to upload the values from the device and set the defaults from the device by specifying the `lcaConfigPropOptLoadValues` and `lcaConfigPropOptSetDefaults` options when you call the `UploadConfigProperties()` method. Alternatively, you could update some or all of the configuration property values using data points, and then call `UploadConfigurationProperties()` with the `lcaConfigPropOptLoadValues`, `lcaConfigPropOptSetDefaults` and `lcaConfigPropOptLoadUnknown` options set to upload all values not specifically set, and then set the defaults to the values defined for this device. For optimal performances, you should set all configuration property values in a single transaction. For more information on these tasks, see *Application-specific Configuration Data* on page 125.

6. Bring the device online by setting the `State` property of the `AppDevice` object to `lcaStateCnfgOnline`.

   ```
   MyAppDevice.State = lcaStateCnfgOnline
   ```

7. You can now add the device to connections on your network, as your network design requires. For instructions on creating and managing connections, see *Connecting Devices* on page 138.

NOTE: Depending on the system management mode, the changes you make to each device may or may not be propagated to the physical device on the network as you invoke

each method. For more information on this, see *System Management Mode Considerations* on page 97.

Discovering When Devices are Detached or Replaced

Your LNS application can check if devices are still present in the network by periodically attempting to communicate with them. One way to do so is to invoke the `Test()` method on each `AppDevice` you need to check on. The `Test()` method verifies that the device is present on the network, that it is properly configured, and that no other devices on the network are using the network address assigned to the device.

If the `Test()` method returns `lcaTestResultComm` as the test status, it may indicate that the device has been detached from the network. A device might be detected as detached for several reasons, some of which will indicate the need for repair rather than a removal or replacement. For example, the device might be detected as missing because it has been detached from the network for physical maintenance, or because it has been powered off. In this case, the best response from the application would be to prompt the user to inspect the physical hardware. If the device has been physically removed from the system, and does not need to be replaced, the application should perform a logical device removal to keep the LNS database synchronized with the physical removal, or it can simply decommission the device if a replacement is expected in the future. For more information on this, see *Removing Devices* on page 137.

The `Test()` method can return many possible values (e.g. `lcaTestResultCommSnode`, `lcaTestResultMismatchDomain`, etc.) as the test status if the device being tested is not properly configured. This could happen because of faulty device hardware design. You can try to correct these errors by re-commissioning the device with the `Commission()` or `CommissionEx()` methods. Consult the *LNS Object Server Reference* help file for more information on the `Test()` method.

The `Test()` method is not the only way to determine when devices have been detached from the network. You can use the `BeginAttachmentEvent()` method to subscribe your application to the `OnAttachment` event. Once you register for this event, the LNS Object Server will periodically ping the devices on your network, and fire the event if any are missing. This is much more efficient than the `Test()` method, but not as comprehensive. You can determine the interval at which each device is pinged by writing to the `AppDevice` object's `PingClass` property. You can set this property to any of the following values:

`lcaPingClassMobile`	Check detachment frequently (1 minute by default).
`lcaPingClassTemporary`	Check detachment less frequently (2 minutes by default).
`lcaPingClassStationary`	Check detachment occasionally (15 minutes by default).
`lcaPingClassPermanent`	Never check.

You can set the ping frequency for each of these ping classes by writing to the `System` object's `PingInterval` property. The default ping rate is once every 15 minutes, unless the `System` object's `InstallOptions` property is set to `lcaSharedMedia`, in which case pinging will be disabled.

When you discover that a physical device has been permanently removed from the network, you should remove the `AppDevice` object associated with the device from the LNS database. See *Removing Devices* on page 137 for information on removing devices.

Before doing so, you should note that device detachment might only signal that a repair operation is under way. For example, if a device is discovered missing, but a new device with the same program ID (or the same location field information) is added to the network and attached to the same channel within the next 15 minutes, then you could assume that the new device is a replacement for the removed device. In this case, you need to update the LNS database to recognize the replacement. For instructions on this, see *Replacing Devices* on page 132.

Note that when a router becomes unattached from the network, LNS will usually be unable to communicate with any devices on the far side of the router, and devices on channels connected by the router will be unable to communicate with each other. In this case, LNS will generate an `OnAttachmentEvent` for the router, but not for each affected device.

System Management Mode Considerations

The network installation scenarios described in this chapter make reference to the system management mode (i.e. the `MgmtMode` property of the `System` object) at various points. The system management mode determines whether or not changes made to the LNS database that affect the configuration of devices on the network are propagated to those devices as the changes are committed to the LNS database, or if the configuration changes are propagated to the devices later. The system management mode affects all client applications connected to a network, since the configuration of the network must be managed consistently.

This section provides information you need to be aware of when using the `MgmtMode` property. The section begins with a description of the two values you can assign to the `MgmtMode` property: `lcaMgmtModePropagateConfigUpdates` and `lcaMgmtModeDeferConfigUpdates`.

NOTE: New names for the system management mode settings have been provided in Turbo Edition. The old values still exist in LNS for compatibility reasons, but the documentation refers only to the new values. The new `lcaMgmtModeDeferConfigUpdates` value maps to the old `lcaOffNet` value, and the new `lcaMgmtModePropagateConfigUpdates` value maps to the old `lcaOnNet` value.

lcaMgmtModePropagateConfigUpdates

When the system management mode is set to `lcaMgmtModePropagateConfigUpdates`, changes made to the database that affect the configuration of one or more physical devices are propagated to those devices as the changes are committed to the database. If the modification is performed as part of a transaction, the changes are propagated to the devices during the call to `CommitTransaction()`. If the modification is not performed as a part of a transaction, the changes are propagated as the property causing the change is written to, or the method causing the change is invoked, and the LNS database is updated.

Note that if LNS fails to update the devices, a network service warning in the range 4030-4089 (`lcaErrNsWarningFirst- lcaErrNsWarningLast`) will be returned. In

addition, any changes that affect the configuration of devices that have not yet been commissioned are deferred until those devices are commissioned.

Note that the system management mode cannot be set to `lcaMgmtModePropagateConfigUpdates` while in engineered mode.

lcaMgmtModeDeferConfigUpdates

When the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, changes made to the database that affect the configuration of devices on the network are not propagated to the devices until the system management mode is set back to `lcaMgmtModePropagateConfigUpdates`. If you commission or replace a device with the `Commission()` or `Replace()` methods while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, LNS will not propagate updates to the device until the management mode is set to `lcaMgmtModePropagateConfigUpdates`.

However, you can now use the `ReplaceEx()` method with the `lcaReplaceFlagPropagateUpdates` option to replace devices while the system management mode is still set to `lcaMgmtModeDeferConfigUpdates`. This may be useful if there are configuration changes pending for a large number of devices, and you want to replace a device without waiting for all of those changes to be applied. For more information on replacing devices, see *Replacing Devices* on page 132.

A similar option is also provided for use with the new `CommissionEx()` method, which you can use to recommission a device while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. For more information on commissioning devices, see *Commissioning Devices* on page 121.

When you open a network in engineered mode, it automatically sets the system management mode to `lcaMgmtModeDeferConfigUpdates`. When you re-open that network later, and are attached to the network, the `System` object's `MgmtMode` property will not be automatically reset. The application must explicitly set the system management mode back to `lcaMgmtModePropogateConfigUpdates`.

It is possible to communicate with devices while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, as long as the network was not opened in engineered mode. However, some operations may not behave correctly in this case, because a device's configuration may not be synchronized with the LNS database. When set to `lcaMgmtModeDeferConfigUpdates`, the system management mode also effects certain operations explicitly. For more information on this, see *Affects on Network Management Methods and Properties* on page 100.

Devices that have pending updates will have a commission status of pending (or failed, if the last update attempt failed) when the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. A network service warning in the range 4030-4089 (`lcaErrNsWarningFirst- lcaErrNsWarningLast`) will be returned to indicate this, but the warning will not indicate which devices have not been updated. You can check a device's commission status by reading its `CommissionStatus` property.

Intended Usage of the System Management Mode

The system management mode should generally be set to `lcaMgmtModePropagateConfigUpdates`, except when defining a network in

engineered mode. This will ensure that the configuration of the network is always synchronized with the information stored in the LNS database. However, you may want to change the system management mode to `lcaMgmtModeDeferConfigUpdates` before making extensive changes to a network, in order to minimize the impact of these changes on the operation of the system.

For example, suppose a new floor or wing of a building is being added to the network, and this addition affects a number of connections with devices that are already installed on the network. You may find it more efficient to set the system management mode to `lcaMgmtModeDeferConfigUpdates`, make all the necessary changes in the LNS database, and then set the system management mode back to `lcaMgmtModePropagateConfigUpdates` mode when you are ready to propagate those changes to the network. This allows you to choose a period during which an interruption in network operation would be less intrusive to do so.

Changing the System Management Mode

The system management mode is global and affects all clients that are accessing the system. When changing the system management mode, consider the following:

1. When changing from `lcaMgmtModeDeferConfigUpdates` to `lcaMgmtModePropagateConfigUpdates`, the network will be updated with any changes to the database that have been made by any application since the system management mode was set to `lcaMgmtModeDeferConfigUpdates`. Presumably, whoever set the management mode to `lcaMgmtModeDeferConfigUpdates` had a reason to do so. Propagating device updates while major changes are incomplete could cause the network to function improperly. As a result, you should make sure that all major network management operations initiated on the network by your application or any other are complete when you change the system management mode from `lcaMgmtModeDeferConfigUpdates` to `lcaMgmtModePropagateConfigUpdates`.

2. Changing from `lcaMgmtModePropagateConfigUpdates` to `lcaMgmtModeDeferConfigUpdates` also affects a variety of network management operations. The following sections describe the things you need to be aware of in this case.

Tracking Device Updates

Changing the system management mode from `lcaMgmtModeDeferConfigUpdates` to `lcaMgmtModePropagateConfigUpdates` can take a considerable amount of time, since LNS may have to update a large number of devices with pending updates. If a device has been commissioned, you can determine if it will need to be updated by reading its commission status. If an `AppDevice` object's `CommissionStatus` property is set to `lcaCommissionUpdatesPending` or `lcaCommissionUpdatesFailed`, then that device will be updated when the system management mode is changed from `lcaMgmtModeDeferConfigUpdates` to `lcaMgmtModePropagateConfigUpdates`.

You can monitor the progress of the device updates caused by such a management mode change by subscribing your application to the `OnCommission` event. Invoke the `BeginCommissionEvent()` method on the `System` object to do so. As devices are updated and their commission status changes, the event will be fired.

Tracking System Management Mode Changes

The system management mode is global and affects all clients that are accessing the network. Because of this, you should consider registering your application for the `OnSystemMgmtModeChangeEvent`. You can do so by invoking the `BeginSystemMgmtModeChangeEvent()` method on the `System` object. Once your application registers for this event, it will be fired each time the system management mode changes. You should not change the system management mode after receiving this event, as the client that changed the system management mode probably had a reason to do so, but you can use the event to stay informed of system management mode changes.

Affects on Network Management Methods and Properties

The following methods and properties are also affected by the system management mode. These tasks are described in more detail in Chapter 6, Network Management: *Defining, Commissioning and Connecting Devices*.

- If you use the `Commission()` or `Replace()` methods to commission or replace an `AppDevice` or `Router` object while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, LNS will not validate the device's program interface or channel. However, if you use the `ReplaceEx()` method with the `lcaReplaceFlagPropagateUpdates` option set, or the `CommissionEx()` method with the `lcaCommissionFlagPropagateUpdates` option set, LNS will perform these validation tasks, regardless of the system management mode.
- Background pinging is disabled while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. This is so LNS will not generate misleading ping reports while the LNS database and the physical network are not synchronized.
- Background update retries for operations that failed while the system management mode was set to `lcaMgmtModePropgageConfigUpdates` are disabled while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. Normally, you can enable background update retries by setting the `System` object's `UpdateInterval` property to a positive value.

  In addition, if you invoke the `RetryUpdates()` method while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, only failed updates that occurred while the system management mode was set to `lcaMgmtModeDeferConfigUpdates` will be retried. Failed updates that occurred while the system management mode was set to `lcaMgmtModePropagateConfigUpdates` will not be retried, unless the system management mode is still set to (or has been restored to) the `lcaMgmtModePropagateConfigUpdates` value when you call `RetryUpdates()`.
- If you create a `Router` with the `AddEx()` method while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, you cannot specify the `lcaRouterFlagSplit` value as the `flags` element.
- You cannot call the `Load()` or `LoadEx()` methods on an `AppDevice` while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`.

- You should be aware that in some cases, remote Full client applications cannot open networks when the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. This may be the case if the Full client application has not previously connected to the LNS Object Server, if the Full client application has changed channels, or if changes have been made to the database such that commissioning the Network Service Device may cause inconsistencies in the configuration of physical devices on the network.

  Use the `AllowPropagateModeDuringRemoteOpen` property to determine how LNS will behave in this situation. Set the property to `True` if it is acceptable for LNS to temporarily change the system management mode from `lcaMgmtModeDeferConfigUpdates` to `lcaMgmtModePropagateConfigUpdates` when opening a network from such a client application in these situations. Otherwise, set this property to `False`.

  You should note that when this property is set to `True` and the system management mode is changed, all pending configuration updates will applied to the physical devices on the network. This may result in unwanted changes being propagated to the network. Once this has completed and the network has been opened, the management mode will be restored to `lcaMgmtModeDeferConfigUpdates`.

Chapter 6 - Network Management: Defining, Commissioning and Connecting Devices

This chapter provides additional details on how to perform the network installation tasks introduced in Chapter 5. This includes topics such as importing device interfaces, assigning network addresses, loading device applications, commissioning devices, synchronizing device configuration properties, and creating connections between devices.

Defining, Commissioning and Connecting Devices

Chapter 5 of this document lists the tasks you need to perform when using LNS to install a LONWORKS network. This chapter provides extensive detail on how you should use LNS services to perform each task introduced in Chapter 5. This includes the following sections:

- *Device Interfaces*. Each LONWORKS device has a device interface. The device interface represents the device's functionality on the LONWORK network. In LNS, distinct types of device interfaces are represented by `DeviceTemplate` objects. Generally, you should specify the `DeviceTemplate` object a device should use when you add the device to the LNS database. You can also import the interface from the device into the LNS database later. The *Device Interfaces* section provides instructions to follow when importing a device interface. It also describes the components of a device interface, and the various considerations you may need to make when choosing a device interface for a device.
- *Defining and Commissioning Devices.* Chapter 5 provides an overview of the steps you need to perform when installing devices on a LONWORKS network. This section describes each of these steps in detail. This includes the following topics: address assignment, commissioning devices, loading device application images, configuring devices, and setting devices online.
- *Other Device Management Operations.* This section describes network management tasks you may need to perform after installing a network, such as testing, removing, and replacing devices.
- *Connecting Devices*. This section provides an overview of how LNS manages connections, and describes how you can create connections between devices on your network.

Device Interfaces

Each LONWORKS device consists of a hardware platform, software that exercises that hardware, and a device interface. The device interface, often referred to as the device's external interface, represents the functionality of the device on the LONWORK network. This is the part of the device software that is exposed to the network, which allows other devices and system integrators to benefit from the functionality and data supplied by the device, and provide input to the device as needed.

The device interface consists of LonMark Functional Blocks, network variables, and configuration properties. The device interface does not expose the internal algorithms of a device. Instead, it only exposes the inputs and outputs of the algorithms.

Much of the device interface can be queried over the network by a network tool. The device manufacturer determines the completeness of a queried interface. For example, a device manufacturer could choose to embed network variable names in a device to ensure that the queried network interface includes these names.

In order to make the complete device interface available to the system integrator, the device interface should be documented in device interface files. There are several benefits to using device interface files:

- A device interface file may include information that is not included in the device itself, such as network variable names.
- A device interface file can be used even when the device cannot be accessed by the network engineering tool, such as during the definition phase of an engineered mode installation.
- Importing from a device interface file may be substantially faster than recovering interface information from a device.
- Importing from a device interface file allows LNS to accurately set the default values for network variable characteristics such as message service type and priority. If you recover the interface from the device, LNS will use the current values in the first device with that program as the defaults.
- Importing from a device interface file ensures that configuration property definitions and default values for the device are available to LNS. After recovering an interface from a device, it is necessary to upload the configuration definitions from the device, after the device has been commissioned and the device application is running. Default values can be set to the current values in the device.

The primary device interface file type is a text file with a .XIF extension. Some platforms convert this file to alternate formats for performance optimization. For example, LNS uses a binary device interface file (.XFB extension), and a device interface object file (.XFO extension). Both of these files are created from the data contained within the text device interface file. The device interface files (.XIF and .XFB file extensions) are supplied by the device manufacturer. The optimized interface file (.XFO extension) is automatically created and maintained by LNS when the device interface files are imported. Note that some manufacturers supply the .XFO file with the device, which allows for more efficient importation of the device interface.

When you add a device to an LNS network database, you must create a `DeviceTemplate` object within the database to represent the device's interface. You can create the `DeviceTemplate` object by querying the desired device's external interface over the LONWORKS network, or by reading the device interface files. To read a device interface file into a `DeviceTemplate` object, follow these steps:

1. Initialize the LNS Object Server and open the network and system you plan to install your devices on. These tasks are described in Chapter 4, *Programming an LNS Application.*
2. Access the `System` object's `DeviceTemplates` collection. You can access the `DeviceTemplates` collection through the `System` object's `TemplateLibrary` object.

   ```
   Dim MyTemplateLibrary as LcaTemplateLibrary
   Dim MyDeviceTemplates as LcaDeviceTemplates
   Set MyTemplateLibrary = MySystem.TemplateLibrary
   Set MyDeviceTemplates = MyTemplateLibrary.DeviceTemplates
   ```

3. Call the `DeviceTemplates` collection's `Add()` method to create a new `DeviceTemplate` object. The `programType` parameter you supply to the `Add` method must be set to `lcaProgramTypeXif`.

   ```
   Dim MyTemplate as LcaDeviceTemplate
   Set MyTemplate = MyDeviceTemplates.Add ("T1", lcaProgramTypeXif)
   ```

4. Call the `Import()` method on the new `DeviceTemplate` object to import the device's external interface file. You will provide the location of the device interface files in the `xifPath` parameter you supply to the method.

   ```
   MyTemplate.Import ("C:\LonWorks\Import\Acme\Cooker.xif")
   ```

   Device interface files may reside in any location that can be reached through the file system on the LNS Server PC. Thus, referencing a device interface file as `C:\LonWorks\Import\Acme\Cooker.xif` refers to the cooker.xif interface file in the `LonWorks\Import\Acme` folder on the `C:` drive of the LNS Server PC. Device interface files may reside outside the LNS Server PC's local storage, and can be referred to via mapped network drives or fully qualified filenames. The device plug-in software that comes with most LONWORKS devices should take care of installing the device interface files into the correct locations, and may also create and initialize `DeviceTemplate` objects as part of the plug-in application's registration with the network.

5. Once you have imported the external interface into the `DeviceTemplate` object, LNS will automatically associate the `DeviceTemplate` with all devices that use the imported external interface. LNS makes this determination by reading the program ID values assigned to the devices, as described in the next section. You can also specify the `DeviceTemplate` an `AppDevice` object should use when you create the `AppDevice` object, as described later in this chapter.

Program IDs and DeviceTemplate Objects

A network can consist of multiple devices, some of which implement the same hardware, software, and device interface. For example, a network could contain several hundred light switches of the same make and model. Each of these light switches has a unique Neuron ID, a unique network address (subnet/node ID, etc), and unique values for their network variables and configuration properties. However, all the light switches share the same device interface, and so they use the same `DeviceTemplate` in the LNS database.

The unique identifier of a device interface, and of the associated `DeviceTemplate` in the LNS database, is the program ID. Every LonMark-compliant LONWORKS application device is assigned a unique, 16 digit, hexadecimal standard program ID that uses the following format:

FM:MM:MM:CC:CC:UU:TT:NN

Table 6.1 describes each part of the program ID. You can find out more about the standard program ID fields in the *LonMark Application-Layer Interoperability Guidelines*, which can be downloaded from the web at http://www.lonmark.org/products/guides.htm.

Table 6.1 Program ID Format

Program ID Segment	Description
F	This represents the format identifier. This will be set to 8 for LonMark certified interoperable devices, 9 for draft (not certified) interoperable devices, or a value less than 8 for non-interoperable devices. Note that the LonMark Interoperability Association has reserved format identifiers 0xA..0xF for future use.
M:MM:MM	A series of 5 hexadecimal digits identifying the device manufacturer. Manufacturer IDs are unique to the manufacturer of certified interoperable devices
CC:CC	A series of 4 hexadecimal digits that describe the class and primary function of the device.
UU	2 hexadecimal digits describing the device's usage within its class. This is also known as the device's subclass.
TT	2 hexadecimal digits identifying the physical channel type the device can be used with.
NN	2 hexadecimal digits representing the device model number.

Note: The first five fields described in Table 6.1 are allocated by the LonMark Interoperability Association.

Device Resource Files

While the device interface file details all the network variables, configuration properties and LonMark Functional Blocks initially available on a device, the interface does not specify the internal structure of these elements.

A network variable defines an operational input or output for the device. Each network variable is assigned a *network variable type,* which determines the structure, range, base units, and scaling factors that the network variable uses.

A configuration property specifies a configuration option for a network variable, a LonMark Functional Block, or the entire device. Each configuration property is assigned a *configuration property type,* which defines the structure, range, base units, and scaling factors that the configuration property uses.

Each network variable and configuration property is also assigned one or more format specifications. Alternate format specifications can be supplied to provide unit conversion

between different measurement systems, to provide alternate industry-specific measurement units, or to provide locale-specific formatting for times, dates, or numeric value separator characters.

A LonMark Functional Block groups network variables and configuration properties that are related to a particular function of the device. LonMark Functional Blocks make a device easier to install and configure. Each LonMark Functional Block is defined by a functional profile that defines the network variables and configuration properties that comprise the class of functional blocks. For example, a digital input device with four switches could contain one LonMark Functional Block for each switch. For general information on LonMark Functional Blocks, consult the *LonMark Application-Layer Interoperability Guidelines,* which can be downloaded from LonMark website at http://www.lonmark.org/products/guides.htm.

Functional profiles, network variable types, and configuration property types are defined in device resource files, which are grouped into resource file sets. A complete resource file set consists of a type file (.TYP extension), a functional profile definitions file (.FPT extension), a format file (.FMT extension), and one or more language files (.ENG, .ENU or other extensions).

You can create resource files with the Echelon NodeBuilder Resource Editor tool, which is available for free download to all LonMark members from http://www.lonmark.org. The LonMark Interoperability Association also provides a standard resource file set, containing the most commonly used standard network variable types (SNVT), configuration property types (SCPT) and functional profiles (SFPT). The most recent standard resource file set is included with the LNS Server installation for Turbo Edition.

The device manufacturer typically provides all device resource files that are referred to by the device (with the exception of the standard resource file set).

Each resource file set must be registered with the resource catalog. The catalog is a repository of all resource file sets available on a given machine. Resource file sets must be registered with the catalog that belongs to the LNS Server PC, and will typically need to be registered on each PC running an LNS client application, unless it is a Lightweight client application. The resource catalog is a file that is typically named LDRF.CAT, and is located in the `[Windows Drive]\LonWorks\Types` folder by default.

The device plug-in software that ships with most LNS devices will typically install the device resource files into an appropriate location and perform the registration with the resource catalog at installation time. For manual registration, you can use the LDRFCAT.EXE utility's graphical user interface.

Scope Selectors

Each set of resource files must be associated with a particular program ID, a range of program IDs, or with all program IDs to associate it with a network variable, configuration property, or `LonMarkObject` on a device. The type of association is called the scope of the resource file, and the scope is specified using a scope selector. The scope selector for a resource file specifies what part or parts of a device's program ID should be used when selecting the resource file. The various scope selectors you can use with LNS are described below.

Scope Selector	*Scope Enumeration*	*Scope Definition*
0	`lcaResourceScopeStandard`	Used for resource files containing standard definitions for all devices from any manufacturer. This selector value can only be used for resource files defined by the LONMARK Interoperability Association. The set of resource files named STANDARD.<exten> implements this scope.
1	`lcaResourceScopeClass`	Used for resource files containing standard definitions for all devices of a specified device class from any manufacturer. This selector value can only be used for resource files created by the LONMARK Interoperability Association.
2	`lcaResourceScopeSubclass`	Used for resource files containing standard definitions for all devices of a specified device class and subclass from any manufacturer. This selector value can only be used for resource files created by the LONMARK Interoperability Association.
3	`lcaResourceScopeMfg`	Used for resource files containing user definitions for all devices of a specified manufacturer. This selector value can be used by a manufacturer for resource files that apply to all of the manufacturer's devices.
4	`lcaResourceScopeMfgClass`	Used for resource files containing user definitions for all devices of a specified manufacturer and device class. This selector value can be used by a manufacturer for resource files that apply to all of the manufacturer's devices of a specific device class.

5	`lcaResourceScopeMfgSubclass`	Used for resource files containing user definitions for all devices of a specified manufacturer, device class, and device subclass. This selector value can be used by a manufacturer for resource files that apply to all of the manufacturer's devices of a specific device class and subclass.
6	`lcaResourceScopeMfgModel`	Used for resource files containing user definitions for all devices of a specified manufacturer, device class, device subclass, and model. This selector value can be used by a manufacturer for resource files that apply to a particular version of a single device type.

For example, if a manufacturer released a set of LonMark resource files with all type, format, and language information for all its devices, this set of files would have a scope selector of `lcaResourceScopeMfg`. If a resource file had a program ID of 80:00:01:00:00:00:00:00 and a scope selector of `lcaResourceScopeMfg`, then all applications with 0:00:01 (Echelon) as the manufacturer section of their program ID would use the types in this file.

The Bigger Picture

Figure 6.1 shows the device interface of the fictitious, energy-managed Cooker 2010 device produced by ACME Corporation. The device implements a standard functional profile (SPFTnodeObject), and an ACME-defined functional profile (UFPTcooker).

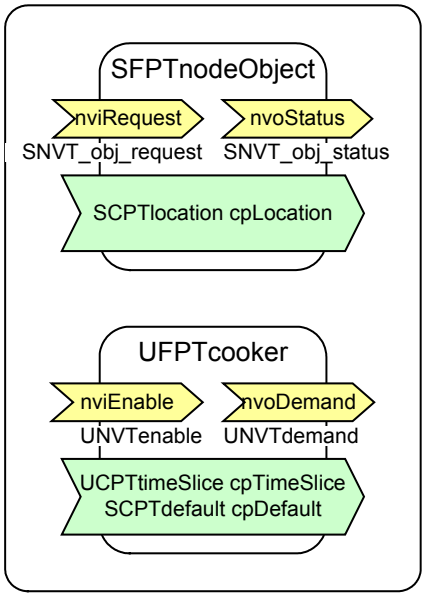


Figure 6.1 Device Interface

SFPTnodeObject implements two network variables: nviRequest of type SNVT_obj_request, and nvoStatus of type SNVT_obj_status. It also implements a single configuration property: cpLocation of type SCPTlocation.

The UFPTcooker LonMark Functional Block implements two network variables: nviEnable of type UNVTenable to enable the heaters, and nvoDemand of type UNVTdemand to request a time-slice of energy to be allocated. In addition, it implements two configuration properties: cpTimeSlice of the ACME-defined type UCPTtimeSlice, and cpDefault of the standard SCPTdefault type.

All SNVT, SCPT, and SFPT types are defined in the standard resource file set, which is automatically available on every LNS-based PC. No extra steps need to be taken to enable those types.

However, the UNVT, UCPT, and UFPT types are defined in user-defined resource file sets, which you must register with the device catalog on each PC that will access devices implementing these types. You can perform many of the basic network management functions without registering the related device resource files. However, monitoring, control, and enhanced diagnostics features may require the resource files to be registered. Figure 6.2 illustrates the device interface and device resource files on a single PC, as related to the fictitious Cooker 2010 device.

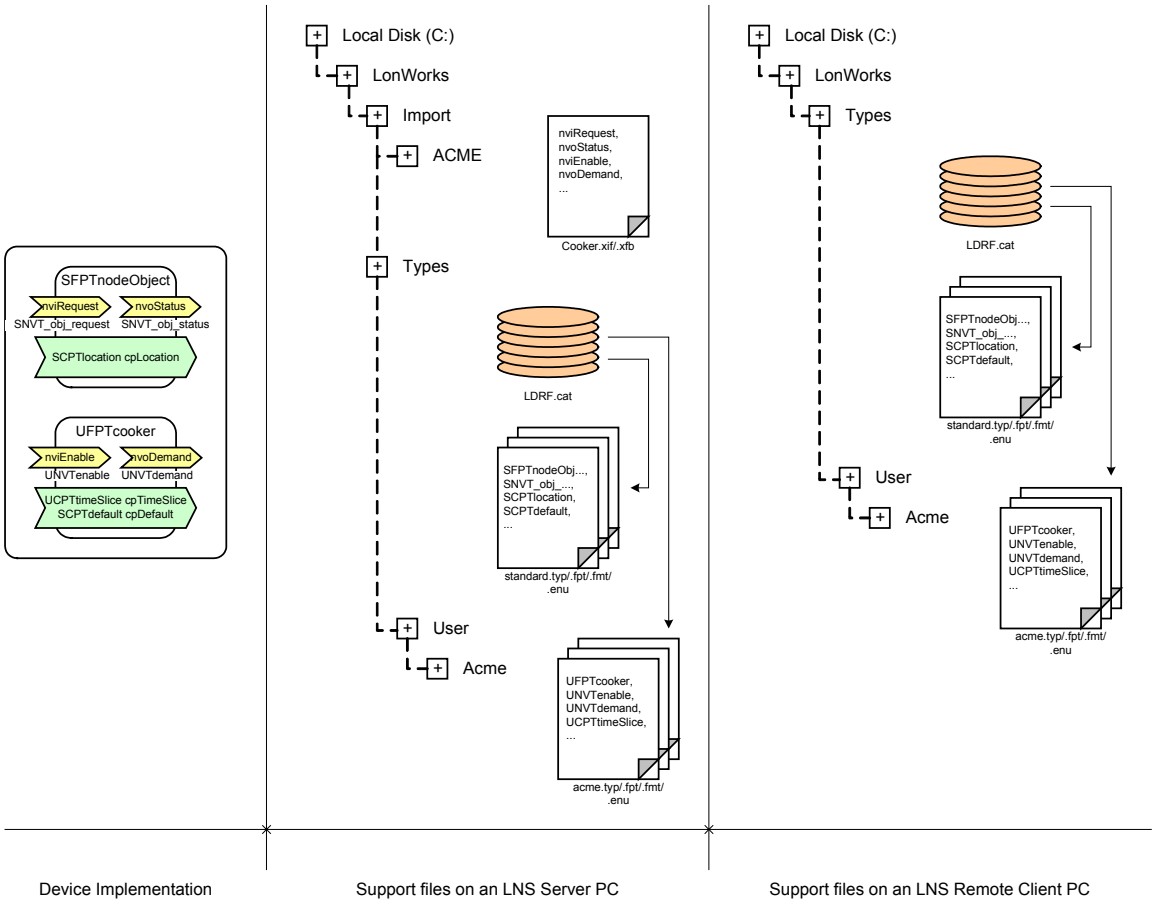


Figure 6.2 Device Interface and Resource Files on a PC

Maintaining Device Interfaces With LNS

The recommended way to manage your device interface and resource file sets is to register all required resource files with the resource catalog, and then to create your `DeviceTemplate` objects by importing the device interface files. Most LNS plug-in applications perform these tasks automatically.

LNS Turbo Edition provides a set of tools that may be used to re-synchronize resources to interfaces, or interfaces to devices, when necessary. You can call the `ResyncToResources()` method on a `DeviceTemplate` to resynchronize the `DeviceTemplate` with the device resource files. This may be necessary if a device's external interface was imported when the resource files for that device were not available in the resource file catalog, or if the device resource files have been updated or modified since the device's external interface was imported. Note that resource file sets should be modified carefully in order to maintain compatibility with existing devices.

You can also call the `ResyncToTemplate()` method on an `AppDevice` to re-synchronize the device with its `DeviceTemplate`. This may be necessary after re-importing the device interface file with the `Import()` method, or after using the `ResyncToResources()` method to resynchronize the `DeviceTemplate` with the device resource files.

See the *LNS Object Server Reference* help file for a complete list of the properties each `DeviceTemplate` object contains, and descriptions of how they might be useful to you.

As stated earlier, each device interface consists of network variables, configuration properties and LonMark Functional Blocks. In LNS, network variables, configuration properties and LonMark Functional Blocks are represented by `NetworkVariable`, `ConfigProperty`, and `LonMarkObject` objects. The device interface as a whole is represented by an `Interface` object, which can be accessed through the `Interface` property of the `AppDevice` object associated with the device. You can use the network variables and configuration properties on each device to monitor and control the device on the network. For information on using LNS to monitor and control a network, see Chapter 9, *Monitor and Control.*

NOTE: In some cases, there may be need to modify the functionality provided by a device interface. For example, controller devices may be used to control other devices. As a result, the number of components required for that device's interface is often an attribute of the network configuration (i.e. how many devices it is controlling), rather than of the device's hardware. Ideally, the resources on these controllers could be allocated dynamically, in order to fit the changing requirements of a given network as devices are added to it. As a result, LNS features support for dynamic interfaces. For more information on this, see *Using Dynamic Device Interfaces* on page 179.

Note that application image files (.NXE, or .APB extensions) that are used for device loading are also typically stored in the server PC's `Import\` folder, and must be referenced using a server-centric path similar to the device interface file path discussed in this section.

Defining and Commissioning Devices

This section describes the steps you need to take when installing and configuring devices on a network with the network installation scenarios described in Chapter 5 in detail. This includes the following sections:

- *Creating AppDevice Objects.* One of the first steps you need to take when using LNS to define and commission a device is to create an `AppDevice` object to represent the device in the LNS database. This section describes how to do so.
- *Neuron ID Assignment*. LNS use a device's Neuron ID to associate an `AppDevice` object with a physical device on the network. In some cases, you may already know the Neuron ID a device uses, and can set the property immediately. In other cases, you might not. This section describes the different ways you can use LNS to determine a device's Neuron ID, and how to store the Neuron ID in the LNS database.
- *Loading Device Application Images.* Before commissioning a device, you may need to load the device's application image. This section describes how to do so.
- *Commissioning Devices.* When you commission a device, LNS uses the device's Neuron ID to associate the `AppDevice` object in the LNS database with the physical device on the network. There are many considerations you need to make when commissioning devices. This section describes those considerations, and describes when you should use the `CommissionEx()` method, which has been added to LNS for Turbo Edition.
- *Configuring Devices.* Each device contains configuration properties that are defined by the device's interface. It is important to make sure that the configuration property information stored in the LNS database matches the configuration property information stored in the physical device on the network. You can use the `DownloadConfigProperties()` and `UploadConfigProperties()` methods to do so. This section describes the considerations you should make when using those methods.
- *Setting Devices Online.* This section describes how to set a device online, and provides guidelines you should be aware of when using LNS to change the state of a device.
- Generally, you can add a device to connections on your network as soon as you have created an `AppDevice` object for the device in the LNS database. However, when using an add-hoc installation scenario in which either the device template or channel is not specified when creating the device, device connections cannot be created until the device has been commissioned. For more information on connecting devices, see *Connecting Devices* on page 138.

Creating AppDevice Objects

One of the first steps you need to take when using LNS to define and commission a device is to create an `AppDevice` object to represent the device in the LNS database. To create an `AppDevice`, follow these steps:

1. Select or create the `Subsystem` object that will contain the new `AppDevice`. As described in *The LNS Programming Model* on page 29, each `Subsystem` represents a

physical or logical segment of your network, and each Subsystem has its own AppDevices and Routers collections. These collections contain the application devices and routers installed on that Subsystem. You can select a subsystem that already exists using the Item property of your Subsystems collection, or you can create a new subsystem with the Add() method.

```
Set MySubsystems = MySystem.Subsystems
Set MySubsystem = MySubsystems.Item("FireSubsystem")
```

OR:

```
Set MySubsystems = MySystem.Subsystems
Set MySubsystem = MySubsystems.Add("FireSubsystem")
```

NOTE: You cannot explicitly add items to AppDevices collections that are contained in the ALL or Discovered subsystems, as these read-only collections are maintained by LNS.

2. Fetch the AppDevices collection from the selected Subsystem:

```
Set MyAppDevices = MySubsystem.AppDevices
```

3. Use the Add() method to add a new AppDevice to the AppDevices collection acquired in step 2.

```
Set NewAppDevice = MyAppDevices.Add("SmokeDetector", _
    MyDeviceTemplate, MyChannel, MySubnet)
```

The Add() method takes four parameters: the device name, a DeviceTemplate object, a Channel object, and a Subnet object. The device name is required, and the Channel and DeviceTemplate objects should be specified whenever possible. You must specify a DeviceTemplate to create planned network variable connections for the device in the engineered installation scenario.

If you do not specify the device template and channel, the device template will be assigned (and recovered from the device if necessary) when the device is commissioned. In this case, the device's interface will be unavailable until the device has been commissioned. Echelon recommends that you specify the template and channel during device creation whenever possible.

If you do not specify the device template, any dynamic network variables on the device's main interface will be uploaded into the LNS database when the device is commissioned, which may not be desirable. However, you can use the MoveToInterface() method to move these dynamic network variables to a custom interface, and then delete them, if desired. For more information on the MoveToInterface() method, see *Using Dynamic Device Interfaces* on page 179.

For more information on device templates, see the *Device Interfaces* section earlier in this chapter. For more information on channel and subnet allocation, see *Managing Networks with Multiple Channels* on page 169.

Echelon recommends that you do not assign a subnet when creating AppDevice objects, as LNS will automatically assign the best subnet to the new device, based on the network topology and resource requirements.

4. You should note that devices can be installed in multiple subsystems. You can use the `AddReference()` method to create a reference to a device that has already been added to another subsystem.

   ```
   Set MyAppDevice = OtherSubsystemAppDevices.Item("Device 1")
   Set MyAppDevices = MySubsystem.AppDevices
   MyAppDevices.AddReference (MyAppDevice)
   ```

   If a device has been added to `AppDevices` collections in multiple subsystems in this fashion, you will need to call the `Remove()` or `RemoveEx()` method on each collection containing the device when you want to remove the device from the network. This does not apply to the ALL or Discovered subsystems. You can use the `AppDevice` object's `Subsystems` property to see which subsystems the `AppDevice` belongs to. For more information on removing devices, see *Removing Devices* on page 137.

Neuron ID Assignment

Every LONWORKS device is assigned a unique Neuron ID when it is manufactured. The Neuron ID differentiates the device from all the other LONWORKS devices in the world, and LNS uses Neuron IDs to uniquely identify the devices on a network. There are three ways to acquire a device's Neuron ID, and write it to the LNS database:

- *Service Pin*
- *Find and Wink*
- *Manual Entry*

Service Pin

Each device has a service pin. When a device's service pin is activated, the device sends a broadcast message containing its Neuron ID and program ID. The method used to activate the service pin varies from device to device. Examples of mechanical methods include activating via a push button, or using a magnetic reed switch. A Neuron C function allows the service pin to be placed under software control as long as the device's application code is running. For example, a device can send a service pin message when it is moved, or when a predefined series of I/O events occur.

You can program your LNS application to receive service pin messages from devices on your network, and determine the Neuron IDs assigned to those devices from those messages. To do so, follow these steps:

1. Invoke the `System` object's `BeginServicePinEvent()` method to register your application for service pin events.

   ```
   MySystem.BeginServicePinEvent()
   ```

2. When the LNS Object Server receives a service pin message, the `OnSystemServicePin` event will be fired. The parameters provided with this event include the Neuron ID, program ID, location, channel handle, network handle, and system handle for the device whose service pin was activated.

3. Assign the Neuron ID acquired in step 2 to the `AppDevice` object you created for the device whose service pin was activated. You can do so by writing to the `AppDevice` object's `NeuronId` property.

   ```
   MyAppDevice.NeuronId = AcquiredNeuronId
   ```

4. Invoke the `EndServicePinEvent()` method to cancel your application's subscription to service pin events.

   ```
   MySystem.EndServicePinEvent()
   ```

Confirmed Service Pin Protocol

The confirmed service pin protocol includes additional steps you can take to ensure that the service pin event your application receives is from the expected device on the network. You should use the confirmed service pin protocol on systems using shared media to ensure that you have not received a service pin message from a device that belongs to another system. You should also consider using the confirmed service pin protocol on systems using private media if multiple technicians are installing different devices simultaneously. For more information on shared media, see *Using Shared Media* on page 167.

To use the confirmed service pin protocol to determine a device's neuron ID, follow these steps:

1. Acquire the device's Neuron ID using the `OnSystemServicePin` event, and assign the Neuron ID to the `NeuronId` property of the `AppDevice` object you have created for the device. Before you write to the `NeuronId` property, you should read the property and preserve a copy of the previous value. These tasks are described in steps 1-3 of the procedure in the previous section.

2. To confirm that you have found the correct device, wink the device by invoking the `Wink()` method.

   ```
   MyAppDevice.Wink()
   ```

   NOTE: When using the confirmed service pin protocol on a router, you should use the `Reset()` method for this step.

3. When a device receives a wink command, it responds in a way that can be easily detected by someone at the device's location. For example, lights can blink, alarms can ring, and displays can flash. Once the device is winked, someone on-site should press the service pin a second time, to acknowledge that the correct device was winked.

   NOTE: When using the confirmed service pin protocol on a router, the router will reset and indicate this via its reset LED.

4. The `OnSystemServicePin` event will be fired a second time. If the `neuronId` element returned by the event matches the one acquired in step 1, then you can assume that you have located the correct device. Otherwise, restore the `AppDevice` object's `NeuronId` property to the original value preserved in step 1.

5. Invoke the `EndServicePinEvent()` method to cancel your application's subscription to service pin events.

   ```
   MySystem.EndServicePinEvent()
   ```

Find and Wink

When it is impractical to activate a device's service pin (for example, if the device is behind a wall or in a false ceiling), you can use the find and wink method to determine its Neuron ID. The LNS Object Server periodically looks for new devices that have been attached to the network, and places them in the `Discovered.Uninstalled` subsystem. When LNS discovers a device and places it in this subsystem, it sets the device's Neuron ID automatically.

To qualify as a "new" device, the device must not be configured. By default, the LNS Object Server searches for new devices once every three minutes. You can change the discovery interval by writing to the `System` object's `DiscoveryInterval` property. You can set the discovery interval to any value between 0 and 65,535 seconds. Set the property to the value 0 to turn off the background discovery process.

Follow these steps to determine which devices the LNS Object Server has discovered, and determine the Neuron ID of those devices:

1. Acquire the predefined `Discovered.Uninstalled` subsystem from the system's `Subsystems` collection:

   ```
   Dim MySubsystems as LcaSubystems
   Dim MySubsystem as LcaSubsystem

   Set MySubsystems = MySystem.Subsystems
   Set MySubsystem = MySubsystems.Item("Discovered.Uninstalled")
   ```

   Note the use of the combined subsystem path in this example. `Subsystem` objects can be retrieved in a single step, using a dot-separated syntax as shown in this example. In this example, the retrieved `Subsystem` object has the name "Uninstalled" and belongs to the "Discovered" subsystem. Whenever a `Subsystem` object's path is known within the subsystem hierarchy, it is more efficient to use the subsystem path to retrieve it, as opposed to using several distinct steps to retrieve one `Subsystem` per hierarchy level.

2. Get the AppDevices collection from this subsystem and iterate through the devices by index, using the Item property. To find the target, examine the properties of each AppDevice in the collection. For example, the program ID, channel ID, and location can be used to find the appropriate devices.

   The location field is a 6-byte field in the Neuron Chip used for storing installation-related information. This property is intended for use in situations where the device's physical location may be read from its I/O pins or programmed into the device prior to installation. For example, if a device is installed in a slot in a card cage, it can read the slot number from the edge connector.

```
Set MyAppDevices = MySubsystem.AppDevices
Counter = 1
MySystem.BeginTransaction()
MaxDevices = MyAppDevices.Count
While not DeviceFound and Counter <= MaxDevices
    Set MyAppDevice = MyAppDevices.Item(Counter)
   DeviceFound = AppDevice.Location = DesiredLocation
    Counter = Counter + 1
End While
MySystem.CommitTransaction()
```

3. To confirm that you have found the correct device, wink the device by invoking the Wink() method. The device should be programmed to respond in a way that can be easily detected by the person installing the device. For example, lights can blink, alarms can ring, and displays can flash. The person performing the installation can then identify the physical application device responding to a particular wink message as the one currently being installed.

   ```
   MyAppDevice.Wink()
   ```

4. Continue performing steps 2 and 3 until the correct device has been found. Then, commission the device, and add it to connections as your network design requires.

5. Note that in a fashion similar to the confirmed service pin protocol described earlier, a local technician could confirm the correct device performed the wink action by activating the device's service pin. Your application could obtain that service pin message via the OnServicePinEvent event, and then compare the originator device's Neuron ID with the value stored in the NeuronId property of the CurrentDevice returned by the event. If both match, your application could assume that the detected device is the desired one.

   In a fully automated approach, your application could use other methods to confirm the device's identity. For example, the application could compare the device's program ID with the expected value, it could query the device's Location property, or inspect the channel it is attached to by reading the AppDevice object's Channel property.

The LNS Object Server can also find configured devices. An LNS application may invoke DiscoverDevices() method on a system to discover configured devices that have not been added to the LNS database. When you call DiscoverDevices(), you must specify the DomainId and backgroundReg parameters. These specify the domain on which devices will be discovered, and whether registration will take place as a background or

foreground task. For more information, see the help page for the `DiscoverDevices()` method in the *LNS Object Server Reference* help file.

NOTE: You should not use find and wink on systems that use shared media. Instead, you should use the confirmed service pin protocol described earlier in this chapter, or you should manually enter the Neuron ID, as described in the next section.

Manual Entry

In many cases, the Neuron ID is manually supplied to the LNS application. Most LONWORKS devices are shipped with the Neuron ID printed on the housing or packaging. The Neuron ID for each device is also frequently supplied on stickers that can be peeled off during physical installation and attached to a device list or floor plan, allowing for the installer to scan these Neuron IDs at a later stage. Alternatively, the installer could be required to manually enter the Neuron ID for each device. Manually entering the data could be as easy as running a bar code reader over the coded location on the plan (representing the device's physical location), followed by running the reader over the bar code containing the Neuron ID of the device at that location. It is also possible to type in the Neuron ID as a series of 6 hexadecimal digits.

If a router is being installed while not connected to the network (e.g. when using the engineered mode installation described in Chapter 5), the Neuron IDs of the near and far sides of the router must be entered. You can do so by writing to the `NeuronId` property of the `RouterSide` object accessed through the `NearSide` and `FarSide` properties of the `Router` object. When installing a router while attached to the network, only the Neuron ID for the near side is required, and LNS will fetch the far side's Neuron ID from the device automatically. Note that the service pin message received from a router always provides the near side Neuron ID.

Loading Device Application Images

Neuron Chip-based devices are usually programmed with an application when they are manufactured. Although loading application images is normally not required for production-level devices, this is a common operation during device development, manufacture, and test. For production-level devices, application images may need to be reloaded into previously-installed devices to repair a damaged application, or to upgrade a device's capabilities.

To load a device's application image, the device's application must be accessible to the LNS Server PC in application binary format (.APB file). Note that you cannot load a device's application image while in engineered mode, and the system management mode must be set `lcaMgmtModePropagateConfigUpdates` when you load an application image.

To load an application image into a device, follow these steps:

1. Set the path to the application image file on the LNS Server PC by writing to the `AppDevice` object's `AppImagePath` property.

   ```
   MyAppDevice.AppImagePath = "c:\DeviceApps\Smkdtctr.apb"
   ```

   Application image files can reside in any location that can be reached through the file system on the LNS Server PC. Thus, setting the `AppImagePath` property to `C:\LonWorks\Import\Acme\Cooker.apb` will refer to the `cooker.apb` file in the

LonWorks\Import\Acme folder on the C: drive of the LNS Server PC. Application image files can also reside outside the LNS Server PC's local storage and referred to via mapped network drives or fully qualified filenames. The device plug-in software that comes with most LONWORKS devices should install the application image files into the correct locations.

2. Call the Load() method on the AppDevice. If the system image version (the Neuron Chip firmware) of the target device does not match that of the application image being loaded, the operation will fail and an exception will be generated. However, some devices support use of the LoadEx() method, which you can use to automatically upgrade the system image version if it is incompatible with the application image being loaded. For more information on the Load() and LoadEx() methods, see the *LNS Object Server Reference* help file.

   ```
   MyAppDevice.LoadEx(lcaLoadOptionsNone)
   ```

   Note: If the device's DeviceTemplate has already been associated with a particular program, loading an application image that uses a different program ID or external interface into the device will cause inconsistencies. If you attempt to load an application image with a different program ID, the device will be left in the unconfigured state after the load, and the NS#38 lcaErrNsProgramidMismatch exception will be thrown. In this case, the LNS Object Server will contain a record of the device's configuration, including the connections in which the old device was a member. To resolve the inconsistency, you must upgrade the device and then recommission it, as described in the *Upgrading Devices* section on page 134. Then, follow the steps above.

   Echelon recommends that you upgrade a device before loading its new application image whenever possible. If the loaded application has the same program ID but implements a different interface, the device will also be left in an unconfigured state. However, since LNS does not support devices with the same program ID and different interfaces, this inconsistency cannot be resolved without either loading a new application or removing the device.

Post-Load State

The state of the device at the end of the application loading process depends on whether the device had been commissioned before the process began, and on the application image that was loaded:

- If the device had not been commissioned prior to the application load, the LNS Object Server will leave the device in the unconfigured state.
- If the device had been previously commissioned, and the device's old application and its new application have the same program ID and the same external interface, the LNS Object Server will restore the device's network image (address and connection information) from the database.
- If the device had been previously commissioned, and the device's old application and its new application have a different program ID or a different external interface, the LNS Object Server will leave the device in the unconfigured state. If the program ID differs, the LNS application can use the Upgrade() method to upgrade the device to use the device template corresponding to the newly loaded application image, and then use the Commission() method to restore as much of

the device's configuration as possible. These features are described in more detail later in the chapter.

- If the device had been previously commissioned, and the device's old application image and its new application image have the same program ID but a different external interface, the LNS Object Server will leave the device in the unconfigured state, and the `NS:#59 lcaErrNsProgramIntfMismatch` exception will be thrown. Per LonMark guidelines, LNS requires that each program ID be associated with only one external interface. This means that all components and properties of each external interface using a given program ID must be identical. However, LNS may not detect all violations of this rule, as it would be very time consuming to validate this on every commission or after every application download.
- An application loading session may be canceled by calling the system's `CancelTransaction()` method from within an `OnSystemNssIdle` event handler, as described in *Using the OnSystemNssIdleEvent* on page 316. If the load is canceled, the device will be left in the applicationless state. In this case, the previous device application must be reloaded to restore the device.

Reloading a Device's Application

You should be aware that in some cases, reloading a device's application may cause the configuration properties on the device to be set to their original, default values. When reloading a device's application, Echelon recommends that you follow this procedure to ensure that the configuration properties in the device are restored to match those in the database in the most efficient manner:

1. Call `DownloadConfigProperties()` on the `AppDevice`. Use the `lcaConfigPropOptClearUpdatePending` value as the `downloadOptions` element.

   ```
   MyDevice.DownloadConfigProperties(lcaConfigPropOptClearUpdatePending)
   ```

2. Call `Load()` or `LoadEx()` to reload the device application, as described previously.

   ```
   MyDevice.LoadEx(0)
   ```

3. Call `DownloadConfigProperties()` again. This time, specify the `lcaConfigPropOptLoadValues` option as the `downloadOptions` element. This will load the configuration properties stored in the LNS database for the device back into the device.

   ```
   MyDevice.DownloadConfigProperties(lcaConfigPropOptLoadValues)
   ```

Commissioning Devices

After you have assigned a device its Neuron ID (and loaded its application image if necessary), you can enable it for communication with other devices on the network and load its network image by invoking the `Commission()` or `CommissionEx()` methods. This does the following:

- Gives the device a network (subnet/node) address in the system's domain if no `DeviceTemplate` or `Channel` were specified when the `AppDevice`

was created. If a DeviceTemplate and Channel were specified, the network address was assigned at that time.

- Extracts the external interface from the device, if no DeviceTemplate object was specified when the AppDevice object was created. If a DeviceTemplate object was specified, the commission process will validate the DeviceTemplate object against the device interface on the physical device.
- Sets all of the unbound address table and network variable configuration table entries in the device to the unbound state.
- Downloads connection information for the device, if the device was added to connections before it was commissioned.
- Sets the device's non-group receive timer to a default value that is based on the network topology.
- Sets or determines the device's channel.
- Sets the device's state to lcaStateCnfgOffline.

Using the Commission and Commission Ex Methods

The CommissionEx() method has been added to LNS for Turbo Edition. There is no functional difference between the CommissionEx() and Commission() methods, other than their behavior while the system management mode is set to lcaMgmtModeDeferConfigUpdates.

If you invoke the Commission() method while the system management mode is set to lcaMgmtModeDeferConfigUpdates, the physical device will not updated with the configuration changes caused by the commission process until the system management mode is set back to lcaMgmtModePropagateConfigUpdates. However, the CommissionEx() method allows you to specify options that will cause the physical devices to be updated if you are recommissioning a device while the system management mode is set to lcaMgmtModeDeferConfigUpdates.

This is useful if there are a large number of configuration changes pending, and you need to recommission a device without waiting for the rest of the changes to propagate to the network. Recommissioning a device while the system management mode is set to lcaMgmtModeDeferConfigUpdates updates the device with the configuration defined for the device when the system management mode was last set to lcaMgmtModeDeferConfigUpdates. This ensures that the network is configuration is consistent. It is not possible to propagate updates to a device while the while the system management mode is set to lcaMgmtModeDeferConfigUpdates if the device has never been commissioned before, as this may introduce network inconsistencies.

When using either method to commission a device while the system management mode is set to lcaMgmtModePropagateConfigUpdates, LNS will validate that the program ID, channel and interface assigned to the device are valid. If you invoke the Commission() method while the system management mode is set to lcaMgmtModeDeferConfigUpdates, LNS will not perform these validation steps. In contrast, the CommissionEx() method allows you to specify options to perform the validation while the system management mode is set to lcaMgmtModeDeferConfigUpdates.

NOTE: As of LNS Turbo Edition, each `DeviceTemplate` object includes a `DeviceValidation` property. The `DeviceValidation` property determines what validation steps LNS will perform when you commission devices using that `DeviceTemplate`. For more information on this, see the next section, *Device Validation Options*.

Echelon recommends that you perform initial commissions only while the system management mode is `lcaMgmtModePropagateConfigUpdates`. This allows LNS to validate the device's interface and channel, and to control the commissioning process. Suppose many devices have been defined and commissioned while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. When the system management mode is set back to `lcaMgmtModePropagateConfigUpdates`, LNS will immediately start commissioning all of these devices. This process may take a long time, and the LNS application will be "locked out" during this process.

See the *LNS Object Server Reference* help file for details on the syntax required when you call `Commission()` and `CommissionEx()`.

Device Validation Options

When you commission a device, LNS will validate that the physical device has the same external interface and program ID assigned to the `AppDevice` object in the LNS database. It will also validate that the physical device is on the channel assigned to the `AppDevice` object. If the physical device is not using the external interface or program ID assigned to it in the database, the commission will fail, and either the `NS#59 lcaErrNsProgramIntfMismatch` or the `NS#38 lcaErrNsProgramidMismatch` exceptions will be thrown. If the physical device is not on the channel assigned to it in the database, the commission will fail and the `NS#72 lcaErrNsWrongChannel` exception will be thrown.

As of LNS Turbo Edition, each `DeviceTemplate` object includes a `DeviceValidation` property. This determines what validation steps LNS will perform when you commission devices that use the `DeviceTemplate`. This will be useful if you are commissioning a large number of devices that are on a slow channel, and you are confident that the devices contain the correct program information and are installed on the correct channel. If you want to reduce the time it takes to commission the device in this situation, you can set the `DeviceValidation` property to any of the following values:

- `lcaDeviceValidationNoChannelValidation`: Do not validate the device's channel.
- `lcaDeviceValidationNoProgramInterfaceValidation`: Do not validate the device's program interface.
- `lcaDeviceValidationNoProgramIDValidataion`: Do not validate the device's program ID.

See the *LNS Object Server Reference* help file for more information on these options. The default is for LNS to perform all validation steps. If you modify this property from the default, you should be sure that the `Channel` and `ProgramId` properties of all `AppDevice` objects using the device template have valid settings before commissioning those devices. Failure to do so may cause improper configuration of the device, and may make it impossible to communicate with the device.

Device Configuration Considerations

Before commissioning a device, you should make sure that the configuration property information contained in the LNS database for the device is complete, and you should download the configuration property values in the LNS database into the physical device.

To do so, call `DownloadConfigProperties()` on the `AppDevice` before you commission it. Use the `lcaConfigPropOptLoadValues` and `lcaConfigPropOptLoadUnknown` download options when you call `DownloadConfigProperties()`. This will set any unknown configuration properties in the `AppDevice` object in the LNS database to their default values, and then set all configuration property information in the physical device on the network to match the information stored in the LNS database. As a result, when the device is commissioned, it will contain current values for any configuration properties that have been explicitly set, and default values for any configuration properties that were unknown before the download.

Note that by using `DownloadConfigProperties()` as described above, you will preserve the information stored in the LNS database for the `AppDevice`, by changing the configuration property information stored in the physical device. In some cases, you may want to do the opposite. You may want to change the configuration property information stored in the LNS database to match the information stored in the physical device on the network.

This can usually be accomplished by calling `UploadConfigProperties()` on the `AppDevice`, but you cannot use this method on a device until it has been commissioned. If you are commissioning a device for the first time and want to preserve the configuration property information stored in the physical device, the solution is to call `DownloadConfigProperties()` with the `lcaConfigPropOptSetUnknown` value as the download option before you commission the device. All configuration property information in the database will be set to unknown status, but the configuration property in the physical device will not be affected. Once the device has been commissioned, you can call `UploadConfigProperties()` with the `lcaConfigPropOptLoadValues` value as the upload option to set the configuration property information in the database to match the configuration property information stored in the physical device.

Echelon recommends that you use the same explicit transaction to call the `DownloadConfigProperties()` method, and the `Commission()` or `CommissionEx()` methods. Otherwise, this procedure will take longer and consume more network bandwidth than when performed in separate transactions.

For more information on transactions, see *Using Transactions and Sessions* on page 65. For more information on how you can use the `DownloadConfigProperties()` and `UploadConfigProperties()` methods, see *Configuring Devices* on page 125.

LNS Licensing Considerations

All applications using the standard LNS Device Credit licensing model will require one LNS Device Credit for each device being commissioned. Therefore, when commissioning devices, you need to make sure that you have a sufficient number of LNS Device Credits available. For more information on this, see Chapter 13, *LNS Licensing*.

Application devices have many different types of configuration information. This data can be roughly organized into two classes, generic configuration data and application configuration data. This section defines these two classes of data, and describes how you should manage each device's configuration.

Generic Configuration Data

Some configuration data is generic to all LONWORKS devices. This includes information required by the network protocol, as well as device properties that are present on all standard application devices. The LNS Object Server reads this configuration data automatically during installation, and exposes it through the properties of the `AppDevice` object. Two examples are the `Priority` and `Location` properties.

You can use the `Priority` property to assign a device a specific priority slot. Set the property to 255 to cause the LNS Object Server to assign the device the next available priority slot. After a device's priority slot is modified, the LNS Object Server automatically resets the device to make the change take effect. A value of 0 indicates the device uses a standard, non-priority communication slot, which most typical devices use.

You can use the `Location` property to set a device's location field. The location field is a 6-byte field in the Neuron Chip used for storing installation-related information. To promote language independence, text should not be stored in this field. Instead, it should contain an index or code that can be mapped to text at the user interface. For example, in a building control system, the floor and room number could be stored in the location field.

The `AppDevice` object contains a variety of other properties containing useful information, such as the device's program ID, authentication setting, configuration state, and channel. When installing a device, you should set these properties, or check that they are set to proper values. For a complete list and descriptions of the properties of the `AppDevice` object, see the *LNS Object Server Reference* help file.

NOTE: The LonMark Interoperability Association recommends that you set the SCPTlocation configuration property on each device to match the subsystem path of the associated `AppDevice` object. This will provide optimum support for network recovery. For more information on this, see *Application-Level Recovery* on page 262.

Application-specific Configuration Data

Application configuration data is defined using *configuration properties*. Configuration properties can be implemented as configuration network variables or configuration parameters. Any device can use configuration network variables. Devices that comply with the *LonMark Application Layer Interoperability Guidelines*, version 3.0 or later, can use configuration parameters. Configuration parameters are stored in files on the device, and are accessed using the LonTalk file transfer protocol or direct memory read/write. The LNS Object Server automatically determines and uses the best transfer option. Configuration properties may be properties of the device, a `LonMarkObject` object, or a network variable.

Configuration property definitions are contained in device external interface files. If a `DeviceTemplate` was created using an external interface file, then configuration property definitions are defined for the `DeviceTemplate`, and for every device using

that template. The configuration property defaults are also defined for the DeviceTemplate, but the configuration properties of the devices using that template are not automatically set to the default values.

If a DeviceTemplate was recovered from a device, the configuration properties for that template will initially be undefined. The ConfigPropertiesAvaliable property will be set to False in this case. You can load the configuration property definitions for the associated device template from the physical device using the UploadConfigProperties() method, as described in the next section.

Downloading and Uploading Configuration Properties

You can use the UploadConfigProperties() method to upload configuration property values from a physical device on the network into the associated AppDevice object in the LNS database. You can use the DownloadConfigProperties() method to download configuration property values stored for the AppDevice object in the LNS database to the physical device that the object represents.

The configuration property data for a device is not automatically added to the LNS database when the device is added to the LNS database and commissioned. Therefore, your LNS application should ensure that the configuration property information stored in the LNS database for a device is synchronized with the configuration information stored in the physical device on the network before commissioning the device. You can use the DownloadConfigProperties() and UploadConfigProperties() methods to do so. For guidelines on this, see *Device Configuration Considerations* on page 124.

After you have associated a device with an interface (by commissioning it, or by specifying a DeviceTemplate and Channel when you create the AppDevice), you can use the DownloadConfigProperties() and UploadConfigProperties() methods for a variety of tasks, including the following:

- Setting configuration properties to their default values.
- Setting unknown configuration property values.

To set the configuration properties for a device to their default values, use one of the following two methods:

- If the device is using a device template based on an external interface file that defines default values for the device's configuration properties, set all the configuration properties to the default values in the external interface file by calling DownloadConfigProperties() with the lcaConfigPropOptSetDefaults and lcaConfigPropOptLoadValues options set. Do not specify the lcaConfigPropOptIncludeMfgOnly option, as this will cause calibration parameters set by the manufacturer to be overwritten. This operation will ensure that devices that were previously installed on another network are returned to their default state.
- Set the default configuration property values in the LNS database to match the current values stored on the physical device on the network by calling UploadConfigProperties() with the lcaConfigPropOptSetDefaults and lcaConfigPropOptLoadValues options set. Note that the UploadConfigProperties() method can only be called after the device has been commissioned and updated by LNS.

To set unknown configuration property values, use one of the following two methods:

- If the device is using a device template based on an external interface file that defines default values for the device's configuration properties, set all unknown configuration properties in the physical device on the network to match the default values stored in the LNS database by calling `DownloadConfigProperties()` with the `lcaConfigPropOptLoadUnknown` and `lcaConfigPropOptLoadValues` options set.

  You can determine the device template's basis by inspecting the `DeviceTemplate` object's `ProgramType` property.

- Set all unknown configuration properties in the LNS database to match the values stored in the physical device on the network by calling `UploadConfigProperties()` with the `lcaConfigPropOptLoadUnknown` and `lcaConfigPropOptLoadValues` options set.

You should note that there are other options you can choose from when calling `DownloadConfigProperties()` or `UploadConfigProperties()`. For example, you can use the `lcaConfigPropOptOnlyDeviceSpecific` option to only download or upload configuration properties with the device-specific attribute set. Or, you can use the `lcaConfigPropOptExcludeDeviceSpecific` option to exclude configuration properties with the device-specific flag set. For more information on device-specific configuration properties, see *Device-Specific Configuration Properties* on page 234.

When downloading configuration properties, you can use the `lcaConfigPropOptClearUpdatePending option` to clear all pending updates on the device affected by the download. Or, you can specify the `lcaConfigPropOptIncludeMfgOnly` option to only include manufacturing-only configuration properties in a download.

There are several other options you can choose from when downloading or uploading configuration property information. For more information, see the *LNS Object Server Reference* help file.

NOTE: Some devices implement configuration properties that cannot be accessed unless the device is in the online state (i.e. the `State` property is set to `lcaStateCnfgOnline`). These are devices that use the file transfer protocol for configuration property access. You can identify these devices by the existence of the SNVT_file_req and SNVT_file_status type network variables in the `NetworkVariables` collection that belongs to the `AppDevice` object's `Interface` object.

Writing Configuration Property Values

You can also use LNS to read or write to the values of the configuration properties on a device one at a time. To do so, call the `GetDataPoint()` method on the `ConfigProperty` object to acquire a `DataPoint` object. If you have not written a configuration property, or set its default value, you cannot read it if any of the following conditions are true:

- The device has not been commissioned.
- The LNS Object Server cannot communicate with the device.
- The device template is based on a definition that was uploaded from the device, and the configuration properties have not yet been uploaded.

For more information on using `DataPoint` objects with configuration properties, see *Using the GetDataPoint Method* on page 235.

Note that there is one case where you may need to read or write to the value of a `ConfigProperty` object directly, without using a `DataPoint` object. Data points are very useful when reading and writing formatted data, or when accessing an entire raw data value as a whole. However, if you want to access arbitrary bytes of raw data to read or write a range of elements in an array configuration property, you should not use a data point. Instead, you should use the `GetRawValuesEx()` and `SetRawValuesEx()` methods of the `ConfigProperty` object. See the *LNS Object Server Reference* help file for more information on these methods.

Setting Devices Online

The final step in installing a device is to set the device online. To do so, set the `AppDevice` object's `State` property to `lcaStateCnfgOnline` as displayed below:

```
MyAppDevice.State = lcaStateCnfgOnline
```

You can set a device online or offline at any point by writing to its `State` property. However, you should be aware that devices cannot receive or respond to network events related to monitor and control while they are offline. For example, if a Network Service Device is offline, then all applications using the Network Service Device will not receive monitor and control events until the Network Service Device is back online. Or, if an application device is offline, then that device will not be able to receive incoming network variable events or respond to network variable polls.

When you consider this, you should note that devices are taken offline while they are being reconfigured. For example, if you remove a connection between an application device and the Network Service Device, both the application device and Network Service Device will be taken offline while the LNS removes the connection. During that time, the Network Service Device will not process network variable updates, nor will it poll network variables, since the configuration of the application device and the configuration of the Network Service Device are in a state of fluctuation.

Although Echelon recommends that you only modify the device's configuration while it is offline, some devices may need to be reconfigured while they are operational. Therefore, you can set the `ConnectionUpdateType` property of an `AppDevice` object to `lcaConnectionUpdateTypeOnline` to prevent the device from being taken offline while changes to network variable connections on the device are being made.

Note: If you are using an explicit transaction to install and configure a device, you must commit the transaction before setting the device online. For more information on transactions, see *Using Transactions and Sessions* on page 65. If you are installing multiple devices at once, you should first install and configure all the devices, and then set them online. This prevents the devices from attempting to communicate on a partially installed network.

Other Device Management Operations

This section describes tasks you may need to perform when managing the devices you have installed on your network. This includes the following:

- *Testing Devices and Detecting Device Failures*
- *Replacing Devices*
- *Upgrading Devices*
- *Decommissioning Devices*
- *Removing Devices*

Testing Devices and Detecting Device Failures

To test the current functionality and state of a device, call the `Test()` method on the appropriate `AppDevice` object. LNS will then perform a series of tests on the device, in this order:

1. LNS will verify that it can address the device by its Neuron ID. If this test fails, it indicates either a problem with the device (the device either is powered off, detached from the network, or defective), a problem with the channel (physical problem or too much network traffic), or a problem with an intervening router (overloaded, offline, powered off, detached from the network, or defective).
2. LNS will compare the network address retrieved from the device with the network address stored in the LNS database for the device. A mismatch indicates a problem with the device. It could also indicate that the device has been configured by an LNS application using a different network database or by a non-LNS network tool.
3. LNS will verify that it can communicate with the device using the subnet/node address assigned to the device in the LNS database. Failure to communicate using subnet/node addressing may indicate a problem with the configuration of one or more routers on the network. This could occur if one or more routers on the network has been configured using a different network or by a non-LNS network tool.
4. LNS will verify that only one device on the network is using the network address specified for the device. Multiple devices with the same network address may indicate that a configured device was moved to this network from another one.
5. LNS will validate the device's program ID, and the device's authentication setting.

When an application device is tested, the details and results of the test are contained in the `LastTestInfo` property of the `AppDevice` that was tested. When a router is tested, the results can be found in the `LastTestInfo` property of the router's `RouterSide` objects. This property contains a `TestInfo` object that contains properties that will

return the expected and actual values of the device attributes tested. If a test fails, the `AuxResultData` property of the `TestInfo` object will indicate whether a Neuron ID, domain ID, subnet/Node ID, or program ID mismatch caused the test to fail.

When you call `Test()` on a router, a router failure can result in failure reports for all of the devices on channels that are accessed via the failed router. For more information on routers and on router failures, see *Managing Networks with Multiple Channels* on page 169.

Note that it can take a significant amount of time to test a device, depending on the complexity of the tests performed in the device's application. For more information on the `Test()` method and the various properties of the `TestInfo` object, see the *LNS Object Server Reference Help File*.

Using the OnAttachment Event

The `Test()` method causes several messages to be sent to a device. An alternative way to test a device that requires fewer messages is for the LNS Server to automatically ping devices using the LonTalk Query Status network diagnostic message. The LNS Object Server will determine whether or not a device has become detached from the network if it fails to respond to a ping, or if it returns an unexpected response to a ping. If a device does not respond to a ping, the LNS Object Server tests the intervening routers to verify that the failure is due to the device, and not to a router along the communications path.

The frequency of the pinging is determined by the device's `PingClass` property. This property may be set to one of the following values:

`lcaPingClassMobile`	Check detachment frequently (1 minute by default).
`lcaPingClassTemporary`	Check detachment less frequently (2 minutes by default).
`lcaPingClassStationary`	Check detachment occasionally (15 minutes by default).
`lcaPingClassPermanent`	Never check.

You can set the interval assigned to each of the four ping classes by writing to the `System` object's `PingInterval` property. The default class is `lcaPingClassStationary`, unless the `System` object's `InstallOptions` property is set to `lcaSharedMedia`, in which case pinging will be disabled.

When the LNS Object Server discovers that a device has become attached or detached, it uses the `OnAttachment` event to report this information. To program your application to test for device failures using attachment events, call the `BeginAttachmentEvents()` method on the `System` object to register your application for the `OnAttachment` event. Filter the `OnAttachment` events your application receives for device failures. These are events where the `IsAttached` parameter returned with the event is `False`, and the device has not been manually removed from the network. You could then subject those devices and routers to more detailed investigation using the `AppDevice` or `Router` object's `Test()` method. However, your `OnAttachmentEvent` event handler should not contain code to examine suspect devices. It should return to LNS as soon as possible, and

notify some other component of your application to perform a thorough test on any suspect devices. For more guidelines on creating LNS event handlers, see *Event Handling* on page 67.

Note that when a router becomes unattached, this will usually prevent LNS from communicating with any devices on the far side of the router, and it will prevent communication between devices on channels connected by the router. In this case, LNS will generate an `OnAttachmentEvent` for the router, but not for each affected device.

Performing Diagnostics on LonMarkObjects

As described in *Device Interfaces* on page 104, each application device contains a group of `LonMarkObject` objects that represent the LonMark Functional Blocks on that device. The LNS Object Server recognizes LonMark Functional Blocks defined in devices that conform to the *LonMark Application-Layer Interoperability Guidelines*, version 3.0 or later. Each `LonMarkObject` has a `Status` property that reflects the current status of the LonMark Functional Block represented by that specific `LonMarkObject`.

Each `LonMarkObject` also contains a `SelfTestResults` property. When your application reads the `SelfTestResults` property, the device application will perform a self-test on the `LonMarkObject`, and return an `ObjectStatus` object describing the results of the self-test. If the self-test takes more than 20 seconds to complete, an exception indicating failure will be raised. In this case, poll the `SelfTestInProgress` property of the `LonMarkObject` until it returns `False`. Then, check the `FailSelfTest` property to determine why the self-test failed. Another property (`LonMarkAlarm`) indicates the current alarm status of the `LonMarkObject` object, if the object supports alarms.

Figure 6.3 depicts these steps.

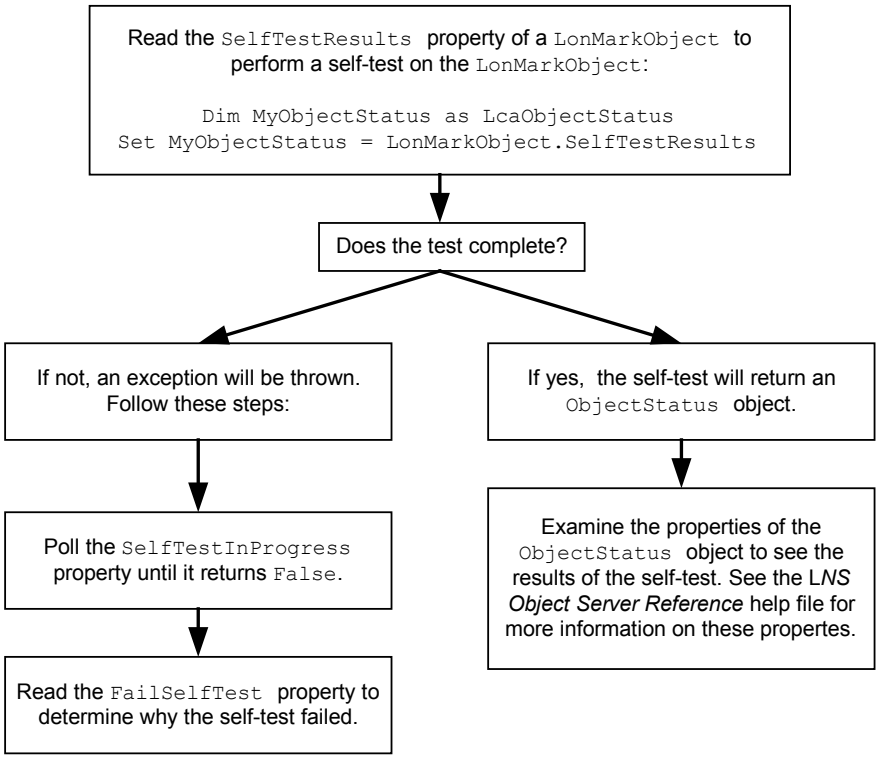


Figure 6.3 Performing a Self-Test on a `LonMarkObject`

NOTE: The `LonMarkAlarm` object supports alarming on devices that implement their alarms through `SNVT_alarm` network variables, but not through the more recent `SNVT_alarm2` type. However, `SNVT_alarm2` type network variables can be monitored and controlled using the standard monitor and control techniques described in Chapter 9 of this document.

To receive meaningful results for the `SelfTestResults` and `Status` properties of the `LonMarkObject`, the device must support these features. If the device does not comply with version 3.0 or later of the *LonMark Application-Layer Interoperability Guidelines*, then the `LonMarkObjects` and `ConfigProperties` collections will be empty for the device, and the properties mentioned in this section will be inaccessible.

Some devices that comply with version 3.0 or later of the *LonMark Application-Layer Interoperability Guidelines* may not support the self-test feature. Reading the `SelfTestResults` property in this case will cause an exception to be thrown. You can use the `ReportMask` property of the `LonMarkObject` object to determine whether a device supports self-tests.

Replacing Devices

This section describes how to replace a device with a new device that contains the same application image and external interface as the old one. Replacing a device in a LONWORKS network is different than replacing a device in a hardwired system. You cannot simply unplug the old device, and plug in a new one to take its place in a LONWORKS network. You must transfer the network image (network address and connection membership) from the old device to the new device. If the old device contained

application configuration data, it might also be necessary to set the configuration properties of the new device to match the configuration property values in the old device.

As described earlier, the AppDevice object represents the logical attributes of a physical device in the LNS database, including its network address, connections, and configuration properties. When the AppDevice object is commissioned, it becomes associated with the physical device on the network. When an AppDevice object is replaced, the AppDevice object is updated to represent a new physical device. To replace a device, install the new device on the network. Then, follow these steps:

1. Obtain the AppDevice object representing the original device in the LNS database.
2. Acquire the new device's Neuron ID using any of the methods described in the *Neuron ID Assignment* section on page 115. Assign the acquired Neuron ID to the NeuronId property of the AppDevice acquired in step 1.
3. If necessary, load the new device's application image, as described in *Loading Device Application Images* on page 119. Use the AppDevice object acquired in step 1.
4. Start a transaction.
5. Call Replace() or ReplaceEx() on the AppDevice being replaced. The LNS Object Server will then perform the following tasks:
    - It deconfigures the old device if it can communicate with it.
    - It loads the network image of the old device into the new device.
    - It leaves the new device offline.

    If you use the Replace() method and the system management mode is set to lcaMgmtModeDeferConfigUpdates, these tasks will not be performed until the system management mode is set back to lcaMgmtModePropagateConfigUpdates. If the system management mode is set to lcaMgmtModeDeferConfigUpdates and you want these tasks performed right away, you should use the ReplaceEx() method with the lcaReplaceFlagPropagateUpdates option as the flags element to perform the replacement.

    When you use the ReplaceEx() method, you can also use the flags parameter to determine whether any of the configuration property values from the old device will be preserved in the LNS database. Otherwise, the configuration properties will be set to the unknown state. Echelon recommends that you use ReplaceEx() and preserve the old configuration property values whenever possible.
6. Download the configuration property information for the new device into the LNS database by calling DownloadConfigProperties() with the lcaConfigPropOptLoadValues option set.
7. Commit the transaction.
8. Set the new device online by setting the State property of the AppDevice object to lcaStateCnfgOnline.

Replacing Network Service Devices

You should not use the Replace() or ReplaceEx() methods on a Network Service Device. Generally, LNS will perform Network Service Device replacements

automatically. However, you may need to manually replace your Network Service Device in some cases. You will need to use the `PreReplace()` method to do so.

For example, you will need to use this method if you open a network remotely from a new PC, and want that client to use the Network Service Device configuration that was previously associated with another remote client PC (effectively moving the remote application and Network Service Device configuration from one PC to another). An exception to this is if the original remote client used a standard network interface, and you move the network interface to the new PC as well. In this case, LNS will automatically associate the new PC with the original client based on the standard network interface's Neuron ID.

You will also need to follow the procedure described below to reattach a Network Service Device to a network if the network has been removed from the `RemoteNetworks` collection for the PC, and you are using a high-performance (Layer 2/VNI) network interface, or if you install a new network interface on the PC.

To re-associate a client with the correct Network Service Device and re-attach the client to the network, follow these steps:

1. Get the Network Service Device to be attached to from the `NetworkServiceDevices` collection.

   ```
   Dim MyNSD as LcaNetworkServiceDevice
   Set MyNSD = MyNetworkServiceDevices.Item("Supervisor Room")
   ```

2. Call `PreReplace()` on the network you want to attach your application to, with the selected Network Service Device as the `sourceNSD` element.

   ```
   MyNetwork.PreReplace(MyNSD)
   ```

3. Close the network, and release all references to the network.

   ```
   MyNetwork.Close()
   ```

4. Call `Replace()` on the network.

   ```
   MyNetwork.Replace()
   ```

5. Call `Open()` on the network to open the network with all previously created monitor sets present.

   ```
   MyNetwork.Open()
   ```

Upgrading Devices

When a device has a new application loaded, or when a device is replaced, the device's external interface may change. You can call the `Upgrade()` method to upgrade the device to be compatible with the updated the external interface with minimal disruption to existing connections and configuration. You can use the `Upgrade()` method on all device types. However, it generally only makes sense to upgrade a device to support functionality similar to the device's original purpose (e.g. to upgrade to a later version of the device).

To upgrade a device, follow these steps:

1. Call the `System` object's `StartTransaction()` method. You should always call the `Upgrade()` method within a transaction. This allows for easy reversal of the upgrade. For more information on transactions, see *Using Transactions and Sessions* on page 65.

2. If the device upgrade includes new hardware, make sure the new physical device is attached to the network, and that the `AppDevice` object's `NeuronId` property is set to the proper Neuron ID.

3. Call the `Upgrade()` method on the `AppDevice` object. This method optionally takes a `DeviceTemplate` object as a parameter. If a `DeviceTemplate` is not supplied, the new external interface will be read directly from the device. If your application is not attached to the network, you must supply a `DeviceTemplate`, or an exception will be generated.

   NOTE: In some cases, you may need to upgrade your Network Service Device, usually because its network interface has changed. Generally, LNS will upgrade the Network Service Device automatically. However, in some cases, you may need to perform the upgrade manually. In this case, you should not supply a `DeviceTemplate` object. For more information, see *Network Interfaces and Network Service Devices* on page 273.

4. The `Upgrade()` method returns an `UpgradeStatus` object. This object contains the `Result` property, which will bet set to `lcaUgResSuccess` if the upgrade was successful. If the upgrade was not successful, call the `CancelTransaction()` method on the `System` object to back out of the upgrade.

5. The `UpgradeStatus` object also contains an `UpgradeInfos` collection. This collection contains one `UpgradeInfo` object for each LonMark Functional Block, network variable, message tag, monitor set, monitor point, and configuration property in the original external interface. Each `UpgradeInfo` object contains a summary of how the old LonMark Functional Block, network variable, message tag, configuration property, monitor set, or monitor point is represented in the new interface. This includes the following properties:

`Class`	This property indicates whether the `UpgradeInfo` object represents a LonMark Functional Block, network variable, message tag, configuration property, network variable configuration property, monitor set, or monitor point.
`FromIndex`	This property returns the device index value assigned to the external interface component represented by this `UpgradeInfo` object in the external interface before the upgrade.
`FromOwnerIndex`	For `UpgradeInfo` objects that represent configuration properties that are contained within `LonMarkObjects` or network variables, this property returns the device index value assigned to the owner `LonMarkObject` or network variable in the external interface before the upgrade. Otherwise, this property returns –1.
`ToIndex`	This property contains the device index value

	assigned to the external interface component represented by this `UpgradeIndo` object in the external interface after the upgrade.
`ToOwnerIndex`	For `UpgradeInfo` objects that represent configuration properties that are contained within `LonMarkObjects` or network variables, this property returns the device index value assigned to the owner `LonMarkObject` or network variable in the external interface after the upgrade. Otherwise, this property returns –1.
`Status`	This property indicates whether the external interface component represented by this object was deleted, retained, or moved during the upgrade. If the component is a network variable or message tag, it also indicates whether the component was removed from some or all of its connections.
`Reason`	This property indicates why the object was deleted, retained, or moved during the upgrade.

6. After examining the `UpgradeStatus` object, call the `CommitTransaction()` method to commit the changes to the LNS database and complete the upgrade. Remember that you can still call `CancelTransaction()` to cancel the upgrade at this point.
7. If you are upgrading the device so you can load a new application, load the new application into the device, as described in *Loading Device Application Images* on page 119.
8. Recommission the device. This completes the upgrade process. For more information on commissioning devices, see *Commissioning Devices* on page 121.

Decommissioning Devices

At some point, you may want to disassociate an `AppDevice` or `Router` object in the LNS database from the physical device on the network, without removing the device from the LNS database. This could allow you to test different designs for a network, without consuming additional LNS Device Credits each time you remove, re-add, and re-commission a device to test a different network design.

To facilitate this, LNS provides a `Decommission()` method. You can call this method on an `AppDevice` or `Router`. This sets the device's `NeuronId` property to `000000000000`, and deconfigures the device or router. Note that manually setting the `NeuronId` property to `000000000000` will not cause a device to be decommissioned.

When you decommission a device, an LNS Device Credit will be returned to your credit pool. When you are ready to restore the device to normal operation, you can recommission the device, and an LNS Device Credit will be charged. For more information on LNS Device Credits and LNS licensing, see Chapter 13, *LNS Licensing*.

Moving Devices and Managing Networks With Multiple Channels

If you are managing a network with multiple channels, there are many special tasks you will need to perform. This includes the installation and configuration of the routers on your network. For information on these tasks, see *Managing Networks with Multiple Channels* on page 169. This section also discusses how you can move a device from one channel to another.

Removing Devices

During the lifetime of a network, you may need to remove devices from service. This could be for a number of reasons. You may not need the device anymore, or you may want to move the device to another network.

A device in a network should be logically removed from the LNS database before it is physically removed from the network. The logical removal process clears device's logical address and authentication key. It also sets the device to the offline state, so that the device can be installed in another network, and then rediscovered by another LNS Object Server, if necessary. If you are moving the device to another network, this also ensures that attaching the device to a new network will not cause address duplication, which could lead to improper system behavior. The logical removal process also updates devices that communicated with the target device, so that they no longer attempt to communicate with it.

To logically remove a device, invoke the `Remove()` method on the `AppDevices` collection containing the device. You will need to specify the device by its name or collection index value. If the device only belongs to one subsystem, this disconnects the device's network variable and message tags, removes the device's network address, and places the device in the unconfigured state. If the `AddReference()` method has been used to place the device in multiple subsystems, the device must be removed from the `AppDevices` collection for all the subsystems it has been added to. You can determine which subsystems a device belongs to by reading its `Subsystems` property. When completely removed, the device is left in the unconfigured state and its service LED (if present) flashes slowly.

This example removes an application device named AppDevice1:

```
Dim MyAppDevices As LcaAppDevices
Set MyAppDevices = MySubsystem.AppDevices
MyAppDevices.Remove("AppDevice1")
```

Removing Devices From Multiple Subsystems

The preceding example removes a device from a single subsystem. Remember that if an `AppDevice` is referenced in multiple subsystems, you will need to remove it from all the subsystems it belongs to in order to completely remove it from a network database. Once you have removed the `AppDevice` from all the subsystems it belongs to, the physical device on the network will be removed from connections, set to the unconfigured state, and be ready to be physically removed from the network. The device will indicate that it has reached the unconfigured state by slowly flashing its service LED. Note that it is not an error if LNS fails to communicate with the device. Since the device might be logically removed as a result of a prior physical removal, LNS will update all connections the

device was associated with accordingly, but will not signal an error if the device itself cannot be communicated with.

To remove a given `AppDevice` object that is referenced in multiple subsystems, collect the list of subsystems it belongs to by reading the `AppDevice` object's `Subsystems` property. This returns a `Subsystems` object. Obtain the `AppDevices` collection for each `Subsystem` object contained in the collection, and remove the `AppDevice` object from each of them. You should delete the `AppDevice` by its name and not its index, as the index will vary for each AppDevices collection it belongs to.

The following example removes an application device named AppDevice1 from all subsystem objects it belongs to:

```
‘ Start a transaction:
MySystem.StartTransaction

‘ Fetch the AppDevice that is to be removed:
Dim MyAppDevices As LcaAppDevices
Dim MyAppDevice As LcaAppDevice
Set MyAppDevices = MySubsystem.AppDevices
Set MyAppDevice = MyAppDevices.Item(“AppDevice1”)

‘ Fetch this device’s Subsystems object:
Dim TheSubsystems As LcaSubsystems
Set TheSubsystems = MyAppDevice.Subsystems

‘ Remove the device from all Subsystem objects it belongs to
Dim CurrentSubsystem As LcaSubsystem
Dim CurrentAppDevices As LcaAppDevices
While TheSubsystems.Count
   Set CurrentSubsystem = TheSubsystems.Item(1)
   Set CurrentAppDevices = CurrentSubsystem.AppDevices
   CurrentAppDevices.Remove(“AppDevice1”)
End While

‘ Release the stale reference held by the MyAppDevice variable:
Set MyAppDevice = nothing

‘ Commit the transaction:
MySystem.CommitTransaction
```

Note that the `MyAppDevice` variable becomes stale during this process. As the last reference to the `AppDevice` is removed from the database, the variable itself refers to an item that no longer exists. Accessing any property or method of the `MyAppDevice` object variable will cause the `LCA:#116 lcaErrStaleObject` exception to be thrown. To prevent this, you should release stale references as shown in the example above.

Connecting Devices

To understand how to use LNS to connect network variables and message tags, you should first consider how Neuron Chip-based devices process incoming messages. There are two pieces of information contained within each device that the Neuron Chip uses to process and qualify an incoming network variable update messages. The first is the message's destination address. The device only processes the message if the message is addressed to the device. If the device determines that the message is addressed to it, and the message is a network variable message, the device checks if the network variable selector in the message matches a network variable selector on the device.

Network variable selectors are 14-bit numbers assigned by the LNS Object Server that identify connected network variables (i.e. network variables that are part of the connection). Devices may use different names to refer to a network variable, or network variables may be located at different offsets within each device's memory, resulting in a different network variable index within each device. However, all network variables in a connection must have the same network variable selector value. In addition, each network variable can have only one network variable selector, unless network variable aliases are used, as described later in this section. This allows the device to uniquely identify each network variable.

This does not mean that all network variable selectors in a network need to be unique. Any network variable selector can be used multiple times in the same network, provided that the devices using the same selector do not share connections or network addresses, so that the selector is unambiguous to all of the devices.

This does not also mean that only a single selector may apply to a given network variable. The LNS Server supports network variable aliases transparently. Network variable aliases allow LNS to map multiple selector values to a single input or output network variable. The number of network variable aliases on each device is defined by the device manufacturer, and cannot be changed by an LNS application, but the LNS Server will take advantage of available aliases, if necessary.

The LNS Object Server defines connections in terms of *hubs* and *targets*. The *hubs* and *targets* are the network variables and messages tags that are bound by the connection. The hub is the center of a connection. The rest of the network variables and message tags in the connection are the targets.

If the hub is an input, then all of the targets must be outputs. Likewise, if the hub is an output, then all of the targets must be inputs. Network variables and message tags may be accessed through an `AppDevice` object's `Interface` property. Network variables that are the hubs of connections can be accessed through an `AppDevice` object's `NVHubs` property. Figure 6.4 shows two connections. The connection on the left would be specified with the nviValue network variable on Device One as the hub, and the nviTemp and nviTemp2 network variables on Devices Two and Three as the targets. The connection on the right would be specified with the nvoStatus network variable on Device Four as the hub and the nvoSetPoint and nvoControl network variables on Devices Five and Six as the targets. This second example also illustrates the fact that `NetworkVariable` objects can share names and thus are uniquely identified only in combination with the objects to which they belong (i.e. `AppDevice` or `LonMarkObject`).

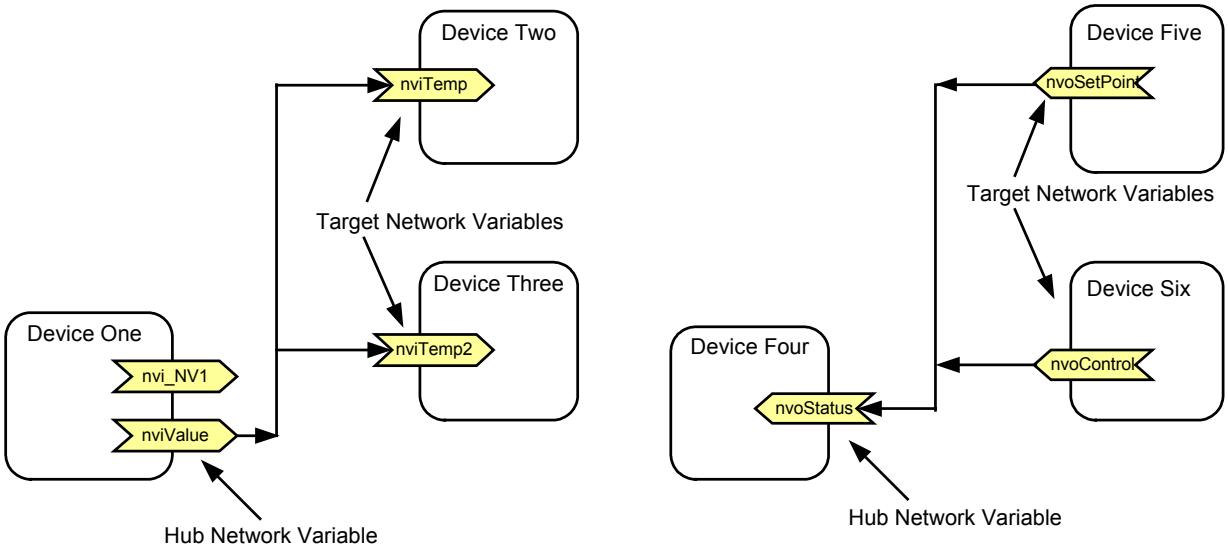


Figure 6.4 Example Connections

Connection Rules

There are several rules you need to consider when creating connections. For network variable connections, the LNS Object Server enforces these rules:

- Network variables can only be connected to network variables.
- The hub must be either the only input or the only output in the connection.
- There must be one hub, and at least one target, in each connection.
- All the network variables in the connection must have compatible types. If SNVTs are used, the LNS Object Server will validate that all members are the same SNVT type. For non-standard network variables types, the LNS Object Server will verify that all members are the same length, if length information is available. If length information for a network variable is not available, the LNS Object Server will treat the network variable as typeless. The LNS Object Server allows typeless network variables to be bound to any other network variable type. It is the responsibility of the LNS application to prevent nonsensical connections from being formed from typeless network variables.
- A polled output network variable can only be connected to polling input network variables.
- Authenticated inputs can only be bound to authenticated outputs. Authenticated outputs may be connected to unauthenticated inputs.
- Devices can have no more than one output and, in some cases, one input network variable within a given connection group (i.e. those connections that share the same network variable selector)

The LNS Object Server allows the use of network variable aliases. Aliasing allows a network variable to have multiple selector values. In addition, the relaxed connection constraints that are available with newer versions of the Neuron Chip firmware sometimes allow selector values to be used more than twice in a given device. Alias table

entries must be defined when the application is created by the device manufacturer, and are located in non-volatile memory. Available alias table entries are used by the LNS Object Server to create connections that would otherwise violate the rule requiring devices to have no more than one input and, in some cases, one output network variable within a given connection group.

Figure 6.5 shows an example of a connection that can only be made with aliases. A single network variable (nvi_Output) on Device One is connected to two separate network variables (nvi_Input1 and nvi_Input2) on Device Two. The LNS Object Server creates one connection using the primary output network variable in Device One, and creates the other using an alias table entry on Device One. The second, implicit connection using the output network variable alias is required because two different selector values are needed to update the input network variables on Device Two. Both device's application programs are unaware of this use of aliases, and the LonTalk protocol automatically ensures that all their network variables behave correctly. When Device One's application program updates its output network variable, messages are sent on both the primary and the alias network variables in two separate transactions. Device Two, in turn, receives two network variable updates, one for each input network variable.

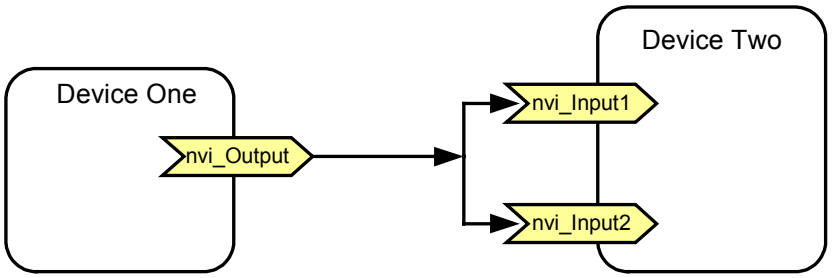


Figure 6.5 Connections Using Network Variable Aliases

For message tag connections, the LNS Object Server enforces these rules:

- Message tags can only be connected to message tags.
- Each device in a connection can have no more than one message tag in the connection.
- In a device's application, there is always one message tag called `msg_in`. The device sends all messages it receives to this tag. The device manufacturer can declare multiple other message tags on each device (usually up to 14). Additionally, you can create dynamic message tags on some devices. You can use these declared and dynamic message tags to send bound messages to other devices. If the hub of a connection is a `msg_in` message tag, then the connection's targets must all be declared or dynamic message tags.
- If the hub is a declared message tag (not the msg_in tag), the targets can be either declared or dynamic message tags, or msg_in tags.
- A declared or dynamic message tag can be in no more than one connection because it's directly tied to a single address table entry in the device.

- The msg_in tag always acts as an input only.
- If there is only one declared message tag in a connection, it acts as an output only.

If there is more than one declared message tag in a connection, all of those declared message tags act as bi-directional ports.

Adding Connections

To create a connection, you must first select a hub network variable or message tag, a set of targets, and optionally, a connection description. You can then use LNS to connect the hub and the targets together, and form a connection. To do so, follow these steps:

1. Select the network variable or message tag that will serve as the connection hub, and fetch it from the applicable collection. To do so, you need to access an `Interface` object from the device that contains the hub network variable or message tag for the connection. For optimum performance, Echelon recommends that you perform the following steps within a transaction.

   ```
   MySystem.StartTransaction()
   Set MyAppDevInterface = MyAppDevice.Interface
   Set MyNvCollection = MyAppDevInterface.NetworkVariables
   Set MyHubNv = MyNvCollection.Item("nvoAlarm")
   ```

2. Optionally, modify the connection parameters that the connection will use by writing to the `ConnectDescTemplate` property of the hub network variable or message tag. For more information on this, and for a description of the default properties that are applied to each connection, see *Connection Descriptions* on page 145.

   ```
   Set MyTemplateLibrary = System.TemplateLibrary
   Set MyConnectTemplates = MyTemplateLibrary.ConnectDescTemplates
   Set MyConnDes = MyConnectTemplates.Add("SvcUnackd")
   MyConnDesc.PropertyOptions = lcaConnPropsServiceType
   MyConnDesc.ServiceType = lcaSvcUnackd
   Set MyHubNv.ConnectionDescTemplate = MyConnDesc
   ```

3. Obtain the target network variables, as described in step 1.

   ```
   Set AnotherInterface = AnotherDevice.Interface
   Set AnotherNvCollection = AnotherInterface.NetworkVariables
   Set MyTargetNv = AnotherNvCollection.Item("nviAlarm")
   ```

4. Call `AddTarget()` on the hub network variable. Supply the `TargetNv` selected in step 3 as the `targetObject` element.

   ```
   MyHubNv.AddTarget(MyTargetNv)
   ```

5. Repeat steps 3 and 4 until the target list is complete, or the number of targets reaches the maximum (25). If you need to add more than 25 connection members, you must complete this procedure to create the connection, and then add additional members, as described in the next section, *Modifying Connections*.

6. Call `Connect()` on the hub network variable to instantiate the connection, and commit the transaction to the LNS database.

   ```
   MyHubNv.Connect()
   MySystem.CommitTransaction()
   ```

Modifying Connections

This section describes how to add members to a connection, remove members from a connection, and delete a connection. To add more members to a connection, follow these steps:

1. Obtain the hub `NetworkVariable` or `MessageTag` for the connection you want to modify.
2. Start a transaction, and then invoke the `AddTarget()` method for each the new targets you want to add. It is not necessary to re-specify existing targets. You must limit the number of targets you add to 25 at a time.
3. When all the targets have been added, invoke the `Connect()` method on the hub network variable or message tag. Then, call `CommitTransaction()` to commit the transaction to the database.
4. If you need to add more than 25 targets to a connection, you can do so by repeating steps 2 and 3. When doing so, you should perform all the steps within the same transaction.

To remove one or more targets connected to a hub, or to remove an entire connection, follow these steps:

1. Obtain the hub `NetworkVariable` or `MessageTag` for the connection you want to modify.
2. Start a transaction, and specify the target network variables or message tags to be removed from the connection using the hub's `AddTarget()` method. You must limit the number of targets you remove at one time to 25.
3. Call the `Disconnect()` method on the connection hub. Then, call `CommitTransaction()` to commit the transaction to the database.

   If no targets are specified in step 1 before the `Disconnect()` method is called, the LNS Object Server will disconnect all targets from the hub, thus deleting the connection. Note that this does not remove the hub network variable or message tag from all connections it is involved in. It only removes the connections for which the network variable or message tag was the connection hub.
4. If you need to remove more than 25 targets from a connection, you can do so by repeating steps 2 and 3. When doing so, you should perform all the steps within the same transaction.
5. To remove a specified network variable or message tag from all connections it is involved in, you will also need to examine the network variable's `NVHubs` property, or the message tag's `MTHubs` property. A network variable's `NVHubs` property contains a collection of the hub network variables for each connection in which it is a target. A message tag's `MTHubs` property contains a collection of the hub message tags for each connection in which it is a target. You can call the `Disconnect()` method on the connection hubs accessed through these collections to remove a network variable or message tag from the connections it is involved in.

Mirrored Connections

LNS allows network variables to participate in multiple connections. As a result, it is possible to create arbitrarily complex network variable connections on a LONWORKS network (subject to the constraints of the LonTalk protocol). A consequence of the superposition of connections is that a network variable may find itself in a "mirrored" connection segment. This situation occurs when one network variable (A) is the hub of a connection containing a target network variable (B), and B is the hub of a connection containing A. Thus, connection segment AB is mirrored by segment BA.

When removing connections, it is important to recognize that the network variables involved in mirrored connections will remain bound until both connections are removed. Mirrored connections may also appear as a side-effect of network recovery.

Listing Connections and Connection Members

LNS provides several connection-related collections you can use to identify what connections a network variable or message tag is involved in:

- Each `NetworkVariable` object contains an `NVHubs` property. This property returns a `NetworkVariables` collection containing the hub network variables of each connection that the network variable is part of. If the network variable is the hub for any connections, it will also appear in the collection.
- Each `NetworkVariable` object also contains an `NVTargets` property. If the network variable is a connection hub, this property returns a `NetworkVariables` collection containing the connection's target network variables. If the network variable is not a connection hub, this collection will be empty. Note that network variables may be involved in connections as both hubs and targets. Thus, to list all network variables to which a network variable is connected, you must iterate through its `NVHubs` and `NVTargets` collections.
- Similarly, each `MessageTag` object contains an `MTHubs` property. This property returns a `MessageTags` collection containing the hub message tags of each connection that the message tag is part of. If the message tag is the hub for any connections, it will also appear in the collection.
- Each `MessageTag` object also contains an `MTTargets` property. If the message tag is a connection hub, this property returns a `MessageTags` collection containing the connection's target message tags. If the message tag is not a connection hub, this collection will be empty.
- Each `AppDevice` object contains an `NVHubs` property and an `MTHubs` property. These properties contain collections of network variable and message tag connection hubs the device contains. This may be useful if you are removing a device from the system, and want to make sure its removal will not affect any connections.
- Each `Network` object has a `Connections` object, which contains collections of all the message tag or network variable connection hubs in the network. You can access these collections through the `Connections` object's `NetworkVariables` or `MessageTags` properties. These collections contain only those `NetworkVariable` or `MessageTag` objects on the network that are currently acting as connection hubs. Like other

collection objects, they may be searched by name or by index. The use of these collections is discouraged for large systems, as it may take a long time to access these collections.

Using the OnNodeConnChange Event

You may want your application to be notified when devices are added to or removed from connections on your system. You can accomplish this with the `OnNodeConnChange` event. To use the `OnNodeConnChange` event, call the `BeginNodeConnChangeEvent()` method on the `System` object.

The `OnNodeConnChange` event will then be fired each time a connection on your system is created or modified, or each time a connection description template being used by a connection on your system is modified.

If you want, you can filter incoming `OnNodeConnChange` events for a specific device using the `ObjectHandle` parameter or a specific type of connection change (i.e. target added, target removed, or connection description template changed) using the `ObjectChangeType` parameter.

Note the `System` object's `MgmtMode` property and the network attachment status determine whether or not connection changes will be propagated to devices on the network. Use the `OnChangeEvent` event or monitor the value of an `AppDevice` object's `CommissionStatus` property to determine when a connection has been committed to the network.

For more information on the `OnNodeConnChange` event, see the *LNS Object Server Reference* help file.

Connection Descriptions

To make it easy to specify the behavior of a connection, the LNS Object Server provides connection descriptions in the form of `ConnectDescTemplate` objects. Each network variable and message tag has a `ConnectDescTemplate` property that returns a `ConnectDescTemplate` object. When a network variable or message tag acts as a connection hub, its current `ConnectDescTemplate` is applied to that connection.

Each connection description template defines basic connection attributes, such as the LonTalk messaging service and authentication setting used by the connection. The default connection description applied to all network variables and message tags has the following settings:

Service type	For network variables, use whatever service the output network variables specify. For message tags, the LNS Object Server assumes acknowledged service.
Priority	For network variables, use priority if the transmitting network variable specifies priority. Do not use for message tags.
Authentication	For network variables, use authentication if a receiving network variable has authentication enabled. Do not use for message tags.
Retry count	Calculated based on topology and service type.

Repeat count	Calculated based on topology and service type.
Repeat timer	Calculated based on topology and service type.
Receive timer	Calculated based on topology and service type.
Transaction timer	Calculated based on topology and service type.
Broadcast options	No broadcast addressing.
Alias options	Use network variable aliases to resolve selector conflicts.

See the *LNS Object Server Reference* help file for more detailed information on these properties. For instructions on how to modify these attributes, and the considerations you should make when doing so, see *Using Custom Connection Description Templates* on page 148.

Chapter 7 - Network Management: Optimizing Connection Resources

This chapter describes advanced topics you may need to consider when managing the connections on your network. This includes considerations you should make when customizing your own connection description template, and guidelines to follow to maximize your network's connection resources.

Using Custom Connection Description Templates

As described in Chapter 6, the LNS Object Server provides connection descriptions in the form of `ConnectDescTemplate` objects that determine the behavior of a connection. Each network variable and message tag has a `ConnectDescTemplate` property containing its `ConnectDescTemplate` object. When a network variable or message tag acts as a connection hub, its `ConnectDescTemplate` is applied to that connection.

To create a new `ConnectDescTemplate` object and apply it to a connection, follow these steps:

1. Invoke the `Add()` method on the `ConnectDescTemplates` collection retrieved from the system's `TemplateLibrary`.

   ```
   Set MyTemplateLibrary = MySystem.TemplateLibrary
   Set MyConnTemplates = MyTemplateLibrary.ConnectDescTemplates
   Set MyNewTemplate = MyConnTemplates.Add("myNewTemplate")
   ```

2. You can customize the template created in step 1 by writing to its properties. Note that you must designate the properties you are modifying as active properties by writing to the `PropertyOptions` property. When building a connection, the LNS Object Server only uses the attributes of the connection description that are designated as active. See the *LNS Object Server Reference* help file for more information on the `PropertyOptions` property.

   The following code sample changes the new `ConnectDescTemplate` object's `RepeatTimer` and `ServiceType` properties, and then sets those properties as active:

   ```
   MyNewTemplate.RepeatTimer = 7
   MyNewTemplate.ServiceType = lcaSvcAckd
   MyNewTemplate.PropertyOptions = lcaConnPropsServiceType OR _
         lcaConnPropsRepeatTimer
   ```

   For descriptions of the various properties of the `ConnectDescTemplate` object, and guidelines to follow when writing to those properties, see the next section, *Setting ConnectDescTemplate Properties*.

3. Once you have modified the `ConnectDescTemplate` object to suit your needs, assign it to the `ConnectDescTemplate` property of a network variable or message tag you plan to use as a connection hub.

   ```
   Set ConnectionHub.ConnectDescTemplate = MyNewTemplate
   ```

4. Create a connection using the selected network variable or message tag as the connection hub. The connection will use the new template. For more information on creating connections, see *Adding Connections* on page 142.

You can also modify the attributes of a connection description that is already being used by a connection hub, and apply those changes to the connection. To do so, follow these steps:

1. Access the `ConnectDescTemplate` object of the hub network variable or message tag for the connection you want to modify

   ```
   Set MyTemplate = ConnectionHub.ConnectDescTemplate
   ```

2. Set the properties of the `ConnectDescTemplate` object, as described in step 2 of the procedure in the previous section. Note that the aliasing options used for network variables in a connection are determined when the network variable is added to a connection. As a result, changing the `AliasOptions` property of a `ConnectDescTemplate` object will not affect the aliasing options applied to network variables that have already been added to connections using that template. It will only affect the aliasing options applied to network variables that are subsequently added to those connections. For more information on using network variable aliases, see the next section, *Optimizing Connection Resources*.

3. Re-assign the `ConnectDescTemplate` object to the hub network variable or message tag. This will cause the attributes of the connection using that `ConnectDescTemplate` to be updated. In previous versions of LNS, it was necessary to re-invoke the `Connect()` method. Note that this operation may take a considerable amount of time to complete, depending on the number of targets in the connection and the network topology.

   ```
   Set ConnectionHub.ConnectDescTemplate = MyTemplate
   ```

Setting ConnectDescTemplate Properties

Table 7.1 lists several of the properties of the `ConnectDescTemplate` object, and the special considerations you need to make when writing to those properties.

Table 7.1 Connection Description Template Properties

Property	Description
`AliasOptions`	Use this property to determine whether or not LNS will use network variable aliases to resolve selector conflicts on a given device. The next section of this document, *Optimizing Connection Resources*, includes a discussion of when and how you should use network variable aliases.

Property	Description
`BroadcastOptions`	This property determines whether LNS will use group or broadcast addressing. You should note that when using group addressing, there are 256 distinct groups per network, and the group addresses consume address table entries in both the receiving and transmitting devices involved in the connection. Most application devices are limited to 15 address table entries, and Network Service Devices that use standard network interfaces are limited to 15 address table entries for use with group addressing. Network Service Devices that use high performance network interfaces support up to 256 address table entries. Broadcast addressing cannot be used with the acknowledged or request messaging services. In addition, domain broadcasting cannot be used with the unacknowledged/repeat messaging service. When LNS uses broadcast addressing, it uses subnet broadcast if all addressed devices are in the same subnet. Otherwise, it uses domain broadcast addressing. You can set the service type for a connection by writing to the `ServiceType` property. Broadcasting with network variables applies only to outputs. The next section of this document, *Optimizing Connection Resources*, includes a discussion of when and how you should use broadcast addressing.

`ReceiveTimer`
`RepeatTimer`
`TransmitTimer`
	The default values for these properties are determined based on the network topology. Echelon recommends that you do not change these properties from their default values. If the default values for these properties are not suitable for your application and you need to change them, Echelon recommends that you use the `Delay` property of each `Channel` object the connection is using to ensure that each message is sent at the correct interval. The `Delay` property allows your application to specify the number of milliseconds expected to send a message and receive an acknowledgment on the specified channel, so that automatic timer calculations made by LNS can be affected accordingly.
	
	These properties accept a range of encoded values from 0 to 15. You can also write the value 254 to the property at any time to restore it to the default. See the *LNS Object Server Reference* help file for more information on these properties, and for the actual values that correspond to each encoded value.

`RepeatCount` `RetryCount`	The default values for these properties are determined based on the network topology. They each accept a range of 0-15.

Property	Description
`ServiceType`	This property allows you choose from the following messaging services: Acknowledged Unacknowledged Repeat Unacknowledged If your application will be sending messages to large numbers of devices at once, one of the unacknowledged messaging services may be desirable, as the acknowledgement messages may generate a significant amount of network traffic. You should only use the request/response messaging service if the devices involved in your connections are designed to send response messages. Certain service types may be desirable over others, depending on the addressing mode and network variable alias usage of your connection. For more information on this, see the next section, *Optimizing Connection Resources*. This property does not apply to message tag connections, as the service type is part of the explicit message and is defined by the application sending the message. Note that the request/response service type is no longer supported by LNS, because network variable updates do not use this service.
`UseAuthenticationFlag`	This property indicates whether or not the connection will use authentication. This property does not apply to message tag connections, as the authentication flag is part of the explicit message and is defined by the application sending the message.
`UsePriorityFlag`	This property indicates whether or not the connection will use priority slots when sending messages. You can write to the `Priority` property of the `AppDevice` objects containing the network variables in your connection to establish which priority slot they will use. This property does not apply to message tag connections, as the priority flag is part of the explicit message and is defined by the application sending the message.

Optimizing Connection Resources

Chapter 6 of this document describes how to create connections with LNS. This procedure is fairly straightforward. However, to optimize the resources available to your system when creating connections, there are many factors you should consider.

Network Design Time

When designing a network, you should seek to achieve the most reliable solution, with the most economic use of network and device resources. In this discussion, which deals specifically with optimizing connection usage, network resources can be thought of as network variable selectors, group identifiers, and channel bandwidth. Address table entries and alias table space are the critical device resources.

When a connection is made, LNS first ensures that it complies with the LONWORKS connection rules:

1. Standard network variables within a connection must have the same type.
2. All network variables within a connection (including user-defined network variables) must have the same length.
3. At least one network variable in a connection must be an input network variable and at least one of them must be an output network variable.

These three rules are fundamental to LONWORKS networks, and all network variable connections must comply with these rules. There are, however, some additional constraints that apply to connections in to LONWORKS networks. These constraints are:

4. Network variables within a connection must share one, and only one, network variable selector, which must be unambiguous.
5. Multiple input and output network variables on a given device cannot share a network variable selector. The same applies to non-polled network variables on nodes earlier than firmware version 6, and to all polled output network variables.
6. Each network variable has one and only one network variable selector. In addition, typically each network variable may have up to one address table entry associated with it.

However, these final three constraints can be worked around with a planned approach, as described in the following sections. Once LNS has approved the compliance of a desired connection with these rules and constraints, it will implement the connection.

You can make sure your connections comply with the rules listed above by writing to the properties of the `ConnectDescTemplate` assigned to the connection. Two key properties are the `AliasOptions` and the `BroadcastOptions` properties. The following sections describe how you can set these properties to maximize your network resources, while complying with the rules and constraints introduced in this section.

Alias Options

The `AliasOptions` property determines how LNS will use network variable aliases in a given connection. This property can be set to the default `lcaAliasForSelectorConflicts` value, or to the `lcaAliasForUnicasts` value.

When set to the default `lcaAliasForSelectorConflicts` value, LNS will allocate one or more input or output network variable aliases to overcome the selector conflicts that are described in rules 4 to 6 above in an attempt to maintain all multicast connections.

The `lcaAliasForUnicasts` option will allow LNS to split a single multicast connection into multiple unicast connections using one or more aliases to the primary output

network variable. This can be used to avoid joining or creating groups, as shown in figures 7.1 and 7.2 later in this section.

For monitor and control applications, the benefit of this approach is that the monitoring tool does not have to join the group. The existing group of nodes 1,2,3 will remain as it is, and node 1 will also propagate network variable updates to the monitoring tool using a unicast connection with Subnet/Node ID addressing. Each time the output network variable is updated, two messages will be sent: one to the group connection, and one using subnet node addressing. The benefit is that address table space on the monitoring node is preserved (becoming a member of a group requires an address table entry to accommodate the group membership information).

The `lcaAliasForUnicasts` option can also be used to avoid multicast connections. The building automation example described later in this chapter introduces a problem that can be avoided by splitting a multicast connection into multiple unicast connections.

Note that there are only 256 distinct groups that can be used for multicast addressing per network, and that by using unicast addressing you can conserve group usage. Group addresses also consume address table entries in both the receiving and transmitting devices involved in the connection, and most application devices are limited to 15 address table entries. Network Service Devices that use standard network interfaces are limited to 15 address table entries for use with group addressing. However, unicast connections may consume more address table entries on transmitting devices.

Typically, multiple unicast messages create extra network traffic compared to a single multicast connection, as all network variable updates will require the transmission of multiple messages onto the network. In contrast, a network variable update in a group connection would only require one message to be sent that would be received by all members of the group.

Note that you can use the `AppDevice` object's `AliasCapacity` to determine the total number of aliases supported by a device, and you can use the `AliasUseCount` property to determine the number of aliases on the device that are already being used by existing connections.

Broadcast Options

The `BroadcastOptions` property determines when a connection should use broadcast addressing. This can be set to any of the following values: `lcaBroadcastNever`, `lcaBroadcastGroup`, and `lcaBroadcastAlways`,

The `lcaBroadcastGroup` value requires some explanation. The `lcaBroadcastGroup` value allows LNS to automatically use broadcast addressing when a multicast connection is required, and no group identifiers are available. This requires that the connection use a messaging service type acceptable for use with broadcast addressing, such as the unacknowledged messaging service or the unacknowledged/repeat messaging service.

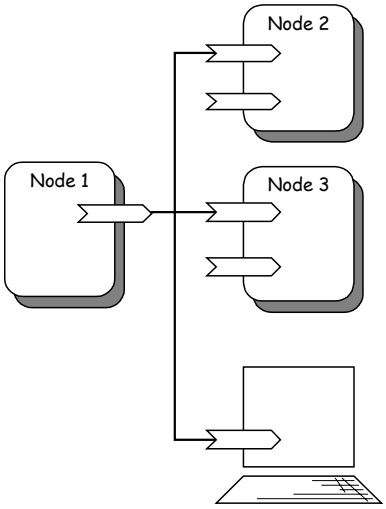


Figure7.1 Monitoring as member of a group

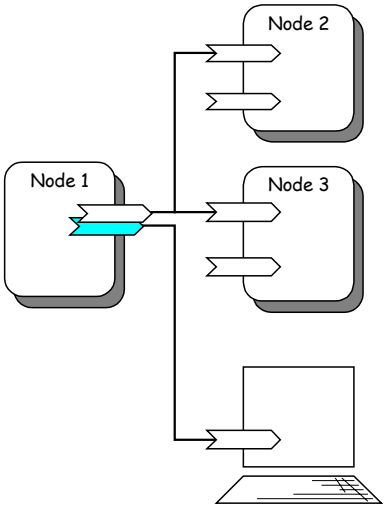


Figure 7.2 Monitoring via network variable alias

Using the AliasOptions and BroadcastOptions Properties

Table 7.2 summarizes the effect of each possible combination of the values of the `BroadcastOptions` and `AliasOptions` properties.

Table 7.2 AliasOptions/BroadcastOptions Combinations

`BroadcastOptions` Values	**`AliasOptions` Values**	
	`lcaAliasForSelectorConflicts`	`lcaAliasForUnicasts`
`lcaBroadcastNever`	This is the default combination applied to all LNS connections. This combination allows for subnet/node addressing and multicast group addressing, but it does not allow the use of broadcast addressing.	This combination splits a single multicast connection into multiple unicast connections using subnet/node addressing, and allocates aliases to the output network variable. If no aliases are available, LNS will use group addressing instead.

BroadcastOptions Values	**AliasOptions Values**	
	lcaAliasForSelectorConflicts	lcaAliasForUnicasts
lcaBroadcastGroup	If the connection is using the acknowledged messaging service, this combination allows for subnet/node addressing and group addressed multicast connections. The unacknowledged/repeat messaging service also allows subnet broadcast addressing to be used, and the unacknowledged messaging service further allows the use of domain broadcast addressing.	This combination splits a single multicast connection into multiple unicast connections using subnet/node addressing, and allocates aliases to the output network variable. If no aliases are available, LNS will use group addressing instead. If no aliases and no group IDs are available and the service type is unacknowledged, LNS will use subnet or domain wide broadcast addressing. If no aliases and no group IDs are available and the service type is unacknowledged/repeat, LNS will use subnet broadcasting if possible.
lcaBroadcastAlways	This combination is recommended if multicast connections must be used, and group addressing needs to be avoided.	This combination splits single multicast connections into multiple unicast connections, but uses broadcast addressing for each of the unicast connections. This has the advantage of potentially re-using address table space on the sending node, at the expense of network bandwidth.

Example Connection Scenario: Building Controls

Figure 7.3 depicts a connection created to manage a lighting system in an office building. The example includes 9 lamps in the ceiling, identified with letters A through J. There are also 4 occupancy sensor devices identified with letters R through U, each connected to the four surrounding lamps. Thus, occupancy sensors R and S necessarily are both connected to lanterns B and E, sensors R and T share lamps D, E, and so on. This common scenario presents an interesting problem.

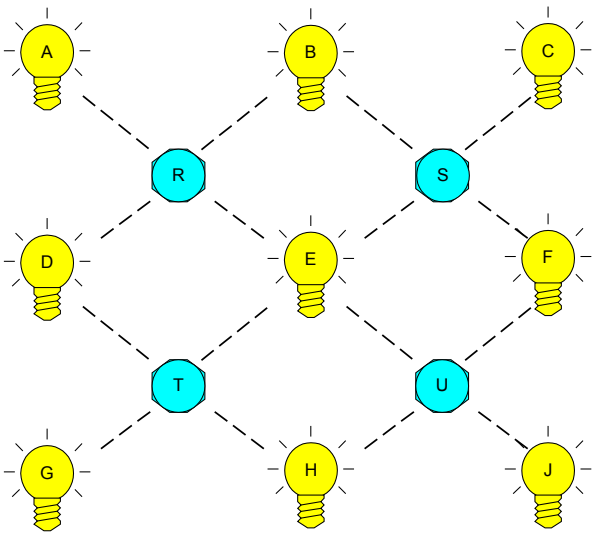


Figure 7.3 Ceiling Lighting With Occupancy Sensors And Connections

Assuming each occupancy detector has only one relevant output network variable, a multicast connection will be used to connect sensor R to lamps A, B, D and E. By default, LNS will use a group to accomplish this. A group ID will be allocated, and one address table entry will be used on each of the five participating devices.

When connecting the sensor S to lights B, C, E and F, another group will be created. This requires another address table space on each of these five devices, and another group identifier.

Each light (apart from the ones in the outermost row and column) will have to have four address table entries to accomplish these connections. On a large floor with more than four occupancy sensors, this will quickly exhaust available group IDs, and eventually the integrator might fail to add new sensor/lamps segments due to a lack of available group identifiers.

Broadcast addressing provides a way to avoid the use of group identifiers in this situation. Most of the lights will reside in the same subnet as the occupancy sensors, and the combination of subnet broadcast addressing and the unacknowledged/repeat messaging service may also seem like an attractive option. However, in order to comply with rule 6 of the connection rules introduced in this chapter, you would have to allocate the same selector Z to both intersecting sensors (e.g. sensors R and S). This is required so that the input network variables on lights B and E do not violate rule 6.

This has an unpleasant side-effect: each network variable update from, say, sensor R will not only effect the lights A, B, D, and E, but also lights C and F. The effect is known as a *network variable leakage,* in this case caused by a phenomenon known as *intersecting broadcasts*. LNS can detect this problem beforehand, and will refuse to connect the second sensor (and any further sensor) due to the detection of a leak in this scenario.

A solution to the problem is to use aliases for unicast connections, instead of using multicast connections. Since each unicast connection can have its own unique selector, the leakage problem will disappear, and the group identifier will remain available for other connections.

This is accomplished at the expense of alias table space and address table space on the occupancy sensor devices, and at the expense of network bandwidth. The different

unicast connections will be processed as separate transactions, causing more network traffic than a single multicast update would have produced.

Solving Problems With Your Connection Scenarios

When planning connections for large systems, or for systems accommodating challenging scenarios, the system integrator might still face difficulties when managing network variable connections. This includes problems such as shortages of group identifiers, aliases, and address table space. The following sections discuss some of these difficulties and their possible remedies.

Shortage of Groups

A single domain has up to 256 distinct groups. However, each group identifier can be used with multiple connections, as long as the connections remain unambiguous. This technique, known as *group overloading*, allows for more than 256 groups to be used based on the 256 distinct group identifiers. Your application can query the number of group identifiers currently allocated in the network by reading the properties of the `NetworkResources` object, which can be obtained from the `System` object's `NetworkResources` property. Consult the *LNS Object Server Reference* help file for more information on this.

To conserve group usage, your application can do the following:

- Replace group connections using the acknowledged messaging service with multiple unicast connections by setting the `AliasOptions` property to `lcaAliasForUnicasts`. This requires sufficient address table and alias table space on the transmitting device. You should note that splitting a multicast connection into multiple unicast connections extends the total transaction time and the total packet count, compared to a single multicast.

- You could combine group connections using the unacknowledged/repeat messaging service into a single connection that uses the unacknowledged/repeat messaging service and subnet broadcast addressing. However, all destination devices in the connection must belong to the same subnet. The source device does not have to belong to the same subnet. This configuration allows up to a 127 devices, the subnet maximum, to participate in the multicast connection. You can also assign a subset of the nodes on a given channel to a specific subnet to facilitate subnet broadcast addressing.

- Polling fan-in connections always require a group connection. Such connections should be avoided. You should use polling strategies only after careful consideration.

- If group addressing is required (for example, for routing purposes), the existing connections can be reconsidered. You might be able to identify an existing group (e.g. an acknowledged multicast connection) that you could change to use the unacknowledged/repeat messaging service with subnet broadcast addressing, or individual unicast connections as described above, thus freeing group identifiers.

Shortage of Address Table Space

Addressing types with broad scopes generally consume fewer address table entries than addressing modes that target only a single destination device. For example, if a device

propagates all outgoing packets using domain broadcast addressing, the device only needs one address table entry per target domain since LNS only supports a single domain, plus one for turn-around connections directed to itself.

Therefore, a good way to conserve address table space is to avoid using group connections, and instead build connections that use subnet broadcast addressing and the unacknowledged/repeat messaging service. This strategy saves group identifiers, and also creates an address table entry that is likely to be re-usable by other connections originating from the same device.

However, the address table does not only accommodate the destination address. The second set of data being kept in each address table entry is the set of transport properties like the repeat count, repeat timer and the transmit timer. In summary, address table shortages can be avoided or overcome by the following means:

- Use broadcast addressing whenever possible to obtain re-usable address table entries.
- Avoid group connections (for the group membership information itself is kept in the address table on each device that is a member of said group).
- Keep variations on transport properties to a minimum.

When groups must be avoided by splitting a group-addressed multicast connection into multiple unicast connections as explained in the previous section, it requires alias table space, and may also quickly consume a large amount of address table space on the sender device. This is because each unicast connection may require its own address table entry.

One way to conserve address table entries is to combine the `lcaBroadcastAlways` and `lcaAliasForUnicasts` values, as described in Table 7.2. This combination will help avoid the effect of address table consumption resulting from the use of multiple unicast connections instead of a single (group-addressed) multicast connection. In this case, LNS will split the multicast connection into multiple unicast connections, but it will also use broadcast addressing for each of these unicast messages. Assuming at least two targets reside on the same subnet, this will result in re-use of address table entries.

Shortage of Aliases

Since aliases are defined at device development time, their number is finite and fixed. At integration time, the tools available are those that re-gain aliases, and those that avoid alias consumption.

Consider the case of a new, connection C_2, and assume C_2 requires an alias on a particular device. Assume the device supports aliases, but all available ones are exhausted by previously defined connections. These connections might contain a connection C_1 that can be disconnected and reestablished, using different connection policy preferences, resulting in C_1^*. This re-designed connection must connect the same set of senders and recipients, but may not require an alias on the device in question.

Summary of Resource Shortage Recommendations

Resource limitations cannot easily be overcome at integration time, but most limitations can be overcome with some trade-off. If a network is characterized by three critical resources (e.g. group IDs, address table entries and alias table space), shortages in one of

these pools can typically be overcome by allowing other types of resources to be consumed.

For example, group IDs can be saved by using network variable aliases, and vice versa. Preferring aliases over group addressing, for example, can also impact the third resource: such a change might not only change the total number of address table entries required, but can also redistribute these address tables entries differently among the devices that participate in the connection.

Predictive Strategies

The previous section described ways to deal with resource shortages as they occur. However, it is best to avoid such a shortage and the related repair work by planning ahead. System integrators should establish an understanding of the characteristics of the current connection scenario and of probable extensions to that scenario, and create connections based on such analysis.

In most cases, the system integrator should know what other connections are to be created in the future. This should include detailed knowledge about a particular number of future connections, as well as an understanding of the characteristics of the complete connection scenario.

For example, consider an office building. The building might consist of any number of floors with several rooms on each floor. Most inner-room networks will be very similar (a light, a switch, a thermostat, etc), and most connections will operate within the local room. On the other hand, a small number of connections will affect many devices in many rooms, such as the intruder or smoke alarming systems, the sun position sensor, etc.

Such a network might be best divided into a few building-wide segments, and a number of very similar, if not identical, subsystems separated from the backbone and building-wide segments by router devices. Thus, network traffic local to a room can be concealed within that room. Also, because subnets cannot span routers, each room (or small number of rooms) will have a local subnet, which allows inner-room connections to use repeated messaging service and subnet broadcast addressing. This will preserve group IDs for use with the building-wide segments.

The strategy detailed in the following flow chart avoids the use of group connections, instead using subnet broadcast addressing for multicasts whenever possible. This strategy is a generalization of the building control example.

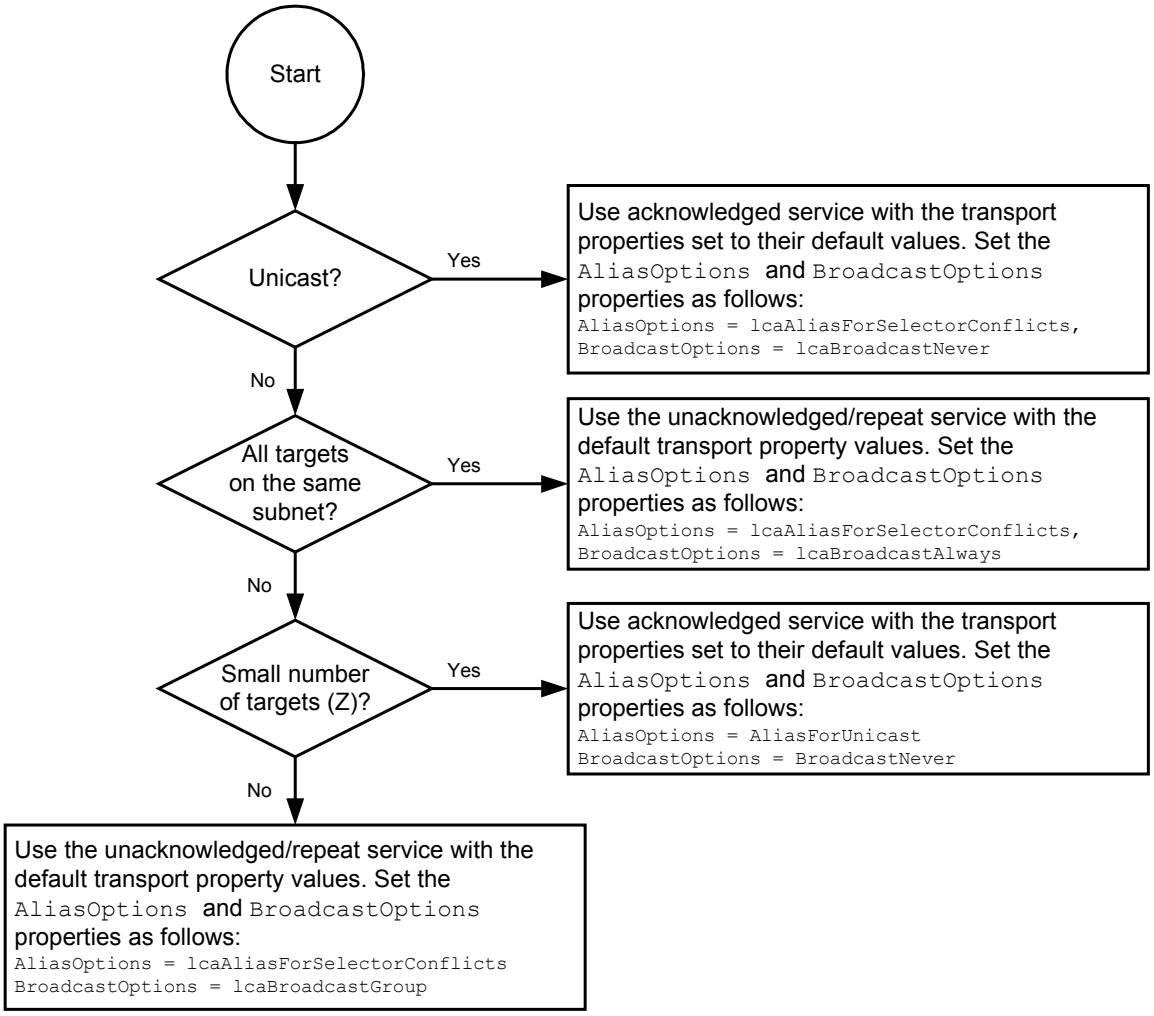


Figure 7.4 Automated Predictive Connection Strategy

The decision-aid presented in this flowchart is deliberately vague in the last decision. If a multicast connection targets a small number of devices on multiple subnets, it might be best to break up the group-addressed multicast connection into multiple unicast connections using aliases, thereby preserving groups. No guidance is given for the cut-off figure Z. So, exactly how many aliases should be used by a single connection?

The answer to this question requires knowledge of the available aliases on the source device (this defines the maximum for Z), and the number of similar connections sourced from the same device. Generally speaking, it is advisable to keep Z fairly low, perhaps less than 5, to limit the impact on the total transaction time introduced with the use of aliases.

You application can read the `AppDevice` object's `AliasCapacity` and `AliasUseCount` properties to determine the used and available alias table entries on a given device.

Conclusion

Your LNS application should manage LONWORKS connections transparently, creating reliable connections with minimum system overhead. However, awareness of what connections will be added to a network in the future and of the network resources those connections will consume is essential when creating a network with a large number of connections.

Chapter 8 - Network Management: Advanced Topics

This chapter describes advanced network management topics such as how to manage a Network Service Device, how to manage a network with multiple channels, how to create custom device interface components, and how to use LNS to change a network variable's type.

Managing Network Service Devices

This section describes special tasks you may need to perform when managing the Network Service Devices on your network. This includes the following:

- *Upgrading a Network Service Device*
- *Moving a Network Service Device*

Upgrading a Network Service Device

In some cases, it may necessary to upgrade a Network Service Device when you change its network interface. Generally, LNS will perform this upgrade automatically, as soon as the system is opened.

However, you can prevent LNS from automatically upgrading the Network Service Device your client application is using by setting the `Flags` property to `lcaFlagsManualNsdUpgrade`. By default, this flag is not set. When the flag is set, you will need to manually perform the upgrade by calling the `Upgrade()` method on the `AppDevice` object that represents your client's Network Service Device. There are several factors you will need to consider when doing so. For more information, see *Network Interfaces and Network Service Devices* on page 273.

Moving a Network Service Device

Whenever a device is moved from one channel to another, it affects the configuration of that device, and possibly the configuration of the devices that are connected to it. If a Network Service Device that is being used by a Local client application is physically moved from one channel to another, the LNS application must tell the LNS Object Server that the Network Service Device is being moved. This is done using the `PreMove()` and `PostMove()` methods. Moving the Network Service Device will typically assign the Network Service Device a new address corresponding to a subnet on the new channel, recalculate transaction timers and update connections accordingly.

The procedure to follow when moving a Network Service Device is the same as if you were moving any other application device. However, unlike moving a normal application device, LNS will not be able to verify that the Network Service Device is on the proper channel after the move. To move a Network Service Device, perform the following steps. This procedure can be performed by a Local client application, a Lightweight client application, or a Full client application. However, you cannot perform this procedure with a Full client application if it is using the Network Service Device being moved.

1. Invoke the `StartTransaction()` and `BeginSession()` methods to start a session.

   ```
   MySystem.StartTransaction()
   MySystem.BeginSession()
   ```

2. Obtain the `AppDevice` object that represents the Network Service Device to be moved. Use the `MyVni` property to do so.

   ```
   Set NSDAppDevice = MyNetwork.MyVNI
   ```

3. Call `PreMove()` on the `AppDevice`. When you call `PreMove()`, you must specify the destination channel as the `newChannelObject` element. You can optionally specify the destination subnet as the `newSubnetObject` element. Echelon recommends that you leave the `newSubnetObject` element empty, as LNS will then allocate the most suitable subnet for the destination location:

   ```
   NSDAppDevice.PreMove(channelObject, NOTHING)
   ```

   NOTE: You can use the `PreMove()` method to move a device from one subnet to another, without switching channels. In this case, specify the device's current channel as the `newChannelObject` element, and the new subnet as the `newSubnetObject` element.

4. Physically move the Network Service Device.

5. End the current session, and start a new one. Then, call `PostMove()` on the `AppDevice` to complete the operation.

   ```
   MySystem.EndSession()
   MySystem.BeginSession()
   NSDAppDevice.PostMove()
   MySystem.EndSession()
   MySystem.CommitTransaction()
   ```

LNS applications that are designed for mobile use, such as a diagnostics tool for a service technician, may find it difficult to follow the procedure described above. Typically, the application will be used in location A, terminated, and started up in a different location B later (the next service case). Since location B may not be known at the time when location A is left, the tool may not be able to call `PreMove()` before physically moving the device.

If the LNS application is a Local or Lightweight client application whose Network Service Device has been moved, the calls to the `PreMove()` and `PostMove()` methods can be performed after the network interface has been physically moved. If you are performing this procedure with a remote Full client application, it is not necessary to call the `PreMove()` or `PostMove()` methods in this case, as the move is performed automatically when the system is opened. However, you must open the network via the `Networks` collection (and not the `RemoteNetworks` collection). See the next section for more details on this.

Remote Full Clients

There are several other factors to consider if a remote Full client application has been using the Network Service Device you are moving. When a remote Full client application opens a database using the `Networks` collection (as opposed to the `RemoteNetworks` collection) and no other application currently using that Network Service Device has the database open, the Network Service Device connects with the LNS Object Server. As part of this connection process, the LNS Object Server determines which channel the Network Service Device is using. If the Network Service Device had been created previously on a different channel, which would be the case if it has been moved, the LNS Object Server will move it to a new channel and assign it a new address, updating connections and timers appropriately. Thus, the LNS Object Server handles the move automatically for you in this case.

You should note that when a Full client application opens a system using the `Networks` collection (as opposed to the `RemoteNetworks` collection), the LNS Object Server automatically determines the channel that the application's Network Service Device is attached to. However, the LNS Object Server cannot determine the correct channel if the channel is one of several channels connected by routers that are configured as repeaters or permanent bridges (i.e. the `Class` property is set to `lcaRepeater`, `lcaPermanentRepeater` or `lcaPermanentBridge`). This is one of the reasons that Echelon encourages the use of configured routers. However, in this scenario, you can set the `System` object's `RemoteChannel` property before opening the `System` to specify the channel that the Network Service Device is attached to.

It is also possible to move a Network Service Device operating as a remote Full client using the `PreMove()` and `PostMove()` methods, as described in the *Moving a Network Service Device* section. However, these methods interrupt the communication between the LNS Server and remote Full client applications attached to the LNS Server. Therefore, the use of these methods on a Network Service Device that is being used by remote Full client applications is discouraged. However, you can move a Network Service Device that is being used by a remote Full client application from one PC to another, as described in the next section.

Using the PreReplace Method

Under normal circumstances, when a remote Full client application re-opens a network, any network variables, connections and monitor sets created previously on the network by the Full client will still be available, as long as the Network Service Device configuration for the Full client still exists in the LNS network database. To ensure that the Network Service Device is never deleted from the LNS database, it must be configured as a permanent device on the network. You can do so by setting the `LcaNsdType` property of the `NetworkServiceDevice` object to `lcaNsdTypePermanent`. However, under some circumstances, the correlation between the Full client and the configuration of the Network Service Device may be lost. In these cases, you can use the `PreReplace()` method to re-associate the client with the correct Network Service Device.

You will also need to use the `PreReplace()` method if you open a network remotely from a new PC, and want that client to use a Network Service Device configuration that was previously associated with a remote client running on another PC (effectively moving the remote application and Network Service Device configuration from one PC to another). An exception to this is if the original remote client used a standard network interface, and you move the network interface to the new PC. In this case, LNS will automatically associate the Network Service Device in the database with the client based on the standard network interface's Neuron ID.

You will also need to follow the procedure described below to re-attach a Network Service Device to a network if the network has been removed from the `RemoteNetworks` collection for the PC, and you are using a high performance (Layer 2/VNI) network interface, or if you install a new network interface on the PC.

To re-associate a client with the correct network service device and re-attach the client to the network, follow these steps:

1. Open the system, and get the Network Service Device to be attached to the network from the `NetworkServiceDevices` collection.

   ```
   Set MySystems = MyNetwork.Systems
   Set MySystem = MySystems.Item(1)
   MySystem.Open()
   Set MyNSDCollection = MySystem.NetworkServiceDevices
   Set MyNSD = MyNSDCollection.Item("MyNSD")
   ```

2. Call `PreReplace()` on the network to be attached to, with the selected Network Service Device as the `sourceNSD` element.

   ```
   MyNetwork.PreReplace(MyNSD)
   ```

3. Close the system, and release all references to the system.

   ```
   MySystem.Close()
   ```

4. Close the network, and release all references to the network.

   ```
   MyNetwork.Close()
   ```

5. Call `Replace()` on the network.

   ```
   MyNetwork.Replace()
   ```

6. Re-open the network.

   ```
   MyNetwork.Open()
   ```

Using Shared Media

A network is said to contain *shared media* if it shares channels with other independently managed networks. For example, power line and RF networks often use shared media, since all networks plugged into the power line (or transmitting over the radio) share the same media. A network is said to contain *private media* if only one network communicates using the channels. Media such as twisted pair is more easily isolated and tends to be used as private media; however, you can still create a design where multiple, independent networks share a twisted pair channel.

When determining whether your installation is shared or private, you should consider whether you intend to share the media with another network, and whether the end-user may add another network to the media in the future. For example, if the media is power line, it is likely to be shared, now or in the future, even if you are installing only one LNS network.

If you are installing a twisted-pair based stand-alone alarm system, then it will probably always be private. However, networks intended for private use often use a shared high-performance backbone, for example when connecting multiple buildings on a site. Subject to the technology used for this shared channel, each network sharing this channel can use private media, or become a shared media network. Generally, it is desirable to use private media, due to more efficient network use and simplified installation and maintenance. For shared backbones using a LONWORKS/IP channel, network design should allow for one virtual LONWORKS/IP channel exclusive to each of the participating

networks, so that each network can maintain its private media status while sharing a physical TCP/IP connection.

When creating a system that uses private media, you should set the system's `InstallOptions` property to `lcaPrivateMedia`. When creating a system on a shared media system, set the `InstallOptions` property to `lcaSharedMedia`. Since there will be multiple networks sharing the media, you should also specify a unique domain ID for your system. When the `lcaSharedMedia` option is selected and no domain ID is specified, the LNS Object Server will select the 6-byte Neuron ID of its network interface as the domain ID. This approach is recommended since it ensures that systems on shared media will have unique domain IDs. Also, keep in mind the following:

- When using the engineered mode installation scenario, the Network Service Device used to create the database may be different than the one used to commission the system. You should either use the Neuron ID of the network interface on the Network Service Device that commissioned the system as the system's domain ID, or set a unique 6-byte ID using your own algorithm.
- In an installed system, if you replace a network interface (e.g. as part of a repair operation), the new network interface will have a different Neuron ID than the old one. As a result, you should not rely on the current network interface's Neuron ID to indicate the domain ID. Instead, you should use the `DomainId` property of the `System` object to determine the system's domain ID.
- If your LNS application is used to install multiple networks, you should not derive the domain ID from the network interface, as this would result in domain ID duplication. Instead, you should use the Neuron ID of any device that is to be installed into the respective network. Alternatively, your application could choose a 6-byte random number to produce a domain ID with a high probability of uniqueness.

In a system that uses shared media, the LNS Object Server disables background discovery and device pinging. These activities are undesirable in a shared media system for the following reasons:

- An LNS Object Server can discover devices that belong to a neighboring system. Thus, discovery is not a reliable means of identifying devices to install.
- If multiple LNS Object Servers are aware of the same device, their communication with that device may create race conditions that result in communication failures with the device, or improper configuration of the device's network image.
- The LNS Object Server background discovery and pinging tasks generate periodic packets on the network. The traffic increases with the number of LNS Object Servers issuing them. Since shared media also tend to be low-speed media such as power line, this "extra" traffic can result in an excessive load on the network.

These guidelines should be followed when installing devices on a system that uses shared media:

- *Do not use find and wink installation*. Since you cannot be assured that devices you discover "belong" to your system, you should not use this method to identify devices. This means that you should disable

background discovery by setting `DiscoveryInterval` property to 0. This is done automatically when the `InstallOptions` property is set to `lcaSharedMedia`.

- *Use the confirmed service pin algorithm for device installation*. When using shared media, there is always a small chance that when a service pin message is received, it is from a device in a neighboring system. The confirmed service pin algorithm is designed to ensure that devices being installed belong to the correct system. For more information on this, see *Neuron ID Assignment* on page 115.

The above process should greatly minimize the already low probability of installing the wrong device on a network. Another way to avoid this is to ask the user to explicitly enter the Neuron ID of each device you install on the network. In all other cases, Echelon recommends that you perform system-level verification after all connections have been made when using shared media. If you discover that an incorrect device has been added to your system, use the `Replace()` method to associate the `AppDevice` object with the correct physical device, as described in *Replacing Devices* on page 132.

The `InstallOptions` property must be set before opening the system for the first time. Setting this property at any other time has no effect. If you initially choose private media, and decide you want to use shared media later, you can accomplish this by following these steps:

1. Set the system's `DomainId` property to match the Neuron ID of the Network Service Device that the LNS Server PC is using.
2. Disable automatic discovery and pinging by setting the `DiscoveryInterval` and `PingIntervals` properties to 0.
3. Disable automatic service pin registration by setting the system's `RegisterServicePin` property to `False`.

NOTE: A LONWORKS/IP channel can be considered private media if it is used by a single network. For each network in a control system, create a separate LonWorks/IP channel to avoid difficulties common to shared media control networks.

Managing Networks with Multiple Channels

LONWORKS networks may contain multiple channels, interconnected by routers or logical repeaters. This section describes the considerations you need to make when managing a network with multiple channels. There are several reasons that you might want to use multiple channels on a network:

- The chosen networking medium for your network has physical layer constraints, such as wire length or device count, and you want to create a network that is larger than these constraints would allow if the network only contained a single channel. Each channel is individually subject to the physical-layer constraints, but the use of multiple channels allows you to extend the distance or device count. Segments of a TP/FT-10 channel connected by a physical layer repeater are considered a single channel.
- The traffic load on your network might approach the channel capacity for your networking medium. Use of routers can partition the local traffic

from traffic that has to span multiple channels, which allows more effective use of available bandwidth.

- You want to provide for fault tolerance in the event of a physical-layer fault, such as a short or open circuit. Only devices on the faulty channel would be affected by such faults. Devices on other channels are isolated from the fault by the routers or logical repeaters.

Overview of Router Types and Operation

A router connects two channels. Physically, a router contains two transceivers (one for each channel it is connected to), and two processing modules. The transceivers and processing modules receive packets from each channel, and decide whether or not to forward the packet to the other channel. For more details on the functionality of LONWORKS routers, see the LONWORKS *Router User's Guide*.

A router counts as a device on each of its channels for the purposes of addressing and physical layer design constraints, but not with respect to the LNS Device Credit limits enforced by the LNS Object Server. For more information on LNS Device Credits, see Chapter 13, *LNS Licensing*.

A network consisting of LONWORKS routers and channels must be completely connected, i.e. it must provide a path for all devices to exchange messages with one another. With the exception of redundant routers (see the *Explicitly Controlling Subnet Allocation* section below), a network should have no logical loops created by routers. Physical loops are allowed on a single channel when free topology transceiver channels are in use. Physical loops between multiple channels may occur due to leakage on separate radio frequency or power line channels. However, since no logical loops are allowed, two messages having the same source and destination will always travel through the same set of channels. Figure 8.1 displays an example topology with four channels that are interconnected by three routers.

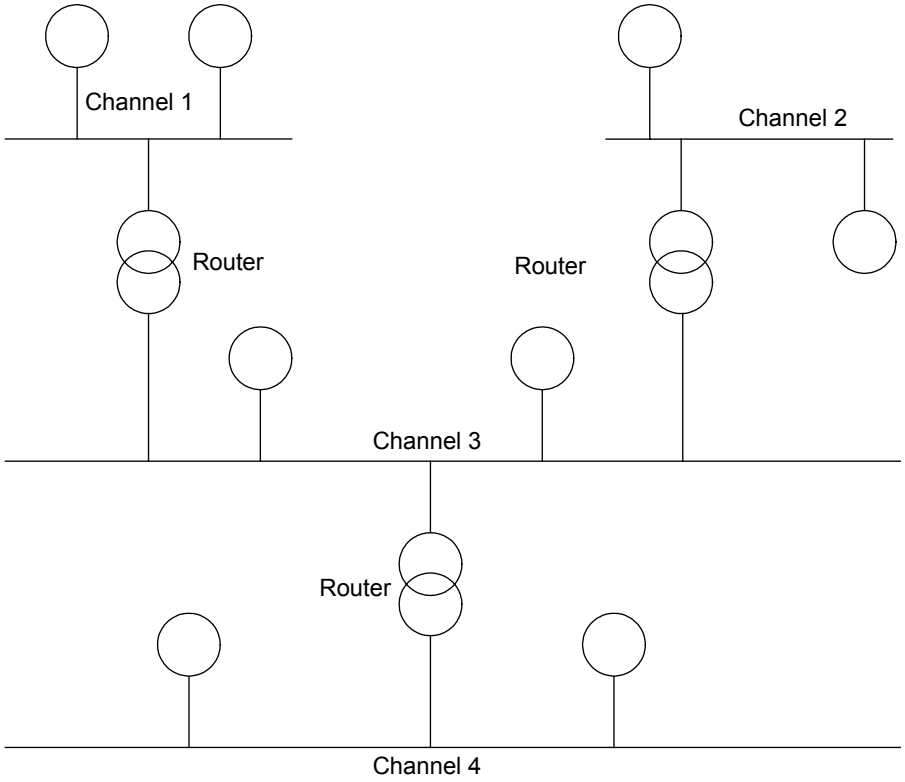


Figure 8.1 A Network with Multiple Channels

A LONWORKS router can be configured as one of four router types:

- A *repeater* forwards all valid packets received on one channel to the other channel, without regard for address. Repeaters extend the physical reach of a channel, while preventing corrupted packets from causing problems. You should not confuse a router configured as a repeater with a physical repeater. Repeaters cannot be used in topologies with physical loops, as a given message could be repeated endlessly in this case. Physical repeaters, in contrast, act as simple signal boosters and noise filters to extend the physical reach of a channel, without providing any message routing, validation or filtering.
- A *bridge* forwards a valid packet received on one channel to the other channel if the packet is sent on a domain that the bridge belongs to. In a single domain network, a bridge functions in the same manner as a repeater. Bridges cannot be used in topologies with physical loops, as a given message could be repeated endlessly in this case.
- A *learning router* forwards packets based on internal routing tables. These tables contain one entry for each subnet in the application domain. Learning routers have their routing tables in volatile memory so that after they are reset, the router forwards packets addressed to all subnets in the application domain. Whenever a learning router receives a packet from one of its channels, it uses the source subnet ID to learn the network topology. It then sets the corresponding routing table entries to indicate that the subnet in question can be discovered in the direction from which the packet was received.

As of LNS Turbo Edition, LNS does not support defining or modifying a router as a learning router. If an application defines a router as a learning router, or changes a routers class to `lcaLearningRouter`, the LNS Object Server will automatically change the class to `lcaConfiguredRouter`. See the next paragraph for a description of configured routers, and the advantages they provide.

- A *configured router* forwards packets based on internal routing tables. Configured routers have their routing tables in non-volatile memory. The LNS Object Server configures and manages the routing tables based on its knowledge of the network topology. In addition, a configured router can be configured to selectively forward group-addressed messages when it is known that all members of the group are on one side or the other of the router. LNS manages the subnet and group forwarding tables in configured routers automatically. **This is the most efficient router type to use, as it also allows the LNS Object Server to automatically determine the channel each device is attached to, supports physical loops, and reduces unnecessary network traffic**. Configured routers also support the use of redundant routers (see the *Explicitly Controlling Subnet Allocation* section below), which provide for redundant message paths.

In addition, you can define repeaters and bridges as permanent routers, meaning that their types cannot be changed after they are installed. When a repeater or bridge router is defined as permanent, the LNS Object Server knows that it will never be changed into a configured router later, and it will allow the same subnet to exist on both sides of the router. This is permitted by the router, since bridges and repeaters do not perform forwarding based on subnet address.

If a router is defined as a non-permanent bridge or repeater, LNS will enforce the rule that no subnet can appear on both sides of the router or bridge, even though the router itself does not enforce this rule. This allows the LNS application to change the router class later, without creating subnet conflicts. See the *Explicitly Controlling Subnet Allocation* section below for more information on the role of subnets and the rules regarding their allocation.

With the exception of learning routers, you can use LNS to configure a LONWORKS router as any of the router types described in this section. The most efficient router type to use is the configured router, as this allows the LNS Object Server to automatically determine the channel to which each device is attached, supports physical loops, and reduces unnecessary network traffic.

Explicitly Controlling Channel Allocation

Unless you are using the ad hoc or automatic installation scenario, you should explicitly define the channels in your system, and specify those channels as devices and routers are added to the system. When using the ad hoc installation scenario, you can omit the channel definitions and allow LNS to create channels for you. When using an automated installation scenario based on device discovery mechanism, LNS automatically creates each `AppDevice` with the correct channel assignment. When explicitly defining a channel, you must specify the channel's media type, by transceiver ID. The LNS Object Server includes the `ConstTransceiverId` constant, which includes enumerations for each of the possible transceiver types you can use for this purpose.

When LNS discovers a router, it will automatically create a channel for the far side, and will choose an existing channel for its near side. The far side channel may in fact be a duplicate of a user-defined channel. Assuming that your LNS application defined a logical definition of this router and specified the user defined channels, this discrepancy will be resolved when the router is commissioned, and LNS will delete the channel that it had created for the router's far side.

Explicitly Controlling Subnet Allocation

Subnets are the second component of the three-component LonTalk domain/subnet/device addressing hierarchy. The subnet address is the level at which routers decide whether or not to forward a packet, so the same subnet cannot appear on both sides of a configured or learning router. By default, the LNS Object Server creates a new subnet when a subnet is full, or when a non-permanent router is added to the system. A subnet is considered full as soon as there are 127 devices assigned to it.

Your application can explicitly create subnets by calling the `Add()` method on the system's `Subnets` collection, and allocate a device to a specific subnet when the device is added to the system with the `AppDevices` collection's `Add()` method. Reasons for controlling subnet assignment include directing the use of subnet broadcasting, and the anticipation of topology changes.

If your application explicitly controls subnet assignment in this manner, certain constraints must be observed. Figure 8.2 helps illustrate these concepts:

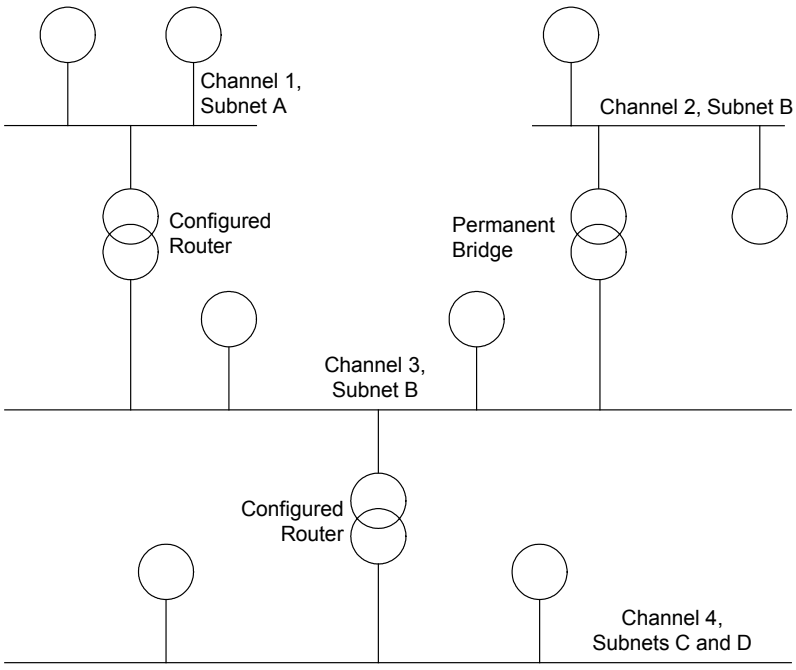


Figure 8.2 Subnets, Channels and Routers

1. Each channel can be assigned to more than one subnet. In figure 8.2, Channel 4 is assigned to Subnets C and D.

2. Each subnet can only be assigned to a single channel, unless it is assigned to multiple channels that are connected by permanent bridges or permanent repeaters. In Figure 8.2, Subnet B spans Channels 2 and 3,

which are connected by a permanent bridge.

3. The final rule allows for redundant routing when using configured routers. Redundant router topologies provide fault tolerance by providing more than one routing path from one channel to another. They are also required when all devices on a given channel may not be able to hear one another (referred to as an *ear shot* problem), e.g. on a radio frequency channel.

   An example of a redundant routing topology is shown in Figure 8.3 below. Both routers can forward packets originating on Subnet A and destined for Subnet B. Any open circuit in either Channel 1 or Channel 2 still leaves the network logically connected.

   The redundant routing topology provides a backup means of communication through redundant paths in the form of routers and the resulting redundant packets. For every packet sent from a device on Channel 1, Subnet A to a device on Channel 2, Subnet B, two packets will be delivered (one packet from each of the configured routers). This will occur for every point at which a backup router exists. Furthermore, acknowledgments are multiplied.

   For example, consider a network consisting of 3 channels that employs redundant routers between each channel. Sending a single acknowledged message that spans all three channels will result in 2 acknowledged messages on the second channel, and 4 on the third channel. Each of these four messages will be acknowledged, resulting in 8 acknowledgements on the second channel and 16 on the first channel. This situation worsens when authenticated messaging is used, since an authenticated transaction consists of 4 separate messages (the initial message, a challenge, a reply, and an acknowledgment). In the example given above, a single acknowledged authenticated message would result in 4 acknowledged messages on the third channel, 16 challenges on the first channel, 64 replies on the third channel and 256 acknowledgments on the first channel. Echelon recommends that you limit the number of redundant routers created by the user, and warn the end user of the effects of setting up redundant routers.

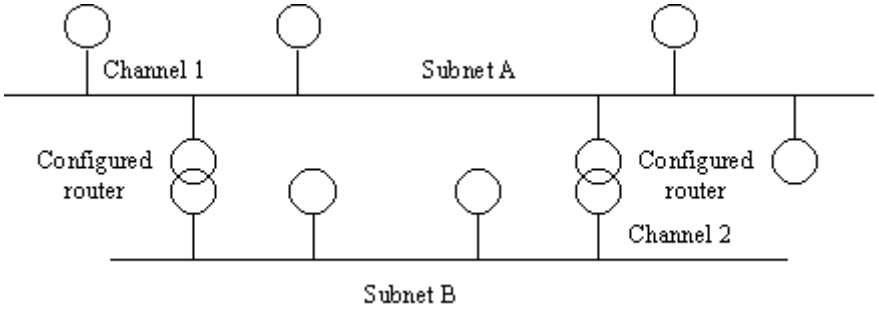


Figure 8.3 Redundant Routing Topology

Installing and Configuring Routers

When installing and configuring routers, your application can treat routers in much the same way as application devices, using the steps described in Chapters 5 and 6 of this

document. Each Subsystem has a Routers collection containing the Router objects that represent your network's routers, just as each Subsystem has an AppDevice collection containing the AppDevice objects for the network.

Routers have a similar set of properties (e.g. State, Location) and methods (e.g. Add, Commission) as application devices. And as with application devices, they can be identified using the automatic discovery, service pin, or manual entry methods described in the *Neuron ID Assignment* section on page 115. When using manual entry, enter the near router side's Neuron ID (if connected to the network). The near router side is the side that is closest to the Network Service Device. When commissioning a router while not connected to the network (or connected but unable to communicate with the near side of the router), the Neuron IDs of both router sides must be entered.

Note that routers do not support the confirmed service pin installation protocol as it is described in Chapter 6, as they do not support the Wink() method. Instead, you should use the Reset() method for confirmation of router devices when using the confirmed service pin installation of s router. This will not work on routers that do not provide a visual or audible reset notification (such as a Reset LED).

When you call the Routers collection's Add() method to define a router in the LNS database, you can define its router class by specifying the routerType element. The most efficient router type to use is the configured router (lcaConfiguredRouter). This allows the LNS Object Server to automate most topology management tasks. The LNS Object Server also automatically allocates and assigns new subnets as needed. As was discussed in the previous section, the application can manually control subnet allocation if required. Note that after you created a Router object, you can change its class by writing to its Class property.

When using redundant routers, the LNS Object Server cannot automatically determine the channel to which the far side is attached. In this case, your application must explicitly specify the channel within the Add() invocation.

Installation Order

The order in which routers and devices are added to the LNS database is important. The LNS Object Server must be able to communicate with a device or router in order to configure it. If the LNS Object Server is always attached to the same single channel, then the routers on that channel must be installed and configured before application devices and routers on any channel adjacent to the router can be installed. In this way, the installed network fabric grows outward from the LNS Object Server's network attachment point. Full client applications must also be able to communicate with the LNS Object Server to operate. A continuous physical and logical path must therefore exist between each Full client application's network interface, and the network interface on the LNS Server PC. If multiple network segments are to be installed before the complete network topology is established, then the network interface on the LNS Server PC may be moved from segment to segment as it is installed.

Installing Devices With Multiple Channels

As devices (and routers) are installed on a network with multiple channels, it is necessary to determine the channel on which each device resides. When using the engineered mode installation scenario, you must always specify the device's channel when you add the device to the LNS database. This allows the LNS Object Server to

allocate a logical address, compute routing tables, and calculate connection timers for the device. When the device is commissioned, the LNS Object Server will verify, as best it can, that the device is indeed attached to the expected channel. If it is not, an exception will be thrown.

When devices are being added to the LNS database as they are discovered (automatic installation), or as they are physically installed (ad hoc installation), LNS will automatically determine the device's channel using the channel isolation process described in the next section. If the system exclusively uses configured routers, LNS can always determine the device's channel.

If LNS cannot uniquely determine a device's channel, it will use one of the possible channels as the channel when commissioning the device. If the system uses repeaters or permanent bridges, your application should always specify the `Channel` to use when you create each `AppDevice` object, since the channel reported by the LNS Object Server may be incorrect.

Channel Isolation Process

When you add a device to the LNS database, LNS uses the channel isolation process to determine the channel the device should use, or to validate that the channel you have assigned to the device is valid.

All router types other than repeaters perform address translation on messages with a source subnet equal to 0, translating the source subnet to be the subnet of the router side that first receives the message. The channel isolation process takes advantage of router source address translation by examining the translated source subnet, and thus identifies the source channel. In some cases LNS may have to temporarily place the device in the unconfigured state in order to perform channel isolation.

LNS can only isolate down to a logical channel segment when routers configured as repeaters or permanent bridges are used. A logical channel segment is a set of channels connected to each other by routers configured as repeaters or permanent bridges. This is because:

- Repeaters do not perform source translation.
- Subnets may legally span permanent bridges.

In both cases, it is not possible for LNS to determine which router within the logical channel segment received a packet first. Installing a device on the wrong channel, but on the correct channel segment, may have negative consequences. Routing will not be affected, but LNS may calculate the layer 4 timers incorrectly, which could result in unnecessary retries or message failures. As a result, Echelon recommends the use of configured routers, as LNS will always be able to identify the correct channel when configured routers are used. If this is not possible, the installer should make sure to install each device on the correct channel, or use the `PreMove()` and `PostMove()` methods to move any devices to their correct channel.

Note that if you are running a remote Full client application that is on a logical channel segment containing routers of class `lcaRepeater`, `lcaPermanentRepeater` or `lcaPermanentBridge`, you will need to set the `RemoteChannel` property prior to opening the system to specify the channel the client is using.

Resolving Installation Failures

There are a few cases where attempts to register, add, or commission a device on a network with multiple channels may fail or have negative side effects. This usually happens when you attempt to install a device that was previously configured as part of another network without deconfiguring it. You should also consider the following:

- If the device to be installed is configured on a subnet in the system's domain that violates the logical topology (i.e. a subnet on wrong side of configured or learning router), LNS may need to force the device to the unconfigured state to communicate with it. Thus, accessing a configured device may result in the device becoming unconfigured.
- If learning routers are in use and the scenario described in case 1 is attempted, the operation may result in the learning router not routing messages to the conflicting subnet. The learning router must be reset to restore normal operation. This is why it is important to use the `PreMove()` method during device movement when learning routers are in the system as described later in this chapter.
- If a device violates the logical topology (as in case 1) and is authenticated, it will not be possible to install it. This is why it is important to use the `PreMove()` method during authenticated device movement as described later in this chapter.
- If a device has the same address as the network interface on the LNS Server PC, then the LNS Object Server will not be able to communicate with that device, since the target device ignores all messages received from a device with its own subnet/device ID. In this case, the device must be deconfigured. You cannot use LNS to do so, since the LNS Object Server will not be able to communicate with the device. So, you must choose another means to do so, e.g. by using a tool that is configured on a different domain. During the development process, Echelon recommends that you choose a different system domain ID to avoid this condition following the creation of a new network database. Note that the LNS Object Server uses a large node ID (e.g. 127) to help prevent this problem from occurring.

Moving Devices and Routers Between Channels

At some point, you may need to move the devices on your network from one channel to another. To do so, follow this procedure:

1. Call `StartTransaction()` to start a transaction, and then call `BeginSession()` to start a session.

   ```
   MySystem.StartTransaction()
   MySystem.BeginSession()
   ```

   NOTE: If you are only moving a single device, you do not need to use a session, but for optimal performance you should use a transaction to perform these steps, unless the device uses authentication.

2. Obtain the device to be moved from the `AppDevices` collection. Or, obtain the router to be moved from the `Routers` collection. The example code in this section applies to an application device:

   ```
   Set MyAppDevice = MyAppDevices.Item("node12")
   ```

3. Call `PreMove()` on the device selected in step 2. You must specify the new channel for the device with the `newChannelObject` element, and you can optionally specify the new subnet for the device with the `newSubnetObject` element. Note that if you do not specify the new channel for the device, LNS will attempt to automatically determine the channel using the channel isolation process described previously in this chapter.

   ```
   MyAppDevice.PreMove(TargetChannel, NOTHING)
   ```

4. Physically move the device to its new location. Repeat steps 2, 3 and 4 for each device being moved.

5. If any of the devices being moved use authentication, end the current session, commit the transaction, and then start a new transaction and session. Then, call `PostMove() on each device` to complete the operation.

   ```
   If MyAppDevice.AuthenticationEnabled Then
      MySystem.EndSession()
      MySystem.CommitTransaction()
      MySystem.StartTransaction()
      MySystem.BeginSession()
   End If
   MyAppDevice.PostMove()
   MySystem.EndSession()
   MySystem.CommitTransaction()
   ```

If you move a previously configured device to a new network, and the device has network management authentication enabled, LNS will not typically be able to install the device. In addition, if the device has the same network address as your client's Network Service Device, LNS will not be able to communicate with the device.

There are special considerations you need to make when moving a Network Service Device. For more on this, see *Moving a Network Service Device* on page 164. In addition, moving a configured device from one channel to another, or to a new network, may cause communication problems between the device and the network. For more information on this, see the previous section, *Resolving Installation Failures*.

Removing Routers

While managing your network, it may be necessary to remove routers from the network. To do so, invoke the `Remove()` method on the applicable `Routers` collection as follows:

```
Dim MyRouters As LcaRouters
Set MyRouters = MySubsystem.Routers
MyRouters.Remove("Router1")
```

There are several things you should consider before removing a router. LNS allows routers to be removed if the removal causes channels to become disconnected, but only if doing so does not break the path between connections, permanent monitor sets, or disrupt communication between the LNS Server and a remote Full client application.

If you attempt to remove a router and the operation fails because it would break a connection, the `NS#73 lcaErrNsInsufficientRouters` exception will be thrown. If you attempt to remove a router and the operation fails because it would break a permanent monitor set, the `NS#181 lcaErrNsInsufficientRtrsForMnc` exception will be thrown. If you attempt to remove a router and the operation fails because it would

break communication between the LNS Server and a remote Full client, the `NS#182 lcaErrNsInsufficientRtrsForNsi` exception will be thrown.

Using Dynamic Device Interfaces

As described in the *Device Interfaces* section in Chapter 6, each LONWORKS device contains a device interface that represents the device's functionality. The device interface consists of network variables, configuration properties and LonMark Functional Blocks.

In LNS, network variables, configuration properties and LonMark Functional Blocks are represented by `NetworkVariable`, `ConfigProperty`, and `LonMarkObject` objects. The device interface as a whole is represented by the `Interface` object, which can be accessed through the `Interface` property of the `AppDevice` object associated with the device.

In some cases, there may be a need to modify the functionality provided by a device interface. For example, some controller devices are used to control other devices. The number of components required on a controller device's interface is often an attribute of the network configuration (i.e. how many devices it is controlling). Ideally, the resources on these controllers could be allocated dynamically to fit the changing requirements of a given network as devices are added to it. As a result, LNS Turbo Edition features additional support for dynamic interface components, meaning that you can use LNS to add custom interface components to devices that support dynamic interfaces. This section describes how you can use those features.

Accessing a Device Interface

A device interface is represented by an `Interface` object. The `Interface` objects contained by an application device include the device's main interface, as well as custom interfaces that have been added to the device dynamically. You can access the main interface of a device through its `Interface` property. The main interface contains the device interface installed with the device by its manufacturer, as well as all the network variables and LonMark Functional Blocks defined in the custom interfaces that have been added to the device. Main interfaces are static interfaces that cannot be modified directly.

You can access and modify each of the custom interfaces that have been added to a device through the device's `Interfaces` property. Figure 8.4 shows the relationship between the device's main interface and its custom interfaces. Note that the network variables and LonMark Functional Blocks included in the custom interfaces are also included in the device's main interface.

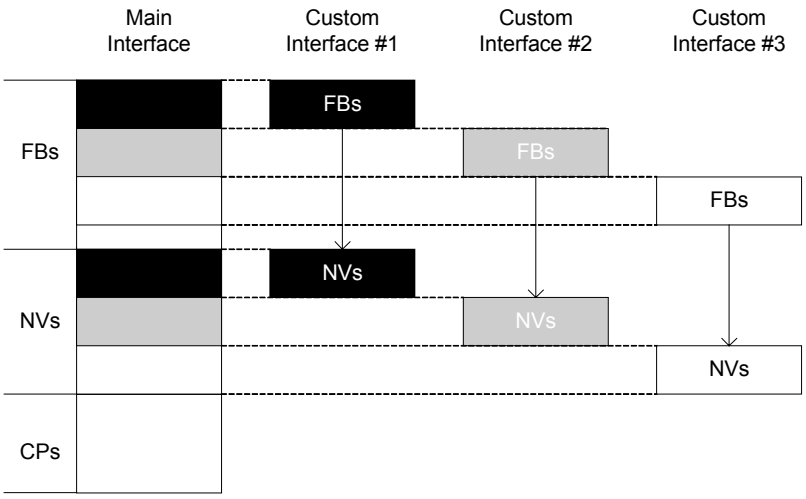


Figure 8.4 Device Interfaces

Table 8.1 describes some of the properties of the `Interface` object. Some of these properties apply to the device's interfaces as a whole. Consult the *LNS Object Server Reference* help file for a complete list, and for more extensive descriptions of each property.

Table 8.1 Interface Object Properties

Property	Description
`ConfigProperties`	This property contains the `Interface` object's `ConfigProperties` collection (i.e. the configuration properties included in the interface that apply to the device as a whole).
`ConfigPropertiesAvailable`	This property indicates whether configuration property definitions are available for the specified `Interface`. If True, then configuration property definitions for the `Interface` have been uploaded from the device or imported from an external interface file. If `False`, the device may not implement any configuration properties at all, or LNS has not yet been able to determine whether the device implements configuration properties.
`DynamicLonMarkObjectCapacity`	This property indicates the number of dynamic `LonMarkObjects` you can add to the custom interfaces on this device, including those you have already added.
`DynamicMessageTags`	This property contains the `Interface` object's collection of dynamic message tags (i.e. the message tags that have been added to the interface).
`LonMarkObjects`	This property contains the `Interface` object's `LonMarkObjects` collection (i.e. the `LonMarkObject` objects included in the interface).

Property	Description
MaxNVSupported	This property specifies the maximum number of network variables (static or dynamic) that the Interface object can contain.
MessageTags	This property contains the Interface object's collection of static MessageTag objects (i.e. the message tags originally included in the interface).
NetworkVariables	This property contains the Interface object's NetworkVariables collection (i.e. the network variables included in the interface).
StaticNVCount	This property indicates the number of static network variables on the Interface object.
SupportDynamicNVsOnStaticLMOs	This property indicates whether static LonMarkObject objects on the device's interfaces support the addition of dynamic network variables.

You can use LNS to add custom interfaces to devices that support dynamic interfaces using the Interfaces property. And depending on what other types of dynamic objects it supports, you can also modify its interfaces by adding or removing objects from its NetworkVariables, LonMarkObjects, and MessageTags collections, as described in the following sections.

Adding a Custom Interface to a Device

You can create custom interfaces on any device that supports dynamic network variables, dynamic message tags, or dynamic LonMark Functional Blocks. To create a custom interface on a device, follow these steps:

1. Access the device's collection of custom interfaces.

   ```
   Dim MyInterfaces as LcaInterfaces
   Set MyInterfaces = MyAppDevice.Interfaces
   ```

2. Call Add() to add a new custom interface to the device. Specify the name of the interface by filling in the interfaceName element. The name of each custom Interface on a device must be unique. If you specify a name that is already being used on the device containing this collection, the LCA:#3 lcaErrDuplicateKey exception will be thrown.

   ```
   Dim MyNewInt as LcaInterface
   Set MyNewInt = MyInterfaces.Add("NewInt", NOTHING)
   ```

   NOTE: You can specify an existing Interface object as the sourceInterfaceObj element. If you do so, the new Interface object will be created with the same NetworkVariables collection as the Interface referenced as the sourceInterfaceObj. If the sourceInterfaceObj is NOTHING, as in this example, an empty Interface object will be created.

3. You can now add dynamic network variables, LonMark Functional Blocks

and message tags to the new `Interface` as you desire, provided that the device supports them. For more information on these tasks, see the following sections.

Adding LonMark Functional Blocks To a Custom Interface

A LonMark Functional Block represents a collection of network variables and configuration properties on a device that perform a related function. For example, a digital input device with four switches could contain one LonMark Functional Block for each switch. For general information on LonMark Functional Blocks, and how you can use them to configure and manage a device, consult the *LonMark Interoperability Guidelines,* which can be downloaded from the web at http://www.lonmark.org/products/guides.htm.

In LNS, LonMark Functional Blocks are represented by `LonMarkObject` objects. Some custom device interfaces support dynamic LonMark Functional Blocks, which means that you can add them to the interface manually. You can determine if an interface supports dynamic LonMark Functional Blocks by reading the interface's `DynamicLonMarkObjectCapacity` property. If the device interface supports dynamic LonMark Functional Blocks, the `DynamicLonMarkObjectCapacity` property will be set to a value greater than 0. Note that you cannot add `LonMarkObjects` to a device's main interface, although the `DynamicLonMarkObjectCapacity` property of that `Interface` may be set to a non-zero value.

To add a dynamic `LonMarkObject` to a device, follow these steps:

1. Access the custom interface that you want to add the `LonMarkObject` to. If necessary, create a new custom interface, as described in *Adding a Custom Interface to a Device* on page 181.

   ```
   Dim Interfaces as LcaInterfaces
   Dim MyNewInt as LcaInterface
   Set MyInterfaces = MyAppDevice.Interfaces
   Set MyNewInt = MyInterfaces.Item("NewInt")
   ```

2. Make sure that the interface supports the addition of dynamic `LonMarkObject` objects by reading the `Interface` object's `DynamicLonMarkObjectCapacity` property. If it does, acquire the `Interface` object's `LonMarkObjects` collection.

   ```
   Dim LonMarkFunctionalBlocks as LcaLonMarkObjects
   Set LonMarkFunctionalBlocks = MyNewInt.LonMarkObjects
   ```

3. Call the `Add()` method on the `LonMarkObjects` collection to create a new `LonMarkObject` object. The `Name` and `ProgammaticName` properties of the new `LonMarkObject` object will be set to match the name you specify as the `fbName` element. You should note that the name assigned to the `LonMarkObject` objects on each device must be unique. If you attempt to use a name that is already used on the device, the operation will fail, and the `LCA#3 lcaErrDuplicateKey` exception will be thrown.

   ```
   Dim MyLMFB as LcaLonMarkObject
   Set MyLMFB = LonMarkFunctionalBlocks.Add("new LMFB",500)
   ```

Configuring LonMark Functional Blocks

You can assign existing dynamic network variables to dynamic LonMark Functional Blocks using the `AssignNetworkVariable()` method, and you can unassign them using the `UnassignNetworkVariable()` method. Alternatively, you can create new network variables that will be automatically assigned to the network interface using the procedure described in the *Creating Dynamic Network Variables* section on page 184.

This may be useful when working with sophisticated generic controller devices. Consider the case of a generic device that controls an entire home. As features such as a new set of lighting scene controllers are added to the home, related LonMark Functional Blocks or network variables can be added to the generic home controller device as needed.

You can also move a dynamic `LonMarkObject` from one custom interface on a device to another using the `MoveToInterface()` method.

Adding Message Tags To a Custom Interface

As of LNS Turbo Edition, you can add dynamic message tags to any device that supports monitor sets, and use those message tags in conjunction with message monitor points to send explicit messages from that device to a group of devices, as with static message tags.

For example, consider the case of a Network Service Device. Network Service Devices do not contain static message tags. However, you can add dynamic message tags to the `AppDevice` object that represents a `NetworkServiceDevice`. Once you have done so, you could connect the message tag to the devices you want to send messages to. Following that, you could create a permanent message monitor point on the Network Service Device that specifies the new dynamic message tag as its monitor target (`targetDevice` element). You could then open the monitor set, and use the message monitor point to send messages to the devices connected to the message tag. The advantage of this approach, as opposed to creating message monitor points for each device, is that a single message can be sent to all devices using group or broadcast addressing, as determined by the connection description used to create the connection. For more details on how to accomplish these tasks, see *Adding Message Monitor Points to a Monitor Set* on page 197.

Not all `Interface` objects support dynamic message tags. If you attempt to add a message tag to a message tag collection on a main interface, or a custom interface on a device that does not support dynamic message tags, the `LCA#119 lcaErrInterfaceNotModifyable` exception will be thrown.

To add a dynamic message tag to a device, follow these steps:

1. Access the custom interface on the device that you want to add the message tag to. If necessary, create a new custom interface, as described in *Adding a Custom Interface to a Device* on page 181.

2. Access the custom interface's `DynamicMessageTags` collection.

   ```
   Dim MyMessageTags as LcaMessageTags
   Set MyMessageTags = MyInterface.DynamicMessageTags
   ```

3. Call the `Add()` method to add a new message tag to the collection.

   ```
   Dim MyNewTag as LcaMessageTag
   ```

```
Set MyNewTag = MyMessageTags.Add("newName")
```

4. For information on using message tags and message monitor points for monitor and control operations, see Chapter 9, *Monitor and Control.*

Creating Dynamic Network Variables

You can add network variables to custom interfaces on devices that support dynamic network variables, to dynamic `LonMarkObject` objects, or to static `LonMarkObject` objects that support dynamic network variable assignment. If you attempt to add a network variable to a static `LonMarkObject` or a device's main interface, then the LCA#119 `lcaErrInterfaceNotModifyable` exception will be thrown, unless the device supports the addition of dynamic network variables to static `LonMarkObject` objects. You can determine whether a `LonMarkObject` is static or dynamic by reading the object's `IsDynamic` property.

You should note that network variables contained within the same custom `Interface` objects must have unique user names (`Name` property). In addition, some devices, such as the *i*.LON 100 Internet Server, require that all network variables within the device have unique programmatic names (`ProgammaticName` property). If you attempt to assign a duplicate user name or programmatic name to a network variable on such a device or interface, the operation will fail, and the `LCA#132 lcaErrUniqueNvNameRequired` exception will be thrown.

To add a network variable to a custom interface or to a LonMarkObject, follow these steps:

1. Access the custom interface you want to add the network variable to. If necessary, create a new custom interface, as described in *Adding a Custom Interface to a Device* on page 181. Or, access the `LonMarkObject` you want to add a network variable to.

   ```
   Dim MyInterfaces as LcaInterfaces
   Dim MyNewInt as LcaInterface
   Set MyInterfaces = MyAppDevice.Interfaces
   Set MyNewInt = MyInterfaces.Item("newInt")
   ```

2. Access the interface or `LonMarkObject` object's `NetworkVariables` collection.

   ```
   Dim myNVs as LcaNetworkVariables
   Set myNVs = MyNewInt.NetworkVariables
   ```

3. Call `Add()` to add a new network variable to the collection. For descriptions of the parameters you need to specify, see the *LNS Object Server Reference* help file.

   ```
   Dim NewNV as LcaNetworkVariable
   Set NewNV = myNVs.Add("nv3",nvType, nvDirection, _
           nvOptions, memberNumber,manufacturerAssigned)
   ```

Note that you can also use the `MoveToInterface()` method to move dynamic network variables from one interface to another. This may be useful if you want to remove a dynamic network variable from a device's main interface. You cannot use the `Remove()` method to remove a network variable from `NetworkVariables` collection on a device's main interface, even if it is a dynamic network variable. However, you can use the

MoveToInterface() method to move a dynamic network variable from the main interface to a custom interface. Once you have done so, you could remove the network variable from the custom interface, and its removal would be propagated to the main interface.

Tracking Custom Interface Changes

You can use the OnNodeIntfChangeEvent event to keep track of when a device's interface is modified. The OnNodeIntfChangeEvent event is fired whenever a device's interface is changed. Depending on the type of interface change, the OnNodeIntfChangeEvent will be generated as the changes are made to the LNS database, or as they are propagated to the physical device. You can register your application for this event by calling the BeginNodeIntfChangeEvent() method on the System object.

For more information on this, see the *LNS Object Server Reference* help file.

Changeable Network Variable Types

As of LNS Turbo Edition, each NetworkVariable object contains a TypeSpec property. The TypeSpec property provides access to a TypeSpec object. LNS uses the TypeSpec object to identify the base type a network variable should use. You can write new values to the properties of the TypeSpec object to change the network variable's type, if the network variable supports changeable types.

To change a network variable's type, follow these steps:

1. Check that the network variable supports changeable types. You can read the NetworkVariable object's ChangeableTypeSupport property to do so. If the ChangeableTypeSupport property is set to lcaNvChangeableTypeSdOnly or lcaNvChangeableTypeSCPT, the network variable supports changeable types.
2. Access the network variable's TypeSpec object through the TypeSpec property.
3. Set the program ID, scope, and name of the new type you want to use by writing to the ProgramId, Scope, and TypeName properties of the TypeSpec object.
4. Optionally, invoke the Lookup() method on the TypeSpec object to make sure that the program ID, scope and name entered in step 3 reference a valid type.
5. Read the IsComplete property to make sure that the TypeSpec object is complete. This step is generally not necessary when changing a network variable's type, unless you are creating a new network variable or changing a network variable's type from a type that was received from another network variable. Consult the *LNS Object Server Reference* help file for more information on the IsComplete property.
6. Pass the modified TypeSpec object back to the TypeSpec property of the network variable. At this point, LNS will use the values entered in step 2 to find the definition of the type in the resource files, and assign values to

the Index, Length, and ObjectType properties of the TypeSpec object. If LNS is unable to find the resource file for the program ID entered in step 2, the LCA#154 lcaErrUnavailableResourceFiles exception will be thrown. If LNS finds the resource file but is unable to find the type name referenced in step 2, the LCA#155 lcaErrNotFoundInResourceFiles exception will be thrown. Be sure that the network variable can support the new type before assigning it. If the length of the new type is too long for the network variable, then the LCA#156 lcaErrTypeLengthTooLong exception will be thrown.

SCPTnvType Configuration Properties

If the ChangeableTypeSupport property is set to lcaNvChangeableTypeSCPT, then the network variable supports changeable types via a SCPTnvType configuration property. This is the recommended LonMark-compliant way to implement changeable type network variables on a LONWORKS device. If this method is supported, LNS will automatically update the value of the SCPTnvType configuration property when the network variable's TypeSpec property is changed. If the single SCPTnvType configuration property is declared for multiple network variables, changing the type of one of those network variables changes the type of all of them.

However, the device can refuse the desired type. For instance, a device could implement a generic PID controller that supports a wide range of numeric data types (float, signed long, etc) for the process value, setpoint, and control value network variables. Changing these network variables to a non-numeric type such as a string (e.g. SNVT_asc_string) may not be meaningful in the context of the application. The device will report this error condition using its NodeObject LonMark functional block, and will not change the network variable type.

This presents a problem when configuring a device in engineered mode. When you change network variable types in the definition phase while the application is not attached to the network, and then create network variable connections based on the desired, new, network variable type, these connections may malfunction if one of the participating devices rejects a type change request once it has been set online. Echelon recommends that integrators only use device-specific plug-in software to change network variable types, as such software has built-in knowledge of the network variable types that a given device supports. In turn, you should design your LNS application so that it does not support changing network variable types to any arbitrary type, at least not without warning the user of these implications.

You must also be aware that changing the type of a single network variable can have a snowball effect. Some devices are designed to implement multiple network variables of changeable type, but with the restriction that some or all of these network variables must have the same type. For instance, the aforementioned generic PID controller could be implemented to support a wide range of numeric network variable types, but could require that setpoint, control and process value network variables always use the same network variable type. Unfortunately, LNS has no method to determine if such a relationship exists. Consequently, changing one network variable may leave the related network variables with the inappropriate type and format selection. However, the device manufacturer can ensure that a set of network variables always have the same type by using a single SCPTnvType for all of them.

Likewise, a configuration property of an inheriting type could apply to a changeable type network variable. Configuration properties of inheriting types derive their type from the

network variable they apply to. For example, configuration properties for default, minimum, or maximum values are often defined as to be of an inheriting type. Such configuration properties will also change their data type if the type of the changeable type network variable changes, and the related data formatting may change.

To control all these issues with minimum error, Echelon recommends that integrators only use device-specific plug-in software to change network variable types, and that you design your LNS application with these implications in mind.

If you design a generic LNS application that may be used to configure or install any arbitrary device type, Echelon recommends that you implement LNS director functionality in your application. An LNS director application can detect and use plug-in software, and can therefore delegate the delicate task of network variable type changes (and similar delicate device configuration tasks) to specialized plug-in software. See Chapter 12 of this document for more information on LNS director applications.

Chapter 9 - Monitor and Control

This chapter describes the LNS features you can use to monitor and control the devices on your network. Monitor and control is the process of reading and writing network variables, and sending explicit messages.

Introduction to Monitor and Control

By definition, device applications only have a local view of the network. They know what data they receive from the network (usually without regard to which devices produce the data), and they know what data they produce (usually without regard to which devices will consume the data).

However, in most control systems there is a need for an application that provides an overall view of the system. For example, in a process control system, the operator interface provides users with supervisory control over the entire process, and displays the current status of the system. Or, in a building control system, there might be multiple control panels from which users can view the current setting of any point in the system, and change the value of any set point in the system. You can use the monitor and control services provided with LNS for these purposes.

Monitoring is the process of fetching and receiving data from devices on the network, while controlling is the process of writing data to those devices. Both involve subordinate tasks such as data formatting, connecting devices, and address change tracking (to ensure that data is not lost due to address changes). In a LONWORKS network, data can be retrieved from application devices using network variable connections, or fetched from devices with regularly scheduled polling messages and explicit polls. Basic formatting is accomplished automatically by processing network variable types according to predefined formatting files. Additional formatting or processing can be performed by the network application.

You can use monitor sets to implement monitor and control operations in your LNS application. A monitor set is a collection of *monitor points*. A *network variable monitor point* represents a single network variable on a device that is being monitored by an LNS application, and you can use the monitor point to read and write that network variable's value. You can use a *message monitor point* to send an explicit message to a device with your LNS application, or to receive messages sent by application devices to your LNS application.

As of LNS Turbo Edition, there are two kinds of monitor sets:

- Permanent monitor sets. Permanent monitor sets are stored persistently in the LNS database, and can be used in multiple client sessions.
- Temporary monitor sets. Temporary monitor sets are not stored persistently in the LNS database, and are only used in a single client session.

Depending on how you need to monitor devices, you may want to use either type of monitor set. If you are writing a device plug-in application, and only want to monitor the values of the network variables on the device while you configure it, a temporary monitor set will suffice, as you will only need to monitor the device for a short period of time.

On the other hand, you might be managing a very large network with hundreds or even thousands of devices, and you need an application to monitor and control those devices persistently. In that case, you should use permanent monitor sets. You could define a group of monitor sets as you install the network, and then use them to monitor and control your devices when your network begins operation.

At some point, you may also want to read or write the value of single network variable without monitoring its value beforehand. You could do this by directly reading or writing

to the value of the network variable, without creating a monitor point to represent the network variable.

This chapter describes the LNS features you can use to perform these monitor and control operations. Table 9.1 describes each section included in this chapter. Echelon recommends that you review the entire chapter before developing your application, so that you are aware of all the monitor and control capabilities LNS provides.

Table 9.1 Monitor and Control

Section	Description
Temporary and Permanent Monitor Sets	This section describes the major differences between temporary and permanent monitor sets.
Creating Monitor Sets	This section describes how to create a temporary or permanent monitor set.
Managing Monitor Sets	This section describes the initial steps you need to take to use a monitor set for monitor and control operations after you have initially created it. This includes the following: 1. Add network variable and message monitor points for the devices you plan to monitor to the set. 2. Define the monitoring options for the set.
Opening and Enabling Monitor Sets	This section describes the steps you need to take to open a monitor set and enable it for monitor and control operations.
Using Network Variable Monitor Points	This section describes several ways you can use network variable monitor points to monitor and control a network.
Using Message Monitor Points	This section describes how you can use message monitor points to monitor and control a network.
Developing Remote Monitor and Control Applications	This section provides guidelines you will need to follow when writing remote applications for monitor and control.
System Management Mode Considerations	This section describes how the system management mode impacts the behavior of some monitor and control operations.
Directly Reading and Writing Network Variables	At some point, you may want to read or write the value of single network variable without monitoring its value beforehand. You can do this by directly reading or writing to the value of the network variable through a `DataPoint` object, without creating a monitor set. This section describes how you can do so.

Section	Description
Using Configuration Properties In a Monitor and Control Application	In some cases, you may want to write an application to control the values of the configuration properties on the devices on your network. This section provides guidelines to follow when doing so.
Data Formatting	This section describes how your application can format the data it obtains through monitor points and data points.

Temporary and Permanent Monitor Sets

As of LNS Turbo Edition, there are two separate types of `MonitorSet` objects: permanent `MonitorSet` objects, which can be used in multiple client sessions, and temporary `MonitorSet` objects, which can only be used in a single client session. This section describes when you should use each type.

Permanent Monitor Sets

Each `Network` object contains a `MyVNI` property, which returns an `AppDevice` object representing your application's Network Service Device. You can use this `AppDevice` to access all the `MonitorSet` objects that are stored in the LNS database for your Network Service Device (i.e. the monitor sets that have been created for applications running on your PC). Echelon recommends that you use the `MyVNI` property to access `MonitorSet` objects when you need to create or modify the configuration of the `MonitorSet` objects. To access `MonitorSet` objects for actual monitor and control operations, you should use the `CurrentMonitorSets` property of the `Network` object.

The `CurrentMonitorSets` property returns a collection of all the `MonitorSet` objects on the network that are currently stored in your Network Service Device. This will be useful if you have created monitor sets while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. Although those monitor sets exist in the LNS database (and can be accessed through the `MyVni` property), they will not be commissioned into the Network Service Device, and cannot be enabled or used for monitor and control operations until the system management mode is set to `lcaMgmtModePropagateConfigUpdates`. When the system management mode is set to `lcaMgmtModePropagateConfigUpdates`, the new monitor sets will be commissioned into your Network Service Device and added to the `CurrentMonitorSets` collection, and the `CurrentMonitorSets` collection will contain the same collection of monitor sets as the `MyVni` collection.

The collection accessed through the `CurrentMonitorSets` property allows access to all the monitor sets you can currently use on a network. The collection accessed through the `MyVni` property allows access to these monitor sets, and those not yet commissioned into your Network Service Device. All the monitor sets obtained through the `CurrentMonitorSets` property are runtime monitor sets, meaning that you can open them and enable them for monitoring operations. However, you cannot change their configuration when you access them through the `CurrentMonitorSets` collection. As noted previously, you should use the collection obtained through the `MyVni` property when you need to change the configuration of your client's local `MonitorSet` objects.

Use permanent monitor sets when you need to create monitor points that will be used often, or in multiple client sessions. If you need monitor points that will only be used once, or in a single client session, you should use temporary monitor sets.

Temporary Monitor Sets

Temporary monitor sets are opened automatically by LNS as they are created, and can only be accessed from the client that created them. They cannot be accessed from the permanent `MonitorSets` collections described in the previous section. When a client releases a temporary monitor set, or when the client session in which a temporary monitor set was created ends, the temporary monitor set and all the monitor points it contains are deleted automatically.

If you need to create a group of monitor points that you can use in multiple client sessions or that you intend to use multiple times, you should use the permanent monitor sets described earlier in this chapter. However, temporary monitor sets take less time and network resources to create. This may be useful if you only need to monitor a device while you are installing it on a network, or if your application will not need to monitor the device on a regular basis.

The properties and methods that can be used on temporary monitor sets and temporary monitor points are generally the same as those that can be used on permanent monitor sets and permanent monitor points. However, there are a few exceptions to this rule.

Temporary `MonitorSet` objects cannot be created or used by Independent client applications. The `Open()` and `Close()` methods have no effect on temporary `MonitorSet` objects, because temporary `MonitorSet` objects are opened as soon as they are created, and closed as soon as the client application releases them, or the session in which they were created ends. For more information on creating monitor sets, see the *Creating Monitor Sets* section later in this chapter.

You should also note that temporary monitor sets are not enabled as they are opened. You must explicitly enable temporary monitor sets and temporary monitor points with your application using the applicable `Enable()` method. For this purpose, the `MsgMonitorPoint` object now includes an `Enable()` method. For more information on enabling monitor sets and monitor points, see *Opening and Enabling Monitor Sets* on page 213.

The `DefaultOptions` properties of `MsgMonitorPoint` and `NvMonitorPoint` objects in temporary monitor sets are not accessible. The values applied to these properties are taken from the temporary monitor set's `MsgOptions` or `NvOptions` properties. For more information on monitor and control options, see the *Managing Monitor Sets* section later in this chapter.

Monitor points in temporary monitor sets do not support the use of connection description templates to define certain monitoring options, as monitor points in permanent monitor sets do. As a result, you must set the `connDesc` element to NULL when you use the `Add()` method to add a message monitor point or network variable monitor point to a temporary monitor set.

There is one other variance you should note when using temporary `MonitorSet` objects. Network variable monitor points in temporary monitor sets cannot be automatically bound to the monitoring node. This means that the `UseBoundUpdates` property of all temporary monitor sets and monitor points must be set to `False`. For more information

on the UseBoundUpdates property, see *The Implicit Bound Network Variable Monitoring Scenario* on page 220.

Creating Monitor Sets

The first step when creating a monitor and control application is to create a monitor set. To create a permanent monitor set, follow these steps:

1. Open the ObjectServer, Network, and System you plan to monitor and control, as described in Chapter 4, *Programming an LNS Application*.
2. Get the AppDevice object contained in the Network object's MyVNI property. This contains a collection of the MonitorSet objects that are stored in the LNS database for your client's Network Service Device. This is the collection you should use when creating and configuring a permanent monitor set.

   ```
   Set VniAppDevice = MyNetwork.MyVNI
   Set MyMonitorSets = VniAppDevice.MonitorSets
   ```
3. Invoke the MonitorSets collection's Add() method. This method takes the name of the new monitor set as an argument.

   ```
   Set newMonitorSet = MyMonitorSets.Add("Monitor Set 1")
   ```

   At this point, the new monitor set is closed. You will need to open and enable it before using it for monitor and control purposes, as described later in this chapter.
4. Set the properties of the new MonitorSet object, including the NvOptions property, the MsgOptions property, and the Tag property.

   You can use the NvOptions property, and the MsgOptions property to set the default monitoring options that will be applied to the monitor set. For more information on monitoring options, see *Setting Monitoring Options* on page 200.

   You can use the Tag property to store any information required to quickly and efficiently identify the monitor set, such as the name of the device you plan to monitor with the set. See the *LNS Object Server Reference* help file for more information on the Tag property, and the other properties of the MonitorSet object.

To create a temporary monitor set, follow these steps:

1. Open the ObjectServer, Network, and System you plan to monitor and control, as described in Chapter 4, *Programming an LNS Application*.
2. Call the CreateTemporaryMonitorSet() method on the Network object. The method will return the new temporary monitor set.

   ```
   Dim MyTempSet as LcaMonitorSet
   Set MyTempSet = MyNetwork.CreateTemporaryMonitorSet()
   ```

   At this point, the new monitor set is open, but it has not been enabled. You will need to enable the monitor set before you use it for monitor and control purposes. This is described later in the chapter.
3. Set the properties of the new MonitorSet object, including the

NvOptions property, the MsgOptions property, and the Tag property.

You can use the NvOptions property, and the MsgOptions property to set the default monitoring options that will be applied to the monitor set. For more information on monitoring options, see *Setting Monitoring Options* on page 200.

You can use the Tag property to store any information required to quickly and efficiently identify the monitor set, such as the name of the device you plan to monitor with the set. See the *LNS Object Server Reference* help file for more information on the Tag property, and the other properties of the MonitorSet object.

After you have created a monitor set, you can add network variable and message monitor points to it. The monitor points will represent the network variables and application messages to be monitored and controlled with the set. See the next section, *Managing Monitor Sets*, for more information on this.

Managing Monitor Sets

After you have created a temporary or permanent monitor set, you should perform the following tasks:

1. Add network variable and message monitor points to the set. For more information on these tasks, see *Adding Network Variable Monitor Points to a Monitor Set* on page 195, and *Adding Message Monitor Points to a Monitor Set* on page 197.
2. Configure the monitoring options for the set. These options determine the behavior of the set while it is enabled. For more information, see *Setting Monitoring Options* on page 200.

Generally, the way you will perform each task is the same for temporary monitor sets and permanent monitor sets. Exceptions to this are noted as each task is described.

Adding Network Variable Monitor Points to a Monitor Set

Each MonitorSet object contains an NvMonitorPoints property. The NvMonitorPoints property returns a collection of the network variable monitor points in the monitor set. You can use network variable monitor points to monitor the devices on your network. To create a network variable monitor point, follow these steps:

1. Open the ObjectServer, Network, and System you plan to monitor and control, as described in Chapter 4, *Programming an LNS Application*.
2. Get the device containing the network variable to be monitored:

```
Set MySubsystems = MySystem.Subsystems
Set MySubsystem = MySubsystems.Item("Floor3.Room7")
Set MyAppDevices = MySubsystem.AppDevices
Set MyAppDevice = MyAppDevices.Item("Thermometer")
```

3. Get the network variable(s) you will be monitoring:

```
Set MyInterface = MyAppDevice.Interface
Set MyNVs = MyInterface.NetworkVariables
Set MyNV = MyNVs.Item(“nvoTemperature”)
Set MyNV2 = MyNVs.Item(“nvoAlarm”)
```

4. Repeat steps 2 and 3 until you have acquired all the network variables you want to monitor, and get the monitor set that will contain the monitor points for these network variables. You should have first created a monitor set for this purpose, as described in the *Creating Monitor Sets* section. Remember that when adding network variable monitor points to a permanent monitor set, or when making any persistent changes to a permanent monitor set's configuration, you should access the monitor set through the `MyVni` property.

```
Set MonVNI = MyNetwork.MyVNI
Set MySets = MonVNI.MonitorSets
Set MySet = MySets.Item("Monitor Set 1")
```

5. Access the monitor set's `NvMonitorPoints` collection, and use the `Add()` method to add network variable monitor points to the monitor set. Each time you call `Add(),` you will pass in the network variable to be monitored by the new point as the `nv_target` element.

   Echelon recommends that you use transactions when creating permanent network variable monitor points, as this considerably reduces the time required to create the monitor points. Transactions should not generally be used when creating temporary monitor points since they are not stored in the LNS database, and so starting a transaction would unnecessarily slow down the process. For more information on using transactions with LNS, see *Using Transactions and Sessions* on page 65.

```
MySystem.StartTransaction()
Dim NvMp1 as LcaNvMonitorPoint
Dim NvMp2 as LcaNvMonitorPoint
Set NvMonitorPoints = MySet.NvMonitorPoints
Set NvMp1 =NvMonitorPoints.Add("nvMP1", MyNV, nothing)
Set NvMp2 =NvMonitorPoints.Add("nvMP2", MyNV2, nothing)
```

6. Set the properties of the new `NvMonitorPoint` object. Two properties you should initially set are the `Tag` property and the `DefaultOptions` properties. You can use the `DefaultOptions` property to set the default monitoring options that will be applied to the new monitor point each time it is enabled. See the *LNS Object Server Reference* help file for more information on the `Tag` property, and for a complete list of the properties of the `NvMonitorPoint` object. For more information on monitoring options, see *Setting Monitoring Options* on page 200.

   For permanent monitor points, you should perform these tasks in the same transaction that the monitor points were created in. When you have finished, commit the transaction to the database.

```
MySystem.CommitTransaction()
```

7. You can now enable the monitor point for monitor and control operations. For more information on this, see *Opening and Enabling Monitor Sets* on page 213.

Adding Message Monitor Points to a Monitor Set

Each `MonitorSet` object contains a `MsgMonitorPoints` property. The `MsgMonitorPoints` property returns a collection of the message monitor points in the monitor set.

There are two ways to use message monitor points. You can create a message monitor point to send messages to (and receive messages from) a single device on the network. Or, you can create a message monitor point to send messages to a group of device simultaneously by connecting those devices to your application's Network Service Device. To do so, you will need to create a dynamic message tag on the Network Service Device. This section describes both procedures.

To create a message monitor point to send explicit messages to (and receive message from) a single device on the network, follow these steps:

1. Open the `ObjectServer`, `Network`, and `System` you plan to monitor and control, as described in Chapter 4, *Programming an LNS Application*.

2. Echelon recommends that you use transactions when creating permanent message monitor points, as this considerably reduces the time required to create the monitor points. For more information on using transactions with LNS, see *Using Transactions and Sessions* on page 65.

   ```
   MySystem.StartTransaction()
   ```

3. Get the device that you plan to send explicit messages to, or that you plan to receive messages from. Note that this cannot be the Network Service Device containing the monitor set you plan to add the message monitor point to.

   ```
   Set MySubsystem = MySubsystems.Item("Floor1.Room3")
   Set MyAppDevices = MySubsystem.AppDevices
   Set MyAppDevice = MyAppDevices.Item("Vent")
   ```

   If you only plan to send explicit messages to the device with the message monitor point, proceed to step 6. If you plan to receive messages from the device, you need to perform steps 4 and 5 to connect a message tag on the device to your application's Network Service Device.

4. Get the `MessageTag` object on the device that will be used to send the updates to the monitor and control application:

   ```
   Set MyInterface = MyAppDevice.Interface
   Set MyMessageTags = MyInterface.MessageTags
   Set MyTag = MyMessageTags.Item("mTag1")
   ```

5. Obtain the `msg_in` message tag on your application's Network Service Device, and connect it to the message tag selected in step 4.

   ```
   Set MonVni = MyNetwork.MyVNI
   Set NsdInterface = MonVni.Interface
   Set NsdMessageTags = NsdInterface.MessageTags
   Set NsdMsgIn = NsdMessageTags.Item("msg_in")

   MyTag.AddTarget (NsdMsgIn)
   MyTag.Connect()
   ```

6. Get the monitor set you will use to monitor the messages. You should have first created a monitor set for this purpose, as described in *Creating Monitor Sets*. Remember that when adding monitor points to a monitor set, you should access the set through the `MyVni` property.

   ```
   Set MyMonSets = MonVNI.MonitorSets
   Set MyMonSet = MyMonSets.Item("Monitor Set 1")
   ```

7. Access the monitor set's `MsgMonitorPoints` collection, and call the `Add()` method to add a message monitor point to the monitor set, passing in the `AppDevice` selected in step 3 as the `targetDevice` element.

   ```
   Dim MyMsgPoints As LcaMsgMonitorPoints
   Dim MyMP As LcaMsgMonitorPoint
   Set MyMsgPoints = MyMonSet.MsgMonitorPoints
   Set MyMP =MyMsgMonitorPoints.Add("msgp1",MonVni,nothing)
   ```

8. Set the properties of the new `MsgMonitorPoint` object. Two properties you should initially set are the `Tag` property and the `DefaultOptions` properties. You can use the `DefaultOptions` property to set the default monitoring options that will be applied to the new monitor point each time it is enabled. See the *LNS Object Server Reference* help file for more information on the `Tag` property, and for a complete list of the properties of the `MsgMonitorPoint` object. For more information on monitoring options, see *Setting Monitoring Options* on page 200.

9. If you are started a transaction in step 2, commit the transaction to the database.

   ```
   MySystem.CommitTransaction()
   ```

10. You can now enable the monitor point in monitor and control operations. For more information on this, see *Opening and Enabling Monitor Sets* on page 213.

To create a permanent message monitor point to send explicit messages to multiple devices simultaneously, follow these steps. Note that you cannot use a temporary message monitor points to send explicit messages to multiple devices simultaneously:

1. Open the `ObjectServer`, `Network`, and `System` you plan to monitor and control, as described in Chapter 4, *Programming an LNS Application*.

2. Echelon recommends that you use transactions when creating permanent message monitor points, as this considerably reduces the time required to create the monitor points. Transactions should not generally be used when creating temporary monitor points since they are not stored in the LNS database, and so starting a transaction would unnecessarily slow down the process. For more information on using transactions with LNS, see *Using Transactions and Sessions* on page 65.

   ```
   MySystem.StartTransaction()
   ```

3. Create a dynamic message tag on your application's Network Service Device. You will use this message tag to connect the Network Service Device to the application devices you want to send messages to. For instructions on creating dynamic message tags, see *Adding Message Tags To a Custom Interface* on page 183.

4. Get the device that you plan to send explicit messages to with the new message monitor point.

```
Set MySubsystem =MySubsystems.Item("Third Floor Room 7")
Set MyAppDevices = MySubsystem.AppDevices
Set MyAppDevice= MyAppDevices.Item("Vent")
```

5. Get the `msg_in` message tag on the application device selected in step 4, and connect it to the dynamic message tag you created on the Network Service Device.

```
Set MyInterface = MyAppDevice.Interface
Set MyMessageTags = MyInterface.MessageTags
Set MyTag = MyMessageTags.Item("msg_in")

NsdDynamicMessageTag.AddTarget(MyTag)
```

6. Repeat steps 4 and 5 for all the application devices you want to send explicit messages to. Then, call `Connect()` to instantiate the connection.

```
NsdDynamicMessageTag.Connect()
```

NOTE: You can only add 25 targets to a connection each time you call the `Connect()` method. If you need to add more than 25 targets to the connection, you should add 25 targets and call `Connect()` to create the connection. Then, add the remaining targets, and call `Connect()` again. See Chapters 6 and 7 for more detailed information on creating connections.

7. Access the monitor set's `MsgMonitorPoints` collection, and call the `Add()` method to add a message monitor point to the monitor set. Pass the message tag created on the Network Service Device (`NsdMessageTag`) as the `targetDevice` element.

```
Dim MyMsgPoints as LcaMsgMonitorPoints
Dim NewMsgPoint as LcaMsgMonitorPoint
Set MyMsgPoints = MonSet.MsgMonitorPoints
Set NewMsgPoint = MyMsgPoints.Add _
             ("msgP1",NsdDynamicMessageTag, nothing)
```

8. Set the properties of the new `MsgMonitorPoint` object. Two properties you should initially set are the `Tag` property and the `DefaultOptions` properties. You can use the `DefaultOptions` property to set the default monitoring options that will be applied to the new monitor point each time it is enabled. See the *LNS Object Server Reference* help file for more information on the `Tag` property, and for a complete list of the properties of the `MsgMonitorPoint` object. For more information on monitoring options, see *Setting Monitoring Options* on page 200.

9. If you started a transaction in step 2, commit the transaction to the database.

```
MySystem.CommitTransaction()
```

10. You can now enable the monitor point in monitor and control operations. For more information on this, see *Opening and Enabling Monitor Sets* on page 213.

Setting Monitoring Options

Each MonitorSet object, and each individual MsgMonitorPoint and NvMonitorPoint object, contains properties that allow you to determine the behavior of the monitor set or monitor point while it is monitored and controlled. These properties are listed and described in the following sections.

- MonitorSet.NvOptions and MonitorSet.MsgOptions – Each MonitorSet object contains an NvOptions property, and a MsgOptions property. The NvOptions property provides access to an NvMonitorOptions object that stores the default monitoring options that are applied to all network variable monitor points as they are added to the monitor set. The MsgOptions property provides access to a MsgMonitorOptions object that stores the default monitoring options that are applied to all message monitor points as they are added to the monitor set.

  Changes to the NvOptions and MsgOptions properties are not managed retroactively, meaning that monitor points that have already been added to a monitor set are not affected when these properties are changed. However, for network variable monitor points, there are a few exceptions to this rule. See *NvMonitorOptions Object* on page 202 for more information on this.

- NvMonitorPoint.DefaultOptions and MsgMonitorPoint.DefaultOptions – Each NvMonitorPoint and MsgMonitorPoint object in a monitor set contains a DefaultOptions property. The DefaultOptions property provides access to a MsgMonitorOptions object (or NvMonitorOptions object) that stores the default monitoring options that are applied to that particular monitor point. These are the options that will be used for that monitor point when the monitor set is opened, and a monitoring session begins.

  When a monitor point is added to a monitor set, the DefaultOptions for the point are set to match to the settings stored in the NvOptions or MsgOptions objects for the monitor set. You can modify the monitor point's DefaultOptions property if you want that monitor point to have different default options than the rest of the set. However, if you change these properties while the monitor set is open, the changes will not be reflected until the monitor set is closed and re-opened.

  NOTE: Because temporary monitor points are only used in a single client session, their DefaultOptions are not accessible. If you attempt to acquire the DefaultOptions property through a temporary monitor point, the LCA:#161 lcaErrNotAllowedOnTemporaryObject exception will be thrown. All temporary monitor points are created with the default options defined for the monitor set via the NvOptions and MsgOptions properties.

- NvMonitorPoint.CurrentOptions and MsgMonitorPoint.CurrentOptions – Each NvMonitorPoint and MsgMonitorPoint object in a monitor set contains a CurrentOptions property. The CurrentOptions property provides access to a NvMonitorOptions object (or MsgMonitorOptions object). The CurrentOptions are set to match the values of the monitor point's DefaultOptions object every time the monitor point is enabled.

If you do not want a monitor point to use those default options during a particular monitoring session, you can change the options that will be used by writing to its `CurrentOptions`. All changes made to the `CurrentOptions` property take effect immediately. However, when the monitor set is closed, these changes will not be saved as the default options to be used the next time it is opened. The default options for each monitor point must be maintained using the `DefaultOptions` properties.

NOTE: For permanent network variable monitor points, the default values for some options properties (`ServiceType, Retries, Priority`) are controlled by the connection description specified when the monitor point is created. This is described in more detail in Table 9.1 in the next section.

Network Variable Monitor Point Options

The previous section of this document described the different ways to access an `NvMonitorOptions` object, and how each one affects a monitor set or individual network variable monitor point. To recap, you should follow these steps when setting the monitoring options for your network variable monitor points:

1. Set the defaults for the monitor set by writing to the `NvOptions` property of the `MonitorSet` object. The following code sample sets the default values for the `ReportByException`, `GenerateInitialFetch`, and `PollInterval` properties.

   ```
   Set NvMonitorPointOptions = MySet.NvOptions
   NvMonitorPointOptions.ReportByException = True
   NvMonitorPointOptions.GenerateInitialFetch = True
   NvMonitorPointOptions.PollInterval = 5000
   Set MySet.NvOptions = NvMonitorPointOptions
   ```

   For descriptions of the properties of the `NvMonitorOptions` object, see *NvMonitorOptions Object* on page 202.

2. If you do not want a network variable monitor point in the set to use the default options established in step 1, you can change its default options by writing to its `DefaultOptions` property. Note that these changes will not be applied to a monitor point that is in an open monitor set until the monitor set is closed and re-opened.

   Echelon recommends that you set the `DefaultOptions` property in the same transaction that you create the network variable monitor point whenever possible. The following code sample sets the default values for a network variable monitor point's `ResetPollingIfUpdated` and `SuppressPollingIfBound` properties.

   ```
   Set NvMonitorPointOptions = MonitorPoint.DefaultOptions
   NvMonitorPointOptions.ResetPollingIfUpdated = True
   NvMonitorPointOptions.SuppressPollingIfBound = False
   Set MonitorPoint.DefaultOptions = NvMonitorPointOptions
   ```

   NOTE: Because temporary monitor points are only used in a single client session, their `DefaultOptions` are not accessible. If you attempt to acquire the `DefaultOptions` property through a temporary monitor point, the `LCA:#161 lcaErrNotAllowedOnTemporaryObject` exception will be thrown. All temporary monitor points will be enabled with the default options selected in step 1.

3. When a monitor set is opened, each network variable monitor point in the set will use the default options established in either step 1 or step 2. If you want a network variable monitor point to use different options during a given monitoring session, you can do so by writing to the `NvMonitorPoint` object's `CurrentOptions` property after the monitor session has started (i.e. the monitor set has been opened). Remember that these changes take effect for the current session only.

   The following code sample sets the current values for a network variable monitor point's `PollInterval` property.

```
Set NvMonitorPointOptions = NvMonPoint.CurrentOptions
NvMonitorPointOptions.PollInterval = 1500
Set MonitorPoint.CurrentOptions = NvMonitorPointOptions
```

NOTE: When performing these steps, remember that the `NvOptions`, `DefaultOptions` and `CurrentOptions` properties are not passed by reference. When you modify an `NvMonitorOptions` object returned by any of these properties, the changes will not take effect until the modified object is passed back to the `NvOptions`, `DefaultOptions` or `CurrentOptions` property of the applicable monitor set or monitor point.

NvMonitorOptions Object

The three properties mentioned in this section (`NvOptions`, `DefaultOptions`, `CurrentOptions`) that apply to network variable monitor points all provide access to an `NvMonitorOptions` object. The `NvMonitorOptions` object contains a set of properties that define the behavior of network variable monitor points when they are enabled.

Table 9.2 lists and describes these properties. The default values listed in Table 9.2 are the defaults that are initially applied to the `NvMonitorOptions` object accessed through a monitor set's `NvOptions` property.

Table 9.2 NvMonitorOptions Object

Property Name	Description
`Authentication`	This property is not used. If the `UseBoundUpdates` option for a monitor point is set to `True`, the monitor point's authentication state is determined by the `UseAuthenticationFlag` property of the `ConnectDescTemplate` specified when the monitor point was created. Otherwise, the authentication state is set based on the monitored network variable's current authentication state.
`GenerateInitialFetch`	This property determines whether the values of the network variable monitor points in the monitor set will be fetched when the monitor set is enabled (`True`). If this property is set to `False`, the value of each network variable monitor point in the set will not be updated until a network variable update for that point is received due to an explicit fetch, bound update, or expired poll interval. **Default Value:** False
`PollInterval`	This property determines the interval to use when polling the network variables being monitored by this set. **Default Value:** 1000 ms* *The default value for this property for temporary monitor sets is set to match the value of the `DsPollInterval` property.

Property Name	Description
Priority	This property determines whether or not priority messaging will be used when sending network variable updates during the current monitoring session. If True, priority messaging will be used. Setting this property through the NvOptions property of a permanent monitor set, or through the DefaultOptions property of a permanent network variable monitor point, does not have an effect. The default priority setting for each permanent network variable monitor point is established by the UsePriorityFlag property of the ConnectDescTemplate specified when the network variable monitor point was created. You cannot set this property through the DefaultOptions property of a temporary network variable monitor point, as the DefaultOptions properties of all temporary monitor points are not accessible. However, you can set this property through the NvOptions property of a temporary monitor set to determine the default priority setting that will be applied to all network variable monitor points as they are added to the set. Remember that you can set this property through the CurrentOptions property of a permanent or temporary network variable monitor point to determine whether or not priority messaging will be used when sending network variable updates during the current monitoring session. **Default Value:** For permanent monitor sets, the default value for this property is False. The default value for temporary monitor sets is set to match the value of the DsPriority property.
ReportByException	This property determines if network variable update events for this set will only be reported when the value of a network variable changes. If True, LNS will only report update events to your application when the value of a network variable changes. For example, if the value of a network variable is polled three times and the value does not change until the third poll, only the third poll will be reported to your application via an update event. Network variable update events are described in more detail later in this chapter. **Default Value:** For permanent monitor sets, the default value for this property is True. The default value for temporary monitor sets is set to match the value of the DsReportByException property.

Property Name	Description
ResetPollingIfUpdated	Use this property to determine whether or not the poll interval for each network variable monitor point in the set will be reset whenever a new value for the point is received, either via a bound update or a read operation. If True, the poll interval will be reset. For example, consider a case where the poll interval is set to 100 seconds. If this property is set to True and a new value for a monitor point is received, LNS would reset the polling cycle and wait another 100 seconds to poll that network variable again. If this property is set to False, and LNS receives a new value 35 seconds after the most recent poll, LNS would not reset the polling cycle. The next poll would occur 65 seconds later. **Default Value:** True
Retries	This property determines the number of retries to use for acknowledged, request/response messages, and for repeat service messages during the current monitoring session. Setting this property through the NvOptions property of a permanent monitor set, or through the DefaultOptions property of a permanent network variable monitor point, does not have an effect. The default retry count for each permanent network variable monitor point is established by the RetryCount property of the ConnectDescTemplate specified when the network variable monitor point was created. You cannot set this property through the DefaultOptions property of a temporary monitor point, as the DefaultOptions properties of all temporary monitor points are not accessible. However, you can set this property through the NvOptions property of a temporary monitor set to determine the default retry count that will be applied to all network variable monitor points as they are added to the set. Remember that you can set this property through the CurrentOptions property of a permanent or temporary network variable monitor point to determine what retry count to use when sending network variable updates during the current monitoring session. **Default Value:** For permanent monitor sets, the default value for this property is 3. The default value for temporary monitor sets is set to match the value of the DsRetries property.

Property Name	Description
ServiceType	This property determines the default messaging service to use when explicitly writing the network variable monitor point. This does not apply to polling or implicit or explicit bound monitoring. Setting this property through the NvOptions property of a permanent monitor set, or through the DefaultOptions property of a permanent network variable monitor point, does not have an effect. The default messaging service for each permanent network variable monitor point is established by the ServiceType property of the ConnectDescTemplate specified when the network variable monitor point was created. You cannot set this property through the DefaultOptions property of a temporary network variable monitor point, as the DefaultOptions properties of all temporary monitor points are not accessible. However, you can set this property through the NvOptions property of a temporary monitor set to determine the default messaging service that will be applied to all network variable monitor points as they are added to the set. Remember that you can set this property through the CurrentOptions property of a permanent or temporary network variable monitor point to determine messaging service to use when sending network variable updates during the current monitoring session. **Default Value:** For permanent monitor sets, the default value for this property is the acknowledged message service. The default value for temporary monitor sets is set to match the value of the DsService property.

Property Name	Description
SuppressPollingIfBound	This property determines if polling will be turned off for monitor points whose target network variable is bound to the host device. If True, polling will be turned off. In this case, polling will be turned off even if the monitor point does not use bound updates, as long as the target network variable is bound to the host device through some other means. For example, if an network variable is explicitly connected to a dynamic network variable on the Network Service Device (perhaps via a fan-in connection), bound updates from that network variable will be reported on all the network variable monitor points monitoring the output network variable, and if the SuppressPollingIfBound property is True, the output network variable will not be polled. **Default Value:** True
ThrottleInterval	This property contains the default throttle interval for network variable updates for the set. This sets the maximum interval between successive OnNvMonitorPointUpdateEvent or OnMsgMonitorPointUpdateEvent events that can be generated by the monitor set. If multiple updates are received within the throttle interval, LNS will not fire a second event until the throttle interval has expired, and will then fire an event for the last update received. Thus, if this property is non-zero and multiple updates are received within this interval, events will only be fired for the first and last updates. And so some updates may be lost. This value should never be greater than the polling interval, since that would guarantee that some polled updates would be lost, and would waste network bandwidth. **Default Value:** For permanent monitor sets, the default value for this property is 1000ms. The default value for temporary monitor sets is set to match the value of the DsPollInterval property.
UseAsyncSend	Use this property to determine whether or not LNS will wait for a completion code to return after updating the value of a monitor point, before returning to the user and sending its next update message. If True, LNS will not wait for a completion code. **Default Value:** False

Property Name	Description
UseBoundUpdates	If set to True, this property enables implicit binding. With implicit binding enabled, LNS will attempt to create connections without user intervention. Changing this option through the CurrentOptions property does not have an effect. Before using this property, you should review the *Using Network Variable Monitor Points* section, which describes different ways you can use implicit binding. This property must be set to False for temporary monitor sets and points. **Default Value:** False

NOTE: In the NvMonitorOptions object accessed through the DefaultOptions property of a network variable monitor point, the PollInterval and ThrottleInterval properties will initially be set to lcaDefaultMcpInterval (-1). The GenerateInitialFetch, ReportByException, ResetPollingIfUpdated, and SuppressPollingIfBound properties will be initially set to lcaDefaultMcpOption(-1). This causes the values of those properties to always match the corresponding value stored in the monitor set's NvOptions object. As noted previously, the values of all the other properties of the monitor set's NvOptions object are only applied to monitor points as they are added to the set, and so changes to the monitor set's NvOptions object are not retroactively applied to the monitor points already contained in the set for any other properties.

Message Monitor Point Options

To recap the previous discussion in this chapter, you should follow these steps when setting the monitoring options for your message monitor points:

1. Set the defaults for the monitor set by writing to the MsgOptions property of the MonitorSet object. The following code sample sets the default values for the FilterByCode, and FilterBySource properties.

   ```
   Dim MsgMonitorPointOptions As LcaMsgMonitorOptions
   Set MsgMonitorPointOptions = MyMonitorSet.MsgOptions
   MsgMonitorPointOptions.FilterBySource = True
   MsgMonitorPointOptions.FilterByCode = True
   Set MyMonitorSet.MsgOptions = MsgMonitorPointOptions
   ```

   For descriptions of the properties of the MsgMonitorOptions object, see *MsgMonitorOptions Object* on page 209.

2. If you do not want a message monitor point in the monitor set to use the default options established in step 1, set its defaults by writing to the message monitor point's DefaultOptions property. Note that these changes will not be applied to a monitor point that is in an open monitor set until the monitor set is closed and re-opened.

   Echelon recommends that you set the DefaultOptions property in the same transaction that you create the message monitor point whenever possible. The following code sample sets the default values for a message

monitor point's FilterCode property.

```
Set MsgMonitorPointOptions =_
          MyMsgMonitorPoint.DefaultOptions
MsgMonitorPointOptions.FilterCode = 62
Set MyMsgMonitorPoint.DefaultOptions =_
          MsgMonitorPointOptions
```

NOTE: Because temporary monitor points are only used in a single client session, their DefaultOptions are not accessible. If you attempt to acquire the DefaultOptions property through a temporary monitor point, the LCA:#161 lcaErrNotAllowedOnTemporaryObject exception will be thrown. All temporary monitor points are instantiated with the default options selected in step 1.

3. When a monitor set is opened, each message monitor point will use the default options established in either step 1 or 2. If you want to use different options for a monitor point during a given monitoring session, you can do so by writing to the MsgMonitorPoint object's CurrentOptions property. Remember that these changes will take effect for the current session only.

   You can change these defaults by writing to the CurrentOptions property. The following code sample sets the current values for a message monitor point's Priority property.

   ```
   Set MsgMonitorPointOptions = MsgMonPoint.CurrentOptions
   MsgMonitorPointOptions.Priority = True
   Set MsgMonPoint.CurrentOptions = MsgMonitorPointOptions
   ```

NOTE: The MsgOptions, DefaultOptions and CurrentOptions properties are not passed by reference. When you modify a MsgMonitorOptions object returned by any of these properties, the changes will not take effect until the modified object is passed back to the MsgOptions, DefaultOptions or CurrentOptions property of the applicable monitor set or monitor point.

MsgMonitorOptions Object

The three properties mentioned in this section (MsgOptions, DefaultOptions, CurrentOptions) all provide access to a MsgMonitorOptions object. The MsgMonitorOptions object contains a set of properties that define the behavior of message monitor points when they are enabled. Table 9.3 lists and describes these properties. The default values listed in Table 9.3 are the defaults that are initially applied to the MsgMonitorOptions object accessed through a monitor set's MsgOptions property.

Table 9.3 MsgMonitorOptions Object

Property Name	Description
`Authentication`	This property determines whether authentication will be used when sending explicit messages using this monitor point or monitor set. If this property is set to `True`, authentication will be used. **Default Value:** False
`FilterByCode`	This property indicates whether the `FilterCode` property will be used to filter message tag values. If this property is set to `True`, only message tags with certain message codes will be passed to your client application. You can determine which message codes will be passed to the application by setting the `FilterCode` property described later in this table. **Default Value:** False
`FilterBySource`	This property indicates whether message tag values will be filtered by source device. If this is set to `True`, your application will only receive messages from the `AppDevice` specified as the `targetDevice` element when the message monitor point was created. This value should never be set to `True` if the monitor point is created with a dynamic message tag as the `targetDevice` element. **Default Value:** `True` for message monitor points created with an `AppDevice` as the `targetDevice` element. `False` for message monitor points created with a dynamic message tag as the `targetDevice` element.
`FilterCode`	This property indicates the message code to use as a filter when the `FilterByCode` property is set to `True`. **Default Value:** 0

Property Name	Description
Priority	This property determines whether or not priority messaging will be used when sending explicit messages with this monitor set or monitor point. If True, priority messaging will be used. Setting this property through the MsgOptions property of a permanent monitor set, or through the DefaultOptions property of a permanent message monitor point, does not have an effect. You cannot set this property through the DefaultOptions property of a temporary message monitor point, as the DefaultOptions properties of all temporary monitor points are not accessible. However, you can set this property through the MsgOptions property of a temporary monitor set to determine the default priority setting that will be applied to all message monitor points as they are added to the set. Remember that you can set this property through the CurrentOptions property of a permanent or temporary message monitor point to determine whether or not priority messaging will be used when sending explicit messages with this monitor point. **Default Value:** For permanent monitor sets, the default value for this property is False. For temporary monitor sets, the default value is set to match the DsPriority property of the System object.

Property Name	Description
Retries	This determines the number of retries to use for acknowledged, request/response, or repeat service messages. Setting this property through the MsgOptions property of a permanent monitor set, or through the DefaultOptions property of a permanent message monitor point, does not have an effect. The default retry count for each permanent message monitor point is established by the RetryCount property of the ConnectDescTemplate specified when the message monitor point was created. You cannot set this property through the DefaultOptions property of a temporary message monitor point, as the DefaultOptions properties of all temporary monitor points are not accessible. However, you can set this property through the MsgOptions property of a temporary monitor set to determine the default retry count that will be applied to all message monitor points as they are added to the set. Remember that you can set this property through the CurrentOptions property of a permanent or temporary message monitor point to determine what retry count to use when sending messages using this message monitor point during the current monitoring session. **Default Value:** For permanent monitor sets, the default value for this property is 0. For temporary monitor sets, the default value is set to match the DsRetries property of the System object.
ServiceType	This property specifies the default messaging service to use when sending explicit messages with this monitor set or monitor point. **Default Value:** For permanent monitor sets, the default value for this property is the acknowledged message service. For temporary monitor sets, the default value is set to match the DsService property of the System object.
UseAsyncSend	LNS sets this property automatically for message monitor points. For example, if you are sending a message via the SendMsgWait() method, a response from the device is expected. Therefore, LNS would set the UseAsyncSend property to False. Or, if you are writing to a message monitor point via the OutputDataPoint property, no response is expected. So, LNS would set the property to True.

Opening and Enabling Monitor Sets

After you have created a monitor set and defined its monitoring options, you can open it and enable it for monitor and control operations. Note that you will not be able to use a permanent monitor set that was created while the system management mode was `lcaMgmtModeDeferConfigUpdates` until the system management mode has been changed back to `lcaMgmtModePropagateConfigUpdates`. In addition, configuration changes made to a permanent monitor set made while the system management mode is set to `lcaMgmtModeDeferConfigUpdates` will not take effect until the management mode is set to `lcaMgmtModePropagateConfigUpdates`.

Remember that you should access permanent monitor sets through the `CurrentMonitorSets` property when you plan to open them. To open a monitor set, call the `Open()` method on the monitor set. The `Open()` method takes two parameters: `doEnable` and `doPoll`. The `doEnable` parameter determines whether or not the monitor set should be enabled when it is opened. If `True`, monitoring of all points in the monitor set will be enabled. In this case, the `doPoll` parameter indicates whether or not polling of the monitor set will also be enabled. Monitor set polling is described in more detail later in this chapter.

If the `doEnable` parameter is set to `False`, the monitor set will not be enabled, and the `doPoll` parameter will not have an effect. In this case, you will need to enable the monitor set, or a group of monitor points within the set, later with the `Enable()` method.

NOTE: You do not need to use the `Open()` method to open temporary monitor sets with your application, as they are opened automatically by LNS when they are created. However, LNS does not enable temporary monitor sets at this point. This is so you can set the `Tag` property and the monitoring options for your temporary monitor sets before enabling them. As a result, you need to manually enable all temporary monitor sets with the `Enable()` method, as described in the next section.

Using the Enable Method

You can enable all the monitor points in a permanent monitor set at once by setting the `doEnable` element to `True` when you open the monitor set. If the `doEnable` element is set to False, you can enable the monitor set later by calling the `Enable()` method on the `MonitorSet` object later. You can use the same method to enable a temporary monitor set.

You can also enable monitoring for an individual network variable or message monitor point in a permanent or temporary monitor set by calling the `Enable()` method on the `NvMonitorPoint` or `MsgMonitorPoint` object.

For monitor sets and network variable monitor points, the `Enable()` method takes one input parameter: `doPoll`. If this is set to `True`, polling will be started on the network variable monitor points when it is enabled. For message monitor points, the `Enable()` method takes no input parameters, as you cannot poll message monitor points.

You can use the `Disable()` method to stop all polled and bound monitoring and control for a monitor set or monitor point. If the `Disable()` method is called on a `MonitorSet` object, polled and bound monitoring for all monitor points on the monitor set will be disabled. After this, none of the monitor points in the set can be enabled for monitoring until the `Enable()` method has been called on the `MonitorSet` object again.

You can also call the `Disable()` method on a specific `NvMonitorPoint` or `MsgMonitorPoint` object. This will override subsequent calls to the `MonitorSet` object's `Enable()` method. In other words, if you call the `Disable()` method on a monitor point named `Point A`, and then call the `Enable()` method on the monitor set containing `Point A`, `Point A` would not be enabled, but all other monitor points in the monitor set that have not been explicitly disabled would be. Once you have disabled a monitor point with the `Disable()` method, you can only re-enable it by calling the `Enable()` method on the monitor point, or by closing and re-opening the monitor set it belongs to.

Once a monitor set or monitor point has been opened and enabled, you can use it for monitor and control operations. For guidelines on how you can use network variable monitor points, see *Using Network Variable Monitor Points* on page 214. For guidelines on how you can use message monitor points, see *Using Message Monitor Points* on page 224.

Using Network Variable Monitor Points

This section describes four ways that you can use network variable monitor points to monitor the values of the network variables on your devices:

- *Explicitly Reading the Monitor Point*. In this scenario, the application explicitly reads network variable values individually. This method is most efficient if network variable values need to be read infrequently and unpredictably. For more information on this, see *Explicitly Reading and Writing Network Variable Monitor Points* on page 215.
- *Polled Monitoring*. This method is most efficient when the value of a network variable must be checked regularly, but the application does not need to know immediately if the value changes (for example, outside air temperature). In this scenario, LNS periodically polls the value of the monitor points in the monitor set and reports them to your application via the `OnNvMonitorPointUpdateEvent` event. For more information on this, *Polled Network Variable Monitoring* on page 217.
- *Implicit Bound Monitoring*. This method is most efficient when you are monitoring a network variable whose value will change infrequently, but your application will require immediate notification when the value does change (for example an alarm or failure condition notification). For more information on this, see *The Implicit Bound Network Variable Monitoring Scenario* on page 220.
- *Explicit Bound Monitoring and Control*. In this scenario, network variables are explicitly created on the Network Service Device used by the monitor and control application, and bound to network variables on the devices in your network. This method allows the implementation of fan-in monitoring, which is the process of connecting two or more output device network variables with a single host input network variable. For more information, see *The Explicit Bound Network Variable Monitoring and Control Scenario* on page 221.

There are two ways you can use network variable monitor points to control the value of a network variable:

- *Explicitly Writing the Monitor Point*. In this scenario, the application explicitly writes network variable values individually, one at a time. This

method is most efficient if network variable values need to controlled individually, or if they need to be written infrequently and unpredictably. For more information on this, see *Explicitly Reading and Writing Network Variable Monitor Points* on page 215.

- *Explicit Bound Monitoring and Control*. This method should only be used for fan-out connections, which is the process of a single host output network variable controlling two or more input device network variables. This is most efficient when updating many network variables to the same value at once. For more information, see *The Explicit Bound Network Variable Monitoring and Control Scenario* on page 221.

Explicitly Reading and Writing Network Variable Monitor Points

In this scenario, the application explicitly reads and writes network variable values. This method is most efficient if network variable values need to be read infrequently and unpredictably, or written individually.

To explicitly read or write a network variable monitor point, you must first create a monitor set and add monitor points for the network variables you want to monitor. For more information on these tasks, see *Creating Monitor Sets* on page 194, and *Managing Monitor Sets* on page 195.

Following that, you can open and enable the monitor set, and begin explicitly reading and writing the values of the network variable monitor points in the monitor set. To do so, you need to acquire a `DataPoint` object representing the network variable monitor point. You can access such a `DataPoint` object by reading the `NvMonitorPoint` object's `DataPoint` property.

The `DataPoint` object contains the value data for the monitor point. The value data is stored in three properties on the `DataPoint` object: `Value`, `FormattedValue`, and `RawValue`. All three of these properties point to the same piece of data in memory, but they format the data in different ways:

- The `Value` property formats the data as a raw, scaled numeric value.
- The `FormattedValue` property formats the data based on the LNS Object Server's `CurrentFormatLocale` setting. The data stored in the `FormattedValue` property is scaled and unit-converted based on the `FormatSpec` the data point is using. For more information on `FormatSpec` objects, see *Data Formatting* in page 241.
- The `RawValue` property formats the data as a raw byte array.

Whenever one of these values is set, the values of the other two properties will be updated to reflect this change. However, you should note that reading and writing any of the three value properties does not automatically access or update the network variable on the source device.

Each `DataPoint` object contains a `Read()` method and a `Write()` method. The `Read()` method copies the monitor point value data from the device into the `DataPoint` object. The `Write()` method copies the monitor point value data from the `DataPoint` object to the device. It is possible to automate this process with the `AutoRead` and `AutoWrite` properties. Set the `AutoRead` property to `True` to cause the `Read()` method to be called automatically before any of the three value properties are read. Set the `AutoWrite` property to `True` to cause the `Write()` method to be called automatically after any of the three value properties are set.

Example of Explicitly Reading a Network Variable Monitor Point

To recap the previous discussion, follow these steps to explicitly read a network variable value via a monitor point:

1. Create a monitor set on the device containing the network variables you want to monitor, as described in *Creating Monitor Sets* on page 194. Then, create a monitor point for each network variable you want to monitor, as described in *Adding Network Variable Monitor Points to a Monitor Set* on page 195.

2. Access the monitor set through the `CurrentMonitorSets` collection, and then open and enable the monitor set. In this example, the `doEnable` parameter supplied to the open method is set to `True`, so that the monitor set is enabled as it is opened.

   ```
   Set MyMonitorSets = MyNetwork.CurrentMonitorSets
   Set MyMonitorSet = MyMonitorSets.Item("Boiler Set")
   MyMonitorSet.Open(True, True)
   ```

3. Acquire the monitor set's `NvMonitorPoints` collection, and access the monitor points for the network variable you want to monitor.

   ```
   Set MyNvMonitorPoints = MyMonitorSet.NvMonitorPoints
   Set MyNvMonitorPoint =MyNvMonitorPoints.Item("Pressure")
   ```

4. Acquire a `DataPoint` object to use to explicitly read the value of the network variable monitor point. Then, call the `Read()` method. LNS will read the value of the network variable being monitored, and update the `DataPoint` object's value properties to match the current value.

   ```
   Set MyDataPoint = MyNvMonitorPoint.DataPoint
   MyDataPoint.Read()
   ```

   ```
   Dim readValue as String
   readValue = MyDataPoint.FormattedValue
   ```

   NOTE: If the `AutoRead` property is set to `True`, you do not need to call the `Read()` method in this step.

There are several ways your application can format the data stored in the `DataPoint` object for display. For more information on this, see *Data Formatting* in page 241.

Example of Explicitly Writing a Network Variable Monitor Point

Follow these steps to explicitly write a network variable value via a monitor point:

1. Acquire the network variable monitor point you want to control, as described in steps 1 through 3 of the *Example of Explicitly Reading a Network Variable Monitor Point* scenario.

2. Acquire a DataPoint object to use to write the value of the network variable monitor point. Then, set the value of the DataPoint object and call the Write() method. LNS will update the value of the network variable being monitored with the new value.

```
Set MyDataPoint = MyNvMonitorPoint.DataPoint
MyDataPoint.FormattedValue = “75.0 1”
MyDataPoint.Write()
```

NOTE: If the AutoWrite property is set to True, you do not need to call the Write() method in this step.

Polled Network Variable Monitoring

In the polled monitoring scenario, LNS periodically polls the values of the monitor points and reports the values using the OnNvMonitorPointUpdateEvent event. This method is most efficient when the value of a network variable must be checked regularly, but the application does not need to know immediately when the value changes (for example, outside air temperature).

To use polled monitoring, you must first create a monitor set and add monitor points for the network variables you want to monitor. For more information on these tasks, see *Creating Monitor Sets* on page 194, and *Managing Monitor Sets* on page 195. Note that when you set the monitoring options, the PollInterval property determines the rate at which LNS will poll the value of each network variable being monitored.

Once you have created a monitor set and established a poll interval, you can open and enable the monitor set, or a group of monitor points within the set. When you open and enable the monitor set, make sure that the doPoll parameter is set to True. This will enable polled monitoring for all monitor points on the monitor set. For more information on how to open and enable monitor sets, see *Opening and Enabling Monitor Sets* on page 213.

You can stop polled monitoring for an entire monitor set by calling the Disable() method on the monitor set (this will stop all monitoring). Or, you can stop polled monitoring by calling the Enable() method with the doPoll parameter set to False (this will leave monitoring enabled). In this case, polling will not be enabled for that monitor point, even if it is enabled for the monitor set, until you call Enable() method on the NvMonitorPoint object with the doPoll parameter set to True.

Once polling for a network variable monitor point has been enabled, your application will be informed of the result of each poll via the OnNvMonitorPointUpdateEvent event. If you only want your application to be informed when the value of a network variable monitor point changes, set the ReportByException property to True when you set the monitoring options for the set or point.

The OnNvMonitorPointUpdateEvent event includes three parameters: the NvMonitorPoint object whose value is being reported, a DataPoint object containing the value of the NvMonitorPoint, and a SourceAddress object that you can use to identify the device containing the network variable.

Echelon does not recommend using the SourceAddress object to identify the network variable, since source addresses may change, and translating from a source address to an application device may be time consuming. A more efficient alternative is to set the NvMonitorPoint object’s Tag property to a value you can use to identify the network variable and source device associated with the monitor point when you create it.

To recap this discussion, make sure that you follow these steps when using the polled network variable monitoring scenario:

1. When you create your monitor points, you should consider setting the `Tag` property to a value you can use to identify the source device and network variable.

2. When you set the monitoring options for the monitor set, make sure that the `PollInterval` property is set to the interval at which you want to poll each network variable. In addition, make sure that the `ThottleInterval` property is not set to a greater value than the `PollInterval` property. The `ThottleInterval` property should be set to 0 if you want to be assured of receiving an event for every network variable update.

   See the next section, *Setting the Poll Interval*, for guidelines you should follow when setting the poll interval for your application.

3. If you only want your application to be informed when the value of a network variable monitor point changes, set the `ReportByException` property to `True` when you set the monitoring options for the monitor set.

4. When you enable the monitor set, or when you enable any of the monitor points in the set, make sure that the `doPoll` element is set to True. Your application will be informed of the result of each poll via the `OnNvMonitorPointUpdateEvent` event. A sample `OnNvMonitorPointUpdateEvent` event handler is shown later in the *Example of a Network Variable Event Handler* section on page 219.

Setting the Poll Interval

When setting the poll interval for your LNS application, you should note the following:

1. The total number of data points being polled, combined with the `PollInterval` chosen, results in a specific number of network variable polls per second. The maximum number of polls that your network can sustain is subject to the channel types being used, network topology, use of authentication, size of the data being polled, performance of the devices that contain the polled network variables, and network interface you are using. You should make sure that your poll interval does not cause your network resources to be exhausted.

2. The poll interval that you specify is not a precise value. LNS tries to poll the data at the specified interval, but some network conditions may prevent polling at the exact, desired moment. For example, if too many messages and polling requests are using priority slots, it could cause collisions on the priority slots. Regular network traffic, or transient partial network outages may prevent successful polling at times. Such failure will be reported by the `OnMsgMonitorPointErrorEvent` or `OnNvMonitorPointErrorEvent` events.

3. Typical LONWORKS networks implement a distributed control algorithm by means of distributed devices, and data exchange with network variables. In many of those systems, such as a typical building control system, monitoring applications are used to oversee network operation.

When you design such a monitoring application, you should bear in mind that the network's primary focus is on the execution of the control algorithm, and that any network traffic generated, or any device or network resources consumed by your monitoring approach, should not prevent the network from fulfilling its primary task. You should design network applications to be as unobtrusive as possible.

4. Each individual device on a network responds to a certain number of network variable poll requests per second. The LNS application has substantial processing power and a high performance network interface at hand, and can typically handle more network variable poll requests per second than most LONWORKS devices. When defining the poll interval, make sure not to flood a single device with poll requests by exceeding its individual limit. Not only would the device be unable to respond to the request at the desired schedule, the polling requests would also flood the device's input buffers and might cause transient errors in the operation of the control algorithm.

5. The poll interval is the interval between the last completion event for a poll, and the next poll request. So if you were to set the poll interval for a network variable to be 5 seconds, and it took 0.5 seconds to poll the network variable, LNS would poll the network variable about every 5.5 seconds. In doing so, LNS avoids queuing poll requests for the same network variable, and slows down polling for a particular device when its responses are slow. This also causes less bandwidth to be consumed by polls, because LNS automatically slows down if there is a large number of retries for a given network variable.

6. After a network variable poll fails, the LNS Object Server will reduce the polling interval of the device containing the network variable to once per minute. If multiple network variables were being polled on the device, the polling will be restricted to a single network variable regarded as the "probe network variable." You should note that polling of the probe network variable is randomized over each minute, so these polls may not necessarily occur exactly once every 60 seconds. This has the effect of distributing the probe polling more evenly. The LNS Object Server will resume normal polling of the device as soon as it successfully polls the probe network variable. Depending on the interval, it might recover in less than 60 seconds. Note that normal polling may also resume before the probe network variable is successfully polled if the LNS Object Server detects a message from the device - for example if the device sends a network variable update, or if the LNS Object Server pings the device.

Example of a Network Variable Event Handler

The following shows a sample event handler you could use for the `OnNvMonitorPointUpdateEvent`. In this example, the application checks if the `Tag` property of the monitor point returned by the event matches a certain value to determine whether it should display the monitor point's value. If the tag matches, then the application obtains the monitor point's new value from the `DataPoint` returned by the event. There are several ways your application can format and display the data stored in the `DataPoint` object. For more information on this, see *Data Formatting* in page 241.

```
Dim roomTemp as String
Private Sub lcaObjectServer_OnNvMonitorPointUpdateEvent( _
```

```
            ByVal nvMonitorPoint As Object, _
            ByVal dataPoint As Object, _
            ByVal srcAddress As Object)
      If nvMonitorPoint.Tag = "Room Thermometer" Then
            roomTemp = dataPoint.FormattedValue
      End If
End Sub
```

Note that by using the `OnNvMonitorPointUpdateEvent` event as described in this section, you will receive the events through the LNS Object Server. You can use the `ILcaNvMonitorPointListener` interface to receive the events directly from the network variable monitor point, which may be more efficient. You can only use this technique if your development environment supports multi-threading. For more information on this, see *Tracking Monitor Point Updates* on page 228.

The Implicit Bound Network Variable Monitoring Scenario

In the implicit bound monitoring scenario, LNS implicitly creates a connection from each monitored output network variable in the monitor set to the corresponding network variable monitor point. Every time the value of an output network variable on a device is updated, the device will send a network variable update to LNS through the appropriate connection. LNS will then report the update to the LNS client application via the `OnNvMonitorPointUpdateEvent` event, as with the polled network variable monitoring scenario. The network variable connections created by LNS are "invisible" connections, meaning that you will not see the connections in any of the hub or target collections, and the connections do not use any network variables on the Network Service Device.

NOTE: Network variable monitor points in temporary monitor sets cannot be automatically bound by LNS to the monitoring node. As a result, you can only use the implicit bound network variable monitoring scenario with permanent monitor sets.

Bound monitoring is most efficient when your application needs to be notified immediately when the value of the network variable changes, but the value may not change very frequently (for example an alarm or failure condition notification). You will receive updates sooner using this method than the polled network variable monitoring method, because your application will be notified of network variable updates as soon as they occur. When using polled network variable monitoring, you will only receive `OnNvMonitorPointUpdateEvent` events at the rate specified by the `PollInterval` property.

To use implicit bound monitoring, follow these steps:

1. Create a monitor set and add monitor points for the network variables you want to monitor. For more information on these tasks, see *Creating Monitor Sets* on page 194, and *Managing Monitor Sets* on page 195.
2. When you set the monitoring options for the monitor set, set the `UseBoundUpdates` property to `True` to enable implicit bound monitoring. Note that it is possible for polled and bound monitoring to take place simultaneously. Set the `SuppressPollingIfBound` property to `True`, or set the `PollInterval` property to 0, to ensure that only bound monitoring takes place.
3. Make sure that you set the `ThrottleInterval` property appropriately. This property should be set to 0 if you want to be sure that network variable update events are fired for each update.

4. Enable the monitor set, as described in *Opening and Enabling Monitor Sets* on page 213. Once implicit bound monitoring has started, your application will be informed of each network variable update via the `OnNvMonitorPointUpdateEvent` event. If you only want your application to be informed when the value of a network variable monitor point changes, set the `ReportByException` property to `True` when you set the monitoring options for the monitor point or monitor set.

   The `OnNvMonitorPointUpdateEvent` event includes three parameters: the `NvMonitorPoint` object whose value is being reported, a `DataPoint` object containing the value of the `NvMonitorPoint`, and a `SourceAddress` object that you can use to identify the device containing the network variable.

   Echelon does not recommend using the `SourceAddress` object to identify the network variable, since source addresses may change, and translating from a source address to an application device may be time consuming. A more efficient alternative is to set the `NvMonitorPoint` object's `Tag` property to a value you can use to identify the network variable and source device associated with the monitor point when you create the monitor point.

   See *Example of a Network Variable Event Handler* on page 218 for a sample `OnNvMonitorPointUpdateEvent` event handler.

The Explicit Bound Network Variable Monitoring and Control Scenario

In the explicit bound monitoring and control scenario, the application explicitly creates a host network variable on the LNS application's Network Service Device, connects one or more device network variables to the host network variable, and then creates a monitor point for the device's output network variable.

Fan-in Connections

Consider an alarm system in which a number of detector devices are installed which can send network variable updates indicating when an intruder has been detected. You could use the implicit bound monitoring scenario to create bound monitor points to monitor the detector devices. When one of the detector devices detected an intruder, it would update its network variable, which would cause the `OnNvMonitorPointUpdateEvent` event to be fired. In this case, the LNS application could use the parameters returned with the `OnNvMonitorPointUpdateEvent` event to determine which alarm had been triggered.

However, it may be more natural to connect all of the alarm device output network variables to a single input network variable on the host PC. This type of connection is called a fan-in connection, because multiple output network variables "fan-in" to a single input network variable. Figure 9.1 shows an example of fan-in where a single input network variable on the host is connected to two output network variables on the network.

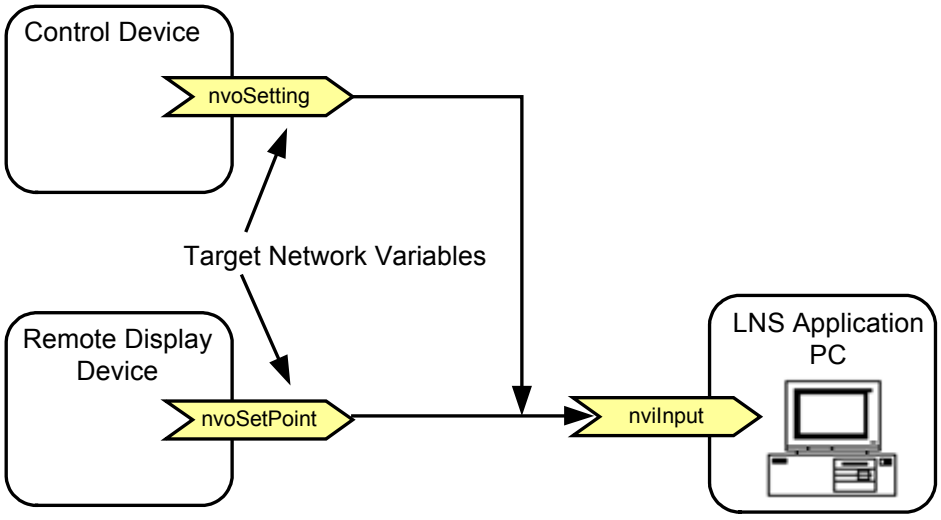


Figure 9.1 Fan-In Connection

When monitoring a fan-in connection, you can monitor the input network variable, the output network variables, or both. If your application needs to identify the source of an update, you can create a monitor point for each output network variable, and set the `UseBoundUpdates` property to `False`, and the `SuppressPollingIfBound` property to `True`, when you configure the options for those points. Each time one of the output network variables is updated, LNS will generate an `OnNvMonitorPointUpdateEvent` for the monitor point associated with the source of the update. If you create a network variable monitor point for the input network variable on the Network Service Device, you will also get an update on that point any time any device involved in the connection sends an update.

Fan-out Connections

Consider a heating system in an office building in which all fans and vents must be shut off and closed due to a supervisor command given at a central monitoring station. You could explicitly write a control value to each of the devices in order to shut them down, but there are two problems with this.

First, it requires knowledge of the maximum number of fan and vent devices to be controlled, and the application will need send the control value to each one separately. This will cause at least one message to be sent for each device, which could cause a large amount of network traffic. Second, it's an inefficient use of resources for the LNS Server, which must maintain the network variable configuration, network variable alias, and self-documentation tables, as well as maintain a record of the application's external interface and connections.

An easier, more flexible, and more efficient solution is to connect a single control value output network variable on the host to each of the input network variables on the fan and vent devices. This type of connection is called a fan-out connection because a single output network variable "fans-out" to multiple input network variables. When making such a connection, it is important that you use an appropriate `ConnectDescTemplate` object. The default connection description for all network variables on a Network Service Device has the `AliasOptions` property set to `lcaAliasForUnicast` by default. This is generally the best setting for fan-in connections to the Network Service Device, as it may prevent creating a group for each output in the fan-in connection that may also be

connected to an input on another device. However, this is usually not the appropriate setting for a fan-out connection, as this will generally result in sending out individual messages for each input network variable.

You could then write to the value of the output network variable to update the values of all of the input network variables on the fan and vent devices at once. Figure 9.2 shows an example fan-out connection where an output network variable is connected to input network variables on two devices on the network.

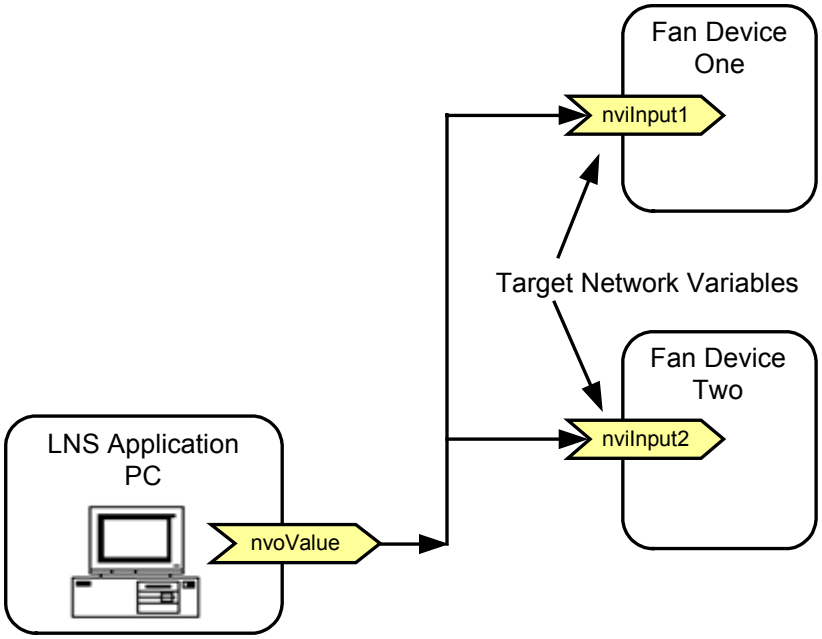


Figure 9.2 Fan-Out Connection

Creating and Using Host Network Variables

Host network variables are network variables that are created on the PC running the monitor and control application (or on the PC running the LNS Server, for a Lightweight client application). These network variables can be used just like network variables on application devices, although they are frequently used to support bound monitoring and control by the LNS application. To use host network variables in this scenario:

1. Create a custom interface and define the network variables to be used on the host PC. You can create these dynamically after the system has been opened. See *Defining Host Network Variables* on page 224 for more information on this.

2. Connect the network variables on the device you want to monitor to the network variable on the host PC, as desired. For information on creating connections, see *Connecting Devices* on page 138.

3. Create a monitor set to monitor the network variables on the devices you want to monitor. Then, implement code for an `OnNvMonitorPointUpdateEvent` event handler, or set up a polling scheme that will explicitly read and write the value of the host network variable.

4. You can now control the input network variables on the network by writing to the host network variable on the PC. Or, you can monitor value changes on the device network variables though the `OnNvMonitorPointUpdateEvent` event.

Defining Host Network Variables

An LNS application can dynamically create and delete local host network variables (i.e. network variables on the Network Service Device being used by the LNS application). All host network variables become part of the local Network Service Device's main interface, which is accessible by reading the `Interface` property contained in the `NetworkServiceDevice` object's `AppDevice` object. It is not possible to add network variables directly to the main interface.

However, you can add network variables to custom `Interface` objects created by LNS applications. To do so, use the `Interfaces` collection returned by the `NetworkServiceDevice` object's `Interfaces` property. For more information on custom interfaces, including how to add and remove network variables from them, see *Using Dynamic Device Interfaces* on page 179.

Using Message Monitor Points

LNS supports the transmission and receipt of application messages and foreign frame messages. Application messages and foreign frame messages are network messages that do not relate to network variables, network management or diagnostic services. Application messages use a 0x00...0x3E message code range, and are sometimes used for non-interoperable data exchange. Foreign frame messages use the 0x40...0x4E message code range and are sometimes used when tunneling packets related to other communication protocols through a LONWORKS network. This allows you to use your application to monitor and control devices that use application messaging.

Application messages are explicitly constructed in the transmitting device, unlike network variable messages, which are implicitly constructed by writing to a network variable or by calling library functions. Message monitoring is implemented by connecting an output message tag on a device on the network to an input message tag on the LNS application's Network Service Device. Note that application messages cannot be polled, and therefore application message monitoring always requires a bound connection.

Message control is implemented with message monitor points. You can use the message monitor points to send explicit messages to the device synchronously or asynchronously. When using message monitor points, you should note that the maximum size of any application message is usually 40 bytes. However, this varies depending on the network buffers and the application buffers in the network interface you are using, as well as on any network or application buffers in any intervening routers along the message's path, and in the application device that is receiving the message. If a message fails as a result of being too large, the failure will appear to be the same as any communication failure, and the device receiving the message will not respond. As a result, you should be careful not to exceed your available network resources when creating an application to send application messages.

Monitoring Message Monitor Points

The first step to take when monitoring application messages is to explicitly connect a message tag on an application device to the msg_in tag on your application's Network Service Device. This will cause any messages sent by the device using the message tag to be delivered to the Network Service Device. The second step is to create a message monitor point to receive the messages. The message monitor point is used to inform LNS that the application is interested in the message, based on the filtering criteria contained in the point's configuration, and to provide information to the application about the message's origin. Finally, the LNS application must open the monitor set containing the message monitor point, and enable monitoring. After that, any qualifying message sent by the application device using the bound message tag will generate an `OnMsgMonitorPointUpdateEvent` event.

It is important to understand that the messages being monitored must be addressed to the Network Service Device, or the Network Service Device must be in the destination address scope of the message. That is, the Network Service Device will only receive messages addressed to itself using standard LonTalk addressing:

- By unicast addressing consisting of the domain/subnet/node Id of the Network Service Device.
- By unicast addressing using the neuron ID of the Network Service Device.
- By multicast addressing, due to group membership
- By subnet broadcasting if the Network Service Device is member of the destination subnet.
- By domain-wide broadcasts if the Network Service Device is member of the same domain.

When a bound message tag connection is used for monitoring, the binding process ensures that messages sent using the message tag will use addressing details that ensure that the destination device will actually receive and process the message. However, unbound input message tags can be used to monitor messages that meet the above criteria. For example, the LNS application could monitor all messages that are sent as a domain broadcast, without explicitly establishing a message tag connection.

If the `FilterBySource` monitoring options property is set to `False`, the monitor point will receive messages without regard to which device sent the message. To restrict messages received by a message monitor point to those sent by the device specified as the `targetDevice` element when the monitor point was created, make sure that you set the `FilterBySource` property is set to True.

Every LONWORKS message contains a 1-byte message code. Only messages with the following message codes can be monitored with the techniques described in this section:

- 0x00 – 0x3E: Application messages
- 0x40 – 0x4E: Foreign Frame messages
- 0x7F: Service Pin messages

Attempting to monitor messages with message codes other than the ones listed above will result in no messages being reported.

To monitor messages that begin with a specific message code, set the `FilterByCode` monitoring options property to `True`, and the `FilterCode` property to the message code to be monitored when you set the monitoring options for the monitor set.

To recap this discussion, follow these steps to monitor messages:

1. Create a monitor set, and add monitor points for the message tags you want to monitor. For more information on these tasks, see *Creating Monitor Sets* on page 194, and *Managing Monitor Sets* on page 195.
2. Connect one or more devices to the host PC, and add corresponding message monitor points to the monitor set as described in the *Adding Message Monitor Points to a Monitor Set* section on page 197. In the case where a remote device propagates an explicitly addressed message with an appropriate destination address, no such connection will be required.
3. Open and enable the monitor set, as described in *Opening and Enabling Monitor Sets* on page 213.
4. Use the `OnMsgMonitorPointUpdateEvent` event to monitor the message monitor points. An example event handler for the `OnMsgMonitorPointUpdateEvent` event is shown below.

Receiving Message Monitor Point Updates

Once you have opened and enabled your message monitor points, your application will begin receiving `OnMsgMonitorPointUpdateEvent` events each time a message is sent to the monitored devices via your message tags. The `OnMsgMonitorPointUpdateEvent` event returns the message monitor point that caused the event to be fired, and a `SourceAddress` object identifying the device containing the message point. It also returns two `DataPoint` objects: `inputDataPoint` and `outputDataPoint`. `inputDataPoint` contains the information that was received from the monitored point. If the incoming message is a request message, you can use the `outputDataPoint` parameter to return a response.

Note that Echelon discourages using the `SourceAddress` object to identify the source of a message. It is more efficient to set the `FilterByAddress` property of the message tag to `True`, and rely on the message tag to identify the sending device.

Example Message Monitor Point Event Handler

The following code example shows how the `OnMsgMonitorPointUpdateEvent` could be used to return data to an external device that requested such data using the request/response messaging service This event handler checks the name of the message monitor point to determine which device sent the message, and if the message is a request it then sends a response to the request.

Note that by using the `OnMsgMonitorPointUpdateEvent` event as described in these sections, you will receive the events through the LNS Object Server. If your development environment supports multi-threading, you can use the `ILcaMsgMonitorPointListener` interface to receive the events directly from the message monitor point. For more information on this, see *Tracking Monitor Point Updates* on page 228.

```
Private Sub lcaObjectServer_OnMsgMonitorPointUpdateEvent( _
            ByVal msgMonitorPoint As Object, _
            UpdateType As Integer, _ByVal inputDataPoint As Object, _
            ByVal outputDataPoint as Object, _
            ByVal srcAddress As Object)
      If msgMonitorPoint.Name = sensorMsgMonPoint _
      AND UpdateType = lcaMonitorEventTypeMsgRequest Then
            outputDataPoint.Value = myResponseData
      End If
End Sub
```

Controlling Message Points

You can send application messages to a device using `MsgMonitorPoint` objects. To obtain a `MsgMonitorPoint` to use to send a single application message to a device, call the `AppDevice` object's `GetMessagePoint()` method. This returns a `MsgMonitorPoint` object that you can use to send application messages to the device. You can do so by writing to the `MsgMonitorPoint` object's `OutputDataPoint` or `RequestDataPoint` properties.

You can use the `OutputDataPoint` property to send a message that does not require a response to the device. To send an application message to a device using this property, set the `DataPoint` object's `Value` property to the value that you want to pass to the device. All `DataPoint` objects returned by the `OutputDataPoint` property have their `AutoWrite` property set to `True`, so the data will be automatically written to the device.

You can use the `RequestDataPoint` property to send a message that requires a response to a device. The response can be requested as a synchronous response, or as an asynchronous `OnMsgMonitorPointUpdateEvent` event that is triggered when the response is received. To send the request message, set the `DataPoint` object's `Value` property to the desired value. To request a synchronous response, call the `MsgMonitorPoint` object's `SendMsgWait()` method. In this case, the response will be returned by this method as a `DataPoint` object. To request an asynchronous response, call the `DataPoint` object's `Write()` method. The response message will be returned via the `OnMsgMonitorPointUpdateEvent` event.

As of LNS Turbo Edition you may send an application message to multiple devices using group or broadcast addressing. To do this you must first define a dynamic message tag on the `AppDevice` object that represents your application's Network Service Device. Once you have done so, connect this message tag to input message tags on all destination devices. Then, create a permanent message monitor point on the Network Service Device that specifies the new dynamic message tag as the monitor target. These tasks are described in the *Adding Message Monitor Points to a Monitor Set* section earlier in this chapter. Once you have performed them, you can then open the monitor set, and use the message monitor point to send application messages from the Network Service Device to the devices that are connected to the dynamic message tag.

Developing Remote Monitor and Control Applications

Monitor and control applications often need to run on a different PC than the PC running the LNS Server, and sometimes need to be able to function when the LNS Server is not attached to the network, or becomes unavailable due to a loss of power or other failure. To support this, LNS allows Independent client applications to open networks in server-independent mode. Independent client applications do not require the LNS Server to be

connected to the PC running the application. Independent client applications read and write data directly to and from the devices on the network without writing to the LNS database.

Temporary `MonitorSets` objects are not available to Independent client applications, and neither are the methods and properties of the `AppDevice` object. However, Independent client applications have access to all permanent monitor sets and monitor points (note that the LNS Server must be present in order to create monitor points).

This means that an Independent client application on a PC with monitor sets defined in its `CurrentMonitorSets` collection can open those monitor sets and use them for monitor and control operations without being attached to the LNS Server. Independent client applications can configure existing monitor points and sets using the `CurrentOptions` (but not the `NvOptions` or `MsgOptions` properties of a `MonitorSet`, or the `DefaultOptions` property of an individual monitor point), and enable the monitor points, but no other network functionality is available.

To open a network in independent mode and perform monitor and control operations, the remote PC must have previously accessed the network as a Full client in server-dependent mode, and created one or more monitor sets using the `MonitorSets` collection accessed through the `MyVNI` property. After this, an Independent client application on the PC can open the network and the monitor points can be monitored and controlled.

For more information on Independent clients and server-independent mode, see *Independent Clients* on page 41. For more information on the differences between the `MonitorSets` collections accessed through the `MyVNI` and `CurrentMonitorSets` properties, see *Permanent Monitor Sets* on page 192.

Tracking Monitor Point Updates

This chapter describes several ways you can use the `OnMsgMonitorPointUpdateEvent` and `OnNvMonitorPointUpdateEvent` events to track updates to your monitor points. You will receive these event updates through the LNS Object Server. Alternatively, you can receive these events directly through the monitor points using the `ILcaMsgMonitorPointListener` and `ILcaNvMonitorPointListener` interfaces.

Using these interfaces results in improved LNS application performance, and provides update and error events in a separate thread. The application remains responsive, and all updates and error logging occurs in the background. This technique can only be used if your development environment supports multi-threading (such as Visual C++). You cannot use this technique with Visual Basic 6.0.

For more information on multi-threading and LNS, see *Multi-Threading and LNS Applications* on page 318.

To use the `ILcaMsgMonitorPointListener` interface, follow these steps:

1. Create a COM object that implements the `ILcaMsgMonitorPointListener` interface.
2. Define the object's behavior when the `UpdateErrorEvent()` and `UpdateEvent()` methods are called. These methods will be called each time the monitor point you select in step 3 is updated.

3. Call the `Advise()` method on the `MsgMonitorPoint` object you want to monitor. Specify the COM object created in step 1 as the `object` element. At this point, LNS will stop generating the `OnMsgMonitorPointErrorEvent` and `OnMsgMonitorPointUpdateEvent` events for the monitor point, and the object specified as the `object` element will start receiving `UpdateErrorEvent` and `UpdateEvent` events for the monitor point. The `Advise()` method must be called from the thread that is managing the `OnMsgMonitorPointErrorEvent` and `OnMsgMonitorPointUpdateEvent` events.

   NOTE: The `OnMsgMonitorPointEvent` event will be fired by the LNS Object Server for each message monitor point contained in a monitor set when the motor set is opened to signal that the message monitor point is available for use. Similarly, the `OnMsgMonitorPointEvent` event will be fired each time a message monitor point is created in a monitor set that is currently open, acknowledging that the message monitor point has been instantiated. You should wait until this event has been fired to retrieve a `MsgMonitorPoint` object and call the `Advise()` method on it.

4. Each time the `UpdateErrorEvent` or `UpdateEvent` events are received, the `UpdateErrorEvent()` or `UpdateEvent()` methods will be called.

5. You can call the `Unadvise()` method on the `MsgMonitorPoint` object at any time to return to the default behavior, where events are invoked in the client thread.

To use the `ILcaNvMonitorPointListener` interface, follow these steps:

1. Create a COM object that implements the `ILcaNvMonitorPointListener` interface.

2. Define the object's behavior when the `UpdateErrorEvent()` and `UpdateEvent()` methods are called. These methods will be called each time the monitor point you select in step 3 is updated.

3. Call the `Advise()` method on the network variable monitor point you want to monitor. Specify the COM object created in step 1 as the `object` element. At this point, LNS will stop generating the `OnNvMonitorPointErrorEvent` and `OnNvMonitorPointUpdateEvent` events for the monitor point, and the object specified as the `object` element will start receiving `UpdateErrorEvent` and `UpdateEvent` events for the monitor point. The `Advise()` method must be called from the thread that is managing the `OnNvMonitorPointErrorEvent` and `OnNvMonitorPointUpdateEvent` events.

   NOTE: The `OnNvMonitorPointEvent` event will be fired by the LNS Object Server for each network variable monitor point contained in a monitor set when the motor set is opened to signal that the network variable monitor point is available for use. Similarly, the `OnNvMonitorPointEvent` event will be fired each time a network variable monitor point is created in a monitor set that is currently open, acknowledging that the network variable monitor point has been instantiated. You should wait until this event has been fired to retrieve an `NvMonitorPoint` object and call the `Advise()` method on it.

4. Each time the `UpdateErrorEvent` or `UpdateEvent` events are received, the `UpdateErrorEvent()` or `UpdateEvent()` methods will be called.

5. You can call the `Unadvise()` method on the network variable monitor point at any time to return to the default behavior, where events are invoked in the client thread.

System Management Mode Considerations

The system management mode is stored in the `MgmtMode` property of the `System` object. This determines whether device configuration changes are propagated to the devices (`lcaMgmtModePropagateConfigUpdates`), or saved for later processing (`lcaMgmtModeDeferConfigUpdates`).

You should be aware of the effects that the system management mode has on monitor and control operations. Changing the database while the system management mode is set to `lcaMgmtModeDeferConfigUpdates` mode affects monitoring operations in different ways. This depends on whether you are using permanent `MonitorSet` objects, or temporary `MonitorSet` objects, and whether you are using bound or unbound monitor and control strategies.

Temporary monitor sets use the LNS database to determine how to monitor a device. Therefore, use of temporary monitor sets while the physical configuration of the network differs from configuration stored in the LNS database is discouraged, since these inconsistencies may lead to monitoring failures, or even invalid data. As a result, Echelon recommends that you do not use temporary monitor sets while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`.

Permanent monitor sets use configuration data in devices that is updated only when the management mode is `lcaMgmtModePropagateConfigUpdates`. This means that any monitor sets and points created prior to setting the management mode to `lcaMgmtModeDeferConfigUpdates` will contain configuration data that is consistent

with the configuration currently stored in the devices, even if database changes have been made subsequently while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. The configuration of the permanent monitor sets will not be updated to reflect the database changes until the management mode is changed to `lcaMgmtModePropagateConfigUpdates`, at which point the devices will also be updated. Thus, barring any update failures, the configuration of the monitor sets will always be consistent with the physical devices on the network. As a result, you can always use permanent monitor sets, regardless of the system management mode. However, Echelon recommends that you use permanent monitor sets while the system management mode is set to `lcaMgmtModePropagateConfigUpdates` whenever possible.

You should also note that monitor points created or modified while in `lcaMgmtModeDeferConfigUpdates` mode cannot be used until the system is set to `lcaMgmtModePropagateConfigUpdates` mode. In addition, all connection changes made while in `lcaMgmtModeDeferConfigUpdates` have no effect until the network is placed in `lcaMgmtModePropagateConfigUpdates` mode.

Similarly, the implicit or explicit creation of bound connections causes device connection changes that will not be propagated to the network while the system management mode is set to `lcaMgmtModeDeferConfigUpdates`. As a result, your monitoring application might fail to receive data, or it might have to resort to polling data. Echelon recommends that you do not initiate bound monitor and control operations when the system management mode is set to `lcaMgmtModeDeferConfigUpdates`.

Directly Reading and Writing Network Variables

At some point, you may want to read or write the value of a single network variable, without monitoring its value before or after the operation. In this case, you should write to the network variable directly, without creating a network variable monitor point. Remember that if you plan on consistently monitoring and controlling a network variable or group of network variables, monitor sets are the most efficient method to use.

Prior to LNS Turbo Edition0, the `NetworkVariable` object's `Value` property was provided for single read and write operations to network variables. However, as of LNS Turbo Edition0, the `Value` property is deprecated and only supported for backwards compatibility. You can now use the `GetDataPoint()` method to create `DataPoint` objects to read and write to the values of network variables. When you do so, your application will have sole access to that data point, and it will manage the format of the data contained in the network variable locally. As a result, it will avoid misinterpreting any formatting changes that may be made to a network variable's value by other client applications.

To acquire a `DataPoint` object you can use to read or write to the value of a network variable, follow these steps:

1. Acquire the `AppDevice` object containing the network variable you want to modify.

   ```
   Set MyAppDevices = MySubsystem.AppDevices
   Set MyAppDevice = MyAppDevices.Item("Device 1")
   ```

2. Obtain the device's `NetworkVariables` collection, and then get the network variable you want to read or write.

```
Set MyInterface = MyAppDevice.Interface
Set MyVariables = MyInterface.NetworkVariables
Set MyNV = MyVariables.Item("Input One")
```

3. Use the `GetDataPoint()` method to create a `DataPoint` object for the network variable. The `GetDataPoint()` method takes a single input parameter that must be set to 0.

   ```
   Set MyDataPoint = MyNV.GetDataPoint(0)
   ```

4. To set the value of the network variable through the `DataPoint` object, set the `Value, RawValue,` or `FormattedValue` properties, and then call the `Write()` method. You do not need to call `Write()` if the `DataPoint` object's `AutoWrite` property is set to `True`.

   ```
   MyDataPoint.Value = 1000
   MyDataPoint.Write()
   ```

5. To read the value of the network variable through the `DataPoint` object, call the `Read()` method and then read the value via the `Value, RawValue` or `FormattedValue` properties. You do not need to call `Read()` if the `AutoRead` property is set to `True`.

   ```
   MyDataPoint.Read()
   Dim Value as String
   Value = MyDataPoint.FormattedValue
   ```

   When reading the value of a `DataPoint` object, there are many ways for an application to determine how the data will be displayed. For more information on this, see *Data Formatting* on page 241. Consult the *LNS Object Server Reference* help file for a complete list of the properties and methods of the `DataPoint` object.

The type of data that is stored in a network variable is determined by its base type. As of LNS Turbo Edition, you can modify a network variable's type by writing to its `TypeSpec` property, as long as the network variable is on a device that supports changeable types. For more information on this, see *Changeable Network Variable Types* on page 185.

Data Points and Enumerated Types

When reading or writing network variables and configuration properties with `DataPoint` objects, you should note that some standard and user-defined types are defined as enumerated data types in the resource files. A network variable or configuration property that uses an enumerated type will only accept values that belong to the enumeration referenced by the type it is using.

For example, the SNVT_defr_term type is based on the "defrost_term_t" enumeration type. As defined in the standard LonMark resource files, the defrost_term_t enumeration type contains values between -1 and 100. However, not all the values in this range are associated with the type's enumeration members. Values from 9 to 99 are illegal, since there are no enumeration members associated with them. Each legal value is assigned an enumeration name. If you have a network variable or configuration property that is using an enumeration data type, you can only write the enumeration names defined for that type to the object's value.

Consider a case where you have acquired a data point for a network variable using the SNVT_defr_term type. In this situation, you will only be able to write enumeration

names defined for the SNVT_defr_term type, such as DFT_TERM_LAST, to the FormattedValue property of the DataPoint. You should also note that this is the only property that enforces range limits (i.e. the Value and RawValue properties do not enforce these restrictions). You can use the Value property of the DataPoint to obtain the numeric value associated with the enumerator, as shown in the following code sample:

```
Dim MyFormatSpec as LcaFormatSpec
Dim MyDataPoint as LcaDataPoint

Set MyFormatSpec = MyDataPoint.FormatSpec
MyFormatSpec.FormatType = lcaFormatTypeNamed
MyFormatSpec.FormatName = "SNVT_defr_term"
Set MyDataPoint.FormatSpec = MyFormatSpec

MyDataPoint.FormattedValue = DFT_TERM_LAST
```

After executing the code shown above, the Value property would return 3 - the numeric value that corresponds to the DFT_TERM_LAST enumeration. Remember that LNS will not perform range checking on the Value property, and no exception will be thrown if you try to write an illegal value to the property.

If you write an illegal value to the Value property at this point (i.e. a value that does not correspond to one of the valid enumeration names for the type) and then read the FormattedValue property, no exception will be thrown. However, the value will not be converted to an enumeration name since there is no enumeration associated with the value. However, if you attempt to write the illegal value to the FormattedValue property at this point, an exception will be thrown.

You could expand the previous example to retrieve all the valid enumeration names that can be assigned to a data point by adding the sample code shown below. You will need to use the MinValue and MaxValue properties to perform this task. These properties contain the maximum numeric value (i.e. the value stored in the Value property) that can be applied to the data point. This code checks every value between the data point's minimum and maximum values. If the data stored in the Value property does not match the data stored in the FormattedValue property, it indicates that the FormattedValue property represents one of the valid enumeration names:

```
Dim i as Long
For i = MyDataPoint.MinValue To MyDataPoint.MaxValue
    MyDataPoint.Value = i
    IF MyDataPoint.FormattedValue <> MyDataPoint.Value THEN
        cboEnumList.AddItem(MyDataPoint.FormattedValue)
    END IF
 Next
```

Using Configuration Properties In a Monitor and Control Application

This section provides guidelines and instructions to follow when creating an LNS application to monitor and control configuration property values. Configuration property access is generally less efficient than network variable access, so for performance reasons, you should only use LNS to monitor and control the values of configuration

properties when it is absolutely necessary. For an overview of configuration property access performance, see *Performance Considerations* on page 240.

Generally, you should perform configuration property management using the `ConfigProperty` objects stored in the LNS database, rather than by writing to the configuration property value directly via network variable writes. If a configuration property is implemented as a configuration network variable and is then modified via a network variable write, the value of the configuration property on the physical device will be updated. However, the value stored in the LNS database will not be updated. Then, if the `DownloadConfigProperties()` method is called, the unsynchronized values in the LNS database will be downloaded to the device, and the changes made to the physical device will be lost. As a result, you need to make sure that the values of the configuration properties on the devices on your network are synchronized with the values stored in the LNS database for those configuration properties.

To facilitate this, LNS Turbo Edition features the ability to access configuration properties via `DataPoint` objects. To do so, the LNS application must first call the `ConfigProperty` object's `GetDataPoint()` method. The `GetDataPoint()` method takes an `options` element, which you can use to determine how the value of the configuration property in the LNS database and in the physical device will be affected by changes made to the `DataPoint` object. The following sections describe how you can use the `GetDataPoint()` method.

NOTE: Prior to LNS Turbo Edition0, the `ConfigProperty` object's `Value` property was provided for single read and write operations to configuration properties. Note the `Value` property is deprecated and only supported for backwards compatibility.

Device-Specific Configuration Properties

Some configuration properties are designed to be modifiable by multiple distributed sources, instead of solely by a central network manager. They are designated as such by the device-specific flag. The value of a device-specific configuration property should generally be read from the device itself, and not from the LNS database. However, LNS does not enforce this restriction. The LNS application must decide whether to use the device-specific configuration property in the device, or in the database. You can use LNS to determine if a configuration property is device-specific by reading its `DeviceSpecificAttribute` property. If a configuration property is device-specific, the `DeviceSpecificAttribute` property will be set to `True`.

By default, the values of device-specific configuration properties will be read from the device. The values of non-device specific configuration properties will be read from the database if the values are available there, and from the device if they are not. You can set the `SourceOptions` property to `lcaDataSourceOptionsfromDevice` at any time to read configuration property values from the device rather than the database, or to `lcaDataSourceOptionsDatabaseOnly` to specify that the values must be read from the database, independently of the device-specific flag.

To provide full support of device-specific configuration properties, the `UploadConfigProperties()` and `DownloadConfigProperties()` methods have options to that allow you to upload or download device-specific configuration properties separately from non-device specific configuration properties. These options are the `lcaConfigPropOptExcludeDeviceSpecific` and `lcaConfigPropOptOnlyDeviceSpecific` flags. They can be combined with the other download/upload flags. For example, to download non-device specific configuration

property values to the device, call the `DownloadConfigProperties()` method with the `lcaConfigPropOptLoadValues` and `lcaConfigPropOptExcludeDeviceSpecific` options set.

For more information on the `UploadConfigProperties()` and `DownloadConfigProperties()` methods, see *Downloading and Uploading Configuration Properties* on page 126.

Using the GetDataPoint Method

You can use a `DataPoint` object to read and write to the value of a configuration property. Each data point has three properties you can use to read and write to the data point's value: the `FormattedValue` property, the `RawValue` property, and the `Value` property. Each of these properties represents the same value, but each one is formatted differently:

- The `Value` property formats the data as a scaled, double float value. The configuration property's type file must be available in order to read the `Value` property.
- The `FormattedValue` property formats the data based on the LNS Object Server's `CurrentFormatLocale` setting. The data stored in the `FormattedValue` property is scaled and unit-converted based on the `FormatSpec` the data point is using. The configuration property's type and formats file must be available in order to read the `FormattedValue` property. For more information on `FormatSpec` objects, see *Data Formatting* in page 241.
- The `RawValue` property formats the data as a raw byte array.

Depending on how you set the `options` element when you call `GetDataPoint()` to create a `DataPoint` object for a configuration property, LNS may read or write the LNS database, the configuration property on the physical device, or both when you read or write the value of the `DataPoint`. This is described in more detail later in this section. To acquire a `DataPoint` to use to read and write the value of a configuration property, follow these steps:

1. Access the `ConfigProperty` you want to monitor and control. In this example, the configuration property "cpCalibration" applies to the device as a whole. Some configuration properties apply to a LonMark Functional Block or network variable, in which case they are accessed via the `ConfigProperties` collection of the parent `LonMarkObject` or `NetworkVariable`.

   ```
   Set MyInterface = myAppDevice.Interface
   Set MyConfigProperties = MyInterface.ConfigProperties
   Set MyCP = MyConfigProperties.Item("cpCalibration")
   ```

2. Call the `GetDataPoint()` method to create a `DataPoint` object for the `ConfigProperty`.

   ```
   Set MyDataPoint=MyCP.GetDataPoint(0,lcaDataSourceOptionsNormal)
   ```

   Some configuration properties are arrays of elements. In this case, you need to create a separate data point to read and write to each element in the array. The first parameter passed to the `GetDataPoint()` method is the `index` parameter. This specifies which element of the array you want this data point to apply to. These arrays are 0-based. If the

ConfigProperty is not an array, specify 0 as the index parameter.

The second parameter is the options element. This determines how LNS will apply changes you make to the DataPoint object to the LNS database, and to the physical device on the network. For more information on this, see the next section, *Data Source Options*.

Note that you can change both of these options later by modifying the DataPoint object's SourceIndex or SourceOptions properties.

3. To set the value of the configuration property through the DataPoint object, set the Value, RawValue or FormattedValue property, and then call the Write() method. Note that you do not need to call Write() if the AutoWrite property is set to True.

```
MyDataPoint.Value = 1000
MyDataPoint.Write()
```

NOTE: The Value, RawValue and FormattedValue properties of DataPoint objects acquired through configuration properties that use enumerated data types have special behavior that you should be aware of. For more information, see *Data Points and Enumerated Types* on page 232.

4. To read the value of the configuration property through the DataPoint object, call the Read() method and then read the value through the Value or FormattedValue properties. You do not need to call Read() if the AutoRead property is set to True.

```
Dim Value as String
MyDataPoint.Read()
value = MyDataPoint.FormattedValue
```

When reading a DataPoint object's value, there are many ways for the application to affect how the data will be displayed. For more information on this, see *Data Formatting* on page 241.

NOTE: Data points are very useful when reading and writing formatted data, or when accessing an entire raw data value as a whole. However, if you want to access arbitrary bytes of raw data to read or write a range of elements in a configuration property array, you should not use a data point. Instead, you should use the GetRawValuesEx() and SetRawValuesEx() methods of the ConfigProperty object. See the *LNS Object Server Reference* help file for more information on these methods.

Data Source Options

When you call the GetDataPoint() method, you will use the options parameter to specify how LNS will manage differences that may exist between the value of the ConfigProperty in the LNS database and the value of the configuration property in the physical device on the network when you read or write to the DataPoint. Table 9.4 describes the values you can apply to the options element.

Note that the value you choose as the options parameter is stored in the DataPoint object's DataSourceOptions property. If desired, you can change this setting later.

Table 9.4 Data Source Options

Value	Description
`lcaDataSourceOptionsNormal`	If you use this option to create a `DataPoint`, or if you set the `SourceOptions` property of the `DataPoint` to this value later, then the value of the `ConfigProperty` will be updated in the LNS database and in the physical device on the network each time you write to the value of the `DataPoint`. When you read the value of the `DataPoint`, the value will be read directly from the device if the source `ConfigProperty` is device-specific. If the source `ConfigProperty` is not device-specific, the value will be read from the database. If its value does not exist in the database, then the value will read directly from the device, as long as the system management is set to `lcaMgmtModePropagateConfigUpdates.` If the source `ConfigProperty` is not device-specific, the value is not in the database, and the system management mode is set to `lcaMgmtModeDeferConfigUpdates`, then the `NS#113 lcaErrNsCpValueNotFound` exception will be thrown when you read the value of the `DataPoint`.
`lcaDataSourceOptionsFromDevice`	If you use this option to create a `DataPoint`, or if you set the `SourceOptions` property of the `DataPoint` to this value later, then the data point is read-only. Reading the value of this data point will always read the value of the source configuration property on the physical device on the network.
`lcaDataSourceOptionsDatabaseOnly`	If you use this option to create a `DataPoint`, or if you set the `SourceOptions` property of the `DataPoint` to this value later, then the value of the data point will always be read from the LNS database. When you write to the data point, the new value will only be written to the `ConfigProperty` object in the LNS database, and not to the configuration property in the physical device. Writing to a `DataPoint` with the `SourceOptions` property set to this value is recommended only when updating the database with a value that has just been read from the device.

Value	Description
lcaDataSourceOptionsTypeDefaultValue	If you use this option to create a DataPoint, or if you set the SourceOptions property of the DataPoint object to this value later, then the DataPoint will always return the default value of configuration properties using the same type as the source configuration property. The default value is generally read from the functional profile template on the device containing the configuration property, or from the type definition for this configuration property type. Data points created with this option set are read-only. Note that this value represents the "type default" defined in the type resource file. The default value of a given configuration property may differ from the default value of its type, since the default configuration property values for a given template are defined in the external interface file and can be set from the current values in the device.

When you create a DataPoint object with the lcaSourceDataOptionNormal option described in Table 9.4, LNS will update the value stored in the database for the source ConfigProperty object and schedule an update to the physical device each time you update the value of the DataPoint object. This keeps the values stored in the LNS database and the physical device on the network in synchronization, and is the recommended way to update configuration property values.

However, in some cases, the values of the configuration properties in the LNS database may become out-of-sync with the values of the configuration properties stored in the devices on your network. The next section describes how you can resolve these situations.

Resynchronizing Configuration Property Values

Assuming you know when the configuration properties on a device in your network are updated, and the LNS database is not updated as well, you can use the UploadConfigProperties() and DownloadConfigProperties() methods to resynchronize the values stored in the LNS database with the new values of the configuration properties in the device (and vice versa). This is described in *Downloading and Uploading Configuration Properties* on page 126.

The DownloadConfigProperties() and UploadConfigProperties() generally affect all the configuration properties on a device. You can also use DataPoint objects to synchronize a single configuration property value in the LNS database with the value of the configuration property on the physical device. To do so, follow these steps:

1. Create a DataPoint object for the configuration property you want to update. Specify lcaDataSourceOptionsfromDevice as the options element when you call GetDataPoint(). The value returned for such a data point will always match the value of the source configuration property in the physical device.

   ```
   Set MyDataPoint = MyCP.GetDataPoint(0, _
          lcaDataSourceOptionsfromDevice)
   ```

2. Call the Read() method to read the value of a data point created with this option set.

   ```
   MyDataPoint.Read()
   ```

3. Set the DataPoint object's DataSourceOptions property to lcaDataSourceOptionsDatabaseOnly. Now, when you write to the data point, the value will be written to the ConfigProperty object in the LNS database.

   ```
   MyDataPoint.DataSourceOptions =lcaDataSourceOptionsDatabaseOnly
   ```

4. Call the Write() method. The value of the source ConfigProperty in the LNS database will then be updated to match the value of the configuration property on the physical device.

   ```
   MyDataPoint.Write()
   ```

Determining When Values Are Out-Of-Sync

If you need to keep the configuration property values displayed by your application up-to-date at all times, you can use the OnCommission event to do so. When a configuration property is modified via an LNS application such as a device plug-in, the OnCommission event will be generated, and the device's CommissionStatus property will be set to lcaCommissionUpdatesPending. Once the configuration property value has been propagated to the device, another OnCommission will be generated and the device's CommissionStatus property will be set to lcaCommissionUpdatesCurrent.

You should note that this event can be generated for a variety of other reasons, such as the creation of network variable connections, and does not necessarily indicate that a configuration property value has been modified. You should also note that the OnCommission event is only generated when the value of the CommissionStatus property changes. Thus, if the device's CommissionStatus property is already set to lcaCommissionUpdatesPending, no OnCommission event will be generated until the value is propagated to the device.

You can also read the ValueStatus property of a ConfigProperty object to determine if there are any pending updates to the configuration property value stored on the physical device. If the configuration property value stored in the database has been modified by an LNS application but has not yet been written to the device, the ValueStatus property will be set to lcaConfigPropertyValueMgmtStatusPendingUpdate.

Performance Considerations

The performance of the monitoring application when reading configuration properties from the database depends on disk performance. When reading configuration property values from the device, rather than from the LNS database, the performance of monitoring configuration properties varies depending on the implementation method.

When configuration properties are implemented within configuration files, the device may provide one of the following three access methods:

- Direct memory read/write.
- The LonTalk file transfer protocol (FTP), with random and sequential access.
- LonTalk FTP with sequential access.

The direct memory read/write is the preferred method for applications running on Neuron-chips or Smart Transceivers, as long as the configuration file fits within an area of directly addressable memory space of the Neuron Chip or Smart Transceiver. This is the most efficient method when reading or writing individual configuration property values. A device implementing direct memory read/write access will have an output network variable of type `SNVT_address`, and will not have a network variable of type `SNVT_file_request` or `SNVT_file_status`.

The LonTalk file transfer protocol is an interoperable way for devices to share data files with one another. The file types 0, 1, and 2 are defined by the LonMark program for specifying configuration parameters. For more information on LonTalk FTP, see the *File Transfer* LONWORKS engineering bulletin, which can be downloaded from:

http://www.echelon.com/support/documentation/bulletin/005-0025-01D.pdf

LonTalk FTP with random and sequential access method requires an implementation of an FTP server on the device, and can be used with any host processor. It can also be used for a Neuron Chip or Smart Transceiver hosted device, if serial non-volatile memory is needed for configuration property storage, the configuration property storage requirements exceed the capacity of directly addressable memory, or if additional data not related to configuration properties is also stored in files on that device. The random access version of the LonTalk FTP protocol allows you to access configuration properties individually. In addition, random access LonTalk FTP requires the device to be online when you read the configuration properties, while directory memory read/write does not. Random access LonTalk FTP also requires a network variable on the device that is dedicated to controlling the position of the file pointer (SNVT_file_pos). LonTalk FTP is the most efficient method to use when reading or writing all of the configuration property values in a device at once, or when reading or writing configuration property values stored in large contiguous blocks within the file.

LonTalk FTP without support for random access is considerably less efficient than the other two access methods, as it does not allow individual configuration property access, and a full transfer of all configuration property values is required for each modified value.

A device implementing one of the two FTP methods will have an output network variables of type `SNVT_file_status`, and an input network variable of type `SNVT_file_req`. A network variable of type `SNVT_file_pos` is used to control the position of the read/write pointer in a file used for random access. Therefore, a device with a `SNVT_file_status` and a `SNVT_file_req` network variable, but without a `SNVT_file_pos` network variable, only implements sequential access.

If a configuration property is implemented using a configuration network variable, the monitoring performance for that configuration property would be similar to the monitoring of an input network variable.

When you write an application to monitor configuration property values, you should take the configuration property access method into account, as the time required to access the configuration properties on your network will impact the overall performance of your application. And although you need to consider these implications, you should remember that LNS supports all methods to access a configuration property transparently. Regardless of whether a configuration property is implemented as a configuration network variable or as a configuration property in files, or whether the device implements an FTP server with or without random access, your LNS application will maintain the configuration property using the same techniques described in this section.

Data Formatting

When displaying the values of `NvMonitorPoint`, `MsgMonitorPoint`, and `DataPoint` objects with your monitor and control application, you should generally use the `FormattedValue` property. The `FormattedValue` property is a variant type that can contain a wide variety of formatted types. This section describes the factors that affect what data will be stored in the `FormattedValue` property, and how your application will display the data stored in the `FormattedValue` property.

FormatSpec Property

The `NvMonitorPoint` and `DataPoint` objects both contain a `FormatSpec` property. This property contains a `FormatSpec` object. Similarly, `MsgMonitorPoint` objects contain an `InputFormatSpec` and an `OutputFormatSpec` property, each of which contains a `FormatSpec` object (one for incoming messages, the other for outgoing messages). Each `FormatSpec` object specifies a base type from the device resource files. The base type determines the type of data that is stored in the monitor point or data point.

All applications can use the Standard Network Variable Types (SNVTs) and Standard Configuration Property Types (SCPTs) contained in the `Standard.TYP` file, which is available on the LonMark website at http://www.lonmark.org/. The data types covered by SNVTs and SCPTs include most standard data types that are used in control networks (temperature, heat, luminosity, pressure, etc). Manufacturers may also create User Network Variable Types (UNVTs) and User Configuration Property Types (UCPTs), which define data types that are specific to a device or devices created by the manufacturer. The standard resource files are automatically installed with every version of LNS, and you can obtain updates to the standard device resource files online at http://www.lonmark.org. User-defined device resource files are normally installed as part of the installation of device-specific plug-in software. When in doubt, contact your device manufacturer for details.

As described in the *Device Interfaces* section in Chapter 6 of this document, each set of resource files must be associated with a particular program ID, a range of program IDs, or with all program IDs to associate it with a network variable, configuration property, or `LonMarkObject` on a device. The type of association is called the scope of the resource file, and the scope is specified using a scope selector. The scope selector for a resource file specifies what part or parts of a device's program ID should be used when selecting the

resource file. See Chapter 6 for more detailed information on program IDs and scope selectors.

The FormatSpec object contains a Scope property, a ProgramId property, an Index property, a FormatName property, and a FormatType property. To use these properties to change the base type of a monitor point or data point, follow these steps:

1. Access the FormatSpec object for the monitor point or data point whose base type you want to change.

   ```
   Dim MyFormatSpec as LcaFormatSpec
   Set MyFormatSpec = MyNvMonitorPoint.FormatSpec
   ```

2. Set the Scope and ProgramId properties to reference the device resource files containing the definition of the base type you want to use.

   ```
   MyFormatSpec.Scope = lcaResourceScopeMfg
   MyFormatSpec.ProgramId = "8000010000000000"
   ```

3. Set the Index property if you want to identify the new type by its index value within the resource file. Or, set the FormatType and FormatName properties if you want to identify the type by its name. Note that the FormatType property must be set to lcaFormatTypeNamed if you want to identify the type by its name.

   ```
   MyFormatSpec.FormatType = lcaFormatTypeNamed
   MyFormatSpec.FormatName = "UNVT_color"
   ```

   NOTE: If you pass an invalid type name to the FormatName property, and then pass the FormatSpec object back to a data point or monitor point, no exception will be thrown, although the FormatSpec object will not reference a valid type. The LNS Object Server will use the previous setting of the FormatName property to determine the data point or monitor point's type. You should note that if you read the FormatSpec object in this situation, the FormatName property will still return the invalid type name.

4. The FormatSpec object is not passed by reference. So, you must pass the modified FormatSpec object back to the data point or monitor point for the changes made in steps 2 and 3 to take effect.

   ```
   Set MyNvMonitorPoint.FormatSpec = MyFormatSpec
   ```

Reading the FormatSpec Object

The FormatSpec object contains several properties you can use to determine how the data stored in your data points or monitor points will be displayed. For example, you can use the UnitsAdder and UnitsMultiplier properties to determine how the scaled value of the data point or monitor point will be unit-converted for display as a formatted value when your application reads the FormattedValue property. These factors can also be used to determine how the formatted value units differ from the base units associated with the data point type.

Table 9.5 introduces the other properties of the FormatSpec object. These properties are read-only, and are set based on the type the FormatSpec object represents. See the *LNS Object Server Online Reference* for more detailed information on these properties.

Table 9.5 `FormatSpec` Object Properties

Property	Description
`AltFormatName`	This property contains an indexed list of all formats that can be applied to the type used by the network variable or data point. For example, in a `FormatSpec` object representing the SNVT_temp_f#US format (i.e. degrees Fahrenheit), this property would contain “SNVT_temp_f#US”, "SNVT_temp_f#SI", and "SNVT_temp_f#US_diff".
`AltFormatNameCount`	This property indicates how many alternate formats are contained in the `AltFormatName` property.
`Precision`	This represents the number of digits that will be used when data is read and displayed using the format specification if the data being displayed is a floating-point type. This property has a range of 0-17. If the data type used by the configuration property or format specification is a single float type, this property defaults to the value of the `FloatPrecision` property of the `FormatLocale` your application is using. If the data type used by the configuration property or format specification is a double float type, this property defaults to the value of the `DoubleFloatPrecision` property of the `FormatLocale` your application is using.
`Units`	This property returns a string describing the units of values using this format.
`UnitsMultiplier`	This property indicates the value by which the scaled value of the data point or monitor point should be multiplied when it is unit-converted for display as a formatted value (via the `FormattedValue` property).

Property	Description
UnitsAdder	This property indicates the value that should be added to the raw value of the data point or monitor point when it is unit-converted for display as a formatted value (via the FormattedValue property). This addition is performed after the raw value is multiplied based on the UnitsMultiplier property.

CurrentFormatLocale

As described in the previous section, the FormatSpec object assigned to a data point or monitor point determines the base type of the data stored in the object. For data points, you can customize how this data will be displayed using FormatLocale objects.

Each FormatLocale object contains a series of properties that reflect a particular geographical area's conventions for data display. These conventions affect how data should be displayed in that area, including factors such as language, measurement system (U.S. or Systeme Internationale), date formats, time formats, and decimal number formats. The settings of a FormatLocale object determine how data accessed through the FormattedValue properties of all DataPoint objects will be displayed when your application uses that FormatLocale object.

Each client application can select which FormatLocale object it will use for a given session by passing a selected FormatLocale object to the CurrentFormatLocale property of the ObjectServer object before opening any networks and formatting any data. Operations that will cause your application to format data include acquiring DataPoint objects, and reading or writing the values of configuration properties and network variables on your network.

By specifying their own FormatLocale object, client applications in different regions can use their own sets of local or absolute formats when displaying data, without affecting the data formatting used by other clients. This greatly reduces the risk of your application returning information that is confusing or misleading due to formatting changes made by another application.

You can access the FormatLocales collection through the FormatLocales property of the ObjectServer object. Initially, the FormatLocales collection contains 4 pre-defined FormatLocale objects:

1. UserDefaultRegionalSettings. This is the default value for the CurrentFormatLocale property. When you use this FormatLocale object, all the properties will be set based on the user-defined Windows regional settings for the user currently logged onto the PC running your application. You can change the regional settings on a PC using the Windows control panel Regional Options applet. Consult the Microsoft Developer's Network (MSDN) documentation of the Win32 GetLocaleInfo() function for more information on the Windows regional settings.
2. SystemDefaultRegionalSettings. When you use this FormatLocale object, all the properties will be set based on the default Windows regional settings on

the PC running your application. The default settings may vary, depending on which operating system is installed on the PC. Consult the MSDN documentation of the Win32 `GetLocaleInfo()` function for more information on the Windows Regional settings.

3. `LonMarkCompatibility`. When you use this `FormatLocale` object, all properties will be set so that formatted data will be displayed based on the LonMark standards used prior to LNS 3.0, when localized formatting was not available. In this case, Systeme Internationale measurement units, LonMark-defined time and date formats, and U.S. options for everything else will be used to display all formatted data.

4. `ISO8601DateAndTime`. When you use this `FormatLocale` object, all properties will be set to be the same as the `LonMarkCompatibility` settings, except for the localized time and date formats, which will be based on the ISO 8601 standard. This standard helps avoid confusion that may be caused by the different national notations used for dates and times, and increases the portability of computer user interfaces.

Creating FormatLocale Objects

The four pre-defined `FormatLocale` objects are read-only, but you can create custom `FormatLocale` objects to suit the specific needs of your application with the `Add()` method of the `FormatLocales` collection. Note that all custom `FormatLocale` objects are instantiated with the same settings as the pre-defined `UserDefaultRegionalSettings FormatLocale` object described above.

To create a custom `FormatLocale` object, and assign it as the `CurrentFormatLocale`, follow these steps. Note that you can only create and configure your custom `FormatLocale` objects, and set the `CurrentFormatLocale` property, before opening any networks and formatting any data with your application. Operations that will cause your application to format data include acquiring a `DataPoint` object, and reading or writing the value of a `ConfigProperty` or `NetworkVariable` object. If you write to the `CurrentFormatLocale` property after performing any of these operations, or attempt to modify the `FormatLocale` object acting as the `CurrentFormatLocale`, the `LCA:#122 lcaErrReadOnlyInContext` exception will be thrown.

1. Open the Object Server as described in Chapter 4 of this document.

2. Access the `FormatLocales` collection.

```
Dim MyFormatLocales as LcaFormatLocales
Set MyFormatLocales = ObjectServer.FormatLocales
```

3. Invoke the `Add()` method to create a new `FormatLocale` object.

```
Dim MyFormatLocale as LcaFormatLocale
Set MyFormatLocale = MyFormatLocales.Add(“myNewFL”)
```

4. Set the properties of the new `FormatLocale` object to match your application’s requirements. These properties are described in Table 9.6.

5. Assign the new `FormatLocale` object to the `CurrentFormatLocale` property.

```
Set ObjectServer.CurrentFormatLocale = MyFormatLocale
```

Table 9.6 lists the various properties of the `FormatLocale` object, and describes how they affect the display of data stored in the `FormattedValue` property. For more detailed information on each property, consult the *LNS Object Server Reference* help file.

Table 9.6 FormatLocale Objects

Property	Description
`CategoryPreferenceList`	You can use the `CategoryPreferenceList` property to establish the format to use when displaying the formatted value of a data point whose base type (as specified by its `FormatSpec` object) contains several alternate formats.
`DateFormatSeparator` `DateFormatSeparatorSource`	The `DateFormatSeparator` property determines what symbol will be used to separate the digits that represent months, days and years when the formatted value is displayed as a date. This applies to format specifications containing the `date()` macro in their text format specification, such as SNVT_date_cal#LO: `text(date(year, month, day))`
`DecimalPointCharacter` `DecimalPointCharacterSource`	The `DecimalPointCharacter` property determines what symbol will be used to indicate decimal places when a formatted value is displayed as a scalar number. This applies to format specifications that use the %f symbol in their text format specification, such as SNVT_temp#US: text("%f", *1.8+32(0:855))
`DoubleFloatPrecision`	The `DoubleFloatPrecision` property determines the default precision that will be used when displaying double-float values.
`FallbackFormat`	The `FallbackFormat` property specifies the default type that should be used to display a network variable data point's formatted value if the data point's actual type cannot be determined.
`FloatPrecision`	The `FloatPrecision` property determines the default precision that will be used when displaying single-float values.
`LanguageId`	The `LanguageId` property determines the language that will be used to display Windows localized settings.

Property	Description
ListSeparatorCharacter ListSeparatorCharacterSource	The ListSeparatorCharacter property determines what character will be used to separate items in the formatted value that are returned as parts of a list. This applies to format specifications that specify the use of a locale-specific separator character, e.g. SCPTsetPnts#SI_LO: `text("%f|%f|%f|%f|%f|%f", occupied_cool, standby_cool, unoccupied_cool, occupied_heat, standby_heat, unoccupied_heat)`
MeasurementUnits MeasurementUnitsSource	The MeasurementUnits property determines the measurement units (Systeme Internationale or U.S.) that will be used to display the formatted values.
ShortDateFormat ShortDateFormatSource	The ShortDateFormat property determines how a formatted value will be displayed if it represents a date. This applies to format specifications containing the `date()` macro in their text format specification, such as SNVT_date_cal#LO: `text(date(year, month, day))`
ShortTimeFormat ShortTimeFormatSource	The ShortTimeFormat property determines how the formatted value will be displayed if it represents a time. This applies to format specifications containing the `time()` macro in their text format specification, such as SNVT_date_time#LO: `text(time(hour, minute, second))`
TimeFormatSeparator TimeFormatSeparatorSource	The TimeFormatSeparator property determines what symbol will be used to separate digits representing hours, minutes and seconds when a formatted value is displayed as a time. This applies to format specifications containing the `time()` macro in their text format specification, such as SNVT_date_time#LO.

NOTE: You can use the various *Source properties to determine whether the value of the property should be manually entered by the application, or whether LNS will set the property automatically based on the system default or user-defined Windows Regional Settings on the PC running your application. For example, the ShortTimeFormatSource property determines how the ShortTimeFormat property should be filled in. These *Source properties default to the `lcaFormatLocaleSourceSystemDefaultRegionalSetting (1)` value, which causes the corresponding property to be initially set based on the system-default Windows Regional Settings. If you change the value of the corresponding property, LNS will automatically update the *Source property to the

`lcaFormatLocaleSourceManualSetting(2)` value. Consult the *LNS Object Server Reference* for more information on these properties.

Chapter 10 - LNS Database Management

This chapter provides information you will need when managing your LNS databases. This includes topics such as validating network databases, performing database back-ups, moving databases, and recovering network databases.

Overview of LNS Databases

As described in Chapter 3, the LNS Server maintains two types of databases: the *LNS global database* and a set of *LNS network databases*. These are high-performance disk-based databases with in-memory caching to optimize repeated access to data.

The LNS global database contains the `Networks` collection, meaning that it contains the names and locations of each of the network databases. The location of the global database is maintained in the Windows Registry, and can be accessed using the `DatabasePath` property of the `ObjectServer` object. This is set to the `ObjectServer\GlobalDb` subfolder of the LONWORKS folder by default. A backup, empty copy of the global database is available in the `ObjectServer\BackupDb` folder. The location of the global database should be set when LNS is installed and then never changed, as LNS applications must access the same global database if they are to interoperate.

Each network defined in the global database and managed by the LNS Server has its own network database. The network database contains the network and device configuration information for that specific LONWORKS network. This chapter provides guidelines you should follow to maintain your LNS databases. This includes the following topics:

- *Automatic Database Upgrade*
- *Backing Up Network Databases*
- *Validating Network Databases*
- *Removing Network Databases*
- *Moving Network Databases*
- *Network Recovery*

Automatic Database Upgrade

If you are upgrading to LNS Turbo Edition from a previous version of LNS, you do not need to worry about compatibility issues between your LNS databases and the software installed with LNS Turbo Edition. All LNS databases since Release 1.0 are compatible with LNS Turbo Edition, and are automatically upgraded when they are opened by an application running on LNS Turbo Edition.

This conversion is a multi-step process that could take a significant amount of time, as global database and all the network databases must all be upgraded when they are first opened with an LNS application that is running on LNS Turbo Edition. You can use the `OnDbConversionEvent` event to track the progress of this procedure. This event is triggered once for each stage of the conversion, and once when the conversion is complete. See the *LNS Object Server Reference* help file for more information on the `OnDbConversionEvent` event.

NOTE: LNS databases are not backward compatible. Once a database is upgraded, the upgrade changes cannot be reversed, and the database cannot be accessed using pre-Turbo Edition versions of LNS.

Backing Up Network Databases

The ability to backup and restore an LNS database is a critical requirement for most systems. LNS provides a network recovery feature you can use to recover a lost database from the network, which is described later in this chapter. However, it is much more reliable to use an archived copy to restore a lost database, or a database that has become corrupted. Different factors can cause database corruption. For example, some hard disk controllers have a write-caching option to improve write performance. Use of this option is not recommended with database management software such as LNS, because a power failure during operation may result in database corruption. You should note that most Windows operating systems enable write-caching by default.

As described earlier in this chapter, the LNS Object Server maintains a network database for each network defined in the global database. The location of the global database is defined by the `DatabasePath` of the `ObjectServer` object, and is set to the `LonWorks\ObjectServer\GlobalDb` directory by default. The location of each network database is defined when each network is created, and you can determine this location later by reading the `DatabasePath` property of the `Network` object.

To backup a global database or a network database, copy the entire contents of the global or network database directory, including all subdirectories. You can reduce the size of the backup by using an archiving utility such as PKZIP to compress and archive the files. Be sure to specify the appropriate options to include subdirectories in the archive.

You can restore both the global and network databases, or just the network database. You can also restore databases to the same PC or a different PC. If you restore both databases, the global database must be restored to the PCs default global database directory, and each network database must be restored to its original location as defined in the global database. Note that if you move an LNS database from one PC to another, there are other files you will need to move as well. For more information on this, see *Moving Network Databases* on page 256.

If you are restoring an existing network database, you should first delete any existing files in the database folder and sub-folders, and then copy the backup files into the folder. **Make sure the network is closed when you do so. This will avoid leaving any "stale" files in the restored database directory.** If you are restoring a network database that is not already on the PC, but is already referenced in the global database, restore it to its original location. To restore a network database that is not already in the global database, restore the database into a new directory, then call the `Add()` method on the `Networks` collection with the `CreateDatabase` parameter set to `False`. This tells the LNS Object Server that this is an existing database that must be registered in the global database. Networks can only be imported by local applications. The following code imports a network named "N1" with database path "`c:\N1`".

```
Dim MyNetwork as LcaNetwork
Set MyNetwork = MyNetworks.Add("N1", "c:\N1", False)
```

Backup Method

To facilitate the process of backing up your network databases, the `Backup()` method has been added to the `Network` object for Turbo Edition. You can call this method on any local network to make a backup copy of the network database, and export the backup copy to a local directory of your choice. The following code sample backs up a network database to the `C:\BackupDBs\LNSNetworkDatabase` folder.

```
MyNetwork.Backup("C:\MyBackupDBs\" & MyNetwork.Name)
```

You can use this method to backup a network database at any time, including while the network is open and clients are attached to it. However, if a remote Full client application accesses the `Networks` collection while the database is being backed up, that network will not appear in the `Networks` collection. In addition, if a remote Full client application attempts to open a network while it is being backed-up, the operation may fail, whereas Local and Lightweight client applications will simply wait for the backup to complete in this case. LNS calls made by client applications already connected to the database when a backup is initiated may not return until the backup is complete, and requests to modify the database will be suspended until the backup is complete.

Echelon recommends that you use this method to backup the network database before validating a network database with the `Validate()` method described in the next section. You can then archive the backed-up database after it is validated.

Validating Network Databases

As described in Chapter 2, *What's New In Turbo Edition,* LNS Turbo Edition features the ability to perform database validations. There are two ways to perform a database validation: to use the LNS Database Validation Tool, or to use the `Validate()` method of the `Network` object.

LNS Database Validation Tool

You can use the LNS Database Validation Tool to validate any of the network databases on a given PC. The tool will process the contents of the selected network database, and report any errors or inconsistencies it discovers while doing so. It can optionally repair some of these errors as part of the validation process. Inconsistencies and errors that may be discovered during the database validation procedure include orphan objects (objects that cannot be accessed through their parent object), broken interfaces, and duplicate objects. You can access the LNS Database Validation Tool by selecting **LNS Database Validator** from the Echelon LNS Utilities group in the Windows Programs menu. Figure 10.1 shows the LNS Database Validation Tool.

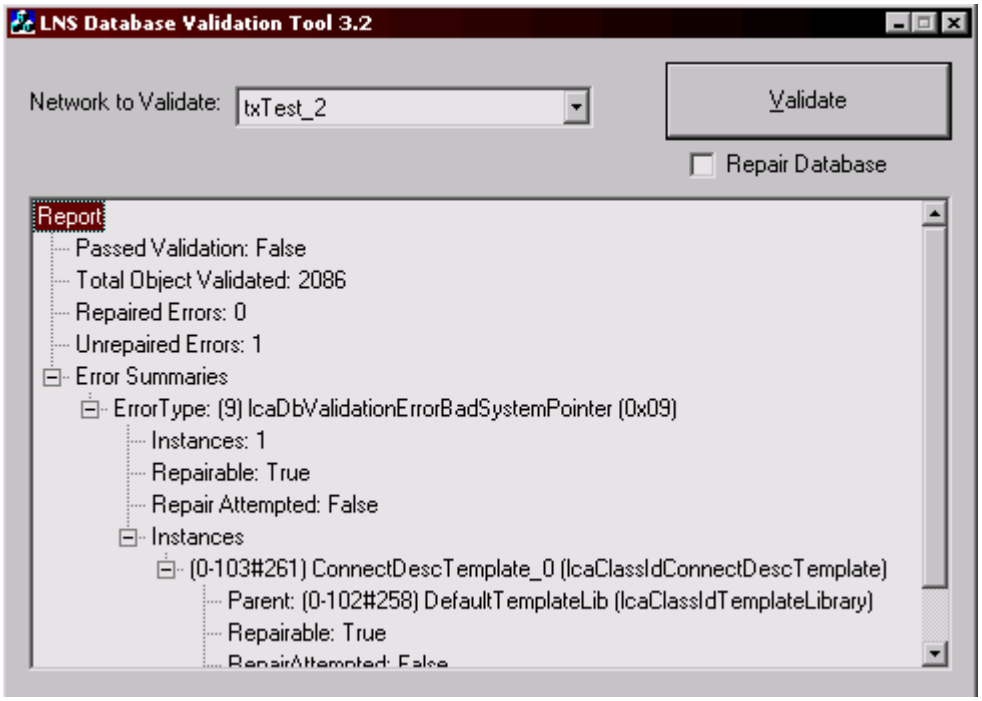


Figure 10.1 LNS Database Validation Tool

To start a database validation, follow these steps:

1. Open the LNS Database Validation Tool and select the network you want to validate from the **Network to Validate:** pull-down list.
2. Optionally click the **Repair Database** check-box if you want LNS to attempt to repair the errors it finds.
3. Click the **Validate** button to start the network validation. Status and result of the validation will appear in the scroll box at the bottom of the dialog.

Consult the online help for the LNS Database Validation Tool for more detailed information on how to use the utility.

Validate Method

You can also use your own LNS application to perform database validations. The `Validate()` method has been added to the `Network` object for Turbo Edition. When you call `Validate()` on a network, LNS will perform a database validation on its network database. The method provides options you can use to specify whether or not LNS will attempt to repair any of the errors it finds.

When the validation has completed, the method returns a `DatabaseValidationReport` object that contains information summarizing the results of the validation. This is the same set of data that would be returned if you used the LNS Database Validation Tool to validate the network database, including descriptions of all the errors it encountered, all the errors that were repaired as part of the validation procedure, and all the errors that were not repaired as part of the validation procedure.

Table 10.1 lists the properties of the DatabaseValidationReport object. See the *LNS Object Server Reference* help file for more details on the DatabaseValidationReport object and its properties.

Table 10.1 DatabaseValidationReport Object

Property	Description
PassedValidation	This property will be set to True if the network passed the database validation. This will occur if no errors were discovered during the validation, or if all the errors discovered during the validation were repaired. This property will be set to False if the network database has any problems that were not repaired during the validation. In this case, you can use the ErrorSummaries property to examine the problems that were not repaired.
ErrorSummaries	This property contains the DatabaseValidationErrorSummaries collection returned by the database validation. This collection contains a group of DatabaseValidationErrorSummary objects, each of which contains properties identifying and describing an error that was encountered, and not repaired, during the database validation. You can use this collection to troubleshoot any problems that still exist in the database after performing a validation.
RepairedErrors	This property contains the number of errors that were successfully repaired by LNS during the database validation.
UnrepairedErrors	This property contains the number of errors that were not repaired by LNS during the database validation.
TotalObjectsValidated	This property contains the number of database objects that LNS validated during the database validation procedure.

Special Considerations

When validating a network database, there are several things you should consider. Depending on the size of the network database, it may take a considerable amount of time to complete the database validation. You can use the OnDbValidationEvent event, or the ProgressUpdate() method of the ILcaProgressListener interface, to check the status of an ongoing database validation.

You can cancel a validation by invoking the CancelValidation() method on the Network object. In this case, the method will not return a valid DatabaseValidationReport object.

The database validation must be initiated locally. While the database validation is in progress, clients will be unable to modify or write to the database. As a result, Echelon

recommends that you perform the database validation while a minimal number of client applications are connected to the database. One suggested approach is to backup the database and restore it with a different name and location, and perform the validation on this restored database. This approach will minimize the disruption caused by the validation, and has the benefit of producing a backup whose validity is known.

While the new LNS database validation feature described in the previous section can be used to repair many of the errors that may occur in your network databases, you should not consider this feature a replacement for consistently backing up your network databases. Echelon strongly recommends regularly backing up and validating the LNS database, to ensure that there is always a recent, valid database backup to fall back upon in the event of database or file system failure or recovery from user error. See the *Backing Up Network Databases* section earlier in this chapter for guidelines on this.

Using the CompactDb() Method

You can also use the CompactDb() method to help maintain your network databases. When you call `CompactDb()` on the `ObjectServer` object, LNS will defragment and reindex the global database. When you call `CompactDb()` on a `Network` object, LNS will defragment and reindex the network database. Defragmenting and reindexing a database might result in smaller database size, and might improve performance of LNS applications that access that database. If database errors are generated when you open a network, it may indicate there are indexing problems on the network database that can be resolved via the `CompactDb()` method.

You should not perform these operations on a database that is open and in use by any client application. In addition, you should backup all databases before calling this method and your PC should have at least twice as much free disc space as the size of the database when you call this method.

Removing Network Databases

To remove a network database that is no longer needed, and to recover all LNS Device Credits that are currently consumed by that network, you need to delete the network from the LNS global database. To do so, follow these steps:

1. Open the system using a valid network interface. Set the system management mode to `lcaMgmtModePropagateConfigUpdates`. This will allow you to recover the LNS Device Credits used to install the network. You will lose the LNS Device Credit used to install any application device that is still commissioned when the network is deleted.

   For more information on LNS Device Credits, see Chapter 13.

   ```
   MySystem.MgmtMode=lcaMgmtModePropagateConfigUpdates
   ```

2. Call the `Decommission()` method on each `AppDevice` and `Router` object in your network (except those `AppDevice` objects that represent Network Service Devices).

   You should decommission all applicable `AppDevice` objects first, and then decommission all `Router` objects. This prevents decommissioned routers from blocking access to application devices that have not yet been decommissioned.

3. Close the system. This detaches the Object Server from the network and shuts down the LNS Object Server.

   ```
   MySystem.Close()
   ```

4. Close the network. This closes the network database.

   ```
   MyNetwork.Close()
   ```

5. Call `Remove()`on the `Networks` collection to delete the network. The `Remove()` method deletes the network database directory, the network database, and the reference to the network within the global database:

   ```
   MyNetworks.Remove(MyNetwork.Name)
   ```

Moving Network Databases

Sometimes, you may need to move a network database from the PC it was created and maintained with. This may be because the LNS Server PC is due for a hardware upgrade, or the integration work continues with the installation phase on site when installing a network using the engineered system installation scenario, or a network database needs to be moved into a data archive for retrieval at a future time.

Moving a LNS network database is a 2-step process. First, you need to remove the database from its original location, and then you need to connect to that database from its new location.

To remove the links to the network database without interrupting operation on the network itself, and without destroying the network database itself, follow these steps:

1. Close the related System and Network objects, if open:

   ```
   MySystem.Close()
   MyNetwork.Close()
   ```

2. Call `RemoveEx()`on the `Networks` collection to delete the network. When you use the `RemoveEx()` method, you can specify whether the network database is to be deleted. If you do not delete the network database, you can restore the network later, without having to re-create the database. The following code removes the network, but leaves the database files intact for use on a different location.

   ```
   MyNetworks.RemoveEx(MyNetwork.Name, -
         lcaNetworkRemovalFlagLeaveFiles)
   ```

   NOTE: In this scenario, you will not decommission any devices in the network before removing the network from the `Networks` collection, as you will want the network to be functional when you finish moving the database and restore it.

The next step is to move the network database to its new PC. When you move a network database, you should include the entire contents of the network database directory, including all subdirectories. You should also include any files from other directories that are referenced in the database such as any download image files (.APB extension), external interface files (.XIF, .XFB, and .XFO extensions), LonMark Resource files (.TYP, .FPT and .FMT extensions), any language resource files with extensions such as .ENG, .ENU, .FRA, etc), any user-defined resource files, icon files (.ICO extension), bitmap files

(.BMP extension), and source files (.NC, .C and .H extensions). It is also a good idea to reinstall plug-in software that applies to device types that are used within this network with your data. Echelon recommends that you backup all these files before performing the move. **NOTE:** Figure 6.2 in Chapter 6 of this document depicts where the external interface files on a PC are stored.

Once you have moved the database to the new PC, you can restore access to the network database on another LNS Server PC by following these steps:

1. Restore the database files into a suitable location on the LNS Server PC's hard drive, e.g. in a folder called "`C:\MyNetwork`".

2. Restore all the required support files to their respective locations. You will need to add any resource file sets that have been moved to the resource file catalog. Any other files referenced in the network database must be placed in the same absolute path location on the new PC. Alternatively, you can update the path property (e.g. `BitmapFilePath` or `IconFilePath` property) referencing the file.

3. Call the `Networks` collection's `Add()` method. You will provide a name for the imported database as the `networkName` element, and a reference to its location as the `databasePath` element. Set the `createDatabase` element to `False`. This adds a reference to the imported database to the global database, without destroying any of its content.

   ```
   Dim MyNetwork As LcaNetwork
   Set MyNetwork = MyNetworks.Add("Zorro Ltd", _
         "C:\MyNetwork", False)
   ```

Network Recovery

The LNS Object Server includes a set of services that allow it to rebuild a network database by scanning an installed network. This process, known as *network recovery*, can be used to:

- Migrate from a LonManager API-based tool to a LNS-based tool.
- Migrate from an NSS-10 managed network to an LNS Object Server managed network.
- Migrate from an NSS-based tool to an LNS-based tool.
- Recover a network database when no backup is available.

Recovery must be initiated by a Local client application. Even though you can use the network recovery feature to rebuild an entire network database, you should not consider it to be a replacement for consistently backing up your LNS databases, as described in the *Backing Up Network Databases* section on page 251. Database backup has these advantages:

- Copying a database directory to restore from a backup is much faster than performing a network recovery. To recover a network database, the LNS Object Server must scan the network to discover all the devices, upload the piece of the overall network configuration that it stored in each device, and then deduce the overall configuration of the network from these pieces. This data collection and reconstruction process grows roughly linearly with the complexity of the network, where complexity is

a function of the number of devices, network variables, channels, and connections.

- Network recovery only recovers attributes and properties stored in the devices on the network. LNS-specific information such as subsystem names and device names is not recovered. In addition, the recovery process will not create an exact duplicate of the original database. If you compare a recovered database to the original database, there will be differences, as there are many objects, properties and attributes that cannot be identified uniquely and unambiguously by the recovery process.

  For example, the handles assigned to devices and routers might differ, and connection hubs and targets may differ if you were to compare the original database with the recovered database. A network recovery may also be incomplete if the network itself is inconsistent. A network could become inconsistent if the network tool failed while updating the configuration of a series of devices. In this case, some devices would contain "new" information while other devices contain "old" information. When recovering this network, there is no way for the LNS Object Server to determine which information is out-of-date.

  For more information on inconsistencies that may exist between the original database and the database created by the network recovery process, see the next section, *Network Recovery Inconsistencies*.
- Network recovery cannot recover network variables, monitor sets or monitor points defined on a Network Service Device, or any connections involving a Network Service Device.
- Restoring a network database from a backup is more reliable. Successful recovery of a database from a given network relies on that network being properly configured, and not saturated with regular network traffic when the recovery is performed. Additionally, authentication can prevent parts of a network from being recovered.

Network Recovery Inconsistencies

It is possible that some network inconsistencies will be created while recovering a network database. Before performing a network recovery, you should review the following list of inconsistencies and limitations of network recovery:

- Recovery of networks with unsupported configurations is not possible. Examples of unsupported network configurations include networks with multiple authentication keys, or networks with multiple domains.
- If a bound network variable has no associated source or target, it is marked as unbound. No address is associated with this network variable. This could occur if only one of two devices that take part in a connection is recovered successfully.
- If a bound declared message tag has no targets, its address table entry is marked as empty.
- Any address table entries that are not associated with a network variable or message tag (source or target) are marked as empty. A lost source or a lost target may create this situation. Group use counts are updated as necessary, and group IDs are freed as necessary.

- The recovery process has no way of determining what network variables are hubs, and what network variables are targets, in the connections on the recovered network. These relationships are arbitrarily assigned.

  Because the hub/target relationships may change, the recovery may not restore the proper connection descriptions to the connections on the recovered network. If the recovery is unable to determine the correct connection description for a connection, it uses the default connection description.
- In many cases, information stored in the network database will be lost, and LNS will make its best guess based on the information it does recover on how to fill in the lost information during the network recovery. This includes most properties of the `System` object.
- If LNS cannot communicate with a device, the device will not be recovered. This will create inconsistencies in any connections that the device was part of. LNS will recover those connections to the extent possible, leaving out the missing device. LNS takes the following actions when this sort of inconsistency is detected:
  - If a bound network variable has no associated source or target, it is marked as unbound. No address is associated with this network variable. The connection is lost.
  - If a bound dedicated message tag has no targets, its address table entry is marked as empty. The connection is lost.
  - Any address table entries that are not associated with a network variable or message tag (source or target) are marked as empty. A lost source or a lost target may create this situation. Group use counts are updated as necessary, and group IDs are freed as necessary.
- If the configuration of a device in the recovered network is inconsistent, then LNS will not recover its connections properly, possibly effecting other connections in ways that were not intended.
- The configuration of your Network Service Device will be not be recovered during this process, as the configuration of the device, including the network variable configuration, network variable selector values, and most of the Network Service Device's address table entries, are stored on the PC containing the network database. As a result, monitor sets on the Network Service Device, and all connections involving the Network Service Device, will not be recovered.
- If you have modified the attributes of the network image of any of the devices in your network outside of LNS, including the channel ID of any of the devices, the LNS Object Server may be unable locate that device on the network during recovery. This may cause the network recovery to fail.
- Any information specific to LNS but without correspondence on the physical network cannot be recovered. This includes user-defined data (`Extension` objects), subsystem assignment, and registration information for plug-in software (`ComponentApp` objects).

For all of these reasons, you should use database backup as the primary means of preventing database loss. In addition, you should examine the recovered database to ensure its stability and consistency after performing a network recovery.

Performing a Network Recovery

To recover a network database, you can use the LNS Database Recovery Wizard. The wizard takes your preferences on several guided pages, runs and controls the recovery process, and formats the resulting database by evaluating device resource files, device interface files, and device interface data. You can access the LNS Database Recovery Wizard by selecting **Start>Programs>Echelon LNS Utilities>LNS Database Recovery Wizard**.

To recover a network database with your own LNS application, follow these steps:

1. Create a new `Network` object using the `Add()` method for the `Networks` collection.

   ```
   Dim RecoveredNet as LcaNetwork
   Set RecoveredNet = MyNetworks.Add("Recvrd","c:\DB", True)
   ```

2. Open the new network.

   ```
   RecoveredNet.Open()
   ```

3. Open the system, and set the `NetworkInterface` object contained in the `System` object's `NetworkServiceDevice` object to identify the network interface for the new network.

   ```
   Set RecoveredSystems = RecoveredNetwork.Systems
   Set RecoveredSystem = RecoveredSystems.Item(1)
   Set RecoveredNSD = RecoveredSystem.NetworkServiceDevice
   Set RecoveredNSD.NetworkInterface = SelectedNI
   ```

4. Call the `System` object's `PrepareToRecoverFromNetwork()` method. If objects have been added to the `Network` object created in step 1, an exception will be thrown. Set the `recoverNetInterface` parameter to `True` if you want the domain signature and authentication key to be recovered from the network interface. Note that this option is only valid when the network interface is a standard network interface whose configuration is consistent with the network. If this parameter is set to `False`, LNS will read the domain signature from the `System` object's `DomainId` property. In this case, you should make sure the `DomainId` property is set to a valid value.

   ```
   RecoveredSystem.DomainId = "32a0cf"
   RecoveredSystem.PrepareToRecoverFromNetwork(False)
   ```

5. Open the system.

   ```
   RecoveredSystem.Open()
   ```

6. If the `recoverNetworkInterface` option was set to `False` and network authentication is being used on the network being recovered, set the `System` object's `AuthenticationKey` property.

7. Optionally add objects, set properties, and call methods. This includes defining `DeviceTemplate` objects, and setting network timers or other properties of the `System` object. If you add device templates at this time, your recovered devices will have more meaningful names for their device templates, and include more accurate default settings.

   Note that you cannot add any objects that consume network resources at this point, such as `AppDevice`, `Router`, `Channel` or `Subnet` objects. This could create conflicts on the network once it is recovered.

8. Call the `RecoverFromNetwork()` method. You can use the `options` parameter to control the behavior of the recovery process. If the network being recovered is a small network, you should specify the `lcaRecoveryOptSmallNetwork` option, which tells the LNS Object Server to use a recovery algorithm optimized for small to medium-sized networks. This algorithm uses domain-wide broadcasts to find devices. The `lcaRcoveryOptSmallNetwork` option will work reliably with network containing up to 64 devices, and may work well with more devices depending on your network topology.

   If the network being recovered has a very high traffic rate, you may want to specify the `lcaRecoveryOptForceOffline` option, which tells the LNS Object Server to force all devices off-line during the discovery process. In this case, all application traffic stops and the devices will be free to listen to the recovery messages. Large networks may generate too much traffic with domain-wide broadcasts so the `lcaRecoveryOptForceOffline` network option uses subnet broadcasts, which take much longer. The devices are left in the `offline` state when recovery is complete. Note that these options may be ORed together.

   ```
   RecoveredSystem.RecoverFromNetwork(False, lcaRecoveryOptSmallNetwork)
   ```

9. After network recovery has been completed, the system management mode will be set to `lcaMgmtModeDeferConfigUpdates`. You need to set the system management mode to `lcaMgmtModePropagateConfigUpdates` before resuming normal operations. You can do so by writing to the `MgmtMode` property of the `System` object.

   ```
   RecoveredSystem.MgmtMode = lcaMgmtModePropagateConfigUpdates
   ```

   Note that you should thoroughly examine the LNS database to make sure that the database is consistent and complete before you set the system management mode to `lcaMgmtModePropagateConfigUpdates.`

10. Call the `System` object's `RestoreLicense()` method to resume the LNS Device Credit licensing model. The method takes a single parameter that indicates whether the recovered network was installed using LNS. This parameter affects whether or not LNS will charge any LNS Device Credits for the recovery. See chapter 13, *LNS Licensing,* for more information on this.

    ```
    RecoveredSystem.RestoreLicense (True)
    ```

11. Configured devices recovered by the network recovery process will be stored in the `Discovered.Installed` subsystem. Unconfigured devices will be stored in the `Discovered.Uninstalled` subsystem. Once the network recovery has completed, you can use the `AddReference()` method to add those devices to other subsystems, as described in Chapter 6 of this document.

During the recovery process, the LNS global database will be locked to prevent accidental modification while recovery is in progress. Other LNS applications should not make changes to the LNS network database while the network is being recovered.

The recovery of a large network may take a long time. You can examine the `System` object's `RecoveryStatus` property to see how a network recovery is progressing. You need to use a separate process to read the `RecoveryStatus` property. For more information on the `OnSystemNssIdle` event, see *Using the OnSystemNssIdleEvent* on page 316.

If the recovery process is interrupted, for example if the service is canceled or the LNS Server PC loses power, you can restart the recovery by invoking the `System` object's `RecoverFromNetwork()` method with the `resumeRecovery` flag set to `True`.

Application-Level Recovery

An LNS application can provide enhanced functionality to the recovery process by uploading additional information from the devices on the network, if the network was designed with this goal in mind. For example, a device may store its subsystem path and device name in a SCPT_location configuration property. You can acquire this path by reading the `Path` property of any `Subsystem` object stored in the `AppDevice` object's `Subsystems` collection. Note that the SCPT_location configuration property may not be implemented in all of your devices.

The syntax of the string should be the subsystem path name :

```
subsystem[.subsystem]
```

For example, the following string specifies an device's logical location as "Building 2", "Room 312":

```
"Building 2.Room 312"
```

To provide an enhanced application-level recovery function, follow these steps:

1. Perform the network recovery, as described in the previous section.
2. Loop through each `AppDevice` object in the `Discovered.Installed` subsystem of the recovered network. Start with the last device and end with the first device in the subsystem's `AppDevices` collection, since the following steps will remove the devices from this subsystem.

   For each device, perform these steps:

   A) Obtain the `LonMarkObjects` collection of the `AppDevice` object's main interface (accessed through the `Interface` property). Search for a `LonMarkObject` object with the `TypeIndex` property set to 0 (SFPTnodeObject). If found, obtain the `ConfigProperties` collection object from this `LonMarkObject`, and try to locate a `ConfigProperty` object with the `TypeIndex` property set to 17 (SCPT_location).

   B) If unsuccessful in the previous step, repeat the previous step, but search all LonMarkObject objects on the device, not just the node object.

   C) If unsuccessful in both previous steps, search the

ConfigProperties collection of the AppDevice for a ConfigProperty object with the TypeIndex property set to 17 (SCPT_location).

D) When you find a SCPTlocation configuration property, parse it for the device's subsystem path and name. Create the subsystem if it does not exist, and then invoke the AddReference() method to move the AppDevice to the new subsystem. Set the Name property of the AppDevice to the proper name, if it is available.

E) If a SCPTlocation configuration property was not found, move the device to a default subsystem of your choice using the AddReference() method.

3. Loop through each Router object in the Discovered.Installed subsystem of the recovered network, and use the AddReference() method move each router to a default subsystem of your choice. Start with the last router and end with the first router in the subsystem's Routers collection, since this step will delete the routers from this subsystem.

Recovery and Mirrored Connections

Mirrored connections contain segments that are mirror images of each other. For example, a connection with hub network variable A and target B is a mirror of a connection with hub B and target A. Typically, mirrored connection segments appear in pairs, created by superimposed connections in complex connections.

Sometimes, network recovery introduces mirrored connections in place of the original connections. This is a side effect of the network recovery process. When recovering a network, the LNS Object Server cannot determine which network variables were hubs and which were targets. It sees only the resultant connections. Thus, the LNS Object Server assigns hubs to connections as best it can, in an attempt to minimize hub usage. It is unlikely that the hub selection algorithm will reproduce the same hub/target set that was used in creating the network. As a result, your application should not make any assumptions as to the hub/target relationship when accessing, removing or modifying connections on a recovered network. Interoperable tools should always examine both the hubs and targets when accessing a connection on a recovered network, since the hub/target relationships on the recovered network depend on the arbitrary results of the network recovery.

Chapter 11 - LNS Network Interfaces

This chapter describes how to configure the various network interfaces you can use with LNS. It also describes the differences between standard and high performance network interfaces.

Network Interfaces Overview

Each Network Service Device contains a network interface. The Network Service Device uses the network interface to communicate with the LONWORKS network, as the network interfaces provides the physical connection between the network and the LNS Server. Each network interface uses LonTalk messaging to communicate with the LONWORKS network and the devices on the network.

This chapter contains information you will need when choosing a network interface. This includes the following sections:

- *Standard and High Performance Network Interfaces*. This section introduces the network interfaces that are commonly used with LNS. Some of these network interfaces (Layer 5 network interfaces) are considered *standard* network interfaces, and others (Layer 2 network interfaces) are considered *high performance* network interfaces. This section describes the functional differences between standard network interfaces and high performance network interfaces.
- *Using xDriver Interfaces*. This section provides details on network interfaces that use the OpenLDV xDriver software to communicate with LONWORKS networks, such as the *i*.LON 10 Ethernet Adapter and the *i*.LON 100 Internet Server. The OpenLDV xDriver software is installed with the LNS Server software during the LNS Turbo Edition installation.
- *Using LONWORKS/IP Interfaces*. You can configure a TCP/IP network card as an LNS high performance network interface, and use it to connect to a network via a LONWORKS /IP channel. In order to do so, you will need one or more LONWORKS/IP channels connected by at least one LONWORKS router, such as an *i*.LON 1000 Internet Server or an *i*.LON 600 LONWORKS/IP Server. To use a TCP/IP card as an LNS network interface, you need to create a LONWORKS/IP Interface device to represent the network card. You can do so with the LONWORKS/IP tab of the LONWORKS Interfaces application in the Windows Control Panel. You will also need to define the LNS PC as a member of the LONWORKS /IP channel. This section describes how you can do so. LONWORKS/IP Interfaces implement the ANSI/CEA-852 layer 3 routing protocol.
- *Network Interfaces and Network Service Devices*. This section describes the ways a network interface might affect the performance of your Network Service Device.

Standard and High Performance Network Interfaces

An LNS network interface consists of two parts: the network interface hardware component, and the network interface software driver. The network interfaces you can use with LNS Turbo Edition are listed below. You can find more detailed information about these network interfaces on Echelon's website at: www.echelon.com/support/documentation/Manuals/.

PCLTA-20	The PCLTA-20 is a standard PCI card. There are four versions of the card that include an onboard transceiver (TP/FT-10, TP/XF-78, TP/XF-1250 or TP-RS485), and one version that accepts a standard modular transceiver (SMX) which may be used with any media type for which an SMX transceiver exists. The PCLTA-20 supports the

	Windows plug-and-play standard. This hardware may be used as an LNS high performance network interface by selecting `PCL20VNI` from the `NI Application` field of the LONWORKS `Plug 'n Play` control panel. It can be used as a standard interface by selecting `NSIPCLTA` from the `NI Application` field.
PCLTA-21	Echelon's PCLTA-21 card is a high performance network interface for desktop and embedded personal computers equipped with a 3V and 5V 32-bit PCI interface and a compatible operating system. This network interface is designed for use in networks that require a PC to monitor, manage, or diagnose a network, and is ideal for industrial control, building automation, and process control applications. The PCLTA-21 card features support for TP/XF-78, TP/XF-1250, TP/FT-10, and RS-485 transceivers, downloadable memory, a network management interface, and Plug n' Play capability with the Microsoft Windows 2000 and Windows XP operating systems. The four versions of the PCLTA-21 interface with integral twisted pair transceivers support the TP/FT-10 (Model 74501), TP/XF-78 (Model 74502), TP/XF-1250 (Model 74503), and TP-RS485 (Model 74504) channels, respectively. This hardware may be used as an LNS high performance network interface by selecting `PCL10VNI` from the `NI Application` field of the LONWORKS `Plug 'n Play` control panel. It can be used as a standard interface by selecting `NSIPCLTA` from the `NI Application` field.
PCC-10	The PCC- 10 is a type II PC (formerly PCMCIA) card that includes an integral TP/FT- 10 transceiver. Other transceiver types can be connected to the PCC-10 via external transceiver "pods". The PCC- 10 can be used as an LNS high performance network interface by selecting `PCC10VNI` from the `NI Application` field of the LONWORKS `Plug 'n Play` control panel. It can be used as a standard interface by selecting `PCC10NSI` from the `NI Application` field.
SLTA-10	The SLTA- 10 is a serial interface with built-in twisted pair transceiver that connects to any host with an EIA-232 (formerly RS232) port. It can also connect to the host remotely using a Hayes compatible modem. You can use the SLTA- 10 for remote applications that cannot use the OpenLDV xDriver software, or for portable hosts that do not contain a type II PC card slot.

Power Line SLTA	The Power Line SLTA is an EIA-232 compatible serial device that allows any PC with an EIA-232 interface to connect to and communicate with a LONWORKS network. The Power Line SLTA can be connected to host PC through a pair of modems and a telephone network, allowing for remote operations. It can be configured to answer incoming calls from remote hosts, or to initiate calls to remote hosts.
LTS-20	The LTS-20 SLTA Core Module is a 40-pin single in-line module (SIM) that is an embeddable version of the SLTA-10. The LTS-20 is functionally equivalent to the SLTA-10. The LTS-20 requires the addition of a LONWORKS transceiver to communicate on a LONWORKS network.
i.LON 10 Ethernet Adapter	You can use the *i*.LON 10 Ethernet Adapter to connect to devices on a LONWORKS network via an IP network. You will need to use the OpenLDV xDriver software subsystem to do so. For more information on this, see *Using xDriver Interfaces* on page 271.
i.LON 100 Internet Server	You can use the *i*.LON 100 Internet Server to connect to devices on a LONWORKS network via an IP network. You will need to use the OpenLDV xDriver software to do so. The *i*.LON 100 Internet Server also contains a large set of applications you can use, allowing you to use it as a network controller and a network interface simultaneously. For more information on this, see *Using xDriver Interfaces* on page 271.

The PCLTA-20, PCLTA-21 and PCC-10 LNS network interfaces can be used as high performance network interfaces, or as standard network interfaces. There are many factors to consider when you decide which type of network interface you should use.

Using an LNS high performance network interface causes much of the LONWORKS protocol processing to be performed on the PC containing the network interface, as opposed to on the network interface hardware. This expands the capabilities of the LNS network interface, and allows you to open multiple networks simultaneously with a single network interface. When using a standard network interface, you can only access a single network at a time.

In addition, when using a high performance network interface, your application can perform up to 250 message transactions simultaneously. Standard network interfaces can only perform one transaction at a time, and so they cannot start a new message transaction while another is completing. The ability to perform multiple transactions simultaneously is highly beneficial in large networks, where you may need to poll the value of hundreds of separate network variables at a time.

NOTE: The *i*.LON 100 Internet Server supports up to 15 simultaneous transactions at a time.

The following section describes some of the other differences in behavior between standard network interfaces and high performance network interfaces.

Addressing

All devices that communicate on a LonTalk channel must have a Neuron ID and a domain/subnet/node address. When using a high performance network interface, these addresses are stored on the PC containing the network interface. Standard network interfaces store these addresses on the network interface.

When a standard network interface receives a LonTalk message, it forwards the message to the PC only if the destination address in the message matches one of the network interface's addresses. High performance network interfaces must forward all messages received by network interface to the PC, as the LonTalk protocol stack running on the PC is responsible for decoding network addresses. This means that high performance network interfaces will send many messages to the PC that are ultimately discarded by the PC. This may have performance implications, depending on the link between the network interface and the PC.

On the other hand, a standard network interface performs more processing than a high performance interface. This could also affect the overall throughput, depending on the network interface's processing speed.

Some network interfaces support making uplink calls when they receive messages addressed to the interface (*i*.LON 10 Ethernet Adapter, *i*.LON 100 Internet Server, and the SLTA-10 models). These are all standard network interfaces, since high performance interfaces do not store their address on the interface. Consult the SLTA-10 documentation for more information on the SLTA-10. For more information on the *i*.LON 10 Ethernet Adapter and the *i*.LON 100 Internet Server in this document, see *Using xDriver Interfaces* on page 271.

LonTalk Transactions

Sending a message using the request, acknowledged or unacknowledged/repeat messaging services initiates a LonTalk transaction. An acknowledged or request message transaction is completed when the acknowledgment or response to the message is received, or when the transaction times out (based on the message transaction timers and number of retries). An unacknowledged/repeat message transaction completes when all of the repeats have been sent.

High performance network interfaces support multiple outstanding transactions for subnet/node and group addressed messages. This means that LNS can send a network variable fetch request using subnet node addressing to different devices simultaneously, without waiting for responses to previously sent requests. This can substantially increase the total polling throughput. LNS utilizes a secondary source subnet/node address when sending Neuron ID/broadcast messages so that it can have a single outstanding Neuron ID or broadcast message transaction, without impacting subnet/node or group addressed messages. When using a high performance interface, LNS can support thousands of simultaneous transactions. However, LNS dynamically adjusts the maximum number of simultaneous transaction when there are signs of network congestion, such as late responses and message failures. Note that LNS does not support multiple outstanding message transactions that use the same destination address, and it does not support multiple outstanding transactions that use Neuron ID or broadcast addressing.

Most standard network interfaces can only support a single outstanding transaction at a time. This means that in order to poll 2 network variables, the first poll must be completed before the second can start. Note that unlike most other standard network interfaces, an *i*.LON 100 Internet Server can support up to 15 simultaneous subnet/node or group addressed transactions.

Number of Groups

When using a high performance network interface, a Network Service Device can be a member of up to 256 LonTalk groups, which is the LonTalk group limit per network. When using a standard network interface, a Network Service Device can be a member of no more than 15 groups, because each group must be configured in a standard network interface's address table entry, and the interface is limited to 15 address table entries.

Supporting Multiple Networks

Each Network Service Device in an LNS network must have a unique Neuron ID and domain/subnet/node address. Standard network interfaces support only a single Neuron ID, and thus can only support a single Network Service Device in a single LNS network at any given time. When using a high performance network interface, all addresses, including the Neuron ID, are stored on the PC containing the network interface. Therefore, a single high performance network interface can be used to support multiple Network Service Devices in multiple networks.

A LONWORKS/IP Interface used to access a LONWORKS /IP channel must use a unique IP address and IP port combination. This allows multiple LONWORKS/IP Interfaces to exist on a single PC with a single IP network card. This in turn allows a single PC to have a LONWORKS/IP Interface device for every LONWORKS /IP Channel it needs to access.

Neuron Ids

When operating as a standard network interface, each network interface has a unique Neuron ID, which is stored in the network interface hardware. Thus, if a Network Service Device is using a standard network interface, the Neuron ID assigned to its network interface will only change when a new network interface is installed and selected as the active network interface.

However, you should be aware that the Neuron ID for a high performance network interface is not stored in the network interface hardware. Instead, LNS generates a separate Neuron ID for each Network Service Device using the network interface, and associates that Neuron ID with the Network Service Device.

This process could result in creating duplicate Neuron IDs (duplicates of other Network Service Devices using high performance interfaces). However, the chances of generating duplicate Neuron IDs in the same network are statistically miniscule. The Neuron ID assigned to each Network Service Device using a high performance network interface is stored in the Windows Registry of the PC running the LNS application. Usually, the same Neuron ID will be used each time that Network Service Device is opened. However, the Neuron ID is not backed up, since this would produce duplicate Neuron IDs. As a result, moving a network from one PC to another (i.e. copying a network database and moving it to another machine), or performing a network recovery with a client using a high performance network interface, will result in generating a new Neuron ID for the network interface.

You should note that if a high performance network interface is replaced with another high performance network interface, the new network interface will use the same Neuron ID as the old one.

Changing the network interface's Neuron ID does not affect the operation of Local or Lightweight clients. However, if the Neuron ID assigned to a Full client's network interface has been modified, it may be necessary to use the `PreReplace()` method to replace the Network Service Device when using it to open a network. For more information on this, see *Using the PreReplace Method* on page 166.

If the Neuron ID of the active network interface on the PC containing the LNS Server has changed, a remote Full client using the `RemoteNetworks` collection may not be able to connect to the server, since the `RemoteNetworks` collection may have the old Neuron ID. In that case the remote Full client can reestablish the connection to the LNS Server by opening the `Networks` collection. For more information on using these collections with remote Full client applications, see *Initializing a Remote Full Client Application* on page 52.

Using xDriver Interfaces

You can use the OpenLDV xDriver to connect network interfaces such as the *i*.LON 10 Ethernet Adapter and the *i*.LON 100 Internet Server to a LONWORKS network via an IP network. The OpenLDV xDriver is an integral part of LNS versions 3.07 and higher, and of OpenLDV Versions 1.0 and higher.

The OpenLDV xDriver can provide authenticated and encrypted connections from an LNS server to hundreds or even thousands of remote LONWORKS networks through network interfaces like the *i*.LON 10 Ethernet Adapter and the *i*.LON 100 Internet Server. As shown in Figure 11.1, the LNS Server accesses each network interface, and the LONWORKS channels they are using, through a TCP/IP connection.

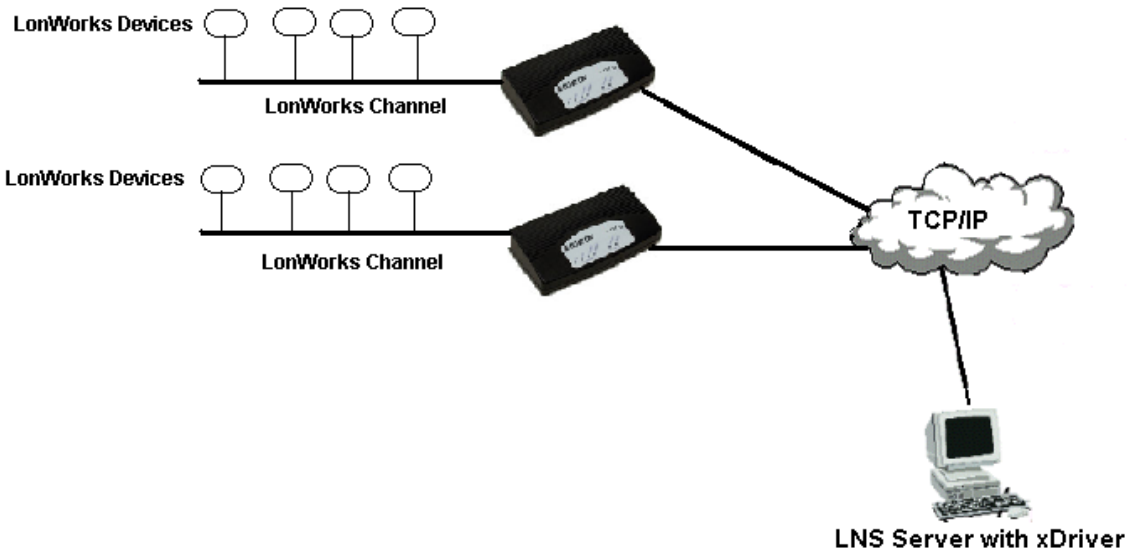


Figure 11.1 xDriver Overview

In Figure 11.1, the LNS Server is using xDriver to connect to two *i*.LON 10 Ethernet Adapters. The OpenLDV xDriver integrates with LNS at the same point as other LNS network interfaces such as the PCC-10 and the PCLTA-20.

The OpenLDV xDriver supports scalable access to many network interface devices. The default xDriver implementation uses the Windows Registry as a database to store the information it requires to connect to each network interface. For small-scale deployments, it is most efficient to use this as the xDriver database.

However, for larger deployments (defined to be more than 100 network interfaces), you should conduct a performance characterization to determine whether the performance achieved with the default Windows Registry based information store is acceptable for your application. If this performance does not meet your needs, Echelon recommends that you extend the default xDriver to use an external database of sufficient performance as your network interface information store. xDriver includes an extension mechanism that will support the external database of your choice.

If you do not plan to extend the default xDriver implementation to use an external database, you can begin using the default xDriver to configure and connect to your network interfaces, as described in Chapter 2, *Using the Default xDriver,* of the *OpenLDV Programmer's Guide, xDriver Supplement*. If you plan to extend the default xDriver implementation, see Chapter 3, *Extending the Default xDriver* of the *OpenLDV Programmer's Guide, xDriver Supplement*.

The *OpenLDV Programmer's Guide* and the *OpenLDV Programmer's Guide, xDriver Supplement* can be downloaded from Echelon's website at:

http://www.echelon.com/support/documentation/default.htm

Using LONWORKS/IP Interfaces

As described earlier in this chapter, you can use an Ethernet network card combined with the TCP/IP protocol as an LNS high performance network interface. This usually requires the use of one or more routers, such as an *i*.LON 1000 Internet Server, or an *i*.LON 600 LONWORKS/IP Server, as shown in Figure 11.2. To use an Ethernet card as an LNS network interface and route LONWORKS messages over an IP network, you should create a LONWORKS/IP Interface device with the LONWORKS/IP tab of the LONWORKS Interfaces application in the Windows Control Panel.

The IP address and port used by the LONWORKS/IP Interface must also be defined as a device on the channel containing the *i*.LON 1000 Internet Server or the *i*.LON 600 LONWORKS/IP Server in the Echelon LONWORKS/IP Configuration Server database.

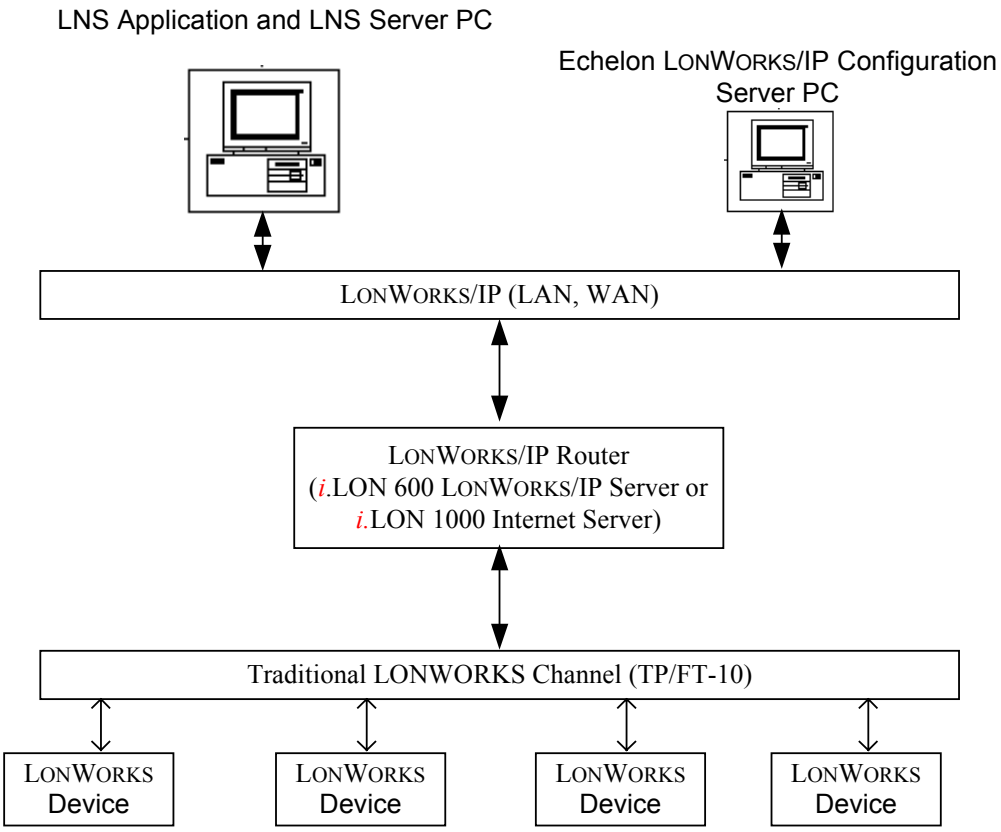


Figure 11.2 LonWorks/IP Interfaces

NOTE: LONWORKS /IP channels in LNS Turbo Edition require the use of the new Echelon LONWORKS/IP Configuration Server. As a result, if an *i*.LON 1000 Internet Server user upgrades from LNS 3.0 to LNS Turbo Edition, they must also upgrade the Configuration Server. To mitigate this requirement, the LNS Server installation for Turbo Edition includes the new Echelon LONWORKS/IP Configuration Server. LNS Turbo Edition and the new Echelon LONWORKS/IP Configuration Server support backward compatibility with the *i*.LON 1000 Internet Server by allowing the creation of, and connection to, *i*.LON 1000 Internet Server compatible LONWORKS /IP channels. If the Echelon LONWORKS/IP Configuration Server is not running on the same PC as the LNS Server installation, the Echelon LONWORKS/IP Configuration Server installer will be available to LNS customers for download on the Echelon web-site.

For detailed instructions on how to use the LONWORKS/IP tab of the LONWORKS Interfaces application, consult the application's online help. For more information on the Echelon LONWORKS/IP Configuration Server, consult the documentation for the *i*.LON 600 LONWORKS/IP Server.

Network Interfaces and Network Service Devices

As described in the *Device Interfaces* section on page 104, all application devices are assigned a device template representing their external interface (i.e.`DeviceTemplate` object) in the LNS database. Each device template has a unique program ID, and specifies a device's programmatic interface.

LNS does not include external interface files for Network Service Devices. However, it does create default `DeviceTemplate` objects for use by Network Service Devices. As described earlier in this chapter, the network interface used by a Network Service Device can affect the Network Service Device's capabilities. Some of these capabilities (e.g. maximum number of groups) affect how LNS will configure the Network Service Device. As a result, Network Service Devices with different network interfaces and programmatic capabilities use different device templates.

Therefore, switching from one network interface to another may require you to upgrade the Network Service Device to use the device template for the new network interface. Normally, LNS will perform the upgrade automatically when the system is opened. However, depending on the configuration of the Network Service Device, switching from a high performance to a standard network interface may result in dropping connections to or from the Network Service Device. Specifically, if the Network Service Device is involved in several multicast connections and uses more than 15 groups, switching to a standard network interface will result in losing some of those connections.

If you do not want your client application's Network Service Device to be upgraded automatically, you can set the `lcaFlagsManualNsdUpgrade` flag in the `Flags` property prior to opening the system. This will prevent LNS from automatically upgrading your client application's Network Service Device. This does not eliminate the need to upgrade the Network Service Device. It only keeps LNS from performing the upgrade automatically.

You can read the `UpgradeRequirement` property of the `AppDevice` object representing a Network Service Device to see if an upgrade is required after the network interface has been changed and the system has been opened. If the property indicates that an upgrade is required, you should start a transaction and perform the upgrade by calling the `Upgrade()` method on the `AppDevice` object. You should not specify a `DeviceTemplate` object when upgrading a Network Service Device.

If the upgrade causes connections to be lost, you can display this information to the user, and the user can decide whether to proceed with the upgrade and commit the transaction, or open with a different network interface (cancel the transaction, and then close the network). If the application keeps the database open without upgrading the Network Service Device, the Network Service Device will not be properly configured, which will result in communication errors during network management and monitor and control operations.

You can examine the `UpgradeStatus` property of the Network Service Device's `AppDevice` object to review the changes made to the device's external interface during the upgrade. For example, the `UpgradeInfos` property contains a collection of `UpgradeInfo` objects, each one containing the status of one external interface component on the old external interface.

The capabilities of a Network Service Device may also be affected by the version of LNS software installed. Therefore, if the `Flags` property is set to `lcaFlagsManualNsdUpgrade`, it may be necessary to upgrade the Network Service Device after installing a new version of LNS. LNS Turbo Edition supports three different types of Network Service Devices, depending on the network interface used. Each uses its own template and program ID. Table 11.1 lists these types by program ID.

Table 11.1 Network Service Devices

Program ID	Usage
90000010103800000	Used when the Network Service Device uses a high performance network interface, a LONWORKS/IP interface, or when LNS is opened in engineered mode.
90000010103800001	Used when the Network Service Device uses a standard network interface.
90000010103800002	Used when the Network Service Device uses a standard network interface that supports enhanced authentication commands, such as an *i*.LON 100 Internet Server.

Note that when a network is opened in engineered mode, the Network Service Device uses the same template as when the network is opened with a high performance network interface, or a LonTalk IP interface. Therefore, LNS will upgrade the Network Service Device when switching from engineered mode to use a standard network interface, or when switching to engineered mode while using a standard network interface. Since the template used by a Network Service Device may change with different releases of LNS, an LNS application should never be designed to expect a Network Service Device to use a particular device template, unless it will only be run in a tightly specified, closed system.

Chapter 12 - Director Applications and Plug-Ins

This chapter discusses the standards and development methodology for creating interoperable LNS director and plug-in applications.

Introduction to the LNS Plug-In Model

All LNS applications achieve a level of interoperability by using the common API provided by the LNS Server, and by using the LNS global database and shared network databases. The shared network databases allow one LNS application to add a new device to a network, and for all other LNS applications attached to the database to immediately know about the new device. This level of interoperability is achieved whether the other applications run on the same PC, on different PCs on the same LONWORKS network, or on PCs connected to the LNS Server and database via a TCP/IP connection.

An LNS network tool could be constructed from a mixture of any of the following components:

- Generic components created by tools vendors for viewing all objects in a system and invoking commands on the objects.
- Device type specific components created by device manufacturers for device configuration, monitoring, and control.
- System specific components created by system manufacturers for system configuration and management.
- Generic components created by tools vendors for alarming, logging, and trending.

In addition to providing a common API and database for these types of applications, LNS defines standard interfaces that applications can use to communicate with each other. These interfaces can be used to categorize LNS applications into the following types:

- *Director applications:* applications that call and initialize other LNS applications. Director applications use the LNS Plug-In API to invoke the other type of LNS application: plug-in applications. An example of a director application is a generic system navigator that allows the user to navigate through the subsystems in a system, select a device, and invoke a command on the device. The command may be carried out directly by the navigator, or the navigator may invoke a device plug-in to carry out the command.
- *Plug-ins:* applications that are called by director applications and are implemented as ActiveX automation servers. They typically implement their own user interfaces, and may be able to operate independently as stand-alone applications. Plug-ins are typically used for device type specific applications (called *device plug-ins*), system-specific applications, and generic applets such as alarming, logging, and trending. Most device manufacturers implement plug-ins to simplify configuration, monitoring, or control of their devices. Plug-ins also implement the LNS Plug-In API.

The interface between director applications and plug-ins consists of standard ActiveX automation interfaces. An application can operate as an LNS plug-in and as an LNS director by implementing both sides of the interface. The number of director applications per system will typically be limited to provide a unified interface to the end-user.

The techniques introduced in this chapter allow an additional level of interoperability to be achieved. They allow network tools to be built from component applications that directly interact with each other. The methods and techniques introduced in this chapter are described in more detail in the *LNS Plug-in Programmer's Guide,* which can be

downloaded from Echelon's website at http://www.echelon.com/. You should consult this manual before developing an LNS component application.

LNS Plug-In API

Director applications call plug-ins using the LNS Plug-In API. This ActiveX-based API defines an automation object that provides a standard interface between a director application and a plug-in. Director applications can launch plug-ins and communicate with them using the methods and properties of the automation object. A set of ActiveX exceptions is defined for passing back error information from the plug-in to the director.

Registering Plug-Ins

LNS plug-ins and device controls must be registered in the Windows Registry. Each plug-in also registers one or more commands in an LNS database so that director applications can find the plug-ins.

A special class of plug-in commands is provided which can register new plug-ins. This allows plug-in registration to be fully automated. The installation program for a plug-in only has to create a single entry in the Windows Registry for the plug-in. The plug-in then performs the tasks required to register all of its commands within the LNS database, and carries out any other initialization steps that it requires. This allows registration to be bootstrapped, so that the first step can occur either before or after the LNS Server has been installed.

Registering a Plug-In in the LNS Database

Plug-in commands are registered in an LNS database using `ComponentApp` objects. Each `ComponentApp` object represents a single command that is handled by the plug-in. A single plug-in can handle many commands. `ComponentApp` objects are contained within the `ComponentApps` collections. The following objects within the LNS Object Hierarchy have `ComponentApps` collections:

- `ObjectServer`
- `System`
- `DeviceTemplate`
- `LonMarkObject`

Registering a Plug-In in the Windows Registry

In accordance with Microsoft's registration requirements for COM components, all plug-ins must be registered in the Windows Registry. This includes generating a unique GUID (Globally Unique Identifier) for each component, registering the GUID and OLE name in the Windows Registry, and registering the type library information. Different development tools provide varying amounts of support for the COM registration tasks. The Visual Basic runtime generates GUIDs and type libraries automatically when a Visual Basic OLE server is run. Visual C++ and MFC provide classes and macros to assist the developer with these tasks.

Registering Plug-In Commands in the Windows Registry

Registration commands for plug-ins must also be registered in the Windows Registry. Registration commands are plug-in commands automatically invoked by a director application to register the plug-in application's commands within an LNS network database. Registration commands may also perform other initialization tasks required by a plug-in, such as creating a device template, importing an external interface file for the device template, and registering device controls with the device template. This is described in more detail in the *LNS Plug-in Programmer's Guide,* which can be downloaded from Echelon's website at http://www.echelon.com/.

Accessing Extension Data

LNS client applications may require data that is not available in the LNS database, so the applications can maintain their own databases. If they do, they must create their own scheme for sharing the data between multiple instances of themselves on a network, and must develop a mechanism for associating the data with data items in the LNS database.

LNS provides `Extension` objects to make maintaining application specific data within the LNS database simpler. The data stored in the `Extension` objects can be associated with objects within the LNS database, and LNS automatically provides a mechanism for accessing the data from multiple clients on a network. The following LNS objects include an `Extensions` collection that can be used for maintaining application specific data: `AppDevice`, `Channel`, `DeviceTemplate`, `LonMarkObject`, `Network`, `NetworkServiceDevice`, `ObjectServer`, `Router`, `Subnet`, `Subsystem` and `System`.

Implementing an LNS Director Application

An LNS director application is an application that can manage and invoke LNS plug-in applications, as described previously in this chapter. Typical LNS director applications are generic network management or monitoring tools that use plug-in software to delegate device-specific tasks such as device configuration. See the *LNS Plug-In Programmer's Guide* for more information about the LNS plug-in interface and API, and about creating LNS plug-in applications. You may also find the example director application included with LNS Turbo Edition useful when reviewing this section. For more information on the example director application in this document, see Appendix C, *LNS Turbo Edition Example Application Suite.*

This section discusses special considerations you will need to make when creating an LNS director application. An LNS director application must be able to perform the following tasks:

- Recognize newly installed or updated plug-in software, and manage the completion of the plug-in registration process.
- For any applicable operation, determine the applicable plug-in software and launch that application.
- Optionally support advanced plug-in management operations, such as temporarily disabling a plug-in, deregistering plug-ins, etc.

To accomplish these tasks, the LNS director application must have access to the Windows Registry, to the LNS Object Server hierarchy, and must implement the client-side of the LNS plug-in API.

Implementing the Client-Side LNS Plug-In API

The LNS Plug-In API is a simple COM interface that defines a list of properties and methods by name. The LNS director application can reference an LNS plug-in application using the plug-in application's registered server name, and in doing so access the plug-in application's methods and properties.

The following code example illustrates how an LNS director application written in Visual Basic can connect to a plug-in application. The plug-in is identified by its registered server name. The proper way to retrieve a plug-in application's server name is discussed later in this section.

```
Dim MyPlugIn As Object
Set MyPlugIn = CreateObject(RegisteredServerName)
```

Note the COM client variable `MyPlugIn` is declared as `Object`. It is important that the LNS director application uses late binding techniques when connecting to plug-in software. The LNS plug-in API contains the definition of methods and properties by name, but details no requirements or restrictions for the related DispID identifiers. Each plug-in will generally use different DispID identifiers when implementing the various properties and methods detailed in the LNS Plug-In API.

Once the director application has attached to the plug-in, the director can access the plug-in application's properties and methods by name. LNS plug-in applications launch in their hidden state, allowing important properties to be set by the director. The director will then typically make the plug-in visible by writing to its `Visible` property, as shown in the following example:

```
Dim MyPlugIn As Object
' connect to the plug-in

Set MyPlugIn = CreateObject(RegisteredServerName)
SetFocus

' insert code to pre-set the plug-in as needed
' ...

' make plug-in visible:
MyPlugIn.Visible = true
```

Note that the director application also calls `SetFocus` after connecting to the plug-in. Some plug-ins may take the focus away from the director when being launched, leaving the director's dialog defocused. Subject to details of the director's implementation, it may be a good idea to reclaim the focus in the interest of an uninterrupted user experience.

The plug-in manages its own lifetime independently, so the director need not terminate the plug-in or reset it to the invisible (hidden) state. However, the director should release its reference to the plug-in when it will not be needed in the foreseeable future, as shown here:

```
Set MyPlugIn = nothing
```

This will release the reference to the plug-in, and will therefore stop the director from accessing the plug-in application's properties and methods through the `MyPlugIn` variable. However, releasing the reference will not terminate the plug-in itself. The *LNS Plug-In Programmer's Guide* contains a section called *How Plug-Ins Know When To Exit* that addresses the plug-in lifetime considerations for the plug-in developer.

The director application is free to hold on to the reference. This will allow you to speed up processing during the network commissioning phase, as plug-ins that are invisible but attached do not need be re-loaded into memory. Therefore, the director can provide a very responsive user-interface.

However, Echelon recommends that you release plug-in references using a timer-controlled scheme to avoid blocking system resources with plug-ins. For example, the director application could automatically release references to a plug-in 15 minutes after the last use of that plug-in, and restart the timer each time the user demands the services of that plug-in.

A complete example implementation of the client-side plug-in API is contained in the example director application described in Appendix C of this document. The related source code can be found in the `Launch()` method that is contained in the `cCommand` class.

Detecting Existing Plug-Ins

Before you can use a plug-in to perform its prime service such as configuring a particular implementation of a device, the plug-in must be fully registered within the Windows operating system, and the LNS system. The *LNS Plug-In Programmer's Guide* provides a discussion of the plug-in registration process. This section provides a brief overview of the process, focusing on the director's role in plug-in registration.

When LNS plug-in software is installed, the installation includes registration of the plug-in application's COM server with the Windows Registry. Among other details, this publishes the plug-in application's *Registered Server Name* to the Windows system. At the same time, the plug-in registers with an LNS-specific part of the Windows Registry, by adding or updating data under the `HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LCA\Plug-Ins` key. The director detects all subkeys stored under `HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LCA\Plug-Ins` in the Windows Registry. Each subkey relates to an LNS plug-in application. Following is an example screenshot, showing the Plug-In Windows Registry key explored in the Windows RegEdit.exe utility, highlighting the Echelon LNS Report Generator plug-in:

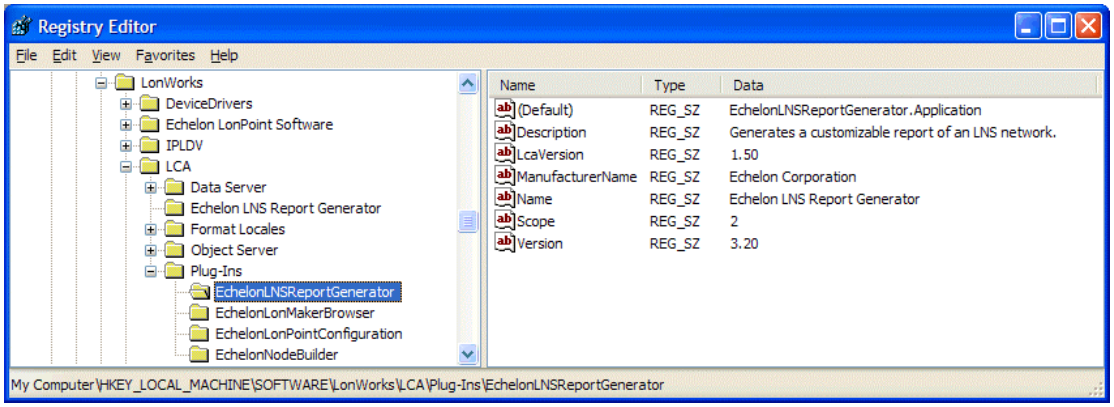


Figure 12.1 LNS Plug-Ins and the Windows Registry

Note that the default value for each plug-in listed in the Plug-In Windows Registry key equals the plug-in application's Registered Server Name. For example, Figure 12.1 shows details for the Echelon LNS Report Generator plug-in, whose default value and registered server name is "EchelonLNSReportGenerator.Application."

The director application must inspect the Plug-In Windows Registry key. For each plug-in found that has not already been fully registered, or that has been previously registered with an earlier version, the director will register the plug-in as described in the next section, *Registering Plug-Ins*.

Optionally, the director may also attempt to connect to all other plug-ins in an attempt to verify the plug-ins presence. Detecting orphan Plug-In Windows Registry keys is described in the *Advanced Plug-In Management Tasks* section later in this chapter.

A complete example implementation that retrieves all unregistered and previously registered plug-ins is contained in the example director application described in Appendix C of this document. Related source code can be found in the `ListPlugins()` method that is contained in the `FrmMain` class.

Registering Plug-Ins

For each plug-in detected in the Plug-In Windows Registry key, the director application investigates each `ComponentApp` object that can be accessed in the relevant `ComponentApps` collection. For plug-ins advertising Scope as 2 (two) in the system registry as shown in Figure 12.1, this is the `ComponentApps` collection that may be retrieved from the `System` object's `ComponentApps` property. Scope 2 is used by plug-in software that requires explicit registration for each network database. Most plug-ins specify Scope as 1 (one) in the system registry. In this case, directors use the `ObjectServer object's ComponentApps` property for maintaining the registration information. This ComponentApps collection is a global collection stored in the LNS global database that is mostly used for managing plug-ins software across multiple director applications. The director performs these tasks:

- For each `ComponentApp` object found in the registration `ComponentApps` collection, the director inspects the `ComponentApp` object's `RegisteredServer` property. If no `ComponentApp` object can be found that relates to the current plug-in stored in the Windows Registry as the registered server name, the plug-in must be registered with LNS.

- If a ComponentApp object that relates to the current plug-in (stored in the Windows Registry as the registered server name) can be found, but its VersionNumber is less than the Version advertised in the Windows Registry, the plug-in must be re-registered with LNS.

To register or re-register a plug-in, the director connects to the plug-in and calls its SendCommand() method with the CommandID argument set to lcaCommandRegister(50). When re-registering updated plug-ins, the director application will then update the ComponentApp object's VersionNumber property to match the current version that is listed in the Plug-In Windows Registry key. When completing the registration of a new plug-in, the director then creates a new ComponentApp object in the global ComponentApps collection, and sets the properties of the new object to reflect the new plug-in. At a minimum, the director sets the plug-ins' CommmandID property to lcaCommandRegister(50), and sets the RegisterServer property to the registered server name.

When the plug-in executes the SendCommand() method, the plug-in completes its own application-specific application needs. This includes importing external interface files, registering device resource files, or any other application-specific steps. Most importantly, the plug-in creates additional ComponentApp objects that detail the services the plug-in provides to the director. The director uses these ComponentApp objects to detect registered plug-ins that are applicable for a certain task. This is detailed in the next section.

A complete example implementation of the plug-in registration is contained in the example director application described in Appendix C of this document. Related source code can be found in the btnRegister_Click() event handler that is contained in the frmMain class.

Detecting Applicable Plug-Ins

A Plug-in can be written to perform any of the operations listed in the ConstCommandIDs constant. These operations may apply to any of the object classes listed in the ConstClassIDs constant. Plug-ins can provide specialized handlers for a wide range of operations on a variety of object classes, although not all combinations make meaningful actions. Typical actions include configuring devices (lcaCommandIdConfigure+lcaClassIdAppDevice) or configuring functional blocks (lcaCommandIdConfigure+lcaClassIdLonMarkObject), but the list of possible actions covers a much wider range.

Whenever the director is about to execute a certain user request that relates to any of the operations possible for plug-in implementation, the director should determine whether one or more plug-ins are available for that specific operation. This makes the list of applicable plug-ins. A typical director implementation would proceed as follows:

- If no applicable plug-in has been found, the director invokes its own generic solution for the task. For example, the LonMaker tool will launch the LonMaker Browser tool to configure devices unless plug-in software is available to accomplish the task.
- If one or more applicable plug-ins have been found, and exactly one plug-in has the related ComponentApp object's DefaultAppFlag set to True, the director invokes that plug-in as the default operation.

In other cases, specifically for cases where no or multiple plug-ins have been marked as the default provider, the director provides some user interface that lets the user to choose between all applicable plug-in applications and the director's own generic solution.

For operations that relate to a device, the `DeviceTemplate` object provides a `ComponentApps` collection. For operations that relate to particular `LonMarkObject` objects within a given device, the `LonMarkObject` object provides a `ComponentApps` collection. Finally, the `System` object's and `ObjectServer` objects' `ComponentApps` properties may be used to register plug-ins that apply to different items such as the `Channel` or `Subsystem` objects, or that apply to a wider range of items. A generic browser tool such as the LonMaker Browser facility is an example of a LNS plug-in application that supports browse and configure commands for all devices, whatever their type (program ID).

Thus, the director application must scan all applicable `ComponentApps` collection objects to collect the list of applicable plug-ins, as illustrated in Figure 12.2.

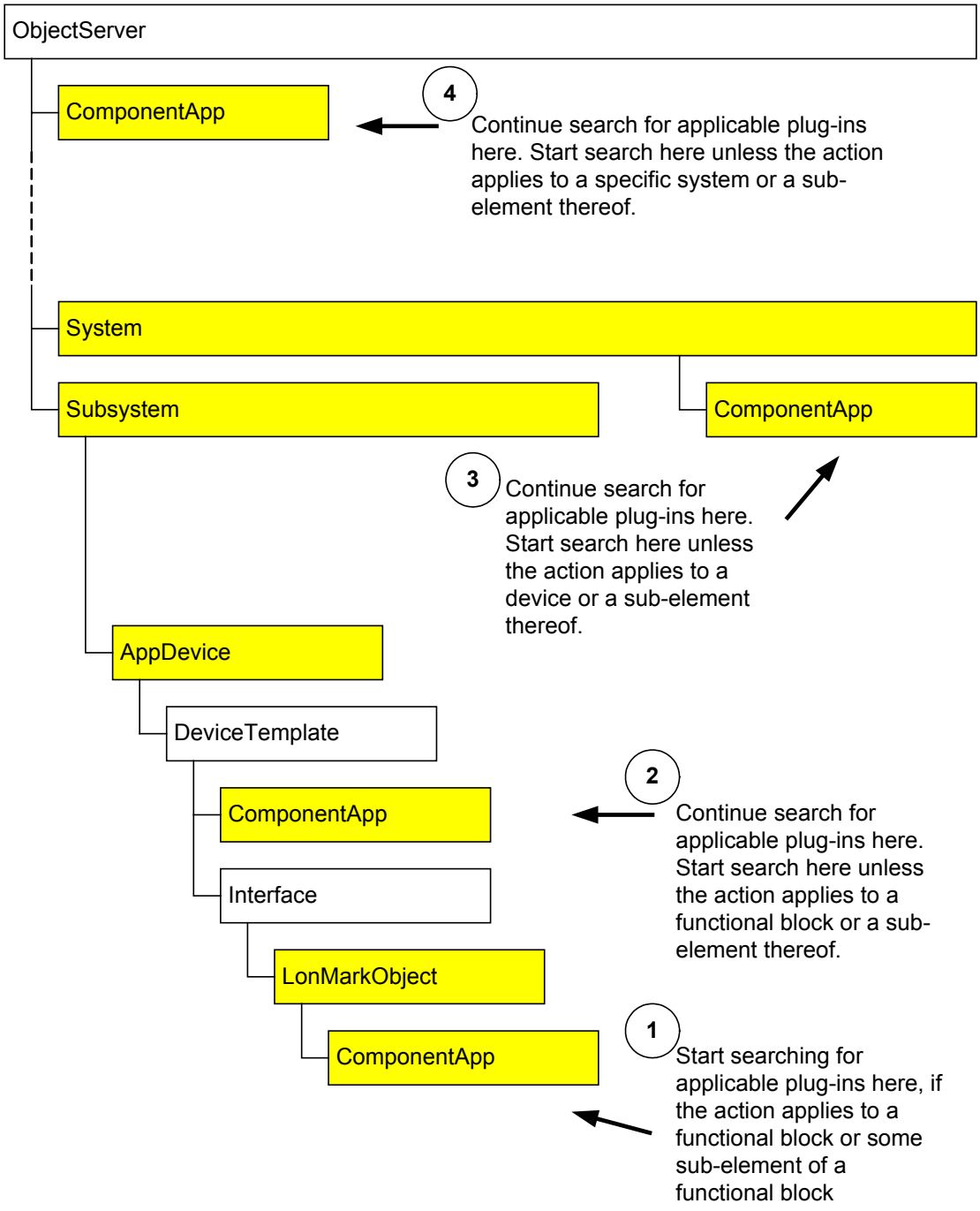


Figure 12.2 The ComponentApps Collection

Launching Plug-Ins

Once the director has detected the applicable plug-ins and chosen one, either automatically or with the help of some suitable user-interface, the director needs to connect to the plug-in. Then, it needs to initialize the plug-in by setting several of its properties. Lastly, it needs to call the plug-in application's `SendCommand()` method to invoke the desired operation.

The `SendCommand()` method and related properties are detailed in the *How Plug-Ins Work - The Details* section in the *LNS Plug-In Programmer's Guide*.

When calling the `SendCommand()` method, you should note that some objects support alternative addressing syntax used with the `objectName` parameter, as detailed in Appendix C of the *LNS Plug-In Programmer's Guide*. If multiple forms are defined, director applications should prefer using the simplest syntax unless ambiguities arise, as not all plug-in software may support all forms of addressing an object.

Advanced Plug-In Management Tasks

LNS Director applications may choose to implement advanced features and operations related to plug-ins, including those briefly described below:

- *Deregistration.* For existing and registered plug-ins, the director might choose to offer a deregistration command. In the presence of the plug-in to be deregistered, plug-in controlled deregistration can be attempted by calling the plug-in application's `SendCommand()` method with `lcaCommandIdUnregister (51)` value. Otherwise, in absence of the plug-in software, the director might scan the network database(s) and remove `ComponentApp` objects that relate to the plug-in to be deregistered. The director could then also remove the related Plug-In Windows Registry key, and complete the uninstallation of the plug-in application. This could be useful if the plug-in has been removed without proper deregistration.

- *Orphan Detection*. Entries might be left in the Plug-In Windows Registry key that no longer relate to existing plug-in software. This results from improper removal procedures. Directors can determine orphan registry keys by trying to connect to the advertised application using its Registered Server Name. Upon failure, directors may offer to deregister the orphan tool as discussed previously. Similarly, directors can investigate each `ComponentApp` object in the LNS global database or in any of the LNS network databases, and try connecting to each advertised plug-in. Upon failure, the tool might offer removal of the orphaned `ComponentApp` objects.

- *Re-registration*. Sometimes, plug-in registration might be broken in an attempt to correct a problem. For example, an integrator could attempt to solve a device-specific problem by removing the related `DeviceTemplate` object from the network database, and then re-creating it from scratch. This could break the plug-in application's registration. A well-written LNS director application should offer a means to re-register a plug-in, even if it seems fully and correctly registered.

- *Pre-launch*. Plug-ins may advertise a pre-launch capability in the Plug-In Windows Registry key by providing a `PreLaunch` subkey set to 1. Plug-ins use this flag to indicate that frequent use of the particular plug-in is anticipated. Director applications are advised to launch the plug-in in the background and before it is actually required, therefore providing for a responsive operation once the plug-in needs to be invoked. However, Echelon recommends that you disconnect from attached plug-ins after a certain timeout period, to prevent the system from being occupied by unused but loaded plug-in software.

The LNS Turbo Edition software includes an example Director application. For more information on this, see Appendix C of this document.

Implementing an LNS Plug-In

An LNS plug-in is implemented as an ActiveX automation server. In Visual Basic, this is done by creating a Sub Main, defining Sub Main as the startup form, and implementing a public Application class.

Implementing an LNS Device Plug-In

Device plug-ins are a special class of plug-ins that are specific to a particular device type (i.e. program ID). Device plug-ins provide an easy way for device manufacturers to provide tailored software for their devices for performing configuration, monitoring, or control functions specific to their devices.

Device plug-ins should be implemented as described in the *Implementing an LNS Plug-In* section earlier in this chapter. This section provides additional guidelines you should follow when implementing device plug-ins. You can also find more information about device plug-ins in the *LNS Plug-In Programmer's Guide* and the *NodeBuilder Plug-In Wizard* documents.

Managing Device Configuration

Configuration data for a device is typically maintained by the configuration properties stored on the device. This ensures that the device and the LNS database have a consistent view of the device's configuration, and allows the device to be easily replaced when it fails. Any configuration data for the device that cannot be stored as a configuration property within the device should be maintained as extension data, as described in the *Accessing Extension Data* section earlier in this chapter. This ensures that a plug-in can find this data, even if the user renames or moves the device or device template within the LNS database. If the configuration data is extensive, the extension data may consist of a key into a device specific extension database. However, Echelon recommends the use of `Extension` objects, since the data stored in `Extension` objects is accessible both locally and remotely.

Chapter 13 - LNS Licensing

This chapter describes the LNS licensing mechanism, including how LNS applications register with the LNS Server and how end-users consume and replenish LNS Device Credits.

Overview of LNS Licensing and Distribution

The LNS licensing model is based on LNS Device Credits. Each LNS Server PC contains a number of LNS Device Credits, and you need one LNS Device Credit for each application device you commission on a network. The LNS Object Server tracks these LNS Device Credits automatically, and removes an LNS Device Credit from the credit pool each time a device is commissioned. Credits may be purchased, added to servers, and transferred between servers using the LNS Server utilities provided with the LNS Application Developer's Kit. LNS also provides a Demonstration Mode which does not require the use of LNS Device Credits.

LNS Turbo Edition supports the standard, automated device credit management system, as well as the legacy capacity-based licensing system used in LNS 1.0 and LNS 2.0. Capacity-based applications built with LNS 1.0 or LNS 2.0 will use the LNS Device Credit mechanism when running on LNS Turbo Edition.

The standard LNS Device Credit management feature offers several cost advantages for the LNS developer. When you redistribute an LNS Server with your product with the LNS Redistribution Kit, the LNS Server will include 64 LNS Device Credits by default. After the end-user installs your product and reboots the target PC, the PC will have the LNS Server installed, and will also have 64 LNS Device Credits by default. The LNS Server redistribution is ready to run after the reboot, and does not require registration with Echelon to operate. You should note that after installing your product and the LNS runtime files, the user will need to initialize the LNS Server license to make the LNS Device Credits visible to the LNS licensing utilities. Initialization can be accomplished by opening a system with any LNS application, such as the LNS Server application.

Note that when you create your redistributable installation package with the LNS Redistributable Maker utility, you can specify more than 64 LNS Device Credits (up to 512). See the *Using the LNS Redistributable Maker Utility* section later in this chapter for more information on the utility.

If your application needs to commission devices, you must ensure that the `SetCustomerInfo()` method has been called on the `ObjectServer` object before doing so. You must do this before you open the system. This begins use of the standard LNS Device Credit licensing mode. LNS requires that one LNS Device Credit is available each time you commission a device on the network. When a device is decommissioned and removed, an LNS Device Credit is returned back into the database. Thus, your end-user does not get charged for deletions - only for the total number of devices managed by the LNS Server. This is described in more detail later in the chapter.

NOTE: If you delete a network with the `Remove()` method, the network database will be deleted. The LNS Device Credits consumed by all commissioned devices in that network will be lost, unless those devices are first decommissioned. For more information on this, see *Removing Network Databases* on page 255.

LNS Device Plug-in applications do not commission devices, and therefore developers of LNS Device Plug-in applications do not need to be concerned about what licensing management mechanism is in effect.

If end-users require more LNS Device Credits, they can use the LNS Server License Wizard to purchase more LNS Device Credits from Echelon. Licensees of the LNS Redistribution Kit product can also re-sell LNS Device Credits, subject to certain

minimum purchase requirements, with the Echelon Software License Generator product separately available from Echelon. Contact your Echelon sales representative for details.

The end-user should also use the LNS Server License Transfer Utility to transfer the LNS Device Credits and the license in order to move the LNS Server to another PC. The LNS Device Credits and license are transferred together. It is not possible to move some of the LNS Device Credits on a PC to another PC. You must move all of the LNS Device Credits and the LNS Server license to the new PC.

The LNS Application Developer's Kit, when registered, can reauthorize as many LNS Device Credits on the development PC as you need for development purposes, without requiring you to request more LNS Device Credits from Echelon. If you need more LNS Device Credits for development purposes, run the LNS Server License Wizard, and follow the prompts. A dialog will appear asking you if you would like to automatically authorize yourself for more LNS Device Credits. Answer "Yes" and you will be granted more LNS Device Credits. This auto-authorization is only available on a PC with a properly registered LNS Application Developer's Kit. Because this capability is recorded in the same license required to use the LNS Server, transferring the license to another PC would move this capability to the target PC.

Additional information about LNS Device Credits may be available on the LNS Home Page at www.echelon.com/lns.

Demonstration Mode

When an LNS application is initialized, it starts running in Demonstration Mode. In Demonstration Mode, each network is limited to a total of four devices, excluding the LNS Server PC, and any Network Service Devices or routers installed on the network. As long as the number of devices is within this limit, the LNS Object Server operates normally in Demonstration Mode. You can add or remove devices, load applications, and create connections between devices.

If you want to distribute demonstrations of your application, you should always distribute them in Demonstration Mode. Check the software license agreement that governs the use of this product to determine if you can distribute LNS Object Servers.

Standard Mode

The Standard Mode is the LNS Device Credit-based licensing model described earlier in this chapter. When operating in Standard Mode, the number of devices operating on the network must be less than or equal to the number of LNS Device Credits installed on the LNS Server PC. As in Demonstration Mode, the number of devices excludes the LNS Server PC, Network Service Devices, and routers.

In addition to the total number of LNS Device Credits, LNS Servers operating in Standard Mode are also assigned a pool of deficit credits. Deficit credits are LNS Device Credits beyond the number purchased that you can use. Typically, 500 deficit credits are allocated to each network.

Networks whose LNS Object Server has a maximum deficit credit pool greater than zero can continue to install devices, even when the regular supply of LNS Device Credits has been exhausted. As soon as the application starts using deficit credits, the LNS Object Server alerts the LNS application via the `OnLicenseEvent` event, starts a 14-day timer,

and begins tracking the number of deficit credits. For more information on the `OnLicenseEvent` event, see *Tracking License Events* on page 293.

If the number of deficit credits in use is reduced to zero, either by installing more LNS Device Credits on the LNS Server PC, or by removing devices from the network, the timer will reset and the LNS Server will return to normal operation. If the timer expires or the number of deficit credits used exceeds the maximum, the LNS Server will no longer operate. In this case, you must upgrade the license by purchasing more LNS Device Credits from Echelon or a distributor. Since the LNS Server will not even be operable, you will not be able to remove devices and regain LNS Device Credits at that point.

The minimum purchase is 50 LNS Device Credits, plus the number of deficit credits you are using. When you purchase the additional LNS Device Credits, the LNS Server License Wizard will automatically suggest a value that is equal to 50 plus the number of deficit credits currently used. It is the responsibility of the LNS application to warn users when they are in deficit mode, so they can purchase more LNS Device Credits or remove devices from the network, as appropriate.

Entering the Standard Mode

To enter the Standard Mode, an application must invoke the `ObjectServer` object's `SetCustomerInfo()` method before it opens the Object Server. The method takes two parameters, a customer ID and key, as follows:

```
ObjectServer.SetCustomerInfo CustomerID, CustomerKey
```

The `CustomerID` and `CustomerKey` values are printed on the back cover of the LNS Application Developer's Kit CD-ROM jewel case. Remember that if you do not call `SetCustomerInfo()` to enter the Standard Mode, your application will remain in Demonstration Mode.

NOTE: Some legacy LNS applications may still use the `SetCapacity()` method. These applications will be automatically switched to use the Standard Mode described in this chapter as soon as the method is called. At that point, the application will use the LNS Device Credits on the LNS Server PC to determine how many devices can be installed on the system, as opposed to using a separate capacity for each system.

Protecting Your Keys

It is very important to protect the integrity of your customer ID and customer key. This information is confidential and should be guarded as securely as any other confidential information that you use. Do not use the customer key or customer ID as any form of product key.

Viewing License Status

LNS applications can determine a system's current licensing status by reading the `System` object's `CreditInfo` property. This returns a `CreditInfo` object contains the following licensing-related properties:

- `DaysRemaining`: The number of days remaining before the license expires. If deficit credits are in use, the value of this property will be 14 or less. The special value of 255 is normal, indicating that no deficit

credits are in use, and therefore there is no time limit. The special value of 254 indicates that the application is operating in Demonstration Mode, or that the application is a legacy (LNS 1.x) application, and there is no time limit.

- `DeficitCredits`: The number of deficit credits currently being used.
- `LicensedCredits`: The number of LNS Device Credits licensed on the LNS Object Server. This number does not include the number of deficit credits allowed.
- `LicenseType`: The type of license in effect. A value of 0 indicates Demonstration Mode is being used, or that you are running a legacy (version 1.x) LNS application. A value of 1 indicates that Standard Mode is being used.
- `MaxDeficitCredits`: The maximum number of deficit credits allowed on the LNS Object Server. If this number is exceeded, the LNS Server license will expire, and the LNS Server will cease to operate until sufficient LNS Device Credits are added.
- `UsedCredits`: The number of LNS Device Credits currently in use. This number does not include the number of deficit credits currently being used.

Tracking License Events

LNS applications should notify users about certain licensing situations, such as when they are using deficit credits and when the LNS Server's license has expired. That way, users can take action before the expired license renders the LNS Server inoperable. For these and other application-specific reasons, an LNS application may need to track licensing events. To do so, LNS provides the `OnLicenseEvent` event.

To register for this event, invoke the `BeginLicenseEvent()` method, as follows:

```
MySystem.BeginLicenseEvent()
```

To cancel the receipt of licensing events, invoke the `EndLicenseEvent()` method, as follows:

```
MySystem.EndLicenseEvent()
```

License Event Types

The `OnLicenseEvent` event returns five parameters. Two of these, the network handle and system handle, identify the network and system on which the licensing event occurred. The event also returns an updated `CreditInfo` object, as well as two integer values indicating the event type and the number of LNS Device Credits debited or credited by that event. The event types and the meaning of the corresponding count values are the following:

- `LIC_DEBIT`: A debit has been charged to the available number of licensed LNS Device Credits due to a device installation. The number of LNS Device Credits charged is returned as a positive integer in the `count` parameter. You can check how many LNS Device Credits are currently in use by reading the `UsedCredits` property of the `CreditInfo` object returned by the event.

- LIC_DEFICIT: A credit or debit has been charged, and the number of licensed LNS Device Credits has been exhausted. As described in the *Standard Mode* section on page 291, the LNS Object Server will continue to run for a 14-day grace period during which the deficits credits can be used. Read the `DaysRemaning` and `DeficitCredits` properties of the `CreditInfo` object returned by the event to determine how many deficit credits are available, and how much longer the LNS Server license is valid for. The `count` parameter contains the number of LNS Device Credits being debited (if positive), or returned (if negative). LNS applications are required to periodically raise a dialog warning the user that the license will shortly expire, once this event is received.
- `LIC_EXPIRED`: The LNS Server's license has expired. This occurs when the maximum number of deficit credits has been used, or when the 14-day grace period initiated when the first deficit credit is used has expired. The application is required to terminate at this point. If the expiration was caused by a debit transaction, the `count` parameter returns the number of devices successfully debited.
- `LIC_CREDIT`: A credit has been applied to the available number of LNS Device Credits due to the removal of a device. This is reflected as a decrease in the number of used LNS Device Credits. The number of LNS Device Credits added is returned as a negative integer in the `count` parameter.

Licensing and Network Recovery

The network recovery feature described in Chapter 10 can be used on any LONWORKS network, whether it was installed by an LNS application or by some other kind of network tool. If the network was installed by an LNS application, the network recovery process should not deduct any LNS Device Credits, because this would result in an overcharge (i.e. a device would be charged once during the original installation and once during the network recovery). On the other hand, recovery performed on a non-LNS network should charge for each device on the recovered network. Unfortunately, there is no way for the LNS Object Server to determine whether or not the network being recovered was installed by an LNS application.

The LNS Licensing Agreement charges the LNS application with the responsibility of indicating whether or not a network was installed by an LNS application. To simplify this process, the `System` object contains the `RestoreLicense()` method. You can call this method after network recovery to restore the license.

The method takes an input parameter (`wasLNS`) that indicates whether or not the recovered network was installed by an LNS application. A value of `True` indicates that the recovered network was installed by an LNS application, and a value of `False` indicates that it was not. For example, the following code indicates that the network was installed by an LNS application, and so restores the license without deducting LNS Device Credits during the recovery:

```
MySystem.RestoreLicense(True)
```

The `System` object provides the `CommissionedDeviceCount`, `UncommissionedDeviceCount`, and `UninstalledDeviceCount` properties. These properties indicate the number of devices that were recovered (commissioned) or discovered in an unconfigured state (uninstalled) during the recovery process. When the

`RestoreLicense()` method is called with the `wasLNS` parameter set to `False`, the LNS Server will debit LNS Device Credits based on the commissioned count.

Licensing and Device Manufacturing

In a manufacturing setting, you may want to load devices with the same configuration information. The `AppDevice` object's `Replace()` method is a convenient way to configure devices quickly in an "assembly line" fashion. The LNS License Agreement specifies that the customer must use an LNS Device Credit for each device configured in this way. However, the `Replace()` method does not automatically deduct LNS Device Credits from the credit pool. To facilitate compliance with the license agreement, the `System` object provides the `DebitLicense()` method. This method deducts the number of LNS Device Credits specified by its `count` parameter.

For example, if 10 replace operations were performed, you could use the following code to debit your LNS Device Credit pool:

```
MySystem.DebitLicense(10)
```

Testing Devices

The current version of the LNS License Agreement provides additional means to create LNS-based manufacturing test tools that do not consume LNS Device Credits. Typical production, calibration and test software will commission devices in a test network, and then perform the necessary production steps. Since most LONWORKS devices should be shipped in the unconfigured state, the production tool will decommission the device at the completion of the burn-in and test phase:

```
MyDevice.Decommission()
```

As decommissioning returns the device credit, the entire manufacture tool consumes just one credit for each device that is under test simultaneously.

Using the LNS License Utilities

The LNS Application Developer's Kit includes two utilities that allow users to add or transfer LNS Device Credits to their LNS Server PC. These utilities are the LNS Object Server License Wizard and the LNS Object Server License Transfer Utility. These applications are described briefly in this section. For more detailed information, see each application's online help.

Using the LNS Server License Wizard

You can use the LNS Server License Wizard to add LNS Device Credits to an LNS Server. To do so, follow these steps:

1. Open the LNS Server License Wizard through the Echelon LNS Utilities group in the Windows Programs menu on the PC running the LNS Server. The utility opens with the dialog shown in figure 13.1.

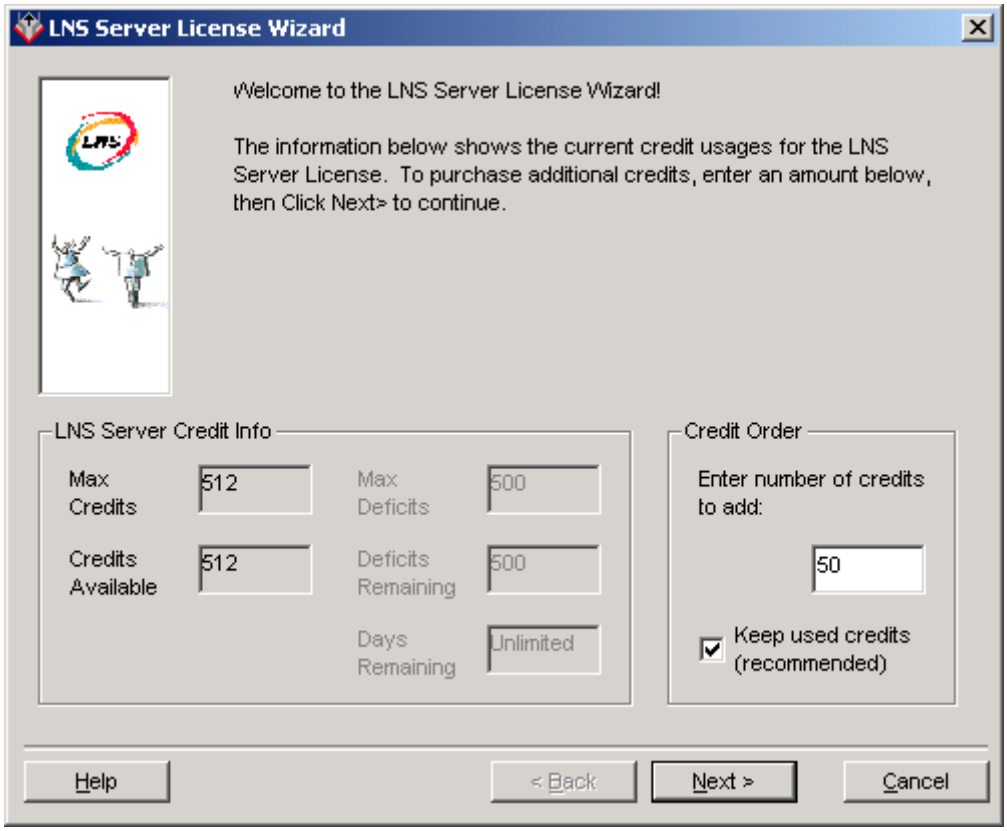


Figure 13.1 LNS Server License Wizard

2. This window contains the current license status of the PC running the utility, and the number of LNS Device Credits available on the PC. Enter the number of LNS Device Credits you want to add in the **Enter credits to add** text-box, and click the **Next** button. This opens the order processing window shown in figure 13.2.

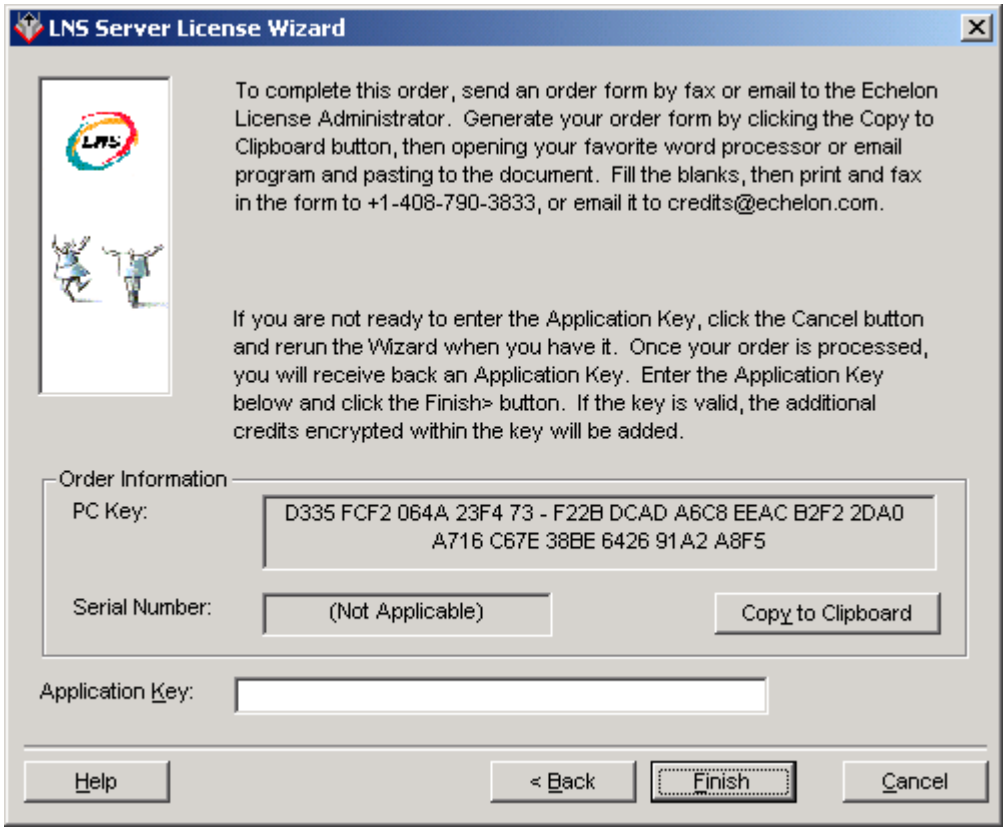


Figure 13.2 Order Processing Window.

3. The order processing window contains the information you need to provide to Echelon to order additional LNS Device Credits. You can copy this information to a Windows clipboard by clicking the **Copy to Clipboard** button. This generates an order form containing the required customer information, which you can paste into a document and send to Echelon (or an authorized distributor) via mail, fax, or email.

Once the order has been processed by Echelon, the Echelon License Administrator will send a new application key. You should enter this key in the **Application Key** textbox, and then click the **Finish** button to add the LNS Device Credits to the LNS Server. While waiting for the new application key, do not commission or remove any devices, or make any other changes to the network you are adding the LNS Device Credits to. This will cause the PC key to change, which will invalidate your new application key.

The LNS Application Developer's Kit provides an option to bypass the application key, and add new LNS Device Credits to your credits pool without making a purchase. However, these credits are only valid on the development PC, and cannot be transferred to LNS Servers that will be redistributed.

Once the credits have been added, the completion window shown in figure 13.3 will appear. Click **Exit** to close the utility.

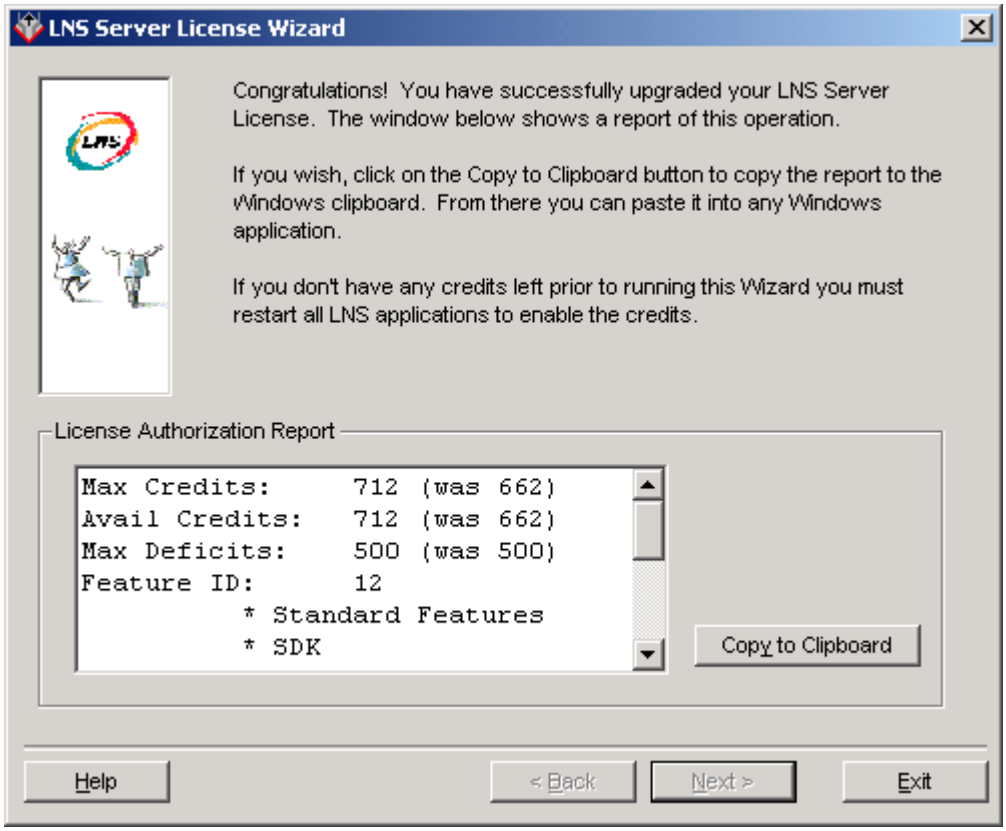


Figure 13.3 Upgrade Completion Window.

Using the LNS Server License Transfer Utility

You can use the LNS Server License Transfer Utility to transfer an LNS Server license between two LNS Server PCs. The transfer is a 3-step process. To begin, open the LNS Server License Transfer Utility through the Echelon LNS Utilities group in the Windows Programs menu, and follow these steps:

1. The initial window displays a welcome message. Click **Next** to open the dialog shown in Figure 13.4.

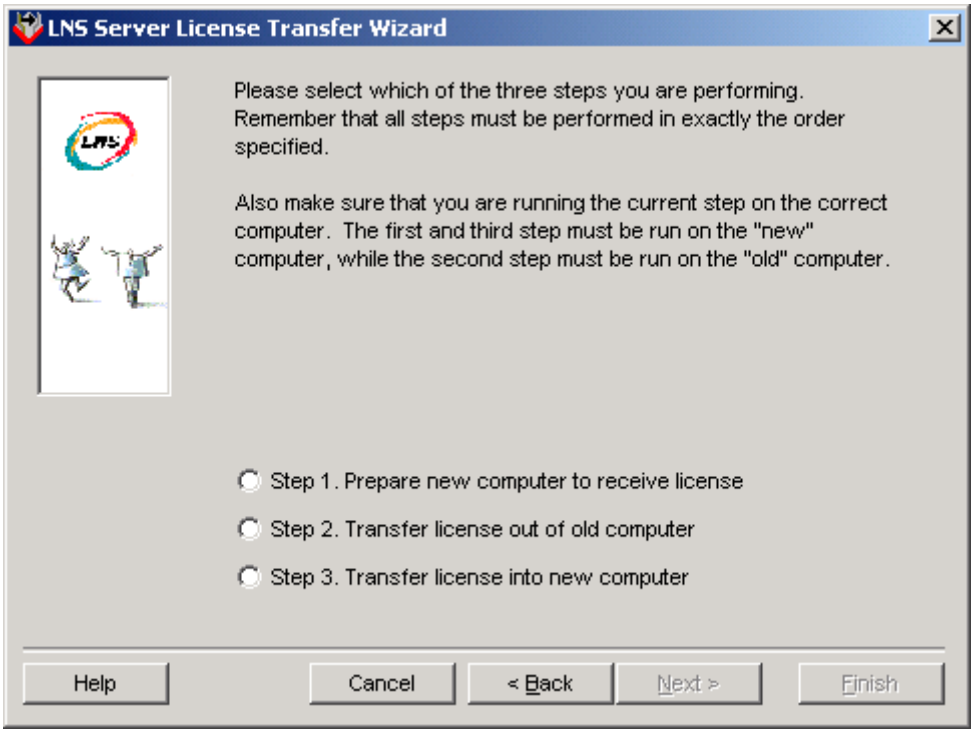


Figure 13.4 Transfer Step Selection Window

2. There are three tasks you will perform with this window. Select the task you are currently performing, and click **Next** to continue.

   These tasks must be performed in the order they are listed. You must perform steps 1 and 3 are on the PC that will receive the transferred credits, and you must perform step 2 on the PC providing the credits. A floppy disk or a shared drive, such as a mapped network drive, is required to complete the transfer.

   When you perform step 1, you will also need to select the drive the LNS Device Credits will be transferred to. You will use the dialog shown in Figure 13.5 to do so.

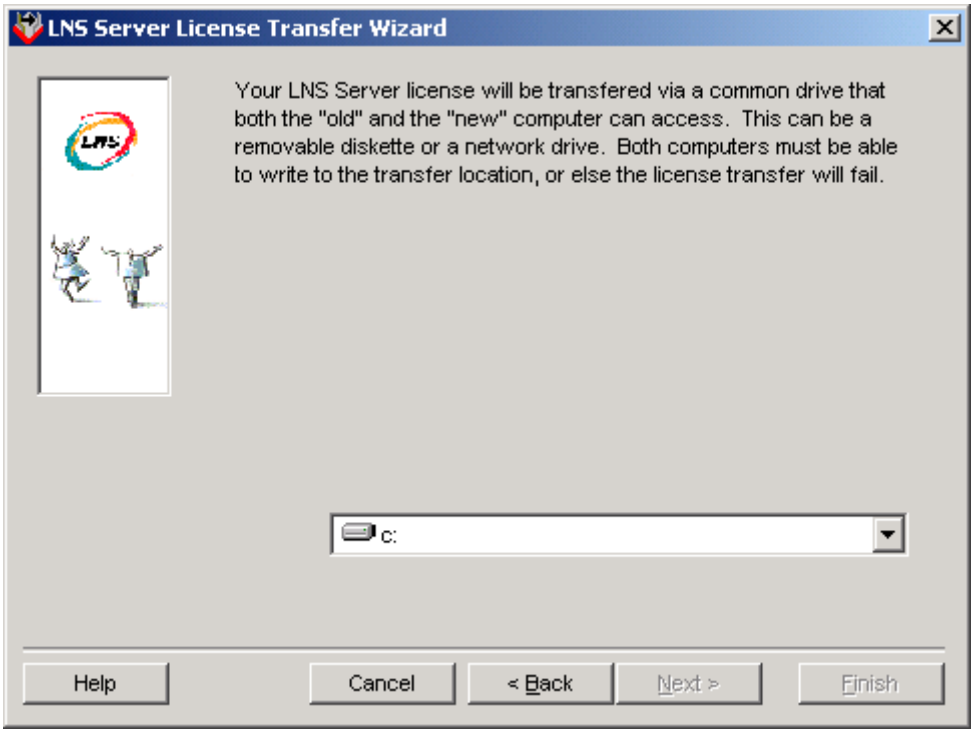


Figure 13.5 Shared Drive Selection Window

3. Select the drive to act as the transfer location from the pull-down list, and click **Back** to return to the dialog shown in Figure 13.4. Then, continue with Step 2 of the transfer operation.

Chapter 14 – Distributing LNS Applications

This chapter describes how you should redistribute your LNS applications, including how to use the LNS Redistributable Maker utility and how to install your LNS application.

Distributing LNS Applications

If you are using the LNS Application Developer's Kit, you can distribute your LNS applications, but you cannot distribute any of the LNS runtime files installed on your development PC. If you want to redistribute LNS applications, you must purchase the LNS Redistribution Kit. When you install this product, the Echelon LNS Redistribution Kit program directory is created. This directory contains the LNS Redistributable Maker utility. This utility is not included with the LNS Application Developer's Kit.

It is the responsibility of the developer to create the installation for an LNS redistributable application. The LNS Redistributable Maker Utility produces an install program that installs all the necessary LNS components, and creates the Echelon LNS Utilities program directory on the target PC.

The required components vary depending on the LNS application. If the application will be used solely as a remote client, then the LNS Server components are not required, but the LNS ActiveX Control components are. In this case, the Redistributable Maker can produce a smaller "remote client" installation. If the application can operate locally and remotely, the Redistributable Maker produces a full LNS installation, which can install both the complete and remote client versions.

Using the LNS Redistributable Maker Utility

The LNS Redistributable Maker Utility is installed as part of the LNS Redistribution Kit. To use the Redistributable LNS Maker Utility, launch the application using the shortcut in the LNS Redistribution Kit program group in the Windows Programs menu. This opens the dialog shown in Figure 14.1.

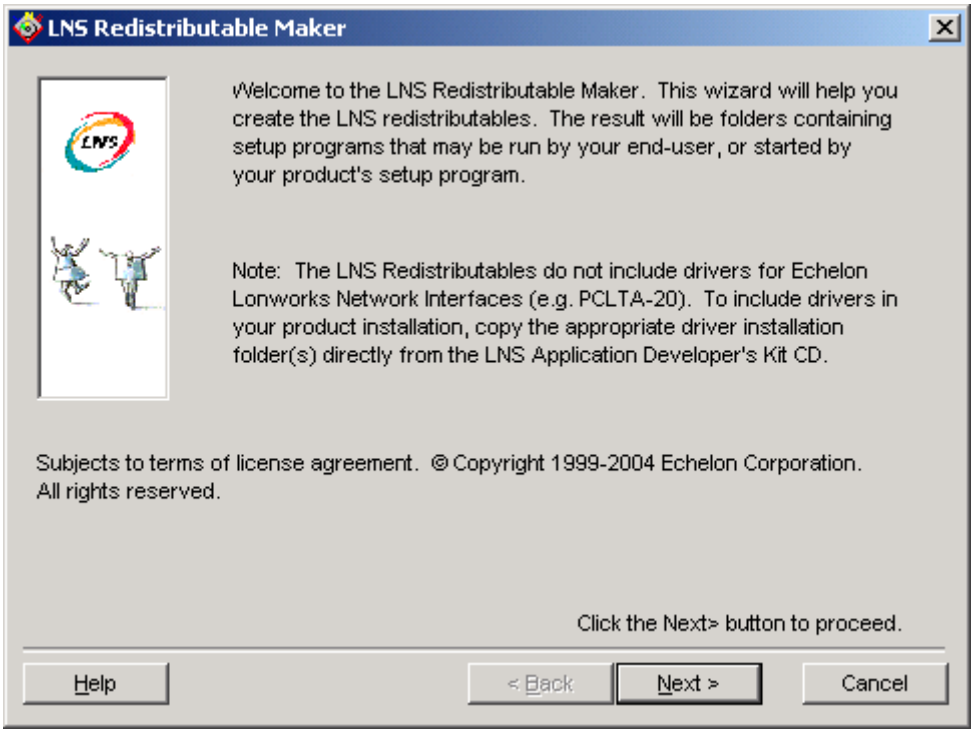


Figure 14.1 LNS Redistributable Maker

1. Click **Next** to begin using the utility. This opens the dialog shown in Figure 14.2.

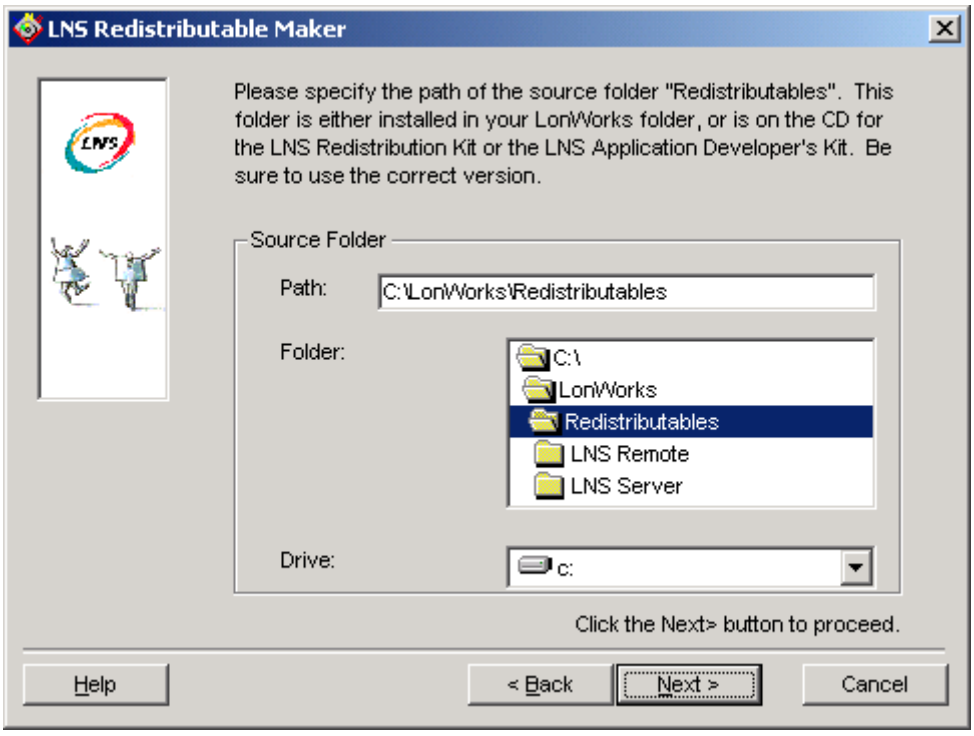


Figure 14.2 Redistributable Source Path Selection Dialog

2. Figure 14.2 illustrates the redistributable source path selection window. Select the path of the source redistributables folder, and click **Next** to continue. This opens the dialog shown in Figure 14.3.

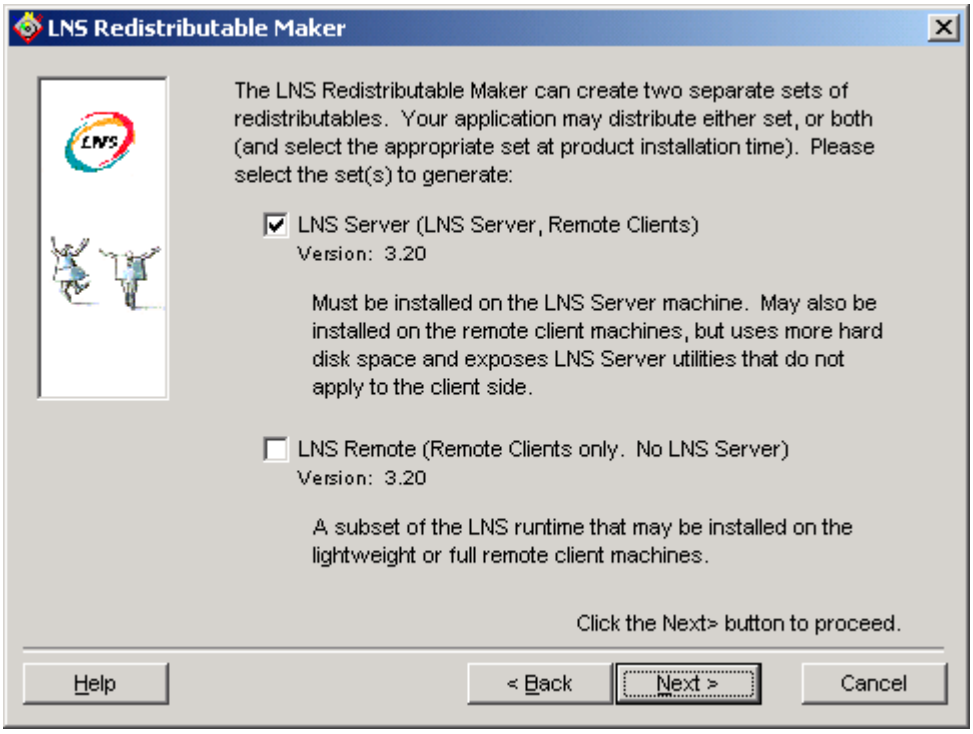


Figure 14.3 Redistributable Selection Dialog

3. Figure 14.3 illustrates the redistributable selection window. Select **LNS Server** or **LNS Remote**, depending on whether your application will operate locally or exclusively remotely. If you select the **LNS Remote** option, your application must set the `RemoteFlag` property to `True` before opening the LNS Object Server, as described in Chapter 4 of this document. If you select **LNS Server**, your application can operate locally or remotely (meaning the `RemoteFlag` property can be set to `True` or `False`).

   Click **Next** to continue. This opens the dialog shown in figure 14.4.

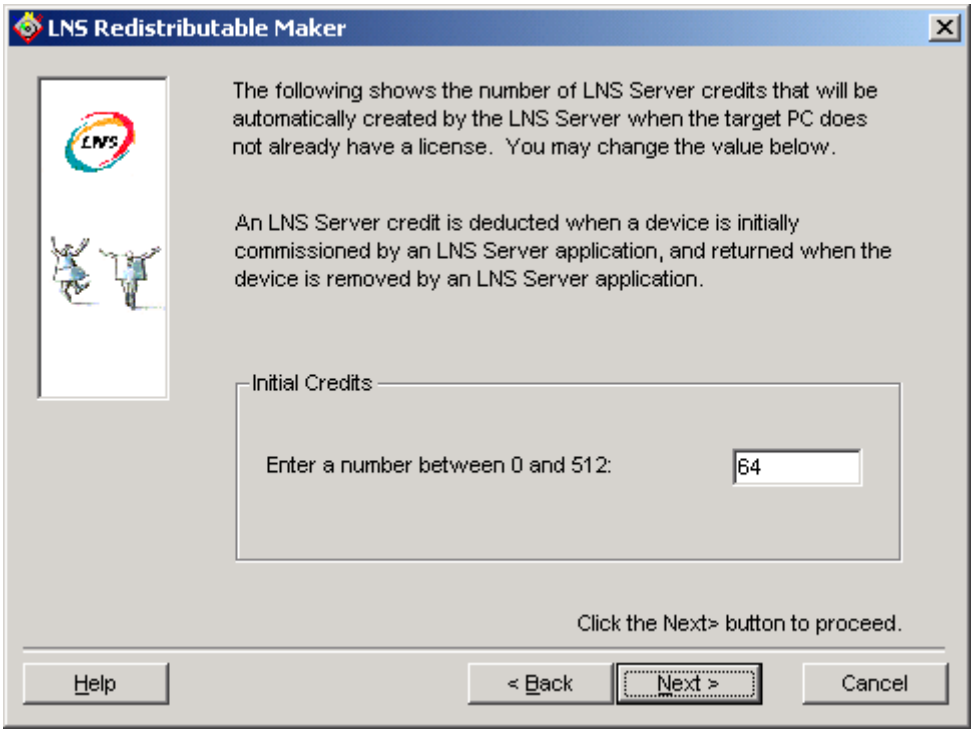


Figure 14.4 Initial Credit Selection Dialog

4. The LNS Redistributable Maker can allocate a number of LNS Device Credits to be available to the LNS Server installed with your application. Use the dialog shown in figure 14.4 to determine how many LNS Device Credits will be available. Once the LNS Server redistributable package is installed and registered with a Type 3 license, the LNS Device Credits allocated in this step will be enabled for the user.

   Additional LNS Device Credits can be added by end-users using the LNS Server License Wizard after they have installed and registered the product with a Type 3 license. Note that if the LNS Server redistributable files are installed on a PC that already has an LNS Server license, the initial LNS Device Credits specified in this dialog box will be disregarded, not added to the license on that end-user's PC.

   Click **Next** to continue. This opens the dialog shown in Figure 14.5.

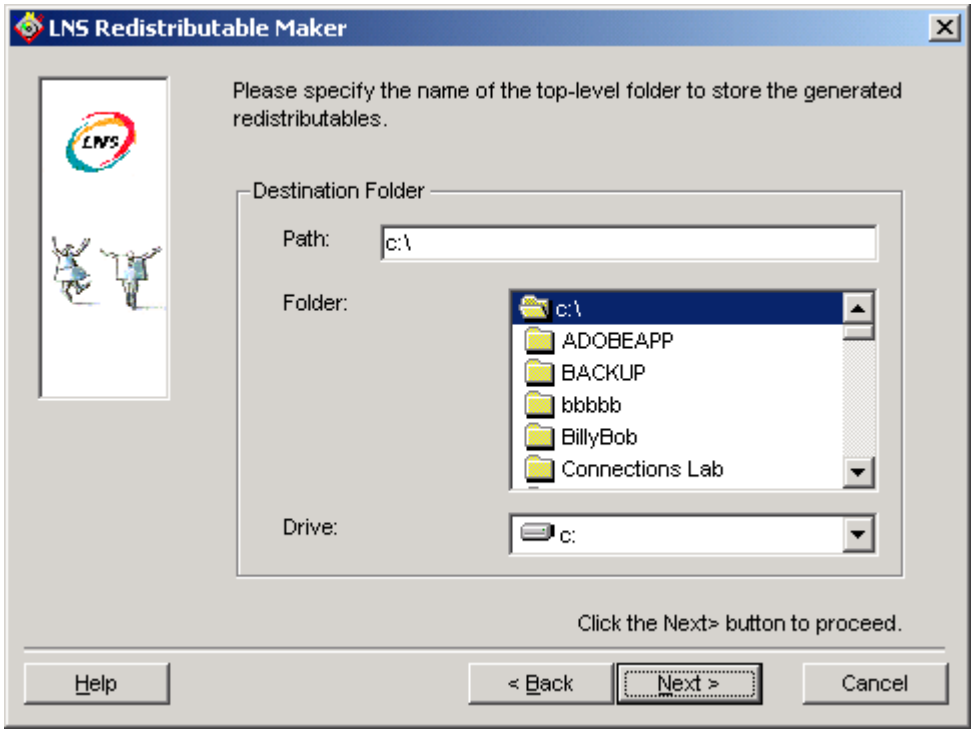


Figure 14.5 Select Setup Package Destination Folder Dialog

5. Specify the location of the setup packages that are generated. If you specify a folder name that does not exist, the LNS Redistributable Maker utility will create that folder automatically. You can produce various redistributable setup packages. For instance, you may want to sell different versions of your product, where each version of the product differs by the number of LNS Device Credits that are initially included with it.

   Click **Next** to continue. You will then be prompted to finish the operation and exit the application.

Adding the LNS Runtime to an LNS-based Product Installation

The LNS Redistribution Kit will allow you to generate sets of files you can use to install either the "Echelon LNS Server" or the "Echelon LNS Remote Client" runtime from your application's installation. These new runtime redistributable packages have been produced with InstallShield DevStudio 9 as Windows Installer installations. They will automatically uninstall older, non-Windows Installer versions of the LNS runtime components before installing the new runtime.

Each set of redistributable files includes the `setup.exe` installation launcher application, and the Windows Installer companion DLL `_SetupLNS.dll`. Depending upon your installation environment, you will probably find it convenient to install the LNS runtime using one of these methods.

The correct approach for your installation depends upon the capabilities of your application's installation. The `setup.exe` application will check for Windows Installer version 2.0 or greater, which is required for the new LNS runtime installations, and update the Windows Installer runtime if it is a lesser version. If your installation is based on Windows Installer, and will update Windows Installer when necessary, it does

not need to use `setup.exe`, and should use the simpler `_SetupLNS.dll`, as described in the following sections.

The `setup.exe` application introduces a noticeable delay after the LNS runtime installation progress dialogs are complete, potentially causing it to appear as if your installation is complete before it actually finishes. For this reason, Echelon recommends using `_SetupLNS.dll` in the Windows Installer environments where it is supported.

Even though Echelon recommends that you use `_SetupLNS.dll` to install the LNS runtime, this document will explain the use of the `setup.exe` option first. Because this option automates less of the installation process, using it requires a more comprehensive description of the LNS runtime installation options, and is valuable for all LNS runtime redistributors.

Using setup.exe

Since `setup.exe` should be used by all environments except Windows Installer based installations, this document cannot assume any particular capabilities exist in the runtime installation other than Windows Registry access. All inputs to the LNS runtime installation and outputs from the LNS runtime installation are described in terms of Windows Registry entries. For proper function, the `setup.exe` file should be kept in the same directory on your source medium as the rest of the LNS runtime installation redistributable files.

To embed the LNS Runtime installation into your LNS-based product installation, follow these steps. Each step is described in detail in the following sections:

1. Preset the LONWORKS path
2. Preset the LNS network database path
3. Check the installed LNS version
4. Check the LNS runtime installation completion status
5. Install the version 3 Microsoft XML Parser

Step 1: Preset the LONWORKS Path

The "LonWorks Path" entry in the Windows Registry must be defined before you launch the LNS Runtime installation, if you want to install to a LONWORKS path different than the default. Although Echelon does not recommend changing this path, it may be necessary in your particular product installation scenario.

The location from which the LNS runtime reads the LONWORKS path information is stored in the following Windows Registry string value:

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LonWorks Path
```

This string value determines the location of the main "LonWorks" installation folder. The LNS runtime files will be installed in subfolders of this folder. If this Windows Registry entry does not exist, the default path will be `[WindowsVolume]\LonWorks`, where `[WindowsVolume]` is the drive where the Windows operating system resides.

Once the LONWORKS path key has been created, you cannot modify it to point to an alternate folder. If the LONWORKS path key is changed after it has been initially set, some or all of the Echelon software installed on your machine could malfunction.

For compatibility with all releases of Echelon products, the value of the LonWorks Path entry should be a full path, including drive designation, and never end in a slash, "\".

Step 2: Preset the LNS Network Database Path

Like LNS 3.0, LNS Turbo Edition requires that you log in as a member of the Administrators user group when running the LNS runtime installations, or any installation that embeds the LNS runtime installations. However, LNS 3.0 also required that you log in as a member of the Power Users group to use LNS. In Turbo Edition, this is no longer the case, as members of any user level can use LNS.

However, users will need the correct access rights to your network database folders in order to read or write the network configurations using LNS. You can configure the LNS runtime installation to allow LNS access to all Windows user levels to avoid this problem.

You must specify the root directory of the network database directory tree in order for the LNS runtime installation to enable network database access to all Windows user levels. Of course, this directory must exist before launching the LNS runtime installation in order to enable access. The network database path location should be set in the following Windows Registry string value:

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LNS Runtime Installer\LNSDBPath
```

The value of the LNSDBPath entry should be a full path, including drive designation, and never end in a slash, "\".

Step 3: Check the Installed LNS Version

The `setup.exe` launcher application will check the installed version of LNS, and run either a Windows Installer full installation or an upgrade installation, depending on which product version is installed.

However, if a greater version of LNS is already installed, it will display the dialog shown in Figure 14.6.

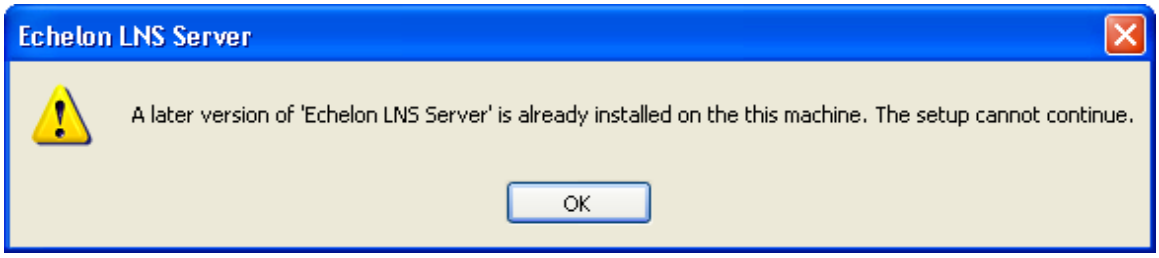


Figure 14.6 Echelon LNS Server Dialog

The statement, "**The setup cannot continue**" may be confusing to the users of your installation, even if you ignore the return code from `setup.exe` and continue your installation.

In order to avoid this dialog, you must check for this case before running `setup.exe`, and avoid any attempt to install over a greater version of the LNS runtime. To do so, your installation must read the LNS runtime version information at the following Windows Registry key:

`HKEY_LOCAL_MACHINE\SOFTWARE\Echelon\<LNS Runtime`[1]`>\Version`

The version Windows Registry key is a string value in the format Mmmbbb, where M is the major version number, mm is the minor version number, and bbb is the build number. The LNS Turbo Edition version number will be in the format "320bbb", where the final build number will be visible in the first few lines of the LNS runtime readme.htm file.

If this Windows Registry key does not exist, the PC either does not have the LNS runtime installed, or the LNS runtime is a pre-Windows-Installer version that can be freely upgraded by LNS Turbo Edition as a full installation. So, the absence of this Registry key is also a signal that the `setup.exe` install can be run.

Step 4: Check the LNS Runtime Installation Completion Status

After `setup.exe` is run, you can check the completion status of the LNS Runtime installation by reading the following Windows Registry key string value:

`HKEY_LOCAL_MACHINE\SOFTWARE\Echelon\<LNS Runtime>\Install Status`

The value will be set to "Success" if the LNS runtime was installed successfully, and does not require a reboot to complete. The value will be set to "Success-RebootNeeded" if the LNS runtime was installed successfully, but a reboot is necessary in order to finalize the installation. Your LNS-based product installation should force a reboot upon completion in this case, or at least inform the user that a reboot is needed to complete the installation.

Step 5: Install the version 3 Microsoft XML Parser

The LNS Turbo Edition runtime requires the version 3 Microsoft XML Parser. While this exists on the newer Windows operating systems that are supported by Turbo Edition, including Windows XP and Windows Server 2003, it does not exist on Windows 2000 (unless it has been updated with Internet Explorer 6.0).

Microsoft's licensing of this redistributable is fairly flexible. However, they do not allow redistribution of these components by users whose customers will then redistribute them again. This prevents Echelon from installing them in the LNS runtime installations.

Users of the LNS runtime installations have two choices. They can either install the version 3 Microsoft XML Parser in their installations, or they can require that systems on which their products are installed include Internet Explorer version 6.0 or greater.

---

[1] <LNS Runtime> should be either "Echelon LNS Server" or "Echelon LNS Remote Client", depending on the LNS runtime installation you are using.

The LNS Application Developer's Kit and LNS Server Edition products both install the version 3 Microsoft XML Parser.

Using _SetupLNS.dll

`_SetupLNS.dll` is a Windows Installer DLL. This means that if it is embedded within your Windows Installer based installation, you can call its functions within your Custom Actions. No parameters are passed to Windows Installer DLLs. They have access to your installation's Windows Installer environment, including any Windows Installer properties declared within your installation, so data can be passed via Windows Installer properties.

To install the LNS Runtime you need to create a Custom Action to call into `_SetupLNS.dll`, which will direct the Windows Installer to run these sub-installations. You should add the `_SetupLNS.dll` that comes with your LNS redistributable components to the Binary Table of your installation. This is done within an InstallShield DevStudio 9 project by going to the "Support Files" item under the "Behavior and Logic" item, and adding the `_SetupLNS.dll` file to the list of files in the "Language Independent" files area.

The Custom Action properties for calling `_SetupLNS.dll` are shown in table 14.1 below. There are two public entry points in this DLL. The entry point **SetupLnsServer** should be used for installing the "Echelon LNS Server". The entry point **SetupLnsRemoteClient** should be used for installing the "Echelon LNS Remote Client".

Table 14.1 Custom Action Properties

Property	Description
DLL Filename	<PATH_TO_HELPERDLLS>_SetupLNS.dll
Function Name	SetupLnsServer
Custom Action Type	Call into an MSI DLL
Location	Stored in the Binary Table
Return Processing	Synchronous (Check exit code)
In-Script Execution	Immediate Execution
Execution Scheduling	Execute only once
Install UI Sequence	After MaintenanceWelcome
Install UI Condition	Not REMOVE

Property	Description
DLL Filename	<PATH_TO_HELPERDLLS>_SetupLNS.dll
Install Execute Sequence	<Absent from sequence>

The `_SetupLNS.dll` component provides the following features:

1. The `_SetupLNS.dll` custom actions take care of all LNS version checking that is required to install an LNS runtime of a particular version. The version that it attempts to install is based on the `_SetupLNS.dll` file version, so this DLL should be re-imported into your installation when you generate a new version of the LNS runtime installation with the LNS Redistribution Kit. For example, if the LNS Redistribution Kit is patched within a future LNS service pack in order to generate new LNS runtime installations, this DLL should also be re-imported.

2. The user may define a Windows Installer property named **LnsDbPath** to pass in the LNS network database path described in the *Using setup.exe* section, instead of writing to the Windows Registry.

3. The DLL will define a Windows Installer property named **LNSREQUIRESREBOOT** and set it to the string value "1" if the LNS runtime installation requires a reboot for completion. Note that your LNS-based product installation should either force a reboot or inform the user that a reboot is necessary. The LNS runtime installation case that requires a reboot will return a success code to the Custom Action that calls it, filtering out the ERROR_SUCCESS_REBOOT_REQUIRED error return from `MsiExec.exe`.

   Table 14.2 shows an example of using this notification from your product installation with an InstallShield DevStudio 9 sequence table entry. The ScheduleReboot standard action may be found by going to the "Sequences" item under the "Behavior and Logic" item, then expanding the actions found in the Installation Execute sequence. When this standard action is run, directly after completion of a successful installation, it informs the user that a reboot is required, and gives the choice of rebooting now or later.

Table 14.2 Using LNSREQUIRESREBOOT Example

ScheduleReboot Standard Action (Installation Execute Sequence)	
DLL Filename	`<PATH_TO_HELPERDLLS>\_SetupLNS.dll`
Sequence Number	6410
Condition	ISSCHEDULEREBOOT or LNSREQUIRESREBOOT
Comments	ScheduleReboot

In order to find the LNS runtime installation files when the Custom Action is called, the `_SetupLNS.dll` makes the following assumptions about the configuration of your installation and installation medium. It assumes that there is a Windows Installer property named **SETUPEXEDIR** defined, and that this property gives an absolute directory location on your source medium. It assumes that it will find the LNS runtime sub-installation in the directory "LNS Server" (for the Echelon LNS Server) or "LNS Remote Client" (for the Echelon LNS Remote Client) of the directory defined in **SETUPEXEDIR**. Recent versions of InstallShield set the **SETUPEXEDIR** property automatically for installations that are run from their standard setup.exe, making it this convention particularly useful for InstallShield users. If you are not using InstallShield for your Windows Installer development, this property may also be defined and set manually within your installation in order to point to the proper medium location of the LNS runtime installation.

The version 3 Microsoft XML Parser installation is also available as a merge module for Windows Installer based installations, and it should be installed separately by your installation, just as in the `setup.exe` case. The merge module, named `msxml3.msm`, is included in both InstallShield DevStudio 9 and recent versions of Visual Studio.

LNS Server and Remote Client Runtime Incompatibility

The LNS Server and LNS Remote Client runtimes are mutually exclusive, and should never be installed on the same PC at the same time. The LNS Server installation is a superset of the LNS Remote Client installation, and may safely be installed after removing any Windows Installer based LNS Remote Client installation on a PC. Likewise, the LNS Remote Client installation can be safely omitted if the Windows Installer based LNS Server installation already exists on the PC.

Neither of the LNS runtime installations currently checks for the existence of the other installation in order to force remedial action in this case, but they may be modified to provide that function in the future. Your LNS-based application installation may provide this check now if this case is likely to arise in your installation scenario.

Windows Installer and InstallShield Caveats

When embedding the new LNS Server or LNS Remote Client runtime installations into your LNS application's installation, do not use the "Nested Installation" option of Windows Installer. This will cause the LNS Server installation to become too tightly coupled to your application. It will not allow the LNS runtime installation to be patched or upgraded, or to be installed by another application on the same PC, so it is not an interoperable way to install the LNS runtime.

Windows Installer does not fully support concurrent installations, but the LNS runtime installations may be run during some portions of some types of Windows Installer based installations. For example, in InstallShield DevStudio 9, the LNS Server may be installed as a Custom Action during some portions of the UI sequence of a "Basic MSI Project." Because of the InstallShield-Windows Installer runtime architecture, the LNS Server can only be installed by an "InstallScript Windows Installer Project" as a Nested Installation. Therefore, an "InstallScript Windows Installer Project" type project may not be used to install the LNS runtime.

Chapter 15 - Advanced Topics

This chapter addresses a number of advanced topics that are not described in the previous chapters. This includes using LNS to perform file transfers, using the `OnSystemNssIdle` event, portable and remote tool development guidelines, multi-threading, avoiding memory leaks with LNS, and creating interoperable LNS client applications.

File Transfer

Each `System` object contains a `FileTransfer` object. This object represents a LonMark file transfer session involving a group of application devices. For example, you might set up a file transfer to upload log files from some of the application devices on your network. You should note that file transfer can only be performed with devices that implement the LonTalk file transfer protocol, and that implement user-files.

NOTE: Configuration property template and value files are not user-files. To access configuration property value files, use the `UploadConfigProperties()` or `DownloadConfigProperties()` methods that are described in Chapter 6 of this document.

You can use the file transfer process to write a file to a device, or to read a file from a device. Both operations require your application to be attached to the network, and for all participating devices to be commissioned and online. To read a file from a device, follow these steps:

1. Obtain a `FileTransfer` object from the `System` object's `FileTransfer` property.

   ```
   Dim MyFileTransfer as LcaFileTransfer
   Set MyFileTransfer = System.FileTransfer
   ```

   Note the `System` object's `FileTransfer` property returns `FileTransfer` objects by value, not by reference. Thus, you get a new `FileTransfer` object each time you query the property.

2. Set the properties of the `FileTransfer` object to specify the characteristics of the file transfer session. This includes the file index of the file to be transferred, the size of the buffer to receive the file, the receive and transmit time-out periods, and the starting position within the file. Note that if you set the starting position to a non-zero value, the file transfer will use random access. This is an optional feature of the LonMark file transfer protocol that is not implemented by all devices.

   ```
   MyFileTransfer.FileIndex = 2
   MyFileTransfer.ReadBufferLength = 40000
   MyFileTransfer.RxTimeOut = 0
   MyFileTransfer.TxTimeOut = 0
   MyFileTransfer.StartPosition = 0
   ```

   If you set the `RxTimeOut` and `TxTimeOut` properties to 0, LNS will calculate the values for those properties automatically, based on the network topology and channel delays on your network.

   See the *LNS Object Server Reference Help* file for a complete list of the properties of the `FileTransfer` object.

3. Select the device you want to read the file from, and call the `AddTarget()` method. Pass in the device containing the file as the `appDeviceObject` element.

   ```
   MyFileTransfer.AddTarget(appDeviceWithFile)
   ```

4. Read the file from the target device into a buffer on your application.

```
Dim fileBuffer as Variant
Dim byte1 as Byte
fileBuffer = MyFileTransfer.ReadFile()
byte1 = fileBuffer(1)
```

The `ReadFile()` method returns an array of bytes of the length transferred. Your application should query the returned Variant to determine the number of available bytes, which could be as little as 0 if an error occurred.

To write a file to a device or group of devices, follow these steps:

1. Obtain a `FileTransfer` object from the `System` object's `FileTransfer` property.

   ```
   Dim MyFileTransfer as LcaFileTransfer
   Set MyFileTransfer = System.FileTransfer
   ```

2. Set the properties of the object to specify the characteristics of the file transfer session. Random access will be used in the file transfer if the `StartPosition` property is set to a non-zero value. This requires that the device you are writing to supports random access file transfer. Sequential file access will be used if the `StartPosition` property is set to 0.

   ```
   MyFileTransfer.RxTimeOut = 0
   MyFileTransfer.TxTimeOut = 0
   MyFileTransfer.StartPosition = 0
   MyFileTransfer.FileIndex = 0
   ```

   If you set the `RxTimeOut` and `TxTimeOut` properties to 0, LNS will calculate the values for those properties automatically, based on the network topology and channel delays on your network. See the *LNS Object Server Reference Help* file for a complete list of the properties of the `FileTransfer` object.

3. Select the device you want to write the file to, and call the `AddTarget()` method. Pass in the device containing the file as the `appDeviceObject` element

   ```
   MyFileTransfer.AddTarget(appDeviceReceivingFile)
   ```

4. Write the file to the device.

   ```
   MyFileTransfer.WriteFile(fileBuffer)
   ```

NOTE: You can use this procedure to write a file buffer to more than one application device at a time, by repeating step 3 for each device to be written to. The file index written to, and the file buffer to write, must be the same for all target devices. In order to write to more than one application device at a time, an output network variable of type `SNVT_file_req` on the LNS Server's Network Service Device must be bound to the file request input network variable on each of the target devices for the file transfer. In addition, an input network variable of type `SNVT_file_status` on the LNS Server's Network Service Device must be bound to the file status output network variable on each of the target devices. If random access is used for the file transfer, then an input network variable of type SNVT_file_pos on the `NetworkServiceDevice` object of the LNS Server must be bound to the file position input NV on each of the targets. If necessary, you can add these network variables to the Network Services Device with your LNS application. See the *Creating Dynamic Network Variables* section for instructions on this.

You should also be aware that LNS does not support writing to the LonMark configuration property template or value files using file transfer. The template file

should never be updated, as this defines the application interface. The configuration property value file must be updated with `ConfigProperty` objects or with the `DownloadConfigProperties()` method to ensure that the LNS database is kept in synch with the values in the configuration value file. For more information on this, see *Downloading and Uploading Configuration Properties* on page 126.

Using the OnSystemNssIdleEvent

You can use the `OnSystemNssIdle` event to allow your application to execute code while the LNS Server is busy completing an operation. This event is useful for refreshing displays, and for providing a cancel option for operations that take a long time.

Call the `System` object's `BeginNssIdleEvent()` method to enable `OnSystemNssIdle` events. This method takes a single parameter that specifies the maximum number of milliseconds between calls to the `OnSystemNssIdle` event handler. This event will then be fired at that interval while your application is waiting for lengthy network operations to complete. Instances of this event will be returned synchronously, and if your application does not handle the event in a timely manner, then your application may hang.

This event may be useful during any lengthy operation, such as when you change the value of the `MgmtMode` property from `lcaMgmtModeDeferConfigUpdates` to `lcaMgmtModePropagateConfigUpdates`, if you are loading a device's application image, or if you are commissioning a device. The main thread of your application will need to wait for these operations to complete before moving on, so you could use this event to refresh your client application's display, so that the user knows it is not stuck. However, your application should be prepared to use this event at any time. Even a simple modification that does not normally consume a large amount of time may be held up because of a lengthy transaction started by another LNS application.

The LNS calls you can make from within the handler for this event are limited to the following. You can access the `ServiceStatus` property to determine the status of the service LNS is trying to perform. If a transaction is taking too long for LNS to execute, you can cancel it by calling the `CancelTransaction()` method from the event handler. Or, if you are performing a network recovery, you could access the `RecoveryStatus` property from the event handle to determine the status of the network recovery.

You could also use an event handler for the `OnSystemNssIdle` event to refresh your application display. For example, Visual Basic will sometimes partially erase a form while the Object Server is busy. To prevent this, the following Visual Basic example implements an `OnSystemNssIdle` event handler that calls the `Refresh()` method for the active form.

```
Private Sub lcaOS_OnSystemNssIdle()
      Screen.ActiveForm.Refresh
End Sub
```

To turn off idle events, call the `System` object's `EndNssIdleEvent()` method.

Developing Remote Tools

All LNS applications should be designed to support both local and remote operation. Since remote operations may take more time to complete than local operations, your

application should provide feedback and the ability to cancel operations. For example, when displaying a list box of devices in the system, you could provide a **Cancel** button and display an hourglass while creating the list.

When operating remotely, your application should also disable operations that are not supported remotely to prevent the user from selecting them. For example, if your application provides a network recovery command, this command should be disabled when running remotely. The *LNS Object Server Reference* help file contains a help page for every LNS object, property, method and event. Each help page indicates whether or not the object, property, method or event it describes is available on remote clients.

To improve remote performance, your application should cache frequently accessed objects such as the current system and subsystem objects, and be careful not to unnecessarily repeat object references. For example, the following statement within a loop will require repeated references to the LNS Server from a remote PC:

```
Dim MyNVs As LcaNetworkVariables
Dim MySubsystem As LcaSubsystem
Dim MyDevices As LcaAppDevices
Dim MyDevice As LcaAppDevice
Set MyDevices = MySubsystem.AppDevices
Set MyDevice = MyDevices.Item("Sunblind")
Set MyNVs = MyDevice.NetworkVariables
Set MyNV = MyNVs.Item(i)
```

Each iteration through the loop will call constructors and destructors on the LNS Server PC for the `AppDevices` collection, the `AppDevice` object, the `NetworkVariables` collection, and the `NetworkVariable` object. To significantly increase performance, fetch the `AppDevices` collection, `AppDevice` object, and `NetworkVariables` collection once outside the loop with the following statements:

```
Dim MyNVs As LcaNetworkVariables
Dim MySubsystem As LcaSubsystem
Dim MyDevices As LcaAppDevices
Dim MyDevice As LcaAppDevice
Set MyDevices = MySubsystem.AppDevices
Set MyDevice = MyDevices.Item("Sunblind")
Set MyNVs = MyDevice.NetworkVariables
```

The statement within the loop can be changed to the following, eliminating most of the constructor and destructor calls:

```
Set MyNV = MyNVs.Item(i)
```

You can subscribe to the `OnChangeEvent` event to keep your application informed of changes to the database that may affect your cached objects.

Developing Mobile Tools

If your network tool is designed to be mobile and it is used in a multi-channel network, it needs to provide special support. This is required to inform the LNS Server that your application may move, and to tell the LNS Server when your tool has moved. If your application is running on the same PC as the LNS Server, then it must inform the system that the LNS Server has moved, so that the LNS Server can update the Network Service Device's network address to be consistent with the channel it has been moved to. These issues only arise on multi-channel networks.

Registering a Mobile Application

Mobile network tools should set the `PingClass` property of their `NetworkServiceDevice` object to either `PingClassMobile` or `PingClassTemporary`. Use `PingClassMobile` if your application is a permanent part of the network (i.e. it will always be somewhere on the network), but may move from place to place. The LNS Server checks for the presence of mobile devices at the highest frequency, by default every minute.

If your application is just being attached to the network for temporary use (as may be the case for a maintenance tool), use `PingClassTemporary`. The Object Server checks for temporary devices at a slightly lower frequency, by default every two minutes.

Moving a Mobile Application to a New Channel

The steps you must take when moving a mobile application from channel to channel depend on whether it is a remote or local LNS application.

If your application is running locally, then it must tell the system that it has moved so the LNS Server can update the Network Service Device's network address to be consistent with the new channel. This is done by calling the `PreMove()` method , and then calling the `PostMove()` method. The `PostMove()` method must be invoked *after* the device has physically been moved.

If your application is running as a remote Full client, you should not use the `PreMove()` and `PostMove()` methods in this fashion. This is because the application would not be able to invoke the `PostMove()` method after it has moved, since intervening routers may not pass on the messages in most cases. Instead, the application should re-attach itself to the network by closing the system and network, and then re-opening the network and system after the device has physically been moved. During the open, the LNS Server will implicitly invoke the `PreMove` and `PostMove` methods to move the NSI along with all of its connections to the new channel.

If your application is running as a remote Lightweight client, there is no need to do anything when moving the application. Since the Lightweight client uses the same Network Service Device as the LNS Server, its location is irrelevant to LNS.

Multi-Threading and LNS Applications

LNS uses Single-Threaded Apartments (STA) only. It does not support Multi-Threaded Apartments. This means that client threads that access LNS are created as STA threads only. If you are developing an LNS application using Visual C++ or Microsoft Visual Studio .NET, you must create the threads that will access LNS in a Single Threaded Apartment.

If you are using Visual C++ or Microsoft Visual Studio .NET and want to write an application that will use multiple threads to access LNS, each thread must initialize COM as an STA thread. You will need to marshal the object references received from LNS in one thread to any other threads the application is using.

If you are using Visual C++ or Microsoft Visual Studio .NET, you can set the `lcaFlagsDirectCallback` flag in the `Flags` property of the Object Server to use direct callback within your application. When using direct callbacks, all event handling

functions will be executed by an internal LNS thread, as opposed to the thread that instantiated the Object Server. While it is usually more efficient to use direct callback, especially when writing a monitor and control application, or an application that you expect to receive a large number of events, you should be aware that this turns the application into a multi-threaded application. This means that these handler functions must not call back into LNS, and the client application must be written to properly deal with multiple threads executing concurrently. Note that this applies to all events, not just monitoring events.

For more specific information on multi-threading with COM, consult the documentation for the development environment you are using with LNS.

Avoiding Memory Leaks with LNS

If you are using a Visual C++ based development environment to create an LNS application that receives LNS events, you need to make special considerations in some situations to avoid memory leaks. Due to Microsoft's COM referencing standards, objects returned as elements in an event are owned by LNS, not by the application that received the event. As a result, the client application needs to increment the reference count for each object received in an event handler in order to keep a copy of that object after the LNS event handler has returned.

Before discussing objects passed by event handlers further, consider how objects that are returned when you invoke an LNS method or access an LNS property are handled. Objects returned by properties and methods are owned by the client application, and LNS increments the reference count for these objects automatically as copies of the objects are created and destroyed. Although LNS increments the object's reference count before passing it to the client application, it is up to the client to call `Release()` on the object before removing it from memory. This decrements the reference count. Different COM client wrappers such as MFC and ATL may handle this differently.

It is important to realize that how objects are freed after being passed through the event mechanism is different than for objects returned through method invocation or property access. LNS does not increment the reference count for objects passed to an application via events. If you want your application to keep an object returned via an event longer than the duration of the event-handling function, you need to call `AddRef()` on that object, so that the object will not be removed from memory when the event-handler exits. You must also call `Release()` when you are done with the object, to ensure that it is properly removed from memory.

All client applications must follow COM reference count rules and properly manage their reference counts in this fashion, so that LNS knows when to appropriately destroy an object. Failure to properly call `AddRef()` when copying an object returned by an event will cause access violations in the client application, as it will hold a pointer to an object that has already been deleted. Failure to properly call `Release()` when removing an object from memory that was returned by an event will result in memory leaks in the client application or in LNS.

The Visual Basic runtime environment manages object reference counts automatically, so you do not need to be concerned with memory leaks as a result of receiving LNS events if you are using Visual Basic. For more information on COM reference count rules, consult the Visual Studio documentation.

NOTE: LNS uses an optimized memory allocator that manages its own local heap. This allocator holds most memory that it allocates for future re-use. Therefore, early in an application's run-time, memory usage may continually climb. However, it will eventually reach a point of stabilization. At this point, the memory allocator will have enough memory in its pool so that it does not need to request more memory from the system. Memory freed internally is kept by the allocator, and then re-used.

Debugging LNS Applications

This section describes a few considerations you should make when debugging an LNS application. If a transaction is interrupted prior to completion for any reason, the LNS databases and the network are returned to their states prior to the start of the transaction. If an application starts a transaction, then it should commit or cancel the transaction within a reasonable time to avoid hanging other applications. No other applications can start a new transaction until the current one is committed or cancelled. While a transaction is in progress, LNS automatically queues requests to start either additional implicit or explicit transactions. If an application shuts down while it has an outstanding transaction, and another application attempts to start a transaction, the LNS Object Server will cancel the abandoned transaction within 30 seconds. Once a transaction is committed or canceled, other transactions on the system can begin.

This behavior can cause problems when debugging your application. For example, LNS may cancel a transaction because your application has started a transaction and is halted in a breakpoint. You can disable this behavior by setting the following Windows Registry DWORD entry to a non-zero value:

```
HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\NSS\Configuration\Transaction
Debugging
```

Set its value to 0 (zero), or remove the entry, to disable the transaction debugging mode. You should only modify this Windows Registry entry for debugging purposes, since disabling this feature may cause all LNS clients to lock up if the value is changed, and an application running a transaction is improperly terminated. For more information on transactions, see *Using Transactions and Sessions* on page 65.

You should also note that if your LNS application is abnormally terminated (due to a crash or terminated by the debugger), there are some LNS processes that may be left running. A few examples are `NSSENG.EXE`, `LCAMON.EXE`, `and PTSERV.EXE`. It could take some of these processes up to one minute to finish after the application abnormally terminates. Restarting your application before these processes have had a chance to shutdown may cause your application to crash or hang. However, Echelon recommends that you do not terminate these processes manually. You should allow them to finish whenever possible. Note that you may have to reboot your PC in case a subsequent start of your LNS application fails.

LNS and Line-Safe Expressions

Programmers sometimes combine several expressions in one line of code, as opposed to explicitly assigning one object variable and performing one method invocation for each part of an operation. For example, to fetch an `AppDevice` object named “Buzzer” from a `Subsystem` object with the subsystem path “Shared Devices.Alarms”, starting from a given `System` object, the step-by-step approach is as follows:

```
Dim MySubsystems As LcaSubsystems
Dim MySubsystem As LcaSubsystem
Dim MyDevices As LcaAppDevices
Dim MyBuzzer As LcaAppDevice

Set MySubsystems = MySystem.Subsystems
Set MySubsystem = MySubsystems.Item("Shared Devices.Alarms")
Set MyDevices = MySubsystem.AppDevices
Set MyBuzzer = MyDevices.Item("Buzzer")
```

The following example combines expressions to retrieve the "Alarm Buzzer" `AppDevice` object with a single line of code:

```
Dim MyBuzzer As LcaAppDevice
Set MyBuzzer = MySystem.Subsystems.Item("Shared Devices.Alarms")._
       AppDevices.Item("Buzzer")
```

Echelon does not recommend using the abbreviated syntax for reasons of quality, performance, and type-safety. When an application accesses a property of an LNS object that is declared with its known type, such as `LcaAppDevice`, the runtime system can access that property quickly and efficiently. This access method is known as *early binding*, and means that the name of the property is translated into the property's numerical identifier at compilation time rather than at runtime.

Because early binding translates names into identifiers at compilation time, the accuracy of a property or method name can be approved at the same time. The following example, invoking a non-existent `Tast()` method on an AppDevice object, will therefore not compile:

```
Dim MyBuzzer As LcaAppDevice
Set MyBuzzer = ....
MyBuzzer.Tast()
```

However, when the compiler evaluates a daisy-chained expression like the one shown above, the compiler must return to the inferior method for the intermediate steps. This method is called *late binding,* and means that the property and method names will be translated into their respective numerical identifiers at runtime. Examine the following single-line expression in detail:

```
Set MyBuzzer = MySystem.Subsystems.Item("Shared Devices.Alarms")._
     AppDevices.Item("Buzzer")
```

The `MySystem` variable is declared as `LcaSystem`, and its `Subsystem` property can therefore be approved at compile-time. However, the COM interfaces that are used with all LNS objects handle references to all types of LNS objects using references to COM's generic IDispatch interface. When the second part of the line-safer expression is evaluated ("`MySystem.Subsystems.Item(...)`"), access to the `Item` property must therefore use late binding. As a result, the following code example, containing deliberate typing errors, will compile correctly, and will only fail if the offending line of source code is being executed.

```
Set MyBuzzer = MySystem.Subsystems.Ixyz("Shared Devices.Alarms").Devices.Get("Buzzer")
```

Although none of the `Ixyz`, `Devices`, or `Get` properties and methods exists, this line of code will compile correctly. If, on the other hand, each assignment is made step-by-step, the incorrect property and method names will fail evaluation at compile time.

Echelon strongly recommends the exclusive use of early binding techniques. Note that LNS director applications may have to use late binding strategies when accessing LNS plug-in software. See the *Implementing an LNS Director Application* section on page 280 for more information on this.

The shorthand using a combined subsystem path such as "`Shared Devices.Alarm`" as shown in the above example is supported directly by the `LcaSubsystems` object. Echelon recommends using a complete subsystem path when known, as opposed to iterating several levels of `Subsystem` objects. Using a subsystem path leads to shorter and faster code, with reduced risk of programming error.

Finally, another frequently used line safer is the use of default properties. While not supported by all development tools, Visual Basic is a popular example of a development tool supporting default properties. While not recommended (as detailed above), the single-line expression to obtain the Buzzer device could be further abbreviated by removing the `Item` properties. The following example will, although not being recommended, still compile and execute correctly.

```
Set MyBuzzer = MySystem.Subsystems("Shared _
Devices.Alarms").AppDevices("Buzzer")
```

The `Item` properties could be removed because this particular property is flagged as the default for the object, and will be automatically queried if no property or method name is explicitly given. Echelon does not recommend using default properties in the interest of source code maintenance. Future developers of your LNS application will have to know about default properties, and which property has the default flag for each particular LNS object, in order to efficiently maintain and enhance the source code. Explicitly stating the correct type, property or method names will make your application's code more self-explanatory and easier to maintain.

LNS and Internet Information Services

When using Microsoft's Internet Information Services (IIS) to connect to an LNS network, there are several things you should consider. Because IIS serves HTTP pages, and HTTP is a stateless protocol, IIS will completely disconnect from the LNS Server after every page is served. This means that the LNS Server must close and restart for each LNS function call you make, which would cause each operation to take a considerable amount of time. As a result, you should use the LNS Server application to provide remote access to the network, and then use a Lightweight client application to perform all operations on the network. By using the LNS Server application to provide remote access to the network, the requirement to connect and reconnect to the LNS Server is eliminated, and replaced by a quick socket connection. This will greatly improve the performance of your application. In addition, this method allows you to run IIS and the LNS Server application on different machines.

Appendix A - Deprecated Methods and Obsolete Files

This appendix lists LNS methods, properties and objects that should no longer be used in LNS Turbo Edition. It also lists files installed with previous versions of LNS that are no longer used, but not removed by the LNS Turbo Edition installation.

Deprecated Methods, Objects, Properties and Events

As of LNS Turbo Edition, some methods, objects, properties and events have been deprecated. This section provides a list of those items. Some have been deprecated because they were never implemented in LNS, or because they are no longer applicable or useful. Others have been deprecated because they have been replaced by new features with better functionality.

Note that many of the objects, methods, properties and events marked as deprecated in the documentation are still implemented in LNS Turbo Edition to maintain backwards-compatibility with applications running on previous versions of LNS. For example, many of the methods of the `ConfigProperty` object have been deprecated as a result of the new data point feature, including the `GetElement()`, `GetElementFromDevice()`, `SetElement()`, and `SetElementFromDevice()` methods. You can still successfully use these methods when running on LNS Turbo Edition. However, Echelon recommends that you use the `DataPoint` object to read and write configuration properties, and so these methods have been marked deprecated.

The following sections list the deprecated objects, properties, methods and events in LNS Turbo Edition. The reason for deprecation can be determined using the following codes:

BA – Better Feature Available. The feature is still implemented in LNS for compatibility purposes, but a more efficient way to achieve its purpose exists.
U – Unimplemented. The feature was never implemented in LNS.
NLA – No Longer Applicable. The feature is still implemented in LNS for compatibility purposes, but is no longer useful or functional because of other changes to the LNS implementation.

For more extensive details on why each of the items in the following list have been deprecated (i.e. what feature to use for those marked BA), consult the *LNS Object Server Reference* help file.

Deprecated Objects

The following objects have been deprecated for Turbo Edition. This means that you should no longer use the object, or any of its properties and methods, in your applications.

`BuildTemplate` Object **(U)**
`BuildTemplates` Object **(U)**
`HardwareTemplate` Object **(U)**
`HardwareTemplates` Object **(U)**
`NetworkVariableField` Object **(BA)**
`ProgramTemplate` Object **(U)**
`ProgramTemplates` Object **(U)**

Deprecated Methods

The following methods have been deprecated for Turbo Edition. Note that the objects these methods apply to have not been deprecated, unless those objects are listed in the *Deprecated Objects* section.

`Build` Method **(U)**
`CloseComponent` Method **(U)**
`Export` Method (`AppDevice` and `DeviceTemplate` Objects) **(U)**
`GetElement` Method **(BA)**
`GetElementFromDevice` Method **(BA)**
`GetField` Method (`NetworkVariable` Object) **(BA)**
`GetRawValues` Method **(BA)**
`GetRawValuesFromDevice` Method **(BA)**
`Link Method` **(U)**
`Lock` Method **(U)**
`OpenComponent` Method **(U)**
`Purge` Method **(BA)**
`RecoverFromNssDb` Method **(U)**
`Remove` Method (`Networks` Collection) **(BA)**
`SetCapacity` Method **(NLA)**
`SetElement` Method **(BA)**
`SetElementFromDevice` Method **(BA)**
`SetRawValues` Method **(BA)**
`SetRawValuesFromDevice` Method **(BA)**
`Unlock` Method **(U)**

Deprecated Properties

The following properties have been deprecated for Turbo Edition. Note that the objects these properties apply to have not been deprecated, unless those objects listed in the *Deprecated Objects* section.

`ActiveXComponent` Property **(U)**
`BuildStatus` Property **(U)**
`BuildTemplate` Property **(U)**
`BuildTemplates` Property **(U)**
`CompatibleNv` Property **(BA)**
`ComplementaryNv` Property **(BA)**
`ConnErrNvMtIndex1` Property **(BA)**
`ConnErrNvMtIndex2` Property **(BA)**
`DataServerHandle` Property **(NLA)**
`DataServerObjectHandle` Property **(NLA)**
`DsAuthenticate` Property **(BA)**
`DsAutoUpdate` Property **(U)**
`DsEventSubscription` Property **(NLA)**
`DsFormatFilesPath` Property **(NLA)**
`DsMessageOwner` Property **(NLA)**
`DsMode` Property **(NLA)**
`DsMonitorTag` Property **(BA)**
`DsPause` Property **(NLA)**
`DsPrecision` Property **(BA)**
`DsPriority` Property (`NetworkVariable` and `NetworkVariableField` objects) **(BA)**
`DsReportByException` Property **(BA)**
`DsRetries` Property (`NetworkVariable` and `NetworkVariableField` objects) **(BA)**

`DsService` Property **(BA)**
`DsUseBoundUpdates` Property **(BA)**
`DynamicNvPersistenceMode` Property **(BA)**
`ExportDirectory` Property **(U)**
`ExportFormat` Property **(U)**
`FormatLocale` Property **(BA)**
`GraphicsDirectory` Property **(U)**
`HardwareTemplate` Property **(U)**
`HardwareTemplates` Property **(U)**
`LockDuration` Property **(U)**
`ProgramTemplate` Property **(U)**
`ProgramTemplates` Property **(U)**
`RawValue` Property (`ConfigProperty` Object) **(BA)**
`RawValueFromDevice` Property (`ConfigProperty` Object) **(BA)**
`RemoteIgnorePendingUpdate` Property **(BA)**
`SingleUserMode` Property **(NLA)**
`TypeDefaultValue` Property **(BA)**
`Value` Property (`NetworkVariable`, `NetworkVariableField`, `ConfigProperty` Objects) **(BA)**
`ValueFromDevice` Property **(BA)**

Deprecated Events

The following events have been deprecated for Turbo Edition.

`OnBuildMessage` Event **(U)**
`OnNetworkServiceDeviceReset` Event **(NLA)**

Obsolete Files

If you are upgrading to LNS Turbo Edition from a previous version of LNS, the installer will leave several obsolete files on your PC. Table A.1. lists the obsolete files left by the LNS runtime installation (i.e. Echelon LNS utilities), and Table A.2 lists the obsolete files left by the LNS Application Developer's Kit installation.

Note that in the following tables, `<LNSDIR>` represents the target directory you have chosen to install LNS to. By default, this is the `[Windows Drive]\LonWorks` directory.

Table A.1 Obsolete Runtime Files

Installation Directory	File Name
<LNSDIR>\bin	Typfilr.exe, Typfilw.exe, Drfusrnv.exe
<LNSDIR>\bin	Xif2to3.exe, xif2to3.txt

Table A.2 Obsolete Application Developer's Kit Files

Installation Directory	File Name
<LNSDIR>\import	Actuator.xif, Actuator.xfb, analog.h, anlgsnsr.apb, anlgsnsr.nc, anlgsnsr.xif, display.h, rtclock.h, scptanlg.apb, scptanlg.nc, scptanlg.xfb, scptanlg.xif, sensor.xif, sensor.xfb, temperat.h
<LNSDIR>\DataServer	Include\dsv.h, Include\LnsDsError.h, Lib\lcadatsv.lib
<LNSDIR>\NetworkServices\include	Neuron.h, ni.h, ni_app.h, ni_callb.h, ni_comm.h, ni_mgmt.h, ni_nmexp.h, ni_nmext.h, ni_slta.h, ni_snvt.h, ns.h, ns_apb.h, ns_event.h, ns_lic.h, ns_msg.h, ns_nativ.h, ns_nmc.h, ns_opts.h, ns_plat.h, ns_props.h, ns_reg.h, ns_spprt.h, ns_srsts.h, ns_srvc.h, ns_xifb.h, nse.h, nse_db.h, nse_dba.h, nse_exa.h, nse_idb.h, nse_int.h, nse_iseq.h, nse_mio.h, nse_seq.h, nse_snd.h, nse_srvc.h, nsx.h, nsx_def.h, nsx_host.h, nsx_prop.h, nsx_qnss.h, nsx_user.h, nsxstdxr.h, platform.h
<LNSDIR>\ObjectServer\include	Lca.h, lca_errs.h, lcaobjsv_i.h, lcaobjsv_i.c
<LNSDIR>\ObjectServer\include\MFC Class Wrappers	Lcaobjsv.h, lcaobjsv3.h, lcaobjsv.cpp
<LNSDIR>\ObjectServer\include	Version.h
<LNSDIR>\ObjectServer\Examples	Sub-directories: LNSexample, LNSexample-VC++, LNSlmapplet, LNSmcapplet, LonMarkDevices, QuickStart
<LNSDIR>\types\include	lcadrf.h, lcadrfw.h
<LNSDIR>\types\lib	lcadrf32.lib, ldrf32r.lib
<LNSDIR>\types\source	Cacheutl.h, cacheutl.c, drfbase.c, drfcat.c, drffpt.c, drfgen.c, drflang.c, drftype.c, drftypt.c, lcadrfi.h, lcadrfiw.h

Appendix B – LNS, MFC and ATL

This section provides information you may find useful when using LNS with MFC and ATL.

LNS, MFC and ATL

Versions 6.0 and higher of Microsoft Visual C++ offer two ways to import the interfaces exposed by the LNS Object Server ActiveX Control into an existing project: the MFC Class Wizard, and the ATL `#import` statement described in Chapter 4 of this document.

As of LNS Turbo Edition, the legacy MFC wrappers, in the files `lcaobjsv.h`, `lcaobjsv3.h`, and `lcaobjsv.cpp`, are no longer included with the LNS Application Developer's Kit. They required the use of many other header files to include LNS constants, and were difficult to keep synchronized with the LNS Object Server type library constants. Instead of using the MFC wrappers, the ATL `#import` statement described in Chapter 4 of this document should be included in a global header file to expose the full set of objects, properties, methods, and constants included with LNS.

The ATL `#import` statement provides a wrapper that is superior to the MFC wrappers. The wrapper uses ATL smart pointers to directly call the functions of the LNS Object Server. In addition, the wrappers generated by the ATL `#import` statement automatically regenerate whenever the LNS Object Server is updated (i.e. when service packs or new versions of the software are released). This is unlike the legacy MFC wrappers, which must be carefully maintained whenever the LNS Object Server is updated.

NOTE: Although it is not recommended, you can still generate the `lcaobjsv.h`, `lcaobjsv3.h`, and `lcaobjsv.cpp` files manually, and use the wrapper files generated by the MFC Class Wizard instead of the new `#import` statement. For more information on this, see the *Generating the Legacy MFC Class Wrapper Files* section later in this appendix.

You can apply a combination of the two approaches to take advantage of all available benefits. You can use the wrappers generated by the MFC Class Wizard to handle events, and use the wrappers generated by the `#import` statement to implement the objects, methods and properties of the LNS Object Server. To do so, follow the steps described in Chapter 4 to import the LNS Object Server Active X Control. Then, to utilize the smart pointer for the Object Server interface, use the following assignment statement. In this statement, the m_ObjectServer variable was declared with code generated by the MFC Class Wizard:

```
Header:
#import "lcaobjsv.ocx" rename_namespace("lns")

Implementation:

lns::_DLcaObjectServerPtr objectServer = m_ObjectServer.GetControlUnknown();

Then you'll need to use the ATL smart-pointers (i.e. lns::ILcaNetworksPtr for the
Networks collection):

lns::ILcaNetworksPtr networks = objectServer->Networks;
lns::ILcaNetworkPtr  network  = networks->Item["mynetwork"];
```

To only use the ATL wrappers, you need to manually add the code to create an instance of the LNS Object Server to your application, as well as the framework for the event handlers. The following example code shows how you could create an instance of the LNS

Object Server and also advise for LNS Object Server events. This code is included in the LNSMonitorCtrl.cpp file included with the LNS Turbo Edition examples suite.

```
HRESULT CLNSMonitorCtrlApp::OpenLca()

{

        HRESULT hResult = S_OK;

        hResult = m_pObjectServer.CreateInstance (_T("LonWorksObjectServer.1"));

        if (hResult != S_OK)

        {

                // failed to create object server instance

                m_IDS = IDS_FAILED_OBJSERVER;

                return hResult;

        }

        // Creating ObjectServer Event Handler...

        m_pEventSink = new CEventSink();

        if (m_pEventSink == NULL)

        {

                // failed to create event sink

                m_IDS  = IDS_FAILED_EVENT_HANDLER;

                return E_FAIL;

        }

        // Route the ObjectServer events to handlers in our EventSink class

        hResult = m_pEventSink->DispEventAdvise(IUnknownPtr(m_pObjectServer));

        return hResult;

}
```

The CEventSink class is based on IDispEventSimpleImpl, and is used to handle LNS Object Server events. This section briefly describes how the class was constructed. See the EventSink.h file included with the LNS Turbo Edition example application suite for complete source code, and reference the Microsoft Developer's Network documentation for more information on IDispEventSimpleImpl.

The event handler prototypes and the event handler information of type ATL_FUNC_INFO for the SINK_ENTRY_INFO macros were created within the EventSink.h file using the _com_dispatch_method calls:

```
#include "LNSMonitorCtrl.h"
static _ATL_FUNC_INFO OnNvMonitorPointEventInfo = {CC_STDCALL,
                                                   VT_EMPTY, 2,
                                                   {VT_DISPATCH, VT_I2}};
static _ATL_FUNC_INFO OnMsgMonitorPointEventInfo = {CC_STDCALL,
                                                   VT_EMPTY, 2,
                                                   {VT_DISPATCH, VT_I2}};

BEGIN_SINK_MAP(CEventSink)
SINK_ENTRY_INFO(IDC_EVENTS,
                lca::DIID__DLcaObjectServerEvents,
```

```
                lca::lcaEventIdNvMonitorPointEvent,
                OnNvMonitorPointEvent,
                &OnNvMonitorPointEventInfo)
SINK_ENTRY_INFO(IDC_EVENTS,
                lca::DIID__DLcaObjectServerEvents,
                lca::lcaEventIdMsgMonitorPointEvent,
                OnMsgMonitorPointEvent,
                &OnMsgMonitorPointEventInfo)
END_SINK_MAP()
```

The wrapper function in `lcaobjsv.tli` provides the necessary fields to fill in the ATL_FUNC_INFO structure:

```
#pragma implementation_key(16)
inline HRESULT lca::_DLcaObjectServerEvents::OnNvMonitorPointEvent(
                    IDispatch * MonitorPoint,
                    short EventType)
{
    HRESULT _result;
    _com_dispatch_method(this,
                         0x13,
                         DISPATCH_METHOD,
                         VT_ERROR,
                         (void*)&_result,
                         {VT_DISPATCH, VT_I2},
                         MonitorPoint,
                         EventType);
    return _result;
}
```

The first 2 parameters will always be `CC_STDCALL` and `VT_EMPTY` to indicate the necessary calling convention and a `VOID` return type. The 3rd parameter specifies number of parameters the event handler received. For example, in the case of the `OnNvMonitorPointEvent` event, the value is 2, for the `MonitorPoint` and `EventType` parameters. Finally, an array of parameter sizes in bytes is listed within {}.

The remaining piece of information necessary from the `lcaobjsv.tli` file is the DISPID specified within the implementation_key macro for the event handler wrapper in the `lcaobjsv.tli` file. This value is used as the third parameter of the SINK_ENTRY_INFO macro. The value is also defined within the lcaobjsv.tli file as the `lcaEventIdNvMonitorPointEvent` constant.

Finally, the actual event handler declaration for `OnNvMonitorPointEvent` event would be:

```
void __stdcall OnNvMonitorPointEvent(LPDISPATCH MonitorPoint,
      short EventType)

{

}
```

Generating the Legacy MFC Class Wrapper Files

This section describes how to generate `the lcaobjsv.h, lcaobjsv3.h`, and `lcaobjsv.cpp` files using Microsoft Visual C++ 6.0. To do so, follow these steps:

1. Open the Microsoft Visual C++ 6.0 development environment. From the **File** menu, select **New.**

2. Select the **Projects** tab on the dialog that opens, and select **MFC AppWizard (exe)**. Then, enter a project name and verify that the **Create a new workspace** option is selected. Click **OK** to continue.

3. This opens the **MFC AppWizrd – Step 1** dialog. Select **Dialog based** as the type of application, and click **Finish.**

4. This opens the **New Project Information** dialog. Click **OK.** At this point, an empty dialog with **OK** and **Cancel** buttons opens.

5. From the **Project** menu, select the **Add to project** sub-menu and then select **Components and Controls.** This opens the **Components and Controls Gallery** dialog.

6. Double-click **Registered ActiveX Controls,** and then select **LonWorksObjectServer.** The **Filename** field will be set to LONWORKSObjectServer.lnk, and the **Path to control** field will usually be set to `c:\lonworks\bin\lcaobjsv.ocx`.

7. Click **Insert** and then click **OK** on the confirmation dialog.

8. This opens the **Confirm Classes** dialog. Verify that the **Class name** field is set to `CLcaObjectServer`. Then, change the **Header File** field from `LcaObjectServer.h` to `LcaObjSv3.h` and change the **Implementation File** field from `LcaObjectServer.cpp` to `LcaObjSv.cpp`. Press **OK** to return to the **Components and Controls Gallery** dialog.

9. Click **Close** to close the dialog. Now, the Toolbox dialog of C++ 6.0 has an extra button at end of list of controls called **LonWorksObjectServer**, as shown below:

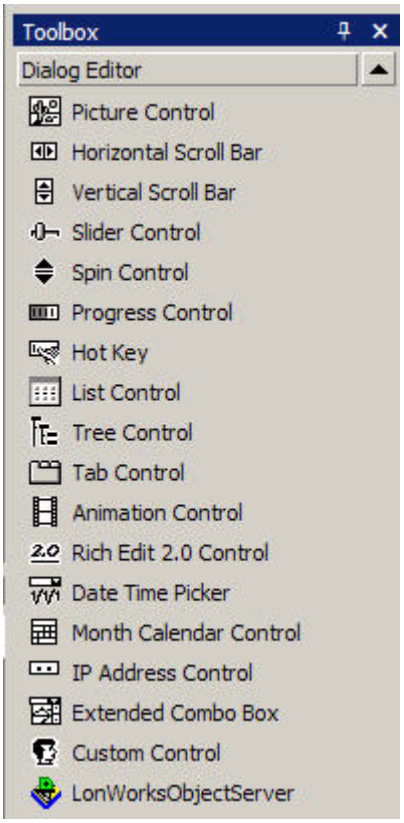


Figure B.1 Visual C++ Toolbox

Drag this new icon anywhere onto the empty dialog mentioned in step 4.

10. Right-click the new icon, and select **Class Wizard** from the pop-up menu that appears. This opens the **MFC Class Wizard** dialog.

11. Select the **Message Maps** page. Then open the **Add Class...** drop-down list and select the **From a type library...** submenu. This opens the **Import from Type Library** dialog. Enter `c:\lonworks\bin\lcaobjsv.ocx` (or whatever the **Path to control** field was set to in step 6) and click **Open.**

12. This opens the **Confirm Classes** dialog. Select all LCA class names, and then deselect the five LCA class names that start with an underscore character, such as `_DLcaObjectServerEvents`. Then, change the **Header File** field from `lcaobjsv.h` to `lcaobjsv3.h`. Press the **OK** button to return to the **MFC Class Wizard** dialog.

13. Optionally, add any desired event handlers.

14. Optionally, select the **Member Variables** page, and add a variable to represent the control ID `IDC_11`.

15. On the **MFC Class Wizard** dialog, click **OK.** The `LcaObjSv3.h` and `LcaObjSv.cpp` files have now been successfully added to the project.

Next, you need to edit the two files to maintain backwards compatibility. To do so, follow these steps:

1. Open the `lcaobjsv3.h` file. From the **Edit** menu, select **Replace.** In the **Find what** field, enter "ILca." Note that the first two letters are capitalized. In the **Replace with** field, enter "Lca." Note that the first letter is capitalized.
2. Click **Replace all**, and then save the changes to the file.
3. Open the `LcaObjSv.cpp` file, and make the same change to this file as described in steps 1 and 2. The class wrappers are now complete, with all LNS Turbo Edition features available.

Finally, you need to add the following statements to your project before the start of your code. Note that you need to reference the complete path of the `lcaobjsv.ocx` file in this statement. This should match the setting of the **Path to control** field in step 6.

```
#import "c:\lonworks\bin\lcaobjsv.ocx" rename_namespace("lns")
using namespace lns;
```

Appendix C – LNS Turbo Edition Example Application Suite

This appendix provides an overview of the example applications included with LNS Turbo Edition. This includes an example network management application, an example monitor and control application, and an example director application.

LNS Turbo Edition Example Application Suite

LNS Turbo Edition features several example applications you will find useful when developing your own applications. This appendix provides an overview of each application. In addition, each application includes comments that provide more specific detail on the example code.

The LNS Turbo Edition example application suite includes the following:

- *Network Management Example*
- *Monitor and Control Example*
- *xDriver Example Applications*
- *Example Director Application*

These example applications demonstrate a number of key LNS features and concepts. The network management and monitor and control examples are C++ applications that create an instance of the LNS Object Server, and uses wrappers generated by the #import statement to implement the objects.

The example applications and all supporting files can be found in the `<LNSDIR>\ObjectServer\Examples` directory, where `<LNSDIR>` represents the target directory you have chosen to install LNS to. Each application has its own respective folders within this directory.

Network Management Example

The network management example application supports operation as a Local, Lightweight, or Full client, and utilizes a variety of LNS features for network management that you can use to build and configure a network.

Device .XIF, .APB and .XFB files used by the network management application are located in the `<LNSDIR>\ObjectServer\Examples\Import` directory. For initial viewing of the network management example application, Echelon recommends using the examples on a network consisting of 3 LTM-10A devices and an *i*.LON 600 LONWORKS/IP Server. Gizmo 4 I/O boards are not necessary as the I/O is limited to the exposed LED on I/O 0 and pushbutton on I/O4. If you are not using LTM-10A device/s, rebuild the .nc files in the `<LNSDIR>\ObjectServer\Examples\NcSource` directory. Replace the .XIF, .APB and .XFB files of the corresponding device in the `Import` subfolder.

The following sections describe the features encompassed by the example network management application. To start the network management example application, select **Echelon LNS Application Developer's Kit>Examples & Tutorials>LNS Network Management Example** from the Windows Programs menu. This opens the dialog shown in Figure C.1.

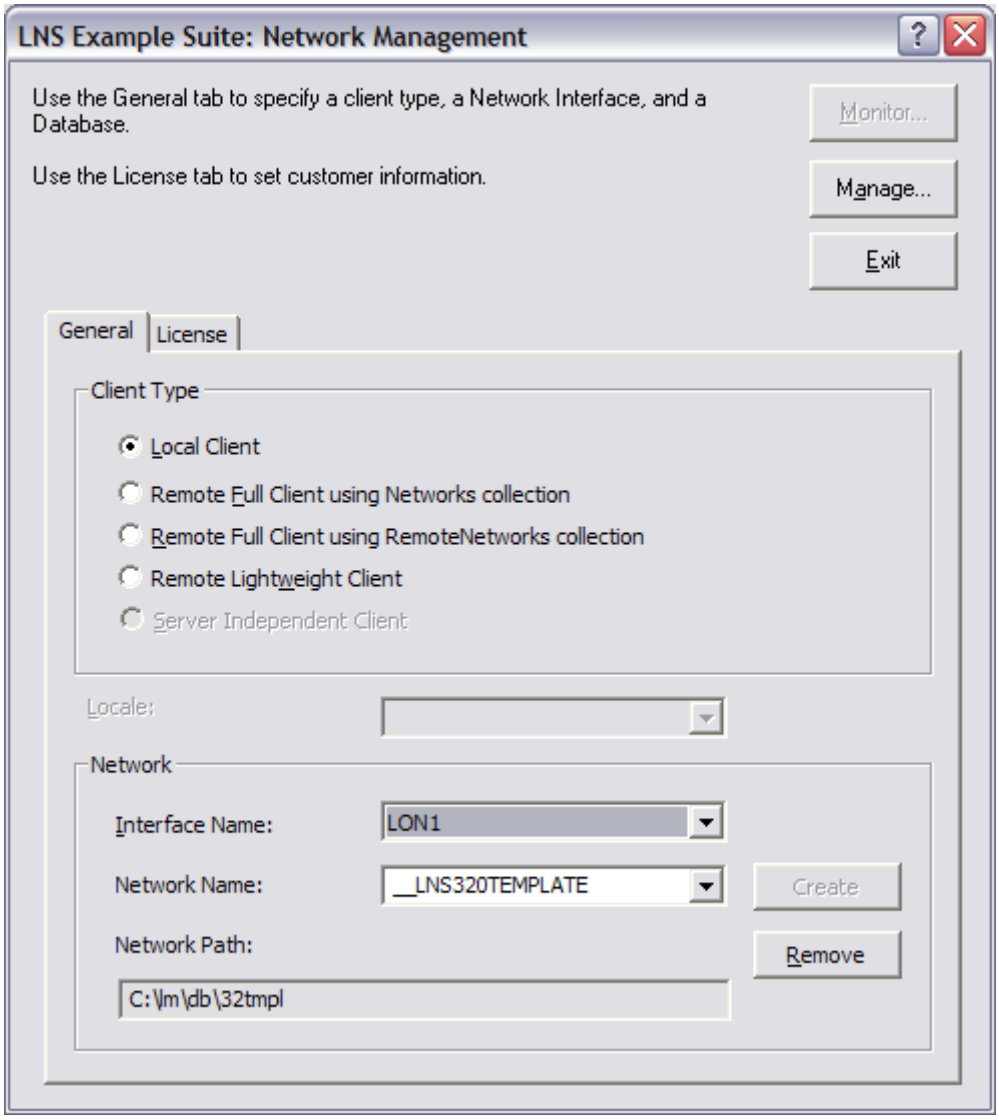


Figure C.1 Example Network Management Application

The following sections describe how you can use this dialog, and reference the location of the source code you can view to see how the tasks you perform are accomplished using LNS.

Initializing a Network

There are several steps you need to take when opening a network with an LNS application, as described in Chapter 4 of this document. To initialize the example network management application, follow these steps from the dialog shown in Figure C.1:

1. From the **General** tab, select an LNS client type. The different client types are introduced in Chapter 3 of this document. When you select a client type, the application will initialize the LNS Object Server. For source code information on this task, see `CLcaHelper::OpenObjectServer` in the `LcaHelper.cpp` file.

In addition, the **Network Name** drop-down box will list all networks accessible by your application as soon as you select a client type. For source code information, see `CDlgInitializeGeneral::SelectClientType` in the `dlginitializegeneral.cpp` file. The client type is passed into `m_iClientType`. The description of the monitor and control example later in this appendix provides more details on each of the client types you can select.

2. From the **Interface Name** drop-down box, select a network interface. This drop-down box lists all configured LONWORKS network interfaces available on your system (i.e. all the network interfaces included in the NetworkInterfaces collection). For initialization source code information, see `CDlgInitializeGeneral::InitNetworkInterfaceName()` in the `dlginitializegeneral.cpp` file.

3. From the **Network Name** drop-down box, select an already existing network from the drop-down list or create a new network by typing in the name of the new network. Although this lists all networks known to the LNS Object Server (i.e. the `Network` collection), you should note that the network management example can only manage networks created with the example application. For source code information, see `LcaHelper::GetNetworks(void)`. For selection and initialization steps see `CDlgInitializationGeneral::InitDatabaseName()` from the `dlginitializegeneral.cpp` file.

4. Once you have selected a client type, a network interface and a network, you need to specify the licensing mode, as described in Chapter 4 of this document. To select the licensing mode, select the **License** tab on the dialog shown in Figure C.1. You will then be able to select either Standard Mode or Demonstration Mode. If you select Standard Mode, you will need to enter the customer ID and customer key.

5. Click **Set Now** to save your changes. This will reinitialize the LNS Object Server, if necessary. For source code information on this, see `OpenObjectServer` in the `LcaHelper.cpp` file.

6. Now, select the **General** tab to return to the dialog shown in Figure C.1. If you selected a new network in step 3, click the **Create** button (or the **Add** button if you are running as a Lightweight client). This creates an example network if you are running as a LNS Full client using the `Networks` collection, or adds an entry to the global database if you are running as a Lightweight client.

   For Local clients, or Full clients using the `RemoteNetworks` collection, a "template network" is created called __LNS320TEMPLATE that will be used as a basis for all future networks. This network is created in the `<LNSDIR>\lm\db\lns32` folder, where `<LNSDIR>` represents the target directory you have chosen to install LNS to. The template network contains device and connection description templates. To generate an example network, a backup of this "template network" is created and then imported with the network name you had specified. The network is first opened with the system management mode set to `lcaMgmtModeDeferConfigUpdates` to prevent the propagation of

device configuration updates to the network. This allows devices, dynamic network variables, and network variable connections to be quickly created in the network database.

The **Network Path** text box displays the current network's `DatabasePath` property. It is read-only before the database name string is edited, and after a new database name is selected. Otherwise, it is possible to alter the path name while the Object Server is initialized as a local client.

In future session with the network management example, you can manage previously created example networks by clicking the **Manage** button. This opens the network, with the system management mode set to propagate all network configuration updates.

Performing Network Management Tasks

Once you have initialized the example network management application, you can perform network management tasks with the application. To do so, select the **Manage** button on the main dialog shown in Figure C.1. This will open a dialog similar to the one shown in Figure C.2:

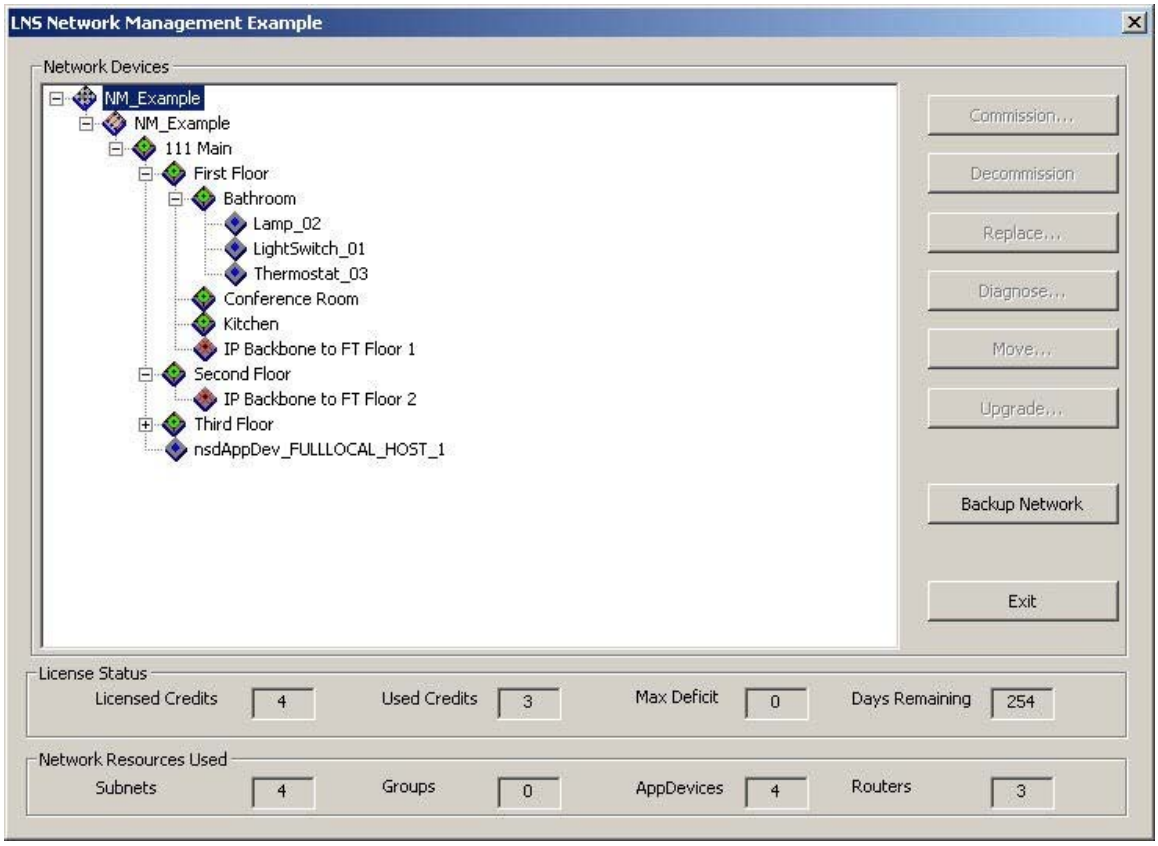


Figure C.2 Performing Network Management Tasks

The tree in the **Network Devices** pane represents the layout of the network, subsystem by subsystem. Each device is displayed in the subsystem that it belongs to. For example,

in Figure C.2, the Lamp_02, Lightswitch_01 and Thermostat_03 devices all belong to the 111 Main.First Floor.Bathroom subsystem.

The bottom part of the dialog provides LNS licensing information relevant to your network, and information on the resources available to your network. For source information on this part of the dialog, see the `lcahelper.cpp` file and `CDlgMain::RefreshStatistics()` in the `dlgmain.cpp` file.

Network management operations that you can perform are listed on the right side of the dialog shown in Figure C.2. If you are using the network setup of LTM-10A devices recommended earlier in this chapter, you can perform all of the tasks described in the rest of this section.

The following procedure describes how you could use the example network management application to commission a device with the network management dialog:

1. If you are using an *i*.LON 600 LONWORKS/IP Server, select the **IP Backbone to FT Floor 1** router in the tree and click the **Commission** button to provide a Neuron ID for the *i*.LON 600 LONWORKS/IP Server. The network management code will identify the near side of the router and associate the Neuron ID with it.

   If you are not using an *i*.LON 600 LONWORKS/IP Server, you should move your Network Service Device to the same floor or room as the devices you plan to manage. To do so, select the NetworkServiceDevice in the **111 Main** subsystem (the **NsdAppDev_FULLLOCAL_HOST_1** device). Then, click the **Move** button and select **111 Main/First Floor/Bathroom** as the destination subsystem. Because the example maintains a relationship between all subsystems, the `AppDevice` object representing the Network Service Device will be move to the appropriate channel and subsystem simultaneously. For source code information, see `CDlgMove::OnOK` in the `CdlgMove.cpp file`, and `LcaNetMgmtHelper::EngineeredMoveOfDeviceToSubsystem()` in the `LcaNetMgmtHelper.cpp` file.

2. Select the Lamp_02 device from the tree hierarchy and click **Commission**. This will associate the `AppDevice` or `Router` object in the LNS database with a physical device on the network.

   For source code information on these tasks, see `CDlgIdentify::OnOk` and `CDlgMain::OnBtnCommission()` in the `dlgmain.cpp` file.

3. After you click the **Commission** button, a dialog will open requesting that you select a device identification method. Select either service pin, manual entry or device discovery as the device identification method. These methods are described in Chapters 5 and 6 of this document.

4. Check the **Load Application** checkbox. This opens a dialog that allows you to specify how the configuration properties on the device will be set. You can choose from the following:

   - Download current values to devices
   - Upload values from device
   - Reset device to default values

- Set device template defaults from device
- Upload value set to unknown in the LNS database
- Set all configuration property values to unknown.

Each of these choices causes the application to call either the `DownloadConfigProperties()` or the `UploadConfigProperties()` methods with the appropriate options set. Consult Chapter 6 of this document, or the *LNS Object Server Reference* help file, for more information on these methods.

5. Click **OK**. If no errors are encountered, focus will return to the dialog shown in Figure C.2. You will notice all pushbuttons on the right side are active. You have successfully commissioned your device.

Table C.2 describes some of the other tasks you can perform from the dialog shown in Figure C.2. To perform them, select a device, and then click the applicable button.

Table C.2 Other Network Management Tasks

Item	Description
Decommission Button	Click the **Decommission** button to decommission a device or router. `CDlgLcaEventHandlerContainer` will receive the resulting `OnLicenseEvent` and forward it to the `CDlgMain` for processing. For source information, see the `ObjServer OnSystemServicePin` handler in the `DlgMain.cpp` file.
Replace Button	Click the **Replace** button to update the Neuron ID for the device and then call the `Replace()` method to replace the device. The LNS Object Server will attempt to decommission the existing device before commissioning the replacement. For source code information, see `CDlgIdentify::OnOk` in the `DlgIdentify.cpp` file.

Item	Description
Diagnose Button	This command exposes the following network management tasks of the `AppDevice` or `Router` objects: • Wink Device (`AppDevice` objects only) • Go Offline • Go Online • Test Device • Reset Device These operations, along with a Results Pane are displayed through the Device Diagnostics dialog that opens when you select the **Diagnose** button.
Move Button	Click the **Move** button to move a device to another subsystem
Upgrade Button	Click the **Upgrade** button to upgrade the device interface, application and device firmware. When you click the **Upgrade** button, the **Upgrade Device** dialog opens, which requires you to enter the following information: • Device Template Name • Device Interface Source
Backup Network Button	This operation performs a backup of the current network. The image path of a backed up network is based on the database path with "_" appended. Also, the backed up network is imported into the `Networks` collection for validation. In case a problem is encountered, the user is prompted to review the diagnostic logic and attempt a database repair.

Source Code Mappings

For your reference, Table C.2 lists the various tasks performed by the example network management application, and the names of the classes containing the code that invokes these tasks.

Table C.2 Example Network Management Application

Task	GUI Code	LNS Interface Code

Initialize and open the LNS ObjectServer	`CDlgInitialize`
	`CDlgInitializeLicense`
	`CDlgInitializeGeneral`
		`CLcaHelper::OpenObjectServer`

Create a network database based on a database template.	`CDlgInitializeGeneral`	`CLcaNetMgmtHelper::CreateNetwork()`
Add an LNS Server reference for a Lightweight client application.	`CDlgInitializeGeneral`	`CLcaNetMgmtHelper::CreateNetworkReference()`

Open a network.	`CDlgMain::OnInitDialog`	`CLcaNetMgmtHelper::OpenNetwork`
		`CLcaHelper::SetSelectedNetworkName(sNetworkName);`
		`CLcaHelper::OpenNetwork()`

Backup and validate a network database.	`CDlgMain::OnBtnBackup()`	`CLcaNetMgmtHelper::NetworkBackup`
Import external interface files.	`DlgUpgrade.xif`	
Create engineered mode devices for which the Neuron ID is provided by user input, and the device interface extracted from an external interface file.	`CDlgIdentify::OnOK`	`CLcaNetMgmtHelper::CreateTemplateNetwork()`
Create ad hoc devices for which Neuron ID is identified by a `OnSystemServicePin` event, or selected from a list of discovered devices, The device interface is extracted from the self documentation information on the device.	`CDlgIdentify::OnOK`	
Manual Neuron ID entry.	`CDlgIdentify_NidByTxt`	

Service-pin Neuron ID discovery.	`CDlgIdentify_NidByServicePin`
	
	`CDlgIdentify_NidByDiscovery::DiscoverDevices`
		`CDlgIdentify_NidByServicePin::OnInitDialog`
		`CDlgLcaEventHandlerContainer::OnCmdMsg`
		`CDlgIdentify_NidByServicePin::OnSystemServicePin`
		`CDlgIdentify_NidByServicePin::OnCancel`
		
		`CLcaNetMgmtHelper::BeginChangeEvent`
		`CLcaNetMgmtHelper::DiscoverDevices Asynchronous`

Task	GUI Code	LNS Interface Code
Creating connections between devices.		CLcaNetMgmtHelper::CreateNetworkVariableConnections_ForEventBasedMonitoring CLcaNetMgmtHelper ::CreateNetworkVariableConnections_ForHVAC CLcaNetMgmtHelper::CreateNetworkVariableConnections_ForLightingControl
Configuring devices using ConfigProperty objects.	CDlgIdentify_CpUpdate	CDlgIdentify_CpUpdate::OnOK
Testing devices.	CDlgDiagnostics	CDlgDiagnostics::OnTest
Moving devices between subsystems and between channels.	CDlgMain::OnBtnMove	CLcaNetMgmtHelper::EngineeredMoveOfDeviceToSubsystem
Adding channels.		CLcaNetMgmtHelper::CreateNetworkObjects
Loading application images.	CDlgUpgrade::OnOK	CLcaNetMgmtHelper ::UpgradeDevice
Upgrading devices.	CDlgUpgrade::OnOK	CLcaNetMgmtHelper ::UpgradeDevice
Licensing tasks.	CDlgMain	CDlgLcaEventHandlerContainer::OnCmdMsg CDlgMain::OnLicenseEvent
Replacing a Network Service Device.	CDlgNsdReplace	CLcaNetMgmtHelper ::SetNetworkServiceDevice

Monitor and Control Example

Once you have created a network with the network management example application, you can use the monitor and control example application to monitor and control the application. The monitor and control example utilizes a variety of LNS features you can use to do so. The following sections describe these features. Remember that you can only use the monitor and control example application with networks created with the network management example application.

To start the monitor and control example application, select **Echelon LNS Application Developer's Kit>Examples & Tutorials>LNS Monitor and Control Example** from the Windows Programs menu. This opens the dialog shown in Figure C.3.

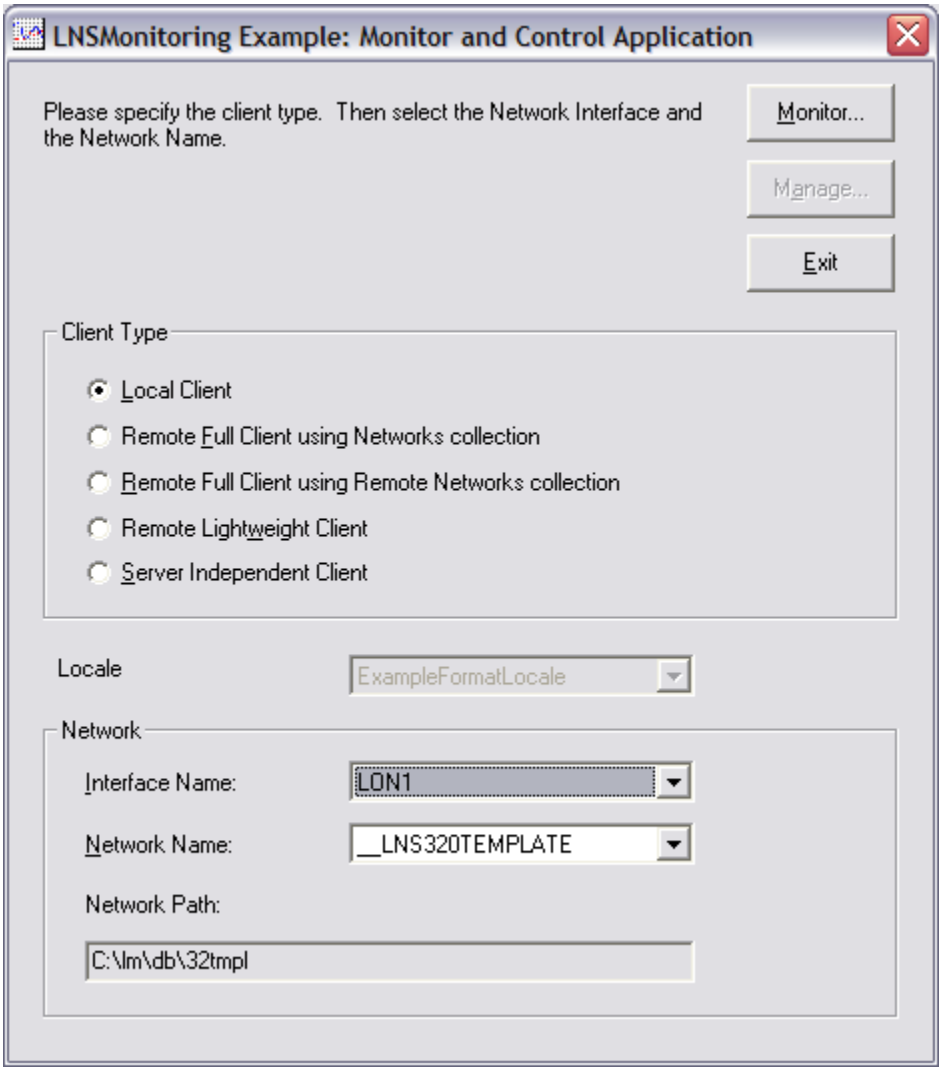


Figure C.3 Example Monitor and Control Application

The following section describes how you can use this dialog, and reference the location of the source code you can view to see how the tasks you perform are accomplished using LNS.

1. First, select a client type. This determines how the example monitor and control application will connect to the LNS Server. The different client types are introduced in Chapter 3 of this document. The client type you select determines which networks and network interfaces will be available from the **Network Name** and **Network Interface** drop-down boxes you will use in the following steps.

   Note that for remote client types, you will need to have the LNS Server running on the same PC that contains the LNS global database. This is the PC on which you created the network with the network management application.

   In addition, if you selected **Remote Lightweight Client**, you will need to perform the following steps before proceeding to step 2:

1. Close the monitor and control application, and launch the network management application from the client PC.

2. Select **Remote Lightweight Client** as the **Client Type** option.

3. Type in the network name

4. The **Network Path** textbox will turn into **Server IP address and port** and will be enabled. Enter this information using the following format: <lns:// xxx.xxx.xxx.xxx:2540/>.

5. Click **Manage**.

6. Close the network management application. Then, restart the monitor and control application and continue with step 2 below.

Table C.3 lists where you can find source code information for each possible selection:

Table C.3 Source Code Information

Selection	**For Source Code Information, See….**
Local Client	`CLcaMonitorDlg::OnSelectLocalClient()` in the `LcaMonitorDlg.cpp` file.
Remote Client Using Networks Collection	`CLcaMonitorDlg::OnSelectFullClientUseNetworksCol()` in the `LcaMonitorDlg.cpp` file.
Remote Client Using Remote Networks Collection	`CLcaMonitorDlg::OnSelectFullClientUseRemoteNetworksCol()` in the `LcaMonitorDlg.cpp` file.
Remote Lightweight Client	`CLcaMonitorDlg::OnSelectLightWeightClient()` in the `LcaMonitorDlg.cpp` file.
Server Independent Client	`CLcaMonitorDlg::OnSelectIndependentClient()` in the `LcaMonitorDlg.cpp` file.

2. Select a network interface from the **Network Interface** drop-down box. Typically, this should be the network interface that the LNS Server PC is using. With certain client types, this option will be selected for you. (i.e. Remote Lightweight Client).

   For source code information, see `CLcaMonitorDlg::OnSelchangeNetworkInterfaceName()` in the `LcaMonitorDlg.cpp` file.

3. Select a network created with the example network management application from the **Network Name** drop-down box. This lists the

names of all networks in the `Networks`, `RemoteNetworks` or `VNINetworks` collections, as dictated by the client type selected in step 1.

If **Server Independent** was selected, the `VNINetworks` collection is used. If **Remote Full client using Remote Networks Collection** was selected, the `RemoteNetworks` collection is used. In all other cases, the `Networks` collection is used.

For source code information, see `CLcaMonitorDlg::OnSelchangeNetworkName()` in the `LcaMonitorDlg.cpp` file.

4. Once you have selected a network, click the **Monitor** button to begin monitor and control operations. Note that the **Locale** field contains the `FormatLocales` collection included with the example. This field is static and read-only.

5. When you click the **Monitor** button, the dialog shown in Figure C.4 opens.

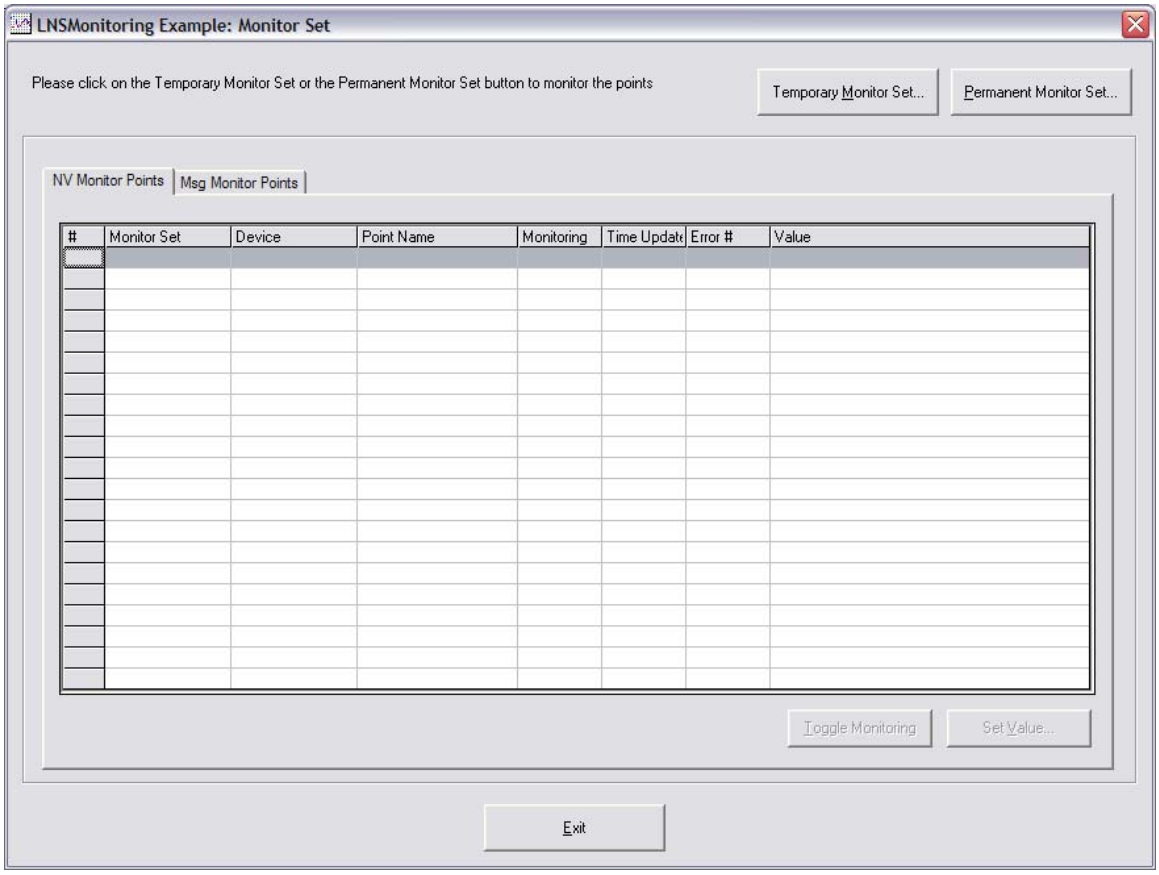


Figure C.4 Monitor Set Dialog

6. Click the **Temporary Monitor Set** button to perform monitor and control operations with temporary monitor sets, or the **Permanent Monitor Set** button to perform monitor and control operations with permanent monitor sets. See Chapter 9 of this document for an overview of the differences between the two types of monitor set.

7. If you clicked the **Temporary Monitor Set** button in step 6, a dialog with a tree view of the AppDevice objects installed on your network will open. Select the `AppDevice` object containing the network variable(s) you want to monitor.

   A temporary monitor set will be created containing a network variable monitor point for every network variable on the devices main and custom interfaces.

   For source information and specific properties set for these objects see `CPrimaryFormDlg::OnTemporaryMonSet()` in the `PrimaryFormDlg.cpp` file. Also, see the `TempMonitorSetDlg.cpp` file.

8. If you clicked the **Permanent Monitor Set** button in step 6, a dialog will display allowing you to open one or more permanent monitor sets. For source information, see `CPrimaryFormDlg::OnPermanentMonSet()` in the `PrimaryFormDlg.cpp` file, and the `PermMonitorSetDlg.cpp` file.

9. Once you have completed step 7 or 8 and created a monitor set, you can choose from the following options:

   - Click the **Query Device Status** button to query the status of the devices using the message monitor points available in the currently selected monitor set.
   - Click the **Toggle Monitoring Button** to enable or disable monitor points.
   - Click the **Set Value** button to write to the value of an input network variable monitor point.

Source Code Mappings

For your reference, Table C.4 lists the various tasks performed by the example monitor and control application, and lists the names of the classes containing the code that invokes these tasks.

Table C.4 Example Monitor and Control Application

Task	GUI Code	LNS Interface Code
Creating monitor sets and monitor points.		`CLcaMonitorHelper::CreateExampleMonitorSets`

Browsing explicitly polled and bound monitor points.	`Opening a monitor set:`
	`CPrimaryFormDlg::OnPermanentMonSet`
	`CPrimaryFormDlg::OnTemporaryMonSet`
	`Handling updates`
	`CPrimaryFormDlg::OnTimer`
		`Opening a Monitor Set:`
		`CLcaMonitorHelper::OpenCurMonitorSet`
		`Assigning an event handler`
		`CEventSink::OnNvMonitorPointEvent`
		`Handling events`
		`CMonitorPointListener:: raw_UpdateEvent`
		`CMonitorPointListener::raw_UpdateErrorEvent`

Task	GUI Code	LNS Interface Code

Use of the `Tag` property for "pigeon-holing" data values		`CLcaMonitorHelper::CreateMonitorPointTag in`
		`CLcaMonitorHelper:: AddNVMonitorPoints`
		`CLcaMonitorHelper:: AddMsgMonitorPoints`
		`CLcaMonitorHelper::AddTemporaryNVMonitorPoints`
		`CLcaMonitorHelper::AddTemporaryMsgMonitorPoints`

Change event tracking.		`Handling events` `CEventSink::OnNvMonitorPointEvent`
Explicitly reading or writing a monitor point.	`CUpdateValueDlg`	`CUpdateValueDlg::OnWrite` `CUpdateValueDlg::DataPointRead`
Sending explicit, synchronous messages using message monitor points.	`CDeviceStatusDlg::OnQueryStatus`	`CLcaMonitorHelper::QueryDeviceStatus`

xDriver Example Applications

The xDriver example applications are described in detail in the *OpenLDV Programmer's Guide, xDriver Supplement,* which can be downloaded from Echelon's website at:

http://www.echelon.com/support/documentation/default.htm

Example Director Application

LNS Turbo Edition also includes an example director application, as referenced in the *Implementing an LNS Director Application* section in Chapter 12 of this document. To start the example director application, select **Echelon LNS Application Developer's Kit>Examples & Tutorials>LNS Plug-In Director Example** from the Windows Programs menu. This opens the dialog shown in Figure C.5.

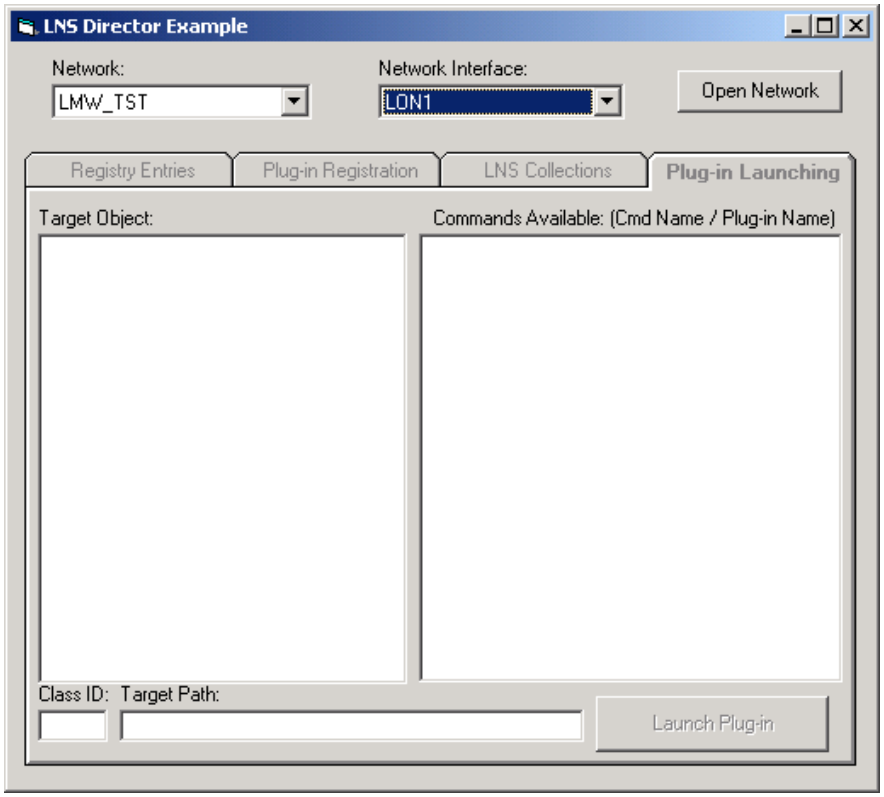


Figure C.5 Example Director Application Dialog

To use the example director application, select the network you want to open from the **Network** drop-down box. Then, select a network interface from the **Network Interface** drop-down box and click **Open Network.** You can then use the 4 tabs of the dialog shown in Figure C.5, as described below:

- Select the **Registry Entries** tab to display the keys and associated class IDs for the plug-in applications associated with the network. These are stored in the (Default) string in the Windows Registry under `HKEY_LOCAL_MACHINE\SOFTWARE\LonWorks\LCA\Plug-Ins`. Click the **Refresh** button reinitializes the internal data structure with registry information and updates the list.

- Select the **Plug-In Registration** tab to register a plug-in application, or to view plug-in applications that have already been registered. This tab contains three lists:

  The **Unregistered Plug-Ins** list contains the plug-in application names found in the Windows Registry for which a `ComponentApp` object with the `lcaCommandIdRegister` command is not found in the LNS database. Select an item from this list, and click the **Register plug-in** button to register the plug-in.

  The **Object Server Registered** list contains the names of plug-in applications that have been registered with the LNS Object Server, and are available to all networks. It lists the associated `RegisteredServer` property of the `ComponentApp` objects for the `lcaCommandIdRegister` command in the `ComponentApps` collection of the Object Server scope next to the plug-in application's name.

The **System Registered** list shows the names of the plug-in applications that have been registered for use with the open network's system. It lists the associated `RegisteredServer` property of the `ComponentApp` object for the `lcaCommandIdRegister` command in the `ComponentApps` collection of the `System` scope.

- Select the **LNS Collections** tab to display the list of all the `ComponentApp` objects in the database, based on scope.
- Select the **Plug-in Launching** tab to invoke commands on target devices on the network. You can select the object to be acted upon from the **Target Object** tree on the left. When you select an object of type `lcaClassIdAppDevice`, all of the commands that apply to the device template for that object will be displayed in the **Commands Available** tree.

  The included commands are the `ComponentApp` objects registered at the `AppDevice`, `System` and `ObjectServer` scopes. Regardless of the object type, the **Class ID** and **Target Path** fields at the bottom of the dialog display information from the selected object. Once an item is selected from the **Commands Available** table, the **Launch Plug-in** button becomes active. This allows the end user to create an instance of the plug-in and send it the selected command.

For your reference, Table C.5 lists the various tasks performed by the example director application, and lists the names of the classes containing the code that invokes these tasks.

Table C.5 Example Director Application

Taks	**GUI Code**	**LNS Interface Code**
Initialize and open the LNS Object Server.	`FrmMain.InitNIs` `FrmMain.InitNetworks`	`FrmMain.InitLNS` `FrmMain.btnOpenNetwork_Click`
List plug-in applications previously registered with Windows via the Windows Registry.	`FrmMain.ListPlugins` `FrmMain.ProcessPlugIns`	`CPluginRegistry.InitFromRegistry`
List plug-ins commands previously registered with the LNS Object Server or System.	`FrmMain.ListPlugins`	`FrmMain.ProcessPlugIns`
Instantiate, initialize, and send a register command to a plug-in application. Note successful registration by adding a `ComponentApp` object.	`FrmMain.btnRegister_Click`	
Send a command to a plug-in application.	`FrmMain.BtnLaunch_Click`	`CCommand.Launch`